Servlets Filtros

¿Qué es un Servlet Filtro?

- Un **filtro** es un objeto que puede **transformar** el ***header* http**, el *contenido de una* ***petición http*** o de una ***respuesta http***.
- Los **filtros** se diferencian del resto de las componentes web JAVA en que usualmente **NO** crean respuestas ni **responden a peticiones http** como lo hacen los servlets http. Los filtros **modifican** o **adaptan** las **peticiones** a un recurso o las **respuestas** de un recurso.
- Los **filtros** proveen una funcionalidad que puede **aplicarse** a cualquier tipo de **recursos web** (estático o dinámico).
- Un **filtro NO tiene dependencias** con el **recurso web** sobre el que se aplica, por lo tanto puede **componerse** con diferentes recursos web.

¿Qué funcionalidades ofrecen los Servlets Filtro?

Las principales tareas que hacen los filtros, son:

- **Consultar la petición http** y **actuar** en consecuencia.
- **Procesar la petición http** a un recurso **antes que se lo invoque**.
- **Interceptar la invocación** a un recurso **después de su invocación**.
- **Bloquear** el par pedido-respuesta.
- **Modificar el header y los datos de la petición http** *wrappeando* la petición original en una versión customizada.
- **Modificar el header y los datos de la respuesta http** proveyendo una versión customizada del objeto respuesta.
- **Ignorar la petición original** y **delegar** la petición a otro recurso web.
- **Aplicar acciones sobre un servlet** o **grupos de servlets** o sobre páginas estáticas, usando uno o más filtros en un orden especificado (encadenamiento de filtros).

¿Qué podemos resolver con Servlets Filtro?

- **Compresión de datos:** enviar menos contenido al cliente web para hacer más rápida la descarga. Esto es deseable especialmente en clientes que tienen una mala conexión a Internet.
- **Autenticación:** bloquear el acceso a usuarios no autenticados.
- **Logging y auditoría:** guardar accesos para estadísticas.
- **Conversión de imágenes:** de formato por ejemplo de GIF a PNG dinámicamente.
- **Encriptación:** encriptar información sensible cuando viaja al cliente.
- **Caching:** minimizar el tiempo necesario para producir una página dinámica.

Servlets Filtro
Generalidades

- Un **servlet filtro** es una **componente web JAVA** gerenciada por un **Contenedor Web**.
- En forma idéntica a otras componentes web JAVA, los **servlets filtro** son **clases JAVA** independientes de la plataforma de ejecución, se compilan a código de bytes (*bytecodes*), se cargan dinámicamente y se ejecutan en un servidor web.
- Los **servlets filtro** son componentes web que permiten transformar el **contenido** y el ***header***, de la ***petición*** y de la ***respuesta*** HTTP.
- Para desarrollar **servlets filtro** se requiere tener más conocimientos del **protocolo http** que para desarrollar servlets http, debido a que típicamente manipulan y modifican campos del ***header***, de la **petición** o de la **respuesta** http.
- Típicamente los **servlets filtros NO crean una respuesta ni responden a una petición** como lo hacen los servlets http, sino que **modifican** o **adaptan** la **petición** a un recurso web o la **respuesta** de un recurso web.
- Los **servlets filtros** están disponibles a partir de la versión **2.3 de la API de Servlets**.

Petición

Verbo Recurso Versión

```
GET /index.html HTTP/1.1
Host: wikipedia.org
Accept: text/html
```

Respuesta

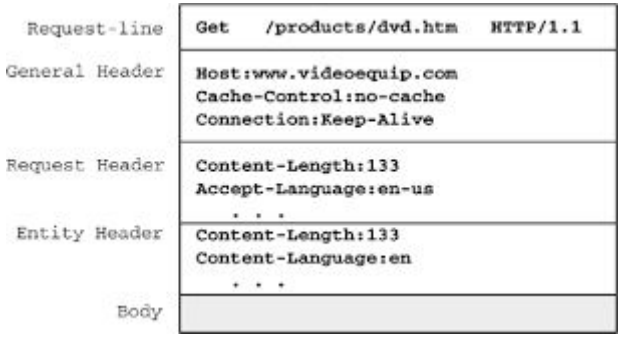
Versión Código respuesta

```
HTTP/1.1 200 OK
Server: wikipedia.org
Content-Type: text/html
Content-Lenght: 2026

<html>
  …
</html>
```

