

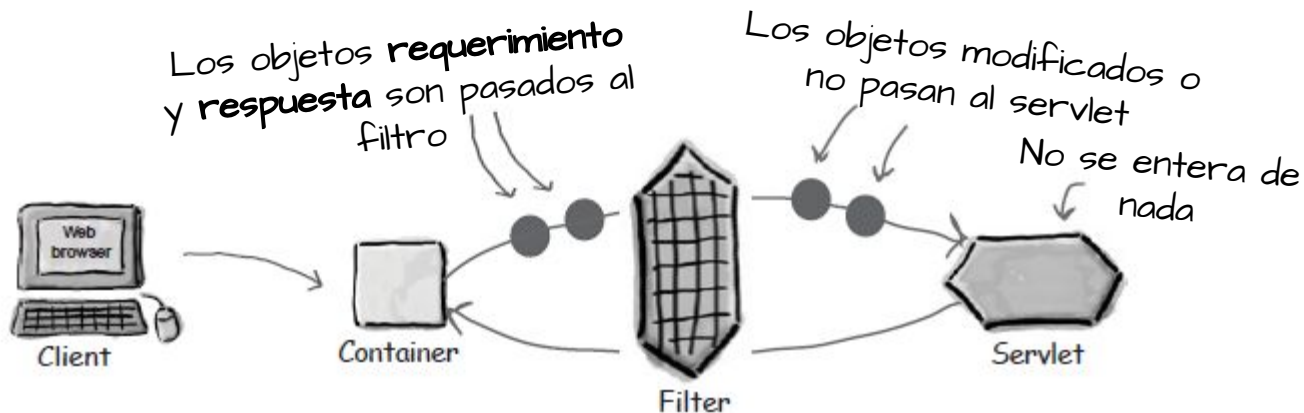
# Filtros

- 1 Características generales
- 2 Configuración
- 3 Ciclo de vida de los Filtros
  - **init()**, **doFilter()** y **destroy()**
  - El objeto **FilterChain**
  - Clases Wrappers.
- 4 Programando requerimientos y respuestas *customizados*.

# Filtros

## ¿Qué son y para qué sirven?

- Un **filtro** es una componente web que puede **interceptar y procesar** *un requerimiento HTTP* antes de que el mismo alcance el servlet y/o **interceptar y procesar** *una respuesta HTTP*, después de que el servlet ha finalizado pero antes de que la respuesta sea devuelta al cliente.
- Los **filtros** forman parte de una **cadena**, donde el último eslabón referencia al recurso solicitado por el cliente.
- Un **filtro** puede elegir pasar el requerimiento al próximo recurso de la cadena o retornar la respuesta al cliente.



En forma idéntica a otras componentes web Java, los filtros son clases Java independientes de la plataforma, que se compilan a código de bytes (bytecodes), se cargan dinámicamente y se ejecutan en un servidor web.

Los filtros están disponibles a partir de la versión 2.3 de la API de Servlets.

# Filtros

## ¿Qué pueden hacer?

Los **filtros** están habilitados para:

- Leer los datos del requerimiento.
- Modificar el requerimiento original antes de pasarlo.
- Modificar y/o manipular los datos de la respuesta que será devuelta al cliente.
- Devolver errores al cliente.

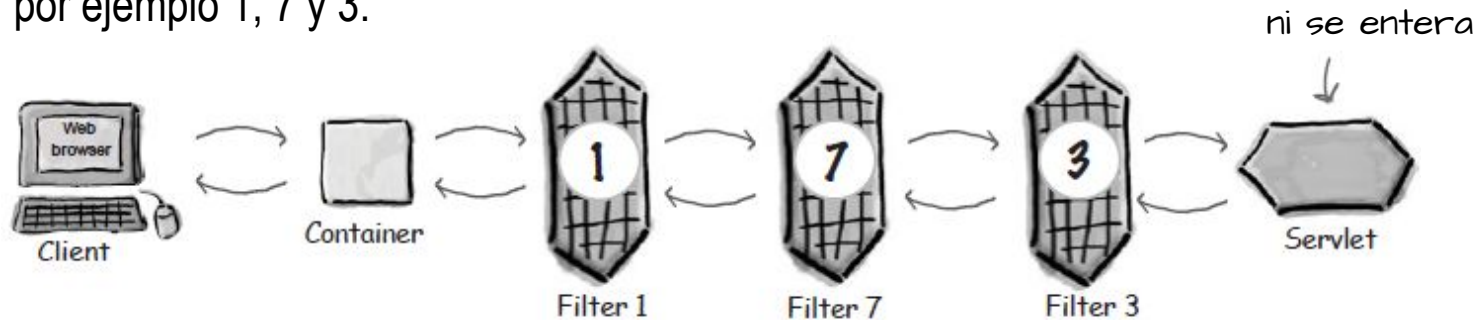
Ejemplos de componentes **filtros**:

