UNIVERSIDAD CARLOS III DE MADRID

Tecnologías informáticas de la Web

TEMA 1 – JSP, Servlet y Filtros

Servlets, Sesiones y Filtros

Jesús Hernando Corrochano

Telmo Zarraonandia Ayo

Curso 2011-12

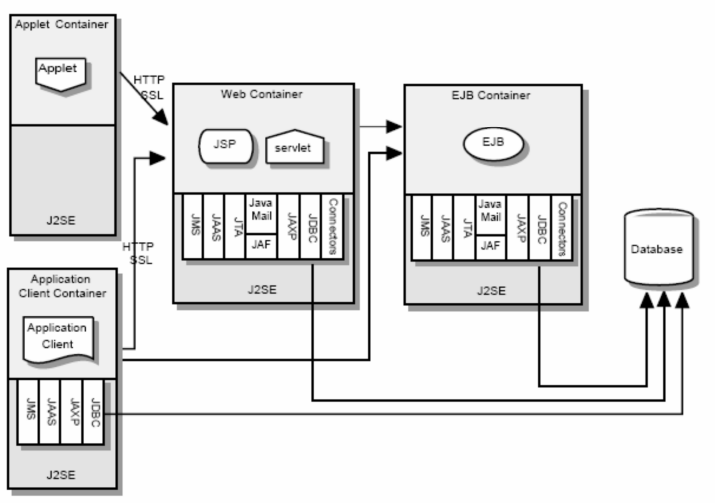
ÍNDICE

Contenedor / Servidor

Un contenedor es un sistema que proporciona un entorno estandarizado en periodo de ejecución, para gestionar los componentes de aplicación, JSP, Servlet,…y cualquier otro tipo de componentes Java. Estos entornos proporcionan una serie de servicios que liberan al desarrollador de ciertas tareas ya estandarizadas.

- Un contenedor es un entorno de ejecución para un componente, el componente vive en el contenedor y este proporciona servicios al componente. De modo similar un contenedor vive dentro de un servidor de aplicaciones que le proporciona un entorno de ejecución para él y para otros contenedores.
- Un servidor de aplicaciones es un producto que implementa (se ajusta) al estándar JEE y ofrece un contenedor Web y un contenedor EJB, como mínimo.
- Los componentes JEE son denominados "objetos gestionados" ya que son creados y gestionados en el interior del periodo de ejecución del contenedor.

La gráfica ilustra los distintos contenedores denotados en la especificación JEE, así como los elementos que destinados a ejecutarse dentro de los mismos.



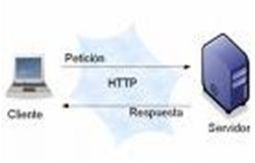
- El contenedor WEB: proporciona mecanismos para interceptar solicitudes HTTP, SMP, FTP, .. .
- Contenedor Cliente de Aplicación: Proporciona servicios para aplicaciones que se ejecutan en una JVM diferente a la del Application Server. Es un nuevo concepto dentro de la especificación JEE y deben ser facilitados por el proveedor de servidor de aplicaciones. Pueden optimizar el acceso a un contenedor del Application Server, teniendo acceso entre otros a JAXP; JDBC; JMS y JAAS en un servidor remoto.
- Contenedor Applet: proporciona servicios para interceptar las solicitudes de programas que se ejecutan dentro de un navegador.
- Contenedor EJB: proporciona servicios para interceptar las solicitudes de lógica de empresa (EJBs)

Servlet: Introducción a los Servlets

¿Qué es un Servlet?

Vamos a describir varias definiciones de servlet:

- Según la tecnología: "Un servlet es una clase en lenguaje Java usada para ampliar la funcionalidad de los servidores web a los que se accede vía modelo de programación request-response."
- Según la arquitectura: un servlet es un componente Web que se ejecuta dentro de un contedor web y genera contenido dinámico.
- Según la programación. Los servlets son pequeñas clases Java independientes de la plataforma compiladas en bytecode que pueden ser cargadas dinámicamente y ejecutadas dentro de un servidor web.
- Por su diferencia con los applets: los applets son programas Java que se descargan desde el servidor y se ejecutan en el cliente. No tiene acceso a otro servidor que no sea el que ha sido descargado. Los servlets, sin embargo, son programas java que se ejecutan en el servidor y reciben peticiones de un cliente web.
- Según el protocolo http. Servlet es un servidor especializado que recibe peticiones http, genera contenido dinámicamente, realiza lógica y devuelve una respuesta http.



Los servlets se ejecutan dentro de un contenedor web (motor de servlets) que traduce las peticiones propias del protocolo http a llamadas a objetos Java (recibe respuestas) y traduce las respuestas procedentes de objetos Java a respuestas nativas del protocolo.

Finalidad de los Servlets.

Aunque los servlets pueden responder a cualquier tipo de petición, se usan comúnmente en aplicaciones alojadas en un servidor Web. Para dichas aplicaciones, la tecnología Java Servlet define clases servlet Http.

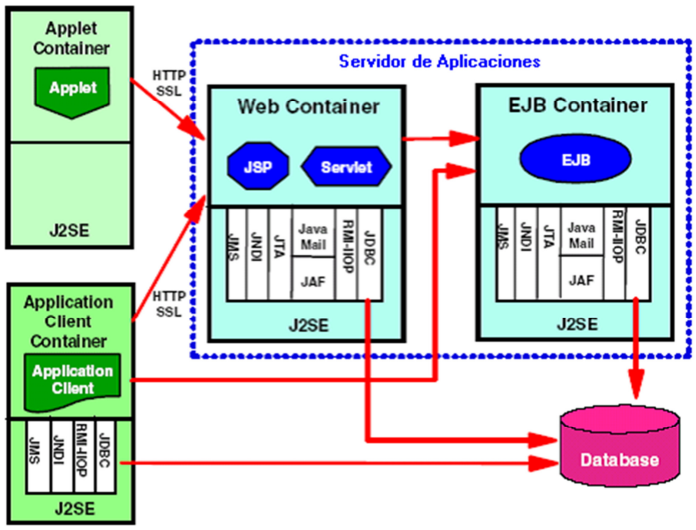
Los paquetes javax.servlet y javax.servlet.http facilitan clases e interfaces para escribir servlets.

Cuando se implementa un servicio genérico (no necesariamente un servicio http), puedes usar o ampliar la clase GenericServlet suministrada con la API Java Servlet. La clase HttpServlet facilita los métodos, como doGet y doPost, para manejar los servicios HTTP necesarios.

Características de los servlets.

- Los Servlets se utilizan para extender o implementar la funcionalidad del servidor.
- Se ejecutan en una máquina virtual (JVM) dentro del proceso del servidor.
- Son módulos de software que se ejecutan dentro del entorno Web de un servidor y proveen servicios de petición/respuesta.

- Son componentes escritos en Java, situados en los servidores e independientes de cualquier protocolo y de cualquier plataforma.
- Los servlets no tienen representación gráfica. Diferencia con los applets que SI tienen representación gráfica.
- Un servlet se instancia la primera vez y se mantiene en memoria esperando nuevas invocaciones (el servidor web tiene una máquina virtual que es la que se ejecuta el Servlet.
- Los servlets pueden ejecutarse en distintos tipos de Servidores (o contenedores Web), y se alojan en los mismos dentro del contenedor Web. Por ejemplo: Java Web Server, TomCat, JRUN, WebSphere, BEA, etc…



Algunas de las características de los Servlets http, que podemos citar son:

- Los servlets http se cargan directamente en el servidor web y ofrecen acceso interactivo a bases de datos y sistemas propietarios.
- API de los Servlets http (paquete javax.servlet.http)
  - HttpServlet (clase que hereda de GenericServlet)
  - HttpServletRequest (hereda de ServletRequest)
  - HttpServletResponse (hereda de ServletResponse)
- Un servlet http puede extender la clase HttpServlet . El método service no hay que programarlo directamente, pues éste en su implementación llama a los métodos doGet o doPost para las dos formas de enviar información a través de HTTP.
- El método service de HttpServlet haría lo siguiente:

```
service(req, res) {

        Si es Get-→ doGet()

        Si es Post→ doPost()
```

- GET y POST son formas de enviar información a través del protocolo http.