Primera línea

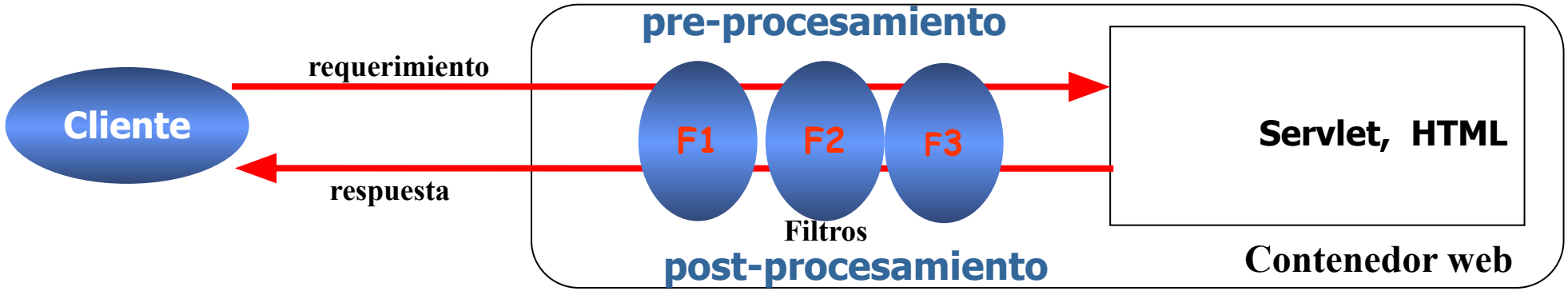
Encabezados

Cuerpo



Servlets Filtro
¿Qué son y para qué sirven?

- Son **componentes web JAVA** que se sitúan entre una **petición** y el **destino** de la petición. El **destino** de la petición puede ser un **recurso web dinámico** (servlets) o **estático** (HTML).
- Los **filtros interceptan** una **petición** enviada a un **recurso web** de la aplicación.
- Forman parte de una **cadena** y el **último enlace** de la cadena apunta al **recurso solicitado**.
- Un **servlet filtro** puede elegir **delegar la petición al siguiente filtro** de la cadena o al **recurso web** solicitado si fuese el último filtro. A su vez, los **filtros podrían modificar** la **respuesta** antes de devolverla al cliente.

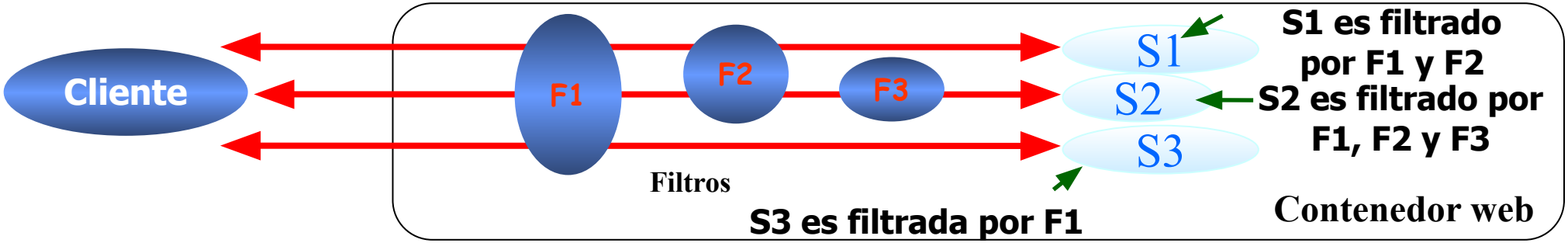


- Los **filtros** permiten **incorporar capas de pre-procesamiento** a una petición y **post-procesamiento** a una respuesta.

Servlets Filtro - Encadenamiento

Los **servlets filtro** se **declaran** usando la anotación **@WebFilter** o en el archivo **web.xml**.

- Se puede declarar que **un recurso web** sea filtrado por 1 o más filtros en un orden determinado. La **cadena de servlet filtros** se configura en el archivo **web.xml**.
- Se puede configurar el mismo **servlet filtro** para aplicar a diferentes recursos web.



- Los **filtros** ofrecen un mecanismo que permite **aplicar múltiples capas de funcionalidad** al par **ServletRequest-ServletResponse**. Con filtros es simple dividir la funcionalidad en múltiples capas lógicas y aplicarlas de la forma que se desee.
- La versión 2.4 de la API de Servlets extendió la utilidad de filtros permitiendo definir filtros que no solamente intercepten peticiones provenientes de un cliente http, sino que también se ejecuten cuando son invocados los métodos **forward()** e **include()** de **RequestDispatcher** o en caso de **error**.