- **Autenticación**: bloquear el acceso a usuarios no autenticados.
- **Logging and Auditoría**: guardar accesos para estadísticas.
- **Conversión de Imágenes**, por ejemplo de formato GIF a PNG
- **Compresión de Datos**: enviar menos contenido al cliente y así hacer más rápida la descarga. Esto es deseable, especialmente en clientes que tienen poco ancho de banda.
- **Encriptación**: de información sensible cuando viaja al cliente.

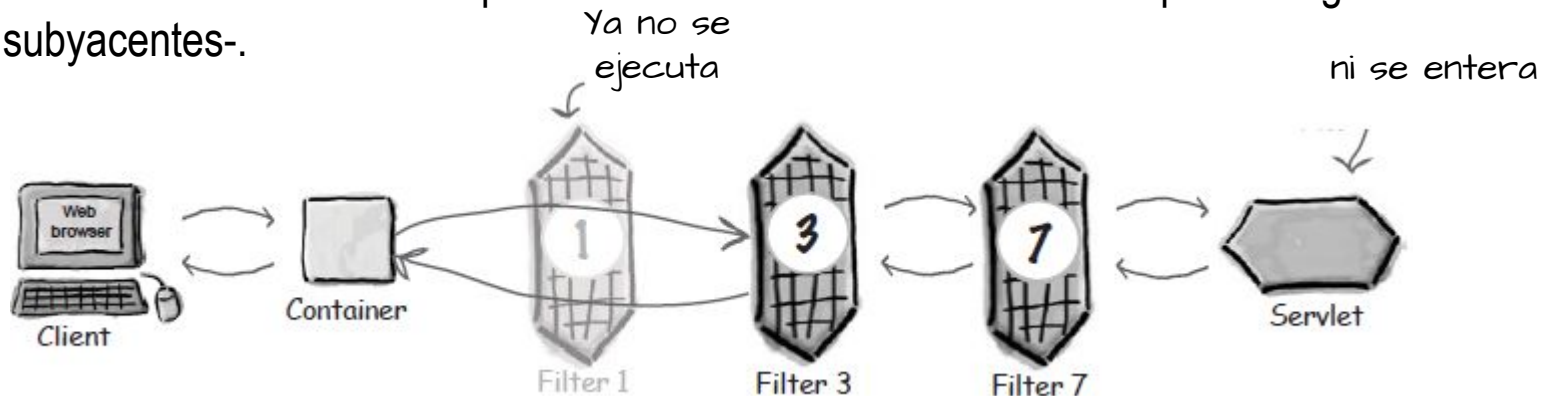
# Filtros

## Configuración

Los filtros son configurados en el archivo descriptor **web.xml**. Se podría configurar por ejemplo, que para un recurso o para un conjunto de URLs se debe ejecutar una cadena de filtros en un orden específico, por ejemplo 1, 7 y 3.



Cambiando la configuración del **web.xml** podríamos quitar un filtro y/o cambiarles de orden, brindando otras funcionalidad a la aplicación web sin necesidad de recompilar código -sin afectar los recursos subyacentes-.



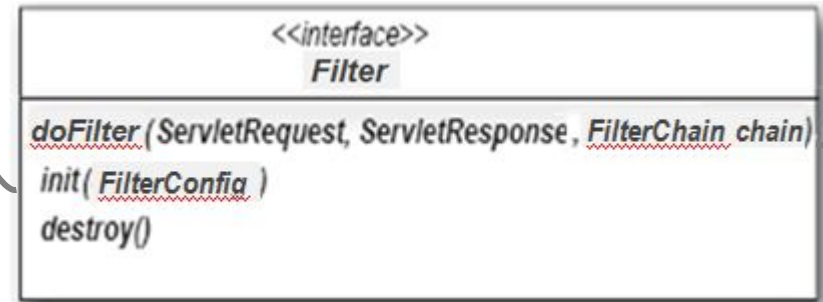
Los filtros permiten incorporar capas de pre-procesamiento y post-procesamiento a un requerimiento y a una respuesta, sin modificar los Servlets.