- GET envía dentro de la URL los parámetros para el servlet. Por tanto, GET permite que se llame al servlet directamente pasándole los parámetros a través de la URL (QueryString)
- POST encapsula los parámetros en la trama que se envía por el protocolo http.

  Para enviar datos a través de POST es necesario tener una página Web con un formulario que se encargue de enviar la información vía POST. Esta forma también se podría utilizar para GET.

- Existen otras peticiones que se pueden hacer: PUT (permite al cliente ubicar un fichero en el servidor, es similar a enviar un fichero por ftp) y DELETE (permite al cliente eliminar un documento o paquete del servidor web).

Ventajas fundamentales de los Servlets

- **Eficiencia.**
  - Instancia permanentemente cargada en memoria por cada servlet. Cada petición se ejecuta en un hilo y no en un proceso.
  - Ejecución de peticiones mediante invocación de un método
  - Mantiene automáticamente su estado y recursos externos.
  - Los servlets son más rápidos que los CGI debido a que utilizan *threads* en lugar de *procesos*.
  - Con los servlets solo hay una copia del servlet en memoria. Cada petición http es gestionada por un subproceso o Hilo.
  - Con los servlets, la máquina Virtual Java permanece arrancada, y cada petición es manejada por un Thread Java de peso ligero, no un pesado proceso del sistema operativo.
  - Con los Servlets, hay **n** threads pero sólo una copia de la clase Servlet. Los Servelt también tienen más alternativas que los programas normales CGI para optimizaciones como los cachés de cálculos previos, mantener abiertas las conexiones de bases de datos, etc.

- **Seguridad.**

  Por ser Java ofrece: máquina virtual, chequeo de tipos, gestión de memoria (garbage collector), excepciones, etc.

  Pudes acceder al API de seguridad de java.

- **Integración (Potencia de la plataforma)**

  Integración fuerte entre servlets y servidor. Los servlets pueden hablar directamente con el Contenedor Web accediendo a las APIs de JEE que de manera ortogonal ofrece el servidor de aplicaciones.

- **Portable.**

  Los Servlets están escritos en Java y siguen un API bien estandarizado.

- **Extensibilidad y flexibilidad**
  - La API de los Servlets es extensible.
  - Filtros (cadenas de Servlets). Los veremos en otras fases de ese plan de formación..
  - Integrable con JSP's.

Estructura de un servlet

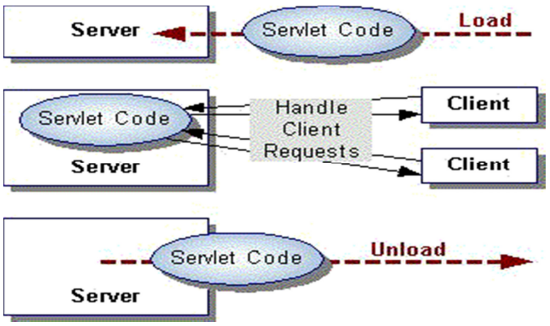
El código básico de un servlet podría ser este:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
        public class Prueba extends HttpServlet
        {
                public void doGet( HttpServletRequest request,HttpServletResponse response )
                throws ServletException, IOException
                {...
                }
        }
```

Pasos:

1. Se importan las clases necesarias
2. Se declara la clase Servlet extendiendo de HttpServlet o de Generic Servlet
3. Se declaran los métodos que se desee sobrescribir. EN este caso doGet(request,response).

Ciclo de Vida de los Servlets



Una vez visto el contenedor web y la estructura básica de un servlet , ya se puede introducir el ciclo de vida o las diferentes etapas en la ejecución de un servlet. El ciclo de vida de un servlet se controla por el contenedor en el cual el servlet ha sido instalado/desplegado. Cuando una petición es mapeada a un Servlet (invocación), el contenedor realiza los siguientes pasos:

- Si la instancia del servlet no existe, el Contenedor:
  - Carga la clase servlet (fichero .class en memoria)
  - Crea una instancia de la clase servlet (constructor del Servlet)
  - Inicializa la instancia del servlet llamando al método init.
- Invoca al método service, pasando un objeto Petición/Respuesta.
- Si el contenedor necesita eliminar el servlet, se finaliza el servlet llamando al método destroy del Servlet.

Es decir, el motor de servlets llama a los métodos init, service y destroy, por este orden. Los desarrolladores de aplicaciones también pueden crearlos "objetos escuchadores", que son objetos que responden a ciertos eventos.

Métodos del ciclo de Vida

El método init()

El contenedor de servlet llama al método init() nada más haberse cargado el servlet en memoria, como hemos visto.

El Servlet se carga en memoria dependiendo de un parámetro de nuestro fichero Descriptor de Despliegue (web.xml): puede cargarse durante el arranque de la aplicación o puede cargarse cuando se reciba la primera petición. Por tanto, la llamada al método init del servlet puede que se realice bastante después que la llamada a su constructor.

El método init() esta sobrecargado:

- init(ServletConfig config). El objeto tipo ServletConfig contiene el método getInitParameter(String) que obtiene los valores de configuración.

```
void init(ServletConfig config){
…}
```

- init(): Sin parámetros

```
void init(){
…}
```

Una vez que el contenedor web carga e instancia el servlet en memoria y antes de entregar peticiones desde el lado cliente, el contenedor web inicializa el servlet. Se puede personalizar este proceso de inicialización para leer datos de configuración persistentes, inicializar recursos, y realizar cualquier otra acción sobrescribiendo el método init de la interfaz Servlet.

El método service

El método service del servlet es llamado cada vez que el servlet recibe una petición del usuario.

En ese momento, el contenedor de servlets pasa una instancia de la clase ServletRequest, que contiene los datos de la petición.

El método service de HttpServlet crea los objetos HttpServletRequest y HttpServletResponse y los pasa como parámetros a los métodos doGet y doPost de nuestros Servlet.