Servlets Filtros - Ciclo de Vida

- El **ciclo de vida** de un **servlet filtro** es conceptualmente idéntico al de servlets HTTP. Tiene tres fases: **inicialización**, **servicio** y **destrucción**.
  - La **inicialización** ocurre solamente **una vez**, cuando el filtro es cargado en memoria.
  - La fase de **servicio** es invocada **cada vez que el filtro se aplica** a una petición y a una respuesta.
  - La **destrucción** ocurre cuando el filtro es descargado de memoria, típicamente cuando **se da de baja la aplicación**.
- Los métodos **init()** y **destroy()** corresponden a la fase de **inicialización** y **destrucción** respectivamente. El método **doFilter()** corresponde a la fase de **servicio**.
- El método **doFilter()** es invocado por el **Contenedor Web** cuando se aplica un filtro a un par **ServletRequest-ServletResponse**. Ambos objetos son pasados como parámetros al método **doFilter()**.
- El **Contenedor Web** gerencia el **ciclo de vida** de cada filtro invocando a los 3 métodos definidos en la interface **jakarta.servlet.Filter**: **init()**, **doFilter()** y **destroy()**.
- El **Contenedor Web** es responsable de la **carga e instanciación** de los **filtros** cuando éstos son **aplicados por primera vez**.
- El **Contenedor Web** crea una **única instancia** de cada **filtro** declarado.
- El **Contenedor Web** invoca al método **doFilter()** en múltiples **threads concurrentes**. El acceso a variables no locales y recursos debe hacerse de manera *thread-safe*.

Programación de Servlets Filtro

La **API de filtros** está definida por las interfaces **Filter**, **FilterChain** y **FilterConfig** del paquete **jakarta.servlet**. Para definir un filtro hay que implementar la interface **jakarta.servlet.Filter.**

```
package jakarta.servlet;

import java.io.IOException;

public interface Filter {

public void init(FilterConfig filterConfig) throws ServletException;

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
                                    throws IOException, ServletException

public void destroy();

}
```

init() es invocado por el Contenedor Web cuando el filtro es cargado en memoria. Se usa para tomar parámetros de inicialización y para ejecutar código de inicialización. Es invocado una única vez.

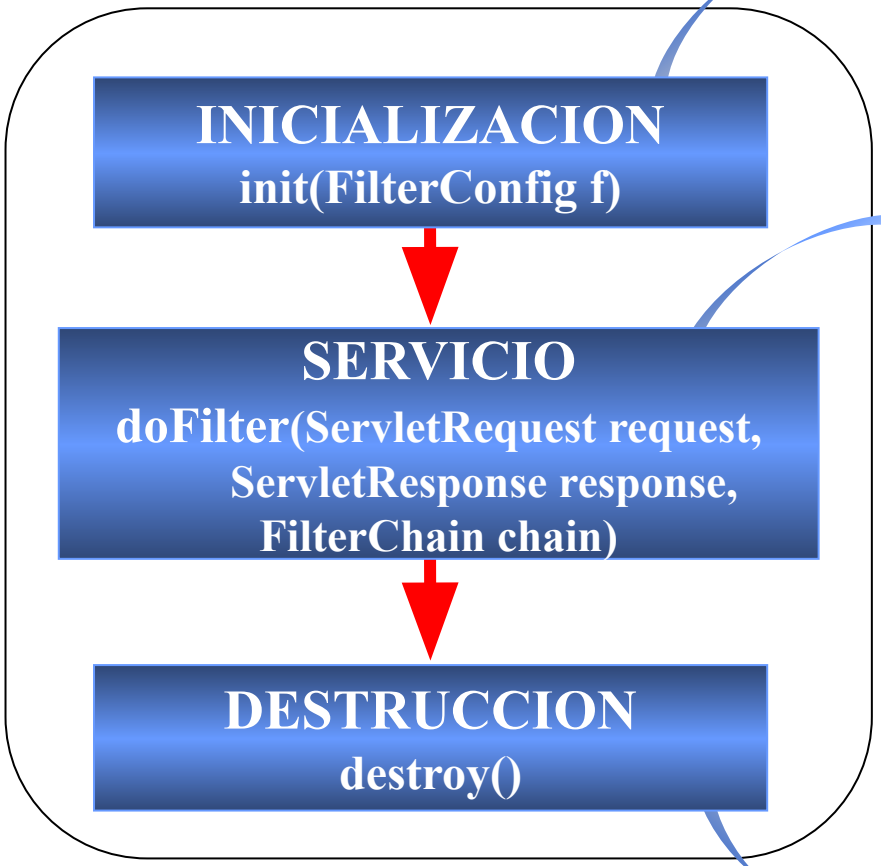
doFilter() recibe el requerimiento, la respuesta y la cadena de filtros. El objeto ServletRequest y ServletResponse son instancias de **HttpServletRequest** y **HttpServletResponse** respectivamente. Este método hace todo el trabajo.

destroy() es invocado por el Contenedor Web cuando el filtro es descargado de memoria. Permite liberar los recursos alocados por el filtro.

El Contenedor Web crea una única instancia por filtro declarado; a su vez el contenedor crea un objeto **FilterConfig** que le pasará como parámetro al **init()**. El objeto **FilterConfig** permite acceder a los **parámetros de inicialización** y al **ServletContext** de la aplicación.

Servlet Filtros
Ciclo de Vida

Un **Servlet Filtro** debe implementar la interface **jakarta.servlet.Filter**. Esta interface define los **3 métodos del ciclo de vida**:



- El Contenedor Web crea un objeto **FilterConfig** y lo pasa al método **init().** Esta referencia posibilita la lectura de los parámetros de inicialización del archivo **web.xml** usando el método **f.getInitParameter("clave").**
- Si el método **init**() **no termina exitosamente** dispara una excepción y el **filtro no podrá ser aplicado**.