```
void Service(HttpServletRequest request, HttpServletResponse response){
..}
```

El método destroy

Es llamado por el contenedor de servlets para informar al servlet que está a punto de ser eliminado. Solo será llamado una vez.

API de los Servlets

javax.servlet

Interfaces
RequestDispatcher
Servlet
ServletConfig
ServletContext
ServletRequest
ServletResponse
SingleThreadModel

Classes
GenericServlet
ServletInputStream
ServletOutputStream

Exceptions
ServletException
UnavailableException

javax.servlet.http

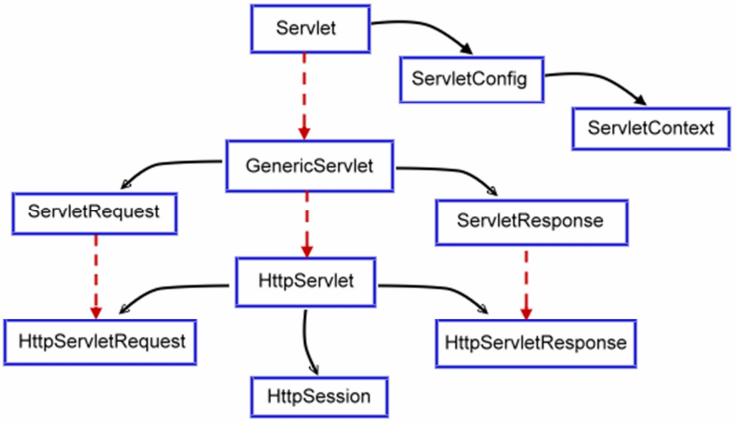
Interfaces
HttpServletRequest
HttpServletResponse
HttpSession
HttpSessionBindingListener
HttpSessionContext

Classes
Cookie
HttpServlet
HttpSessionBindingEvent
HttpUtils

Paquetes javax.servlets.* y javax.servlet.http.*

Este API presenta dos paquetes:

- Javax.servlet.*: clases e interfaces independientes del protocolo
- Javax.servlet.http.*: clases e interfaces específicas del protocolo http (objeto de este módulo)



La Interfaz servlet

Todo Servlet debe implementar la interfaz Servlet que define los métodos del ciclo de vida.

Es decir, para que una clase java se considere un Servlet debe implementar la interfaz Servlet.

Pero alternativamente se puede:

- Heredar de la clase javax.servlet.GenericServlet, independiente del protocolo.
- Heredar de la clase javax.servlet.http.HttpServlet: dependiente del protocolo (objetivo de este módulo)

Ambas clases implementan la interfaz Servlet, por tanto, viendo los principales métodos de esta interfaz analizaremos la funcionalidad de nuestro Servlet.
Métodos importantes de la interfaz:

- *void init(ServletConfig config):*

  Cada vez que se inicia el servlet el servidor web llama a este método pasando un parámetro de la clase **ServletConfig** que guarda información de la configuración del servlet y del contexto del servidor web en el que se ejecuta.
- *void service(ServletRequest req, ServletResponse res):*

  En este método se encuentra la mayor parte de la funcionalidad del servlet.

  Método que recibe un objeto ServletRequest(encapsula la petición) y un objeto ServletResponse(encapsula la respuesta al cliente).

  - **Request**: Petición del cliente al servidor
  - **Response**: Respuesta del servidor para el cliente
- *void destroy()*

  Cada vez que el Servlet desaparece o se destruye del entorno de ejecución.
  Será llamado por el servidor web cuando el servlet está a punto de ser descargado de memoria(repito e insisto: no cuando termina una petición).En este método se han de realizar las tareas necesarias para conseguir una finalización apropiada como cerrar archivos y flujos de entrada de salida externos a la petición, cerrar conexiones persistentes a bases de datos...etc..

La Clase HttpServlet

- Clase que hereda de GenericServlet , que a su vez implementa la interfaz Servlet.
- La clase HttpServlet además de implementar estos dos métodos implementa los métodos doGet, doPost, doPUT, doDelete

Métodos:

Resumen Métodos	
protected void	**doDelete**(HttpServletRequest req, HttpServletResponse resp) Invocado por el contenedor (via el metodo service) para manejar una peticion DELETE
protected void	**doGet**(HttpServletRequest req, HttpServletResponse resp) Invocado por el contenedor (via el metodo service) para manejar una peticion GET
protected void	**doHead**(HttpServletRequest req, HttpServletResponse resp) Recibe una petición HTTP HEAD desde el metodo servicequest from the protected service method and handles the request.
protected void	**doOptions**(HttpServletRequest req, HttpServletResponse resp)

	Invocado por el contenedor (via el metodo service) para manejar una peticion OPTIONS
protected void	**doPost**(HttpServletRequest req, HttpServletResponse resp) Invocado por el contenedor (via el metodo service) para manejar una peticion POST
protected void	**doPut**(HttpServletRequest req, HttpServletResponse resp) Invocado por el contenedor (via el metodo service) para manejar una peticion PUT
protected void	**doTrace**(HttpServletRequest req, HttpServletResponse resp) Invocado por el contenedor (via el metodo service) para manejar una peticion TRACE
protected long	**getLastModified**(HttpServletRequest req) Devuelve en tiempo en milisegundos desde la ultima vez que el metodo HttpServletRequest fue modificado.
protected void	**service**(HttpServletRequest req, HttpServletResponse resp) Recibe una peticion HTTP standard desde el metodo publico service y redirige la acción a los metodos doXXX correspondientes definidos en la clase Servlet.
void	**service**(ServletRequest req, ServletResponse res) Redirige las peticiones del cliente al método service protected.

La Clase ServletRequest

- Clase abstracta que encapsula la petición del cliente al servidor
- Clase abstracta que contiene métodos públicos para la lectura

La Clase ServletResponse

Clase abstracta que encapsula la respuesta del servidor al cliente

La Clase ServletConfig

Interface para pasar información del contenedor de Servlets al Servlet

Métodos importantes:

Resumen de los métodos	
java.lang.String	**getInitParameter**(java.lang.String name) Devuelve un String que contiene el valor del parámetro de inicializacion dado, o null si no existe el parámetro. Los parámetros de inicialización se encuentran en la etiqueta: <servlet> <init-param><param-name></param-name></init-param> </servlet>
java.util.Enumeration	**getInitParameterNames**() Devuelve los nombres de los parámetros de inicializacion como un Enumeration de objetos String o como una Enumeration vacia si no existen parámetros.
ServletContext	**getServletContext**() Devuelve una referencia al ServletContext en el cual el Servlet se está ejecutnado.
java.lang.String	**getServletName**() Devuelve el nombre de la instancia del servlet.

La Clase ServletContext

Interfaz que permite usar información del context donde se ejecuta el Servlet.

Métodos destacados:

Method Summary	
java.lang.Object	**getAttribute**(java.lang.String name) Devuelve el valor atributo del contenedor de Servlets dado un nombre de atributo.
java.util.Enumeration	**getAttributeNames**() Devuelve una Enumeration que contiene los nombres de atributos disponibles.

	Si no existe ningún atributo devolverá una Enumeration vacía.
ServletContext	**getContext**(java.lang.String uripath) Devuelve un objeto ServletContext dada una URI determinada.
java.lang.String	**getInitParameter**(java.lang.String name) Devuelve una String con el valor del atributo que corresponde a la etiqueta <context-param><param-name> que se pasa como parámetro. Si el parámetro no existe en el web.xml devuelve nulo.
java.util.Enumeration	**getInitParameterNames**() Devuelve los nombres de los parametros del del context(web.xml) como una Enumeration. Si no existen los parámetros la Enumeration estará vacía.
int	**getMajorVersion**() Devuelve la version principal de la Servlet API que soporta el contenedor de servlets. En nuestro caso, 2.3
java.lang.String	**getMimeType**(java.lang.String file) Devuelve el tipo MIME del archive especificado. Si no se conoce el tipo MIME devolverá nulo.
int	**getMinorVersion**() Devuelve la version secundaria de la Servlet APIReturns the minor version of the Servlet API .
RequestDispatcher	**getNamedDispatcher**(java.lang.String name) Devuelve un objeto RequestDispatcher que actúa como un wrapper para el servlet especificado.
java.lang.String	**getRealPath**(java.lang.String path) Devuelve el valor de la ruta física del Servlet. Si ponemos "" nos devolvera la ruta desde el raiz del servidor.
RequestDispatcher	**getRequestDispatcher**(java.lang.String path) Devuelve un objeto RequestDispatcher que actúa como un wrapper para el recurso ubicado en el path/ruta dado.

`java.net.URL`	**getResource**`(java.lang.String path)` Devuelve un objeto URL que representa al recurso segun el path facilitado.
`java.io.InputStream`	**getResourceAsStream**`(java.lang.String path)` Devuelve un recurso como un InputStream que representa al path especificado.
`java.util.Set`	**getResourcePaths**`(java.lang.String path)` Devuelve una lista de todos las direcciones a los recursos de la aplicacion web cuyos subdirectorios coinicde con el pasado como argumento.
`java.lang.String`	**getServerInfo**`()` Devuelve el nombre y la versión del contenedor de servlets sobre el que corre el servlet.
`java.lang.String`	**getServletContextName**`()` Devuelve el nombre de la aplicación web que corresponde a este objeto ServletContext como se especifica en el descriptor de despliegue de la aplicación web por medio del elemento display-name.
`void`	**log**`(java.lang.String msg)` Escribe un mensaje en el log del servlet.
`void`	**log**`(java.lang.String message, java.lang.Throwable throwable)` Escribe una mensaje explicativo y una pila de Excepciones en el log del servlet.
`void`	**removeAttribute**`(java.lang.String name)` Elimina el atributo con el nombre dado del servletContext.
`void`	**setAttribute**`(java.lang.String name, java.lang.Object object)` Establece un objeto con el nombre dado en el servletContext.

Ejemplo de utilización ServletConfig y ServletContext