- Este es el método que hace el trabajo del filtro: el Contenedor Web crea 3 objetos que pasa como parámetros al método **doFilter()**:
  - **ServletRequest** y **ServletResponse:** estos objetos representan la **petición** y la **respuesta http**.
  - **FilterChain:** es el objeto que representa la cadena de filtros que está siendo aplicada al par petición-respuesta.

- Lo ejecuta el Contenedor Web cuando el filtro es descargado de memoria, generalmente antes de bajar la aplicación. Se usa para liberar recursos inicializados por el filtro en el método.

Declaración de Servlets Filtros

En el **web.xml** los filtros se declaran con elementos que ofrecen funcionalidad idéntica a los que describen servlets.

Los filtros se definen en el web.xml antes que todos los servlets y después del elemento **<context-param>**.

```
<filter>
    <filter-name>LoguinFiltro</filter-name>
    <filter-class>curso.filtros.LoguinFiltro</filter-class>
    <init-param>
        <param-name>saludo</param-name>
        <param-value>Hola!!!</param-value>
    </init-param>
</filter>
```

El elemento *filter-mapping* define a **qué recursos** de la aplicación web se le **aplicarán los filtros.**
El elemento *filter-mapping* asocia un filtro con un servlet usando el sub-elemento **<servlet-name>** o con un conjunto de recursos usando **<url-pattern>**.

```
<filter-mapping>
   <filter-name>LoguinFiltro</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

El filtro **LoguinFiltro se aplica** a **todos los recursos** de la aplicación web.

```
<filter-mapping>
  <filter-name>LoguinFiltro</filter-name>
  <servlet-name>Facturar</servlet-name>
</filter-mapping>
```

El **filtro LoguinFiltro se aplica** sólo al servlet **Facturar**.

Declaración de Servlets Filtros

@WebFilter es la anotación que se usa para **declarar filtros en una aplicación web**. Esta **anotación** se especifica en la **clase que implementa el filtro** y contiene **metadatos** sobre el filtro que se está declarando.

El filtro anotado debe especificar al menos un **URL pattern** usando los atributos **urlPatterns** o **value**.

```
package curso.filtros.
import jakarta.servlet.Filter;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;
@WebFilter(filterName = “LoguinFiltro",
urlPatterns = {"/*"},
initParams = {
@WebInitParam(name = “saludo", value = “Hola")})
public class LoguinFiltro implements Filter {….}
```

Si el orden en que deben ser invocados los listeners, servlets y filtros es importante para la aplicación, entonces debe usarse el archivo descriptor web.xml
.
Sección 8.2.3 de la especificación de servlets 3.0

El objeto FilterChain

El objeto jakarta.servlet.FilterChain:

- Permite la separación en múltiples **capas de funcionalidad**.
- Para cada petición entrante nueva, el Contenedor Web arma una **cadena de filtros** a aplicar, de acuerdo al orden de declaración en el **web.xml**. El objeto ***FilterChain*** representa la **cadena de filtros** que deben aplicarse al **par pedido/respuesta**.

¿Cómo arma el Contenedor Web la cadena de filtros?

- Cuando recibe una petición **identifica** el recurso web "destino" y luego:
  - **Arma la cadena <servlet-name>:** si hay filtros aplicados al elemento <servlet-name> y el servlet-name es el recurso "destino" -> el contenedor construye la cadena de filtros de dicho servlet-name en el orden declarado en el **web.xml**. El último filtro de la cadena corresponde al último elemento <servlet-name> asociado con un filtro y es este último filtro el que invoca al "destino".
  - **Arma la cadena <url-pattern>:** si hay filtros aplicados al elemento <url-pattern> y el url-pattern coincide con el "destino" -> el contenedor construye la cadena de filtros de dicho url-pattern en el orden declarado en el **web.xml**. El **último filtro de la cadena** corresponde al último elemento **<url-pattern>** asociado con un filtro y es este filtro el que **invoca al primer filtro en la cadena <servlet-name>** o al **"destino"** si no existe la cadena <servlet-name> para dicha petición.
- Para mejorar la **performance** las implementaciones de contenedores cachean en memoria la cadena de filtros, para evitar computarla cada vez que reciben una petición.

El método doFilter()

Un filtro que está ejecutando su método **doFilter()** tiene 2 posibilidades: manejar completamente el par request/response o solamente manipularlo y pasarlo al próximo filtro en la cadena.

- Un **filtro** puede **interrumpir la ejecución de los siguientes filtros** en la cadena simplemente **retornando** del método doFilter(). Esto causa que el flujo de control de la aplicación retorne al filtro previamente ejecutado (si es que había) y finalice enviando la respuesta al cliente.

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws ..
  HttpServletRequest req1=(HttpServletRequest) req;
  HttpServletResponse res1=(HttpServletResponse) res;
  PrintWriter out = req1.getWriter();
  out.println("<HTML>");
        chain.doFilter(req, res);
        Continúa la ejecución de la cadena
  out.println("Hola");
  out.println("</HTML>");
  out.close();
        return;
        Interrumpe la ejecución de la cadena
}
```

Si se quiere modificar la respuesta del servlet, se debe mandar un stream auxiliar para que el Servlet escriba.