```
public void init(ServletConfig conf) throws ServletException {
        System.out.println("Se llama al init()");
        System.out.println("Nombre del Servlet: " +conf.getServletName());
        String parametroConfig=conf.getInitParameter("aplicacion");
        System.out.println("Valor parametroConf: "+parametroConfig);
        Enumeration enu=conf.getInitParameterNames();
        String nombre="";
        while(enu.hasMoreElements()){
                nombre=(String)enu.nextElement();
                System.out.println("Nombre parametro ServletConfig: "+nombre);
                System.out.println("Valor parametro ServletConfig:
                "+conf.getInitParameter(nombre));
        }
        ServletContext servletContex=conf.getServletContext();
        String parametroContexto ="";
                parametroContexto =servletContex.getInitParameter("CursoContexto");
        System.out.println("Valor parametroContexto: "+parametroContexto);
}//Fin metodo init()
```

La Clase GenericServlet

Clase abstracta que implementa las interfaces Servlet y ServletConfig, con el objetivo de crear servlets genéricos independientes del protocolo.

Manejo de peticiones y respuestas

Petición (request) al Servlet HTTP

Cuando se implementa la interfaz *HttpServletRequest* se obtiene acceso a la siguiente información específica de http:

- Encabezado http
- Información de sesión
- Formas de leer datos de formularios enviados por el cliente
  - ✓ Métodos *getParameter* de la interfaz *ServletRequest*. Si el parámetro tiene más de un valor se utiliza *getParameterValues*. Para conocer el nombre de los parámetros tenemos *getParameterNames*.
  - ✓ Método *getReader* que retorna un *BufferedReader* para leer el cuerpo de la petición(sólo para POST).
  - ✓ Procesando una referencia a un *stream* de entrada con los datos binarios, a través del método *getInputStream* de la clase *ServletInputStream* (sólo para POST).
  - ✓ Método *getQueryString* que retorna un String con datos del cliente. Habría que analizar la cadena para identificar los parámetros (sólo para GET).

Clasificación de los métodos de HttpServletRequest:

- Encapsuladas en la interfaz javax.servlet.ServletRequest
- Encapsula información sobre:
  - ✓ Parámetros de la petición
  - ✓ Atributos
  - ✓ Internacionalización
  - ✓ El cuerpo de la petición
  - ✓ El protocolo de la petición
  - ✓ El servidor
  - ✓ El cliente
  - ✓ Redirección de la petición (Dispatchers)
- Para dar soporte a protocolos específicos habrá que crear subinterfaces
- El proveedor del contenedor implementa estas interfaces

Para HttpServletRequest estos métodos son:

- Object getAttribute(String name): devuelve el atributo asociado con el nombre que se le pasa con el argumento, o null si no hay ningún atributo asociado con dicho nombre.
- java.util.Enumeration getAttributeNames(): devuelve una enumeración que contiene los nombres de los atributos presentes en la petición.
- String getParameter(String name): devuelve el parámetro asociado con el nombre que se le pasa como argumento, o null si dicho parámetro no existe.

- java.util.Map getParameterMap(): devuelve un mapa con los parámetros de la petición.
- String[] getParameterValues(String name): devuelve un array de cadenas de caracteres conteniendo todos los valores asociados con un determinado parámetro, o null si dicho parámetro no existe.
- String getContextPath(): devuelve el fragmento de la URL que indica el contexto de la petición.
- Cookie[] getCookies(): devuelve las cookies asociadas con esta petición. s.
- String getHeader(String name): devuelve el contenido asociado con la cabecera http especificada en el parámetro.
- java.util.Enumeration<String> getHeaderNames(): devuelve una enumeración conteniendo todas las cabeceras http de la petición.
- java.util.Enumeration<String> getHeaders(String name): devuelve todos los valores asociados con una determinada cabecera http.
- int getIntHeader(String name): devuelve el valor de una determinada cabecera http como un entero.
- String getMethod(): indica el método http empleado en la petición; por ejemplo GET, POST, o PUT.
- HttpSession getSession(): devuelve la sesión asociada con esta petición. Si la petición no tiene una sesión asociada, crea una.
- HttpSession getSession(boolean create): similar al método anterior, pero sólo crea la sesión si se le pasa el parámetro true.

Respuesta (response) del Servlet HTTP

La interfaz ***HttpServletResponse*** proporciona métodos para dar formato a *streams* HTML:

- Constantes para los códigos de error http
- Métodos para añadir campos a un encabezado HTML. Por ejemplo el método *setContentType* sirve para especificar el tipo del contenido.
- Para enviar datos a un cliente web, el objeto Response proporciona un stream. Antes de obtener el *stream* de escritura, se debe llamar al método *setContentType* para especificar el tipo MIME (*Multipurpose Internet Mail Extension*) que indica el formato del dato que será pasado (texto de página HTML, gif, audio, etc).
- Un método que puede utilizarse para mandar información al cliente es *getWriter* que retorna un objeto de tipo *PrintWriter* para escribir textos al cliente.
- Otro método es *getOutputStream* que retorna un *ServletOutputStream* para enviar datos binarios al cliente.
- Cerrar el *Writer* o el *ServletOutputStream* después de enviar la respuesta permite que el servidor web sepa que la respuesta ha sido completada.

Los métodos más comúnmente empleados de HttpServletResponse son:

- java.io.PrintWriter getWriter() throws java.io.IOException: devuelve un objeto PrintWriter asociado con la respuesta que será enviada al cliente. Habitualmente, crearemos la página web que se va a mostrar al usuario escribiendo contenido en el cauce de salida representado por este objeto.
- void setContentType(String type): establece el tipo de contenido de la respuesta que se va a enviar al cliente. Por ejemplo, "text/html;charset=UTF-8" para indicar que vamos a generar una página web en empleando UTF-8 para codificar los caracteres, o "text/plain;charset=UTF-8" para indicar que lo que vamos a generar es texto plano empleando el mismo encoding que en el caso anterior.
- void addCookie(Cookie cookie): añade una cookie a la respuesta.
- void addHeader(String name, String value): añade una nueva cabecera, con el valor especificado en el segundo parámetro, a la respuesta.
- boolean containsHeader(String name): comprueba si la respuesta contiene o no una determinada cabecera.
- String encodeRedirectURL(String url): prepara una URL para ser usada por el método sendRedirect. Más adelante abordaremos en más profundidad este método.
- String getHeader(String name): devuelve el valor de la cabecera http especificada.
- java.util.Collection<String> getHeaderNames(): devuelve los nombres de las cabeceras http de la respuesta.
- java.util.Collection<String> getHeaders(String name): devuelve los valores asociados con una determinada cabecera http.
- void sendError(int sc): envía al servidor una respuesta con el código de error especificado.
- void sendRedirect(String location): redireciona al cliente a una determinada a URL.
- ServletOutputStream getOutputStream(): devuelve un objeto tipo ServletOutputStream; empleando este objeto es posible construir Servlets que generen respuestas binarias (por ejemplo, una imagen) y no sólo texto.

Ejemplo de modelo petición/respuesta

Para poder enviar información al servlet desde el cliente a través del protocolo HTTP, existen dos formas: GET y POST.

- GET:
  - Envía dentro de la URL los parámetros para el servlet (QueryString)
  - Permite que se llame al servlet directamente pasándole los parámetros a través de la URL.
- POST:
  - Encapsula los parámetros en la trama que se envía por el protocolo http.

- Para enviar datos a través de POST es necesario tener una página Web con un formulario que se encargue de enviar la información vía POST. Esta forma también se podría utilizar para GET.

A continuación mostraremos un ejemplo muy sencillo de una página html que envía información a un servlet y este devuelve otra página como respuesta. La página se llamará **Registro.html**.

Registro.html.