- Un **filtro** puede **continuar la ejecución de la cadena de filtros** invocando al método **doFilter()** del objeto **FilterChain**. El siguiente filtro de la cadena puede ser otro filtro o el destino final de la cadena. Al invocar al método doFilter() de FilterChain la ejecución se detiene y espera que el siguiente recurso de la cadena manipule el request-response. Una vez que el método doFilter() de FilterChain finaliza, el filtro completa la ejecución (post-procesamiento).

Más sobre configuración de Servlets Filtros

El elemento **<filter> del web.xml** puede contener múltiples **parámetros de inicialización.**

```
<filter>
    <filter-name>filtroContador</filter-name>
    <filter-class>misFiltros.FiltroContador</filter-class>
    <init-param>
        <param-name>saludo</param-name>
        <param-value>Buen Día</param-value>
    </init-param>
</filter>
```

Los **parámetros de inicialización** pueden ser leídos a través del objeto **FilterConfig** que es pasado como parámetro en el método **init() del servlet filtro.** Este objeto es semejante al objeto **ServletConfig**.

A diferencia de Servlets http, en filtros no existe un método getFilterConfig(). Se dispone del **objeto FilterConfig** únicamente en el init(). Si se desea acceder en otro método al FilterConfig, se lo debería guardar por ej. en una variable de instancia del filtro.

```
public void init(FilterConfig config)throws ServletException{
  config.getInitParameter("saludo");
 }
 public void doFilter( , ,) throws...{
  this.getFilterConfig().getInitParameter("saludo");
 }
```

Servlets Filtro y RequestDispatcher

- Por defecto, una cadena de filtros está definida para manejar solamente peticiones provenientes del cliente web. Si un requerimiento es delegado usando los métodos **forward()** o **include()** del objeto **RequestDispatcher**, los filtros no son aplicados.
- A partir de la **versión 2.4** de la especificación de Servlets, la configuración por defecto puede modificarse a través del **web.xml** usando el elemento **<dispacher>**

```
<filter>
    <filter-name>FiltroLoguin</filter-name>
    <filter-class>misFiltros.FiltroLoguin</filter-class>
</filter>
<filter-mapping>
    <filter-name>FiltroLoguin</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

Esta configuración indica que el filtro de nombre **FiltroLoguin** será aplicado **solamente** para **peticiones provenientes de clientes web** (REQUEST) y cuando **se invoca al método forward()** del RequestDispatcher (FORWARD).

El elemento **<dispatcher>** permite indicar para un <filter-mapping> si el filtro lo queremos aplicar:

- Cuando el requerimiento proviene de un cliente web, asignándole el valor **REQUEST.**
- Cuando el requerimiento proviene de un forward(), asignándole el valor **FORWARD**.
- Cuando el requerimiento proviene de un include(), asignándole el valor **INCLUDE**.
- Cuando el requerimiento proviene de una página de error, asignándole el valor **ERROR**.
- Si no se especifica el tipo de *dispatcher* el valor por defecto será **REQUEST**.

Ejemplo: Un filtro que computa el tiempo de respuesta de un recurso web

Este ejemplo está basado en uno de los filtros que viene en Tomcat

```
package misFiltros;
import jakarta.servlet.*;
import java.io.IOException;

public class FilterTime implements Filter{
 private FilterConfig config;

 public void init(FilterConfig config)  throws ServletException {
      this.config = config;
 }

 public void doFilter(ServletRequest req, ServletResponse resp,FilterChain chain)
                                                    throws ServletException,IOException {
   // Registramos el tiempo al inicio de  la petición
   long antes = System.currentTimeMillis();
   chain.doFilter(req,resp);
   // Recuperamos el tiempo cuando finaliza la petición
   long despues = System.currentTimeMillis();
   String nomServlet= ( (HttpServletRequest) req).getRequestURI();
   config.getServletContext().log(nomServlet+": " + (despues - antes) + "ms");
 }

 public void destroy() {
   config=null;
 }
} // Fin de la clase del Filtro
```

Con esta configuración el **FilterTime** será aplicado a todos las peticiones:

```
<filter>
  <filter-name>FilterTime</filter-name>
  <filter-class>misFiltros.FilterTime</filter-class>
</filter>
<filter-mapping>
  <filter-name>FilterTime</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