```
<html>
<head><title>Registro TIDW/head>
<body>
<form METHOD="POST" ACTION="/servlet/Registro">
<hr>
<h2>
<center>Registro de cliente<br>
Nombre<INPUT NAME="Nombre" VALUE=""><br>
Correo<INPUT NAME="Correo" VALUE="" ><br>
<INPUT TYPE=Submit Value = "Enviar">
</center>
</h2>
<br>
</body>
</html>
```

El Servlet que recibiría la petición de esta página estará programado en **Registro.java** y tendría el siguiente código:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Registro extends HttpServlet
{
public void doPost(HttpServletRequest req, HttpServletResponse res)
throws ServletException,IOException
{
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        //Leer los parámetros de la petición
        String nomb=(String) req.getparameter("Nombre");
```

```
        String correo=(String) req.getparameter ("Correo");
        //Mandar página de respuesta
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Registro</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("Datos del cliente " );
        out.println("<h2>Nombre: "+nomb+"</h2>");
        out.println("<h2>Correo: "+correo+"</h2>");
        out.println("</body>");
        out.println("</html>");
        out.close(); //indica al servidor que se ha terminado de mandar la información
}
}
```

Escribiendo un Servlet sencillo

A continuación realizaremos un sencillo de ejemplo de un Servlet que recibirá como parámetro un nombre y saludará al cliente que realizo la petición. Para ello construiremos una página Web con un formulario que nos servirá para enviar la petición al servlet.

```
<html>
<head> <title>Ejemplo de servlet</title> </head>
<body>
      <h1>Introduzca su nombre y pulse el boton de enviar</h1>
      <form action="miaplicacion/HolaServlet" method="post">
         Nombre:<input type="text" name="nombre" size="30">
        <input type="submit" name="enviar" value="Enviar">
      </form>
</body> </html>
```

A continuación se muestra el código del Servlet:

```
import java.io.*;
import javax.servlets.*;
import javax.servlets.http.*;
public class HolaServlet extends HttpServlet {
/** Redefinición del metodo init y configuración de los parámetros de inicio*/
public void init(ServletConfig config) throws ServletException {
/** Llamada la metodo init() de la superclase.Esto es imprescindible para la correcta inicialización del servlet y debe realizarse antes que cualquier otra acción*/
super.init(config);
System.out.println("HolaServlet arrancado a las " + new Date());
}
/** Redefinición del metodo destroy sin tareas a realizar en este caso*/
public void destroy() {
System.out.println("HolaServlet detenido a las " + new Date());
}
/** En este caso se ha optado por redefinir el metodo doPost(),pudiéndose igualmente haberse optado por redefinir
```

```
service().Lo que sería incorrecto es redefinir doGet() ya que la
peticion se realizará por el método post*/

public void doPost(HttpServletRequest request,

HttpServletResponse response) throws ServletException {

/** Se obtiene el valor del parametro enviado*/

String name = request.getParameter("nombre");

/** Se establece el contenido MIME de la respuesta*/

response.setContentType("text/html");

   /** Se obtiene un flujo de salida para la respuesta*/
   PrintWriter out;
   try {
     out = response.getWriter();
   }catch (IOException e){
     System.out.println("Error   en   el   canal   de   salida:
     "+e.toString());
   }
   /**Se escribe la respuesta en HTML estándar*/
   out.println("<html>");

   out.println("<head>");

   out.println("<title> Respuesta de HolaServlet </title>");

   out.println("<head>");

   out.println("<body>");

   out.println("<h1>¡Funcionó!:El   servlet   ha   generado   la
   pagina</h1>");

   out.println("<br>");

   out.println("<font color="red">");

   out.println("<h2>Hola " + name + "</h2>");

   out.println("</font>");

   out.println("</body>");

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

   /** Se fuerza del volcado del buffer de la salida y se cierra
   el canal*/

   out.flush();

   out.close();

   } //fin doPost()

   }//fin clase
```

Sesión y estado

Para implementar transacciones comerciales a través de la Web necesitamos:

- Sesión, es decir, el servidor debe ser capaz de diferenciar distintas solicitudes de un mismo cliente.
- Estado, es decir, el servidor debe ser capaz de almacenar/recordar información de anteriores solicitudes de un cliente. HTTP es un protocolo sin estado y no orientado a conexión, esto quiere decir que las conexiones son cerradas después de cada solicitud.

Podemos hablar de métodos de registro de la sesión del tipo de:

- QueryString, también conocido como reescritura URL.
- Formularios ocultos (Hidden)
- Cookies

Todos ellos son métodos poco elegantes y no recomendados para arquitecturas de aplicaciones empresariales.

La especificación del API Servlet requiere que los servidores de aplicaciones implementen el registro de sesión utilizando el mecanismo de cookies, debido a que el servidor configura automáticamente una cookie "jsessionid" para el control de la sesión. Si el browser cliente no admite cookies automáticamente se envía por QueryString.

Algunos Ejemplos:

QueryString: http://www.jesus.com?telefono=333333

Campo oculto: <input type="Hidden" name="uc3m" value="leganes">

Cookies:

```
Cookie myCookie=new Cookie("nombre", "jesus");

myCookie.setMaxAge(60); //espira en 60 segundos

myCookie.setDomain(".uc3.com");

resp.addCookie(myCookie);
```

Lectura de todas las cookies devueltas al servidor:

```
javax.servlet.http.Cookie[] myCookies = request.getCookies();
```

El API Java Servlet, proporciona funciones para registrar y mantener el estado, mediante la interface HttpSession.

Creación

La creación y registro de sesión significa que el contenedor es capaz de asociar una solicitud a un cliente representada mediante el objeto HttpSession. Los métodos de la interface HttpRequest son:

Método	Significado
getSession()	Devuelve el objeto de sesión de la solicitud actual. Si no existe crea uno nuevo
getSession(boolean arg)	Si este argumento es false, en caso de no existir una sesión devuelve null

HttpSession miSesion=req.getSession();

HttpSession

Encapsula la noción de sesión.

Gestión de la vida de la sesión

Debido a que la sesión se controla en el lado del servidor y http es un protocolo sin conexión, ningún servidor puede controlar si el usuario seguirá navegando en la aplicación o no. Cada sesión consume memoria en el servidor, por consiguiente, es conveniente mantener las sesiones vivas sólo en un periodo razonable.