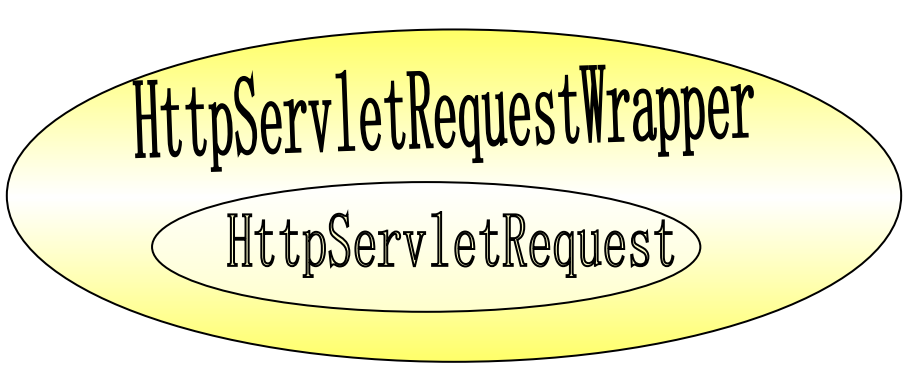
Wrappers

- Una de las características útiles de los filtros es la habilidad para *wrappear* una petición y/o una respuesta. *Wrappear* significa encapsular el objeto petición o respuesta dentro de otro *customizado* o adaptado.
- Es necesario para **manipular** el objeto petición y/o respuesta.
- Para hacer *wrapping* **la especificación 2.3** incorporó clases especiales que podrían usarse igualmente sin filtros con los métodos include() y forward() de RequestDispatcher.
- Un *wrapper* es una implementación del objeto que encapsula.

HttpServletRequestWrappper implements **HttpServletRequest**

HttpServletResponseWrappper implements **HttpServletResponse**

A partir de la API de servlet 2.3 se cuenta con estas dos clases wrappers

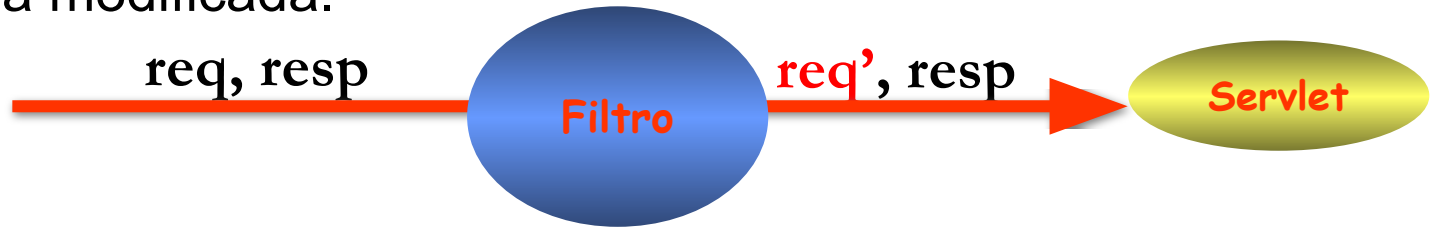


Crear un *wrapper* customizado consiste en extender alguna de estas clases wrapper y sobreescribir los métodos necesarios.

Programar Peticiones o Respuestas customizadas

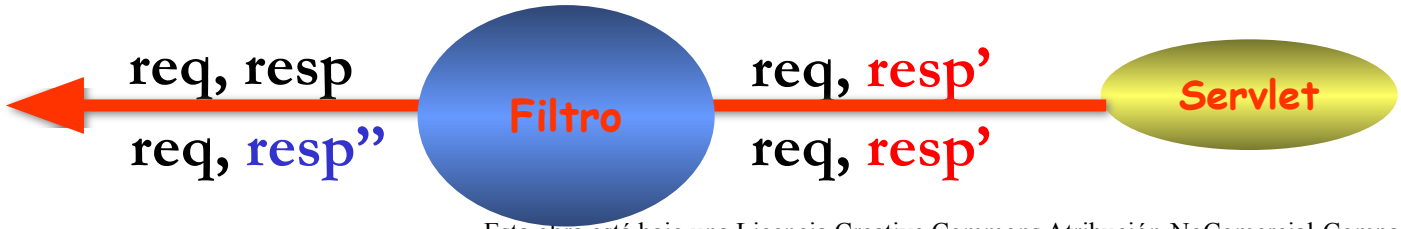
Hay muchas maneras de *customizar* una ***petición*** o una ***respuesta***. Por ejemplo un filtro podría agregar un atributo a la petición o insertar un dato en la respuesta.

a) Un filtro que **modifica una petición** debe interceptar la petición antes de que alcance el recurso web, modificarla y enviarlo modificada (req') en la invocación al método doFilter() para que el próximo filtro en la cadena o el recurso "destino" la reciba modificada.