Método	Significado
getCreationTime()	Devuelve la fecha en milisegundos en la que ha sido creada la sesión
getLastAccessedTime()	Devuelve la fecha y el momento en el que el usuario accedió por última vez
getMaxInactiveInterval()	Devuelve en tiempo en segundos en el que la sesión permanecerá activa entre dos solicitudes
getMaxInactiveInterval(int interval)	Establece en tiempo en segundos en el que la sesión permanecerá activa entre dos solicitudes Ver: <session-timeout> de web.xml
getId()	Devuelve un String exclusivo de identificación de la sesión.
Invalidate()	Invalida la sesión

Es razonable no realizar estas/algunas de estas operativas de manera programática, para ello es más eficiente la utilización del archivo web.xml, en su sección <session-config>.

Gestión del estado

Denominamos gestión de estado, a aquellos métodos que nos permiten almacenar elementos dentro de HttpSession y recuperarlos cuando lo deseemos antes de que expire la sesión.

Método	Significado
getAttribute(String name)	Devuelve el contenido del atributo name o null si no existe dicho atributo
setAttribute(String name, Object attribute)	Almacena un objeto con el nombre name.
getAttributesNames()	Devuelve una enumeración de todos los nombres de todos los atributos de la sesión.
removeAttribute(String name)	Desvincula y elimina el atributo dado de la sesión

Hemos de tener en cuenta que la mayoría de la aplicaciones empresariales serán marcadas como "distribuibles o en cluster". Incluso los que no estén en cluster, los servidores proporciona persistencia a las sesiones, en las que el servidor puede *pasivar* las sesiones menos activas, para optimizar la memoria. Por ello los atributos de la sesión deben ser serializables (implementar la interface java.lang.Serializable).

En estas circunstancias el método setAttribute, puede lanzar InvalidArgumentException.

Manejo de eventos de sesión

Interface HttpSessionListener

Notifica cuando se crea o destruye una sesión.

public interface HttpSessionListener	
Método	Significado
sessionCreated(HttpSessionEvent event)	Este método seré invocado cuando se creé un sesión dentro de nuestra aplicación Web
sessionDestroyed(HttpSessionEvent event)	Este método seré invocado cuando una sesión existente sea destruida

Cualquiera puede implementar esta interface, pero debemos indicarlo explícitamente en el descriptor de despliegue (web.xml). Por ejemplo:

```
<listener>
<listener-class>MiEscuchador</listener-class>
</listener>
```

Interface HttpSessionActivationListener

Esta interface se utiliza para la notificación de activación o *pasivación*.

public interface HttpSessionActivationListener	
Método	Significado
sessionDidActivate(HttpSessionEvent event)	Es invocado por el contenedor después de activar una sesión
sessionWillPassivate(HttpSessionEvent event)	El contenedor invoca este método antes de *pasivizar* la sesión.

Los atributos de sesión pueden implementar esta interface. No es necesaria una entrada en web.xml

La clase HttpSessionEvent

Esta clase sólo tiene un método *public HttpSession getSession()*, por medio del cual podemos acceder a la sesión.

Manejo de eventos de atributos de sesión

HttpSessionBindingListener

La interface ***HttpSessionBindingListener*** se utiliza para notificar a un objeto (el que implementa dicha interface) cuando está siendo añadido o eliminado de la sesión (mediante los métodos setAttribute o removeAttribute)

public interface HttpSessionBindingListener extends java.util.EventListener	
Método	Significado
valueBound()	El contenedor invoca este método en el objeto que está siendo vinculado a la sesión
valueUnBound()	El contenedor invoca este método en el objeto que está siendo desvinculado de la sesión, bien explícitamente o bien invalidando la sesión.

HttpSessionAttributeListener

La interface ***HttpSessionAttributeListener,*** es semejante a la anterior, pero está destinada a notificar el cambio en el estado de una sesión.

public interface HttpSessionAttributeListener extends java.util.EventListener	
Método	Significado
attributeAdded(HttpSessionBindingEvent event)	El contenedor invoca este método cuando un atributo es añadido a una sesión
attributeRemove(HttpSessionBindingEvent event)	El contenedor invoca este método cuando un atributo es eliminado de la sesión.
attributeReplace(HttpSessionBindingEvent event)	Cuando se modifica un atributo

La clase HttpSessionBindingEvent

Representa los eventos de vinculación y desvinculación de sesión.

public interface HttpSessionBindingEvent extends HttpSessionEvent	
Método	Significado
getName()	Devuelve un String con el nombre del atributo vinculado o desvinculado de la session
getValue()	Devuelve un Object con el valor del atributo vinculado, desvinculado o remplazado de la session
getSession()	Devuelve un HttpSession que representa la sesión donde el atributo está siendo vinculado, desvinculado o remplazado.

Contexto de Servlet

Hemos estudiado anteriormente el API Servlet, para el tratamiento específico de la sesión de un cliente, el "*Contexto de Servlet*" es específico de una aplicación Web, es decir, cada aplicación Web dentro de un contenedor tendrá un único contexto de Servlet asociado al contenedor. Por consiguiente nos permite mantener un estado común para todos los clientes de la aplicación.

Cuando una aplicación está en cluster (ha sido marcada como distribuible, en Web.xml:

```
<Web-app>
    <distributable/>
<Web-app>
```

) los contenedores Web replican la sesión en otros nodos del cluster.

La especificación de Servlet no requiere que la información de contexto sea replicada en un cluster, por consiguiente cada nodo mantiene su propio contexto. Así pues hemos de ser conscientes de la distribución o deployment que vayamos a realizar de nuestra aplicación a la hora de utilizar el contexto de Servlet. (Esto mismo es aplicable a las variables y clases estáticas, que son locales a la JVM).

Interface ServletContext

Encapsula el contexto de una aplicación Web.

Un objeto de este tipo puede ser obtenido de la forma:

```
ServletContext miCont=getServletContext();
```

Debido a la limitación expuesta en el apartado anterior, el contexto de Servlet no es muy utilizado, por ello en la siguiente tabla denotaremos los métodos más importantes asociados a esta interface y destinados al control del estado. El significado es semejante al de sesión, pero a nivel de aplicación.

public interface ServletContext	
Método	public Object getAttribute(String name)
	public Enumertion getAttributeNames()
	public void setAttribute(String name, Object)
	public void removeAttribute(String name)

Manejo de Eventos

Interface ServletContextListener

Un objeto de esta interface recibe los eventos de creación y destrucción de un contexto de Servlet. Decir, que el contenedor crea el contexto de Servlet como paso previo a servir cualquier contenido de la aplicación.

public interface ServletContextListener extends java.util.eventListener	
Método	Significado
contextInitialized(ServletContextEvent event)	Será invocado cuando se cree un nuevo contexto
contextDestroyedServletContextEvent event ()	Será invocado cuando se destruya un contexto ya existente.

La clase que implemente este interface ha de ser registrada en web.xml, mediante el elemento <*listener*>

Interface ServletContextAttributeListener

Notifica cuando existe un cambio en el estado de un contexto.

public interface ServletContextAttributeListener extends java.util.eventListener	
Método	Significado
attributeAdded(ServletContextAttributeEvent event)	El contenedor invoca este método en el escuchante cuando un atributo es añadido al contexto.
attributeReplace(ServletContextAttributeEvent event)	Idem cuando es modificado
attributeRemoved(ServletContextAttributeEvent event)	Idem cuando es eliminado

La interface RequestDispatcher

Esta interface encapsula la referencia a otro recurso Web, dentro del ámbito / alcance del mismo contexto de Servlet. Se utiliza para encaminar solicitudes a otros Servlet o JSP, permitiendo delegar el procesamiento de solicitud respuesta en ellos.