req' es subclase de **ServletRequestWrapper** o **HttpServletRequestWrapper**

b) Un filtro que **modifica una respuesta** debe interceptar la respuesta antes de que sea enviada al cliente. Para hacer esto se debe pasar un **stream de respuesta sustituto** (resp') al servlet que genera la respuesta. Este stream, sobre el que se escribirá la respuesta, evita que el servlet cierre la respuesta original cuando termina de ejecutarse y le permite al Filtro modificar la respuesta que escribió el servlet.



resp' es subclase de **ServletResponseWrapper** o **HttpServletResponseWrapper**

Programar Peticiones Customizados

- Para crear un objeto *petición customizado* debemos extender la clase **HttpServletRequestWrapper** y sobreescribir los métodos que se deseen de la petición original (**getParameter()**, **getHeader()**, etc.).
- Algunos ejemplos:
  - Reemplazar/sobreescribir el código de los métodos de **HttpServletRequest** por ejemplo para proveer información de **Auditoria**: guardar en un archivo de logs la información de los métodos invocados de **HttpServletRequest**.
  - Manipular la petición para detectar ataques con **SQL Injection o Inyección de código en general**, por ej. un filtro podría verificar si los valores de los parámetros de la petición contienen palabras claves de SQL u operadores que cambien la lógica o sintaxis de la sentencia SQL. Si esto ocurre podría blanquear el valor del parámetro de la petición y guardar los datos relacionados al ataque en el archivo de logs de la aplicación web.
  - Crear e inicializar objetos con información que posteriormente será desplegada por otros servlets. En este caso, los filtros podrían crear e inicializar **JavaBeans** y ligarlos al alcance *request*.

Ejemplo: Filtrar inyección de código SQL

FiltroLoguin detecta ataques a la base de datos con “Inyección de código SQL”.

Dada la consulta a una tabla de usuarios:

“SELECT * FROM Usuario WHERE USERNAME=” +**user+** “AND PASSWORD=”+**pass**

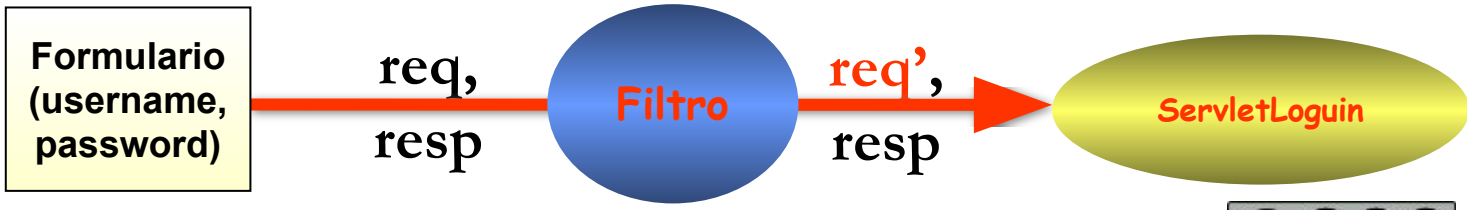
¿Qué pasa si un usuario ingresa en user lo siguiente: '' OR 1=1 -- ?

(-- es comentario)

Se ejecutaría una sentencia SQL como ésta, la cual podría devolver todos los usuarios de la base

SELECT * FROM Usuario WHERE USERNAME='' OR 1=1 --user AND PASSWORD=pass

```
class FiltroLoguin implements Filter {
private FilterConfig config;
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
                                                    throws ServletException,IOException {
  String usr = req.getParameter("username");
  String passw = req.getParameter("password");
  if (!usr.equals("") || !passw.equals("")){

  SQLInjectionRequestWrapper wrappedReq =new SQLInjectionRequestWrapper((HttpServletRequest) req,
                                  config.getServletContext());
      chain.doFilter((HttpServletRequest)wrappedReq, resp);
   }
    else
      chain.doFilter(req,resp);
  }
}
```



Ejemplo: Filtrar inyección de código SQL

Esta clase crea un objeto **wrapper** de la petición original, que sobreescribe los métodos **getParameter()**, **getParameterValues()** y **getParameterMap().** Garantiza que el **servletLoguin** reciba parámetros seguros, en otro caso los blanquea.

```
public class SQLInjectionRequestWrapper extends HttpServletRequestWrapper {
    private ServletContext contexto;
    public SQLInjectionRequestWrapper(HttpServletRequest req, ServletContext ctx) {
          super(req);
          contexto = ctx;
    }
    public String getParameter(String p) {
       String param = super.getParameter(p);
       if (isInjected(param)){
             contexto.log(new Date() +" "+ this.getRemoteHost() + " " + this.getRequestURI());
             return "";
       } else{
             return param;
       }
   }
  public Map getParameterMap() {. . .}
 public String[] getParameterValues(java.lang.String p1) {. . .}
 public static boolean isInjected(String str) {
      String[] dangerKeywords = {"OR","Or","or","--","having", "="};
      for (int i = 0; i < dangerKeywords.length; i++) {
            if (str.contains(dangerKeywords[i])){
                  return true;
            }
      }
     return false;
 }
} // Fin de la clase SQLInjectionRequestWrapper
```