Esta interface tiene dos métodos:

- Forward

  Permite reenviar una solicitud a otro Servlet o JSP, o a un HTML, en el servidor y que sea este el encargado / responsable absoluto de producir la respuesta.

  Puesto que este método reenvía la solicitud, el Servlet llamante no debe realizar ninguna información de salida.

- Include

  Este método permite incluir el contenido producido por otro recurso en la respuesta del Servlet llamante. Es decir, incluye la respuesta del Servlet invocado en la respuesta global del Servlet llamado. Esto permite la composición de respuestas procedentes de múltiples recursos.

Ejemplo:

```
RequestDispatcher reqDis=req.getRequestDispatcher("nombreServlet");

reqDis.forward(req, resp);


RequestDispatcher reqDis=req.getRequestDispatcher("/compra/caja.jsp");

reqDis.include(req, resp)
```

Filtros

Un filtro es un objeto parecido a un Servlet y son controlados por el contenedor Web que puede ser insertado de forma declarativa en el proceso de solicitud-respuesta HTTP. No podemos filtrar solicitudes a EJB, ..). Son útiles porque podemos introducir código en el proceso de interceptación del contenedor Web.

Su ciclo de vida se reduce a cuatro etapas, a la sazón:

- Instanciación: el contenedor instancia cada filtro en el arranque del contenedor, para cada aplicación el contenedor mantiene una instancia por filtro.
- Inicialización: después de instanciar un filtro, el contenedor invoca el método init(), durante esta llamada el contenedor pasa al filtro parámetros de inicialización.
- Filtrado: aquellas solicitudes que requieran un filtro, el contenedor invoca al método doFilter del filtro.
- Destrucción: el contenedor llama al método destroy() antes de retirar el filtro de servicio.

Un filtro es una clase que implementa la interface javax.servlet.Filter, y que intercepta las peticiones a recursos del contenedor Web, previo al procesamiento de los mismos por parte del contenedor. Esta tiene al igual que un Servlet los métodos init() y destroy(), con la misma finalidad. Un filtro posee el método doFilter() que es invocado durante el proceso de interceptación, es decir, siempre que exista una solicitud de un recurso del servidor.

Dentro de web.xml tenemos:

```
<filter>
        <filter-name>FiltroContador</filter-name>
        <display-name>FiltroContador</display-name>
        <filter-class>jhc.FiltroContador</filter-class>
        <init-param>
                <param-name>parametro1</param-name>
                <param-value>valor1</param-value>
        </init-param>
</filter>
<filter-mapping>
        <filter-name>FiltroContador</filter-name>
```

```
        <url-pattern>*.html</url-pattern>
    </filter-mapping>
```

la primera directiva es la que especifica el filtro dentro del descriptor, y la segunda indica al contenedor que tipo de solicitudes deben ser filtradas con el filtro en cuestión.

Destacar que el orden en el que aparecen los filtros en el descriptor de despliegue (web.xml) representa el orden de la cadena de filtro.

API Filtro

Consiste en las siguientes interfaces y clases:

javax.servlet.Filter

Método	Significado
Init(FilterConfig config)	El contenedor invoca este método antes de poner el filtro en servicio. El contenedor envía la información de configuración mediante el objeto config.
doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)	Este método es invocado por el contenedor mientras procesa un filtro. Mediante FilterChain, el filtro instruye al contenedor para que procese el filtro siguiente.
Destroy()	Lo invoca el contenedor antes de sacar a un filtro de servicio.

javax.servlet.FilterConfig

Similar a ServletConfig, permite o proporciona acceso al entorno de filtros.

Método	Significado
getFilterName()	Devuelve el nombre del filtro, especificado en web.xml
getInitParameter(String name)	Devuelve el valor del parámetro, null si no existe
getInitParameterNames()	Devuelve un Enumeration con los nombres de los parámetros. Null si no tiene parámetros de iniciacización.
getServletContext()	Devuelve una referencia al contexto de Servlet asociado a la aplicación.

javax.servlet.FilerChain

permite al filtro invocar el resto de la cadena de filtro (siguiente de la cadena), a través del método doFilter().

Cuando un filtro invoca este método el contenedor invoca al siguiente filtro, si existe, en caso contrario el contenedor invoca al recurso Web de la solicitud.

Tiene un único método:

public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)

throws
ServletException,
IOException:

Ejemplo:

Presentamos el ejemplo de un filtro que cuenta los accesos a una aplicación.

```
package tidw;

import java.io.IOException;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletContext;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

public class FiltroContador implements Filter {

        // Para tener una referencia al contexto de filtro

        FilterConfig config;

 public void init(FilterConfig config) {

       this.config = config;

 }
```

```
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)  throws IOException, ServletException {
        // Tomamos el Contexto de sesión donde almacenaremos el contador
        ServletContext miContexto = config.getServletContext();
        // Vemos si el atributo contador existe dentro del contexto
        // En caso contrario lo creamos e inicializamos a cero
        Integer intContador = (Integer)miContexto.getAttribute("contador");
        if(intContador == null) {
              intContador = new Integer(0);
        }
        intContador = new Integer(intContador.intValue() + 1);
        miContexto.setAttribute("contador", intContador);
        // Instruimos al contenedor para que invoque al siguiente filtro
        chain.doFilter(request, response);
  }
  public void destroy() {}
}
```

JSR 315: Servlet 3.0

Podemos encontrar la especificación en:

JSR 315: Java™ Servlet 3.0 Specification

Cambios más significativos de la especificación 2.5 a la 3.0:

Anotaciones

Muy importante, ya no será necesario editar o hacer el uso tan intensivo de los ficheros descriptores, web.xml. La nueva especificación (JSR 315: Java™ Servlet 3.0 Specification) introduce una serie de anotaciones que se pueden utilizar para describir elementos comunes como Servlets, Filtros, mapeos de URLs, parámetros de inicialización y el context listener de cada Servlet.

Las nuevas anotaciones son:

- @WebServlet – Define un Servlet. Se debe extender de HttpServlet

```
@WebServlet(name="MyServlet", urlPatterns={"/foo", "/bar"})
public class SampleUsingAnnotationAttributes extends HttpServlet{
    public void doGet(HttpServletRequest req,HttpServletResponse
                                                        res) {
}
```

  La línea que indica qué peticiones quiere responder nuestro Servlet es la de la anotación urlPatterns:

```
@WebServlet(name="MyPrimerServlet", urlPatterns={"/MyServlet"})
```

  para indicar los patrones de URL a los cuales desea responder el Servlet es posible emplear * de la siguiente manera:

```
@WebServlet(name=" MyPrimerServlet", urlPatterns={"/ MyServlet/*"})
```

  Hasta Java EE 6 (que incorporó la versión 3.0 de la especificación de los Servlets) todos los Servlets debían extender la clase HttpServlet. En Java EE 6 sigue estando disponible esta opción, pero también es posible convertir en un Servlet a cualquier POJO (Plain Old Java Object) simplemente anotándolo.

- @WebFilter - Define un Filter. Debe implementar `javax.servlet.Filter`.

```
@WebFilter("/foo")
```

```
public class MyFilter implements Filter {

        public void doFilter(HttpServletRequest req, HttpServletResponse
                                                                    res) {

            ...

        }

}
```

- @WebListener – Define un Listener. Se aplica sobre:
  - javax.servlet.ServletContextListener
  - javax.servlet.ServletContextAttributeListener
  - javax.servlet.ServletRequestListener
  - javax.servlet.ServletRequestAttributeListener
  - javax.servlet.http.HttpSessionListener
  - javax.servlet.http.HttpSessionAttributeListener

```
@WebListener

public class MyListener implements ServletContextListener{

        public void contextInitialized(ServletContextEvent sce) {

            ServletContext sc = sce.getServletContext();

            sc.addServlet("myServlet", "Sample servlet",

                                    "foo.bar.MyServlet", null, -1);

            sc.addServletMapping("myServlet", new String[] {

                                                "/urlpattern/*" });

        }

}
```

- @WebInitParam – Define un parámetro de inicialización. Se utiliza para pasar parámetros a un Servlet o un Filtro.

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

  Mismo efecto que en el web.xml:

```
<servlet>
<servlet-name>ConfigurableServlet</servlet-name>
```

```
<servlet-class>tidw.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>
```

Es de destacar, que podemos usar el web.xml para sobrescribir los valores especificados en las anotaciones. La especificación deja abierta la posibilidad de mezclar las dos aproximaciones, es decir el utilizar anotaciones y el descriptor web al mismo tiempo.

Procesamiento asíncrono

Un cambio importante en Servlets 3.0 es la introducción de métodos para realizar procesado asíncrono. Hasta la 1.5 el procesado de un request es completamente síncrono, si la request necesita bloquearse por algún recurso, el thread que está manejando esa request se bloqueará. Asimismo, si queremos mantener la request viva durante un largo espacio de tiempo haciendo streaming al cliente, la request permanecerá bloqueada. Esto hace que los contenedores web no puedan escalar apropiadamente ya que el uso de recursos no es óptimo.

La solución proporcionada en Servlets 3.0 es añadir tres nuevos métodos, *suspend*, *resume* y *complete*, que permitirán suspender una request, reanudarla posteriormente por ejemplo desde otro thread diferente, y completarla.

Es importante que la especificación indica de que tanto el objeto request como el objeto response no son *thread-safe*.

Es configurado bien en:

- El fichero descriptor web.xml: <async-supported>true</async-supported>
- Mediante anotaciones: @WebServlet(asyncSupported=true)
- A nivel de programación: registration.setAsyncSupported(boolean)

Así podemos construir aplicaciones server Push usando directamente el API de Servlets.

Analícese el siguiente código:

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import tidw.asincservlet.MiClaseAsincrona;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import javax.servlet.AsyncContext;
```

```
@WebServlet(name = "/MiPrimerServletAsincrono", urlPatterns = {
"/asincrono" }, asyncSupported = true)
public class MiPrimerServletAsincrono extends HttpServlet {
	private static final long serialVersionUID = 1L;

	public MiPrimerServletAsincrono() {
		super();
		// TODO Auto-generated constructor stub
	}

	protected void doGet(HttpServletRequest request,
			HttpServletResponse response) throws
			ServletException, IOException {
		// TODO Auto-generated method stub
		procesandoElRequest(request, response);
	}

	protected void doPost(HttpServletRequest request,
			HttpServletResponse response) throws
			ServletException, IOException {
		// TODO Auto-generated method stub
		procesandoElRequest(request, response);
	}

	protected void procesandoElRequest(HttpServletRequest request,
			HttpServletResponse response) throws
			ServletException, IOException {

		AsyncContext aCtx = request.startAsync(request, response);
		ScheduledThreadPoolExecutor executor = new
					ScheduledThreadPoolExecutor(10);
		executor.execute(new MiClaseAsincrona(aCtx));

	}
}

package tidw.asincservlet;

import javax.servlet.AsyncContext;

public class MiClaseAsincrona implements Runnable {
	AsyncContext ctx;

	@Override
	public void run() {
		try {
			// Ejecutaremos este hilo cada 6 segundos
			Thread.sleep(6000);
			System.out.println("Se lanza cada 6 segundos");
		} catch (Exception ex) {
			ex.printStackTrace();
		}
		ctx.complete();
	}

	public MiClaseAsincrona(AsyncContext ctx) {
		this.ctx = ctx;
	}
}
```

Fragmentos web

Otra introducción importante son los fragmentos web.

Hasta ahora en las aplicaciones web relativamente grandes teníamos un web.xml con demasiada información, lo que dificultaba su comprensión.

Ahora los fragmentos web permiten crear ficheros web.xml individuales que se distribuyen dentro de un fichero JAR que contendría lo que podríamos llamar una mini-aplicación web.

Es importante destacar que el fichero web.xml debe colocarse dentro del META-INF y el JAR dentro del /lib.

El contenedor de Servlets al arrancar la aplicación se encargará de buscar todos los "fragmentos web" buscando en lib y automáticamente irá componiendo el web.xml de la aplicación, añadiendo entradas por cada uno de los diferentes web.xml individuales. A modo de ejemplo:

```
<web-fragment>
  <servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>tidw.MyServlet</servlet-class>
  </servlet>
  <listener>
    <listener-class>MyListener</listener-class>
  </listener>
</web-fragment>
```

Esta mejora nos ayudará en el desarrollo de aplicaciones en trabajo en grupo y favorece la modularidad de las aplicaciones web.

Otros.

El resto de cambios, ya no tan grandes, son el añadido de nuevos métodos a la interfaz ServletContext de modo que se puedan añadir servlets y filtros programáticamente en tiempo de inicialización; el soporte de cookies HttpOnly que no están expuestas a scripting en lado del cliente, y el añadido de métodos en la request para obtener el ServletContext y el objeto response.

Ejemplo 1

```
@WebServlet(name="CabecerasServlet", urlPatterns={"/cabeceras"})
public class CabecerasServlet extends HttpServlet {
      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>Cabeceras Servlet</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();
        }
    }
@Override
protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
      processRequest(request, response);
}
@Override
protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
      processRequest(request, response);
}
}
```

Ejemplo 2

Supongamos una página HTML con este form incluido:

```
<form method="get" action="/solicitud" name="datos">
      Nombre: <input name="nombre"><br>
      Apellidos: <input name="apellidos"><br>
      Edad:
            <select name="edad">
                  <option>Menor de 18</option>
                  <option>De 18 a 30</option>
                  <option>De 30 a 55</option>
                  <option>Mayor de 55</option>
            </select>
<br>
Hobbies:<br>
      <input name="hobbies" value="lectura"
                                          type="checkbox">lectura<br>
      <input name="hobbies" value="futbol"
                                          type="checkbox">verfutbol<br>
      <input name="hobbies" value="deporte" type="checkbox">jugar
                                                tenis<br>
<br>
<button>Enviar</button></form>
```

Podríamos leer los datos emitidos y responder al respecto con el siguiente servlet:

```
@WebServlet(name=" FormularioServlet ", urlPatterns={"/solicitud"})
public class FormularioServlet extends HttpServlet {
```

```
        protected void processRequest(HttpServletRequest request,
                                      HttpServletResponse response)
                                      throws ServletException,
                                      IOException {
        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>");
              out.println("<title>Servlet que procesa una
                                                  solicitud</title>");
              out.println("</head>");
              out.println("<body>");
              out.println("<h1>" + "Hola " + nombre + " " + apellidos+
                                                        "</h1>");

              out.println("Tu Franja de edad es " + edad + " y tus
              hobbies son:");

              out.println("<ul>");
              for (String hobby : hobbies) {
                              out.println("<li>" + hobby + "</li>");
              }
              out.println("</ul>");
              out.println("Esta solicitud ha sido invocado con Los
                                      siguientes parametros:<br/>");
              out.println(request.getQueryString());
              out.println("</body>");
              out.println("</html>");
        } finally {
              out.close();
        }
}
```