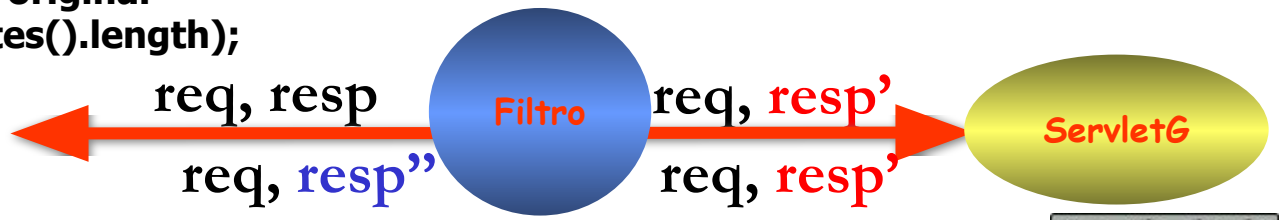
Programar Respuestas Customizados

- Para crear un objeto respuesta ***wrappeado*** se debe extender **HttpServletResponseWrapper** y sobreescribir los métodos que se desee del objeto HttpServletResponse: **getWriter()**, **getOutputStream()**.
- *Wrappear* la respuesta es más difícil que *wrappear* el requerimiento entrante. Naturalmente el objeto **HttpServletRequest** es de sólo lectura, sin embargo el objeto **HttpServletResponse** contiene mucha información generada, incluyendo los datos de salida.
- Si se desea manipular los datos de salida, la información debe ser capturada y luego usar un objeto ***writer* adaptado** para modificarla. En general, esto implica modificar los *headers* de la respuesta, por ejemplo *content-length* y *content-type*.
- El **post-procesamiento** de la respuesta podría consistir en:
  - **Comprimir dinámicamente el contenido de la respuesta**. La idea de este filtro es enviar al cliente menos información. El filtro debe examinar el *header* de la petición original para determinar si el cliente entiende el formato comprimido GZIP, en cuyo caso se aplica el algoritmo de compresión GZIP sobre la respuesta.
  - **Cambiar el formato de la respuesta de GIF a PNG**.
  - **Agregar datos en la respuesta**.

Ejemplo: Agregar datos a la respuesta

FiltroContador inserta en la respuesta un valor de un contador tomado del contexto.

```
class FiltroContador implements Filter {
  private FilterConfig config;

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

   Integer counter = (Integer)config.getServletContext().getAttribute("counter");
   PrintWriter out = resp.getWriter();
   // Construye un wrapper sobre el objeto resp que se recibe como parámetro
   CharResponseWrapper wrapper = new CharResponseWrapper((HttpServletResponse) resp);


   chain.doFilter(req, wrapper);

  // Escribimos en un buffer la respuesta
   CharArrayWriter  caw = new CharArrayWriter();
   caw.write(wrapper.toString().substring(0,wrapper.toString().indexOf("</body>")));
   caw.write("<p>\n<center>"+"Usted es el Visitante: "+"<font color='red'>"+counter+"</font></center>");
   caw.write("\n</body></html>");


   // Le configuramos la longitud a la respuesta original
   resp.setContentLength(caw.toString().getBytes().length);

   out.write(caw.toString());
   out.close();
 }
...
}
```



Ejemplo: Agregar datos a la respuesta

Esta clase crea una respuesta **wrappeada** que sobreescribe el método **getWriter()** (también podría sobreescribir al método **getOutputStream()**) para retornar el stream donde se escribe la salida.

```
public class CharResponseWrapper extends HttpServletResponseWrapper {
    private CharArrayWriter output;

    public CharResponseWrapper(HttpServletResponse resp) {
        super(resp);
        output = new CharArrayWriter();
    }

//este método se sobreescribe para retornar un output donde
//el ServletG escribe la salida

  public PrintWriter getWriter(){
        return new PrintWriter(output);
  }

  public String toString(){
        return this.output.toString();
  }
}
```

Ejemplo: Agregar datos a la respuesta

El **ServletG** es el generador de la página principal de una librería on-line.

resp es el objeto *wrappeado* `CharResponseWrapper`

```
public class ServletG extends HttpServlet implements Servlet {

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

  PrintWriter out=resp.getWriter(); //se obtiene el stream wrappeado (CharArrayWriter)


  resp.setContentType("text/html");
  out.print("<HTML>");
  out.print("<BODY>");
  out.print("<H1><BR>Librería Gandhi</H1><P>");
  out.print("<IMG border=\"0\"src=\"libro.gif\" width=\"123\"  height=\"113\">");
  out.print("<BR><A href=\"/gandhi/Compras\">Comienza a    Comprar</A></P>");
  out.print("</BODY>");
  out.print("</HTML>");
  out.close();          // se cierra el stream wrappeado !!
 }

}
```