# Filtros

## Ciclo de Vida

El Contenedor Web gerencia el ciclo de vida de cada filtro, invocando a los 3 métodos definidos en la interface **javax.servlet.Filter**: **doFilter()**, **init()** y **destroy()**. Tienen, al igual que los Servlets, los métodos **init()** y **destroy()** y disponen del método **doFilter()** similarmente a los **doGet()/doPost()** de servlets.

El método **init(FilterConfig f)** es llamado por el Contenedor (por única vez) cuando el filtro es cargado en memoria y recibe como parámetro un objeto **FilterConfig**, que posibilita la lectura de los parámetros de inicialización del archivo web.xml, usando el método **getInitParameter("clave")**



El método **doFilter()** es invocado por el contenedor cuando se aplica un filtro a un **ServletRequest/ServletResponse**. Recibe el objeto **ServletRequest** y **ServletResponse** que son instancias de **HttpServletRequest** y **HttpServletResponse**, respectivamente y el objeto **FilterChain** que representa la cadena actual de filtros que está siendo aplicada al requerimiento o a la respuesta.

El Contenedor Web, es el responsable de cargar e instanciar a los filtros cuando el filtro es aplicado por primera vez. El contenedor, crea una única instancia por elemento **<filter>** declarado en el web.xml y también crea un objeto **FilterConfig** que le pasa como parámetro al método **init()**. El objeto **FilterConfig** permite acceder a los parámetros de inicialización y al **ServletContext** de la aplicación.

# Filtros

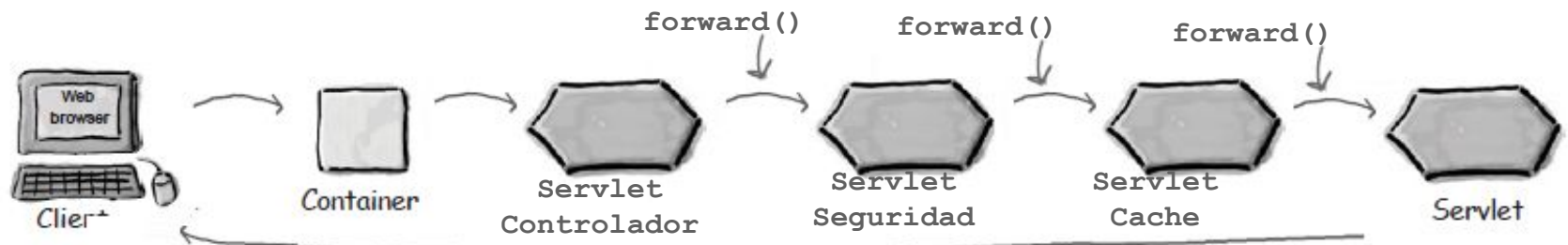
## El objeto FilterChain

- Para cada requerimiento nuevo, el contenedor arma una lista con los filtros que debe aplicar, con el mismo orden que aparecen en el web.xml. El objeto **FilterChain** representa la lista de filtros que deben ejecutarse sobre un requerimiento/respuesta.
- Los filtros proveen un mecanismo para aplicar capas de funcionalidad a un ServletRequest y/o ServletResponse. Esto marca una diferencia con los Servlets, los cuales tienen un único recurso como destino.
- Con filtros es fácil dividir una funcionalidad en capas y acomodarlas como se quiera.



### Cómo haríamos esto sin Filtros?

Deberíamos utilizar un mecanismo para despachar requerimientos



# Filtros

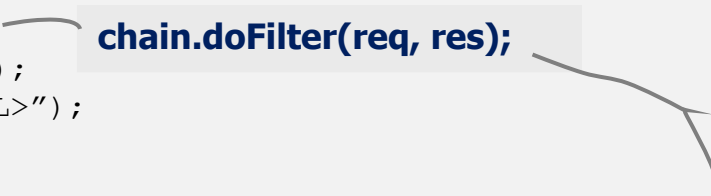
## Implementando la interface Filter: el método doFilter()

Para crear un filtro se debe implementar la interface **javax.Servlet.Filter** y sobrescribir el método **doFilter()** para darle comportamiento. Un filtro ejecutando su método **doFilter()** tiene 2 posibilidades:

### 1 Manejar completamente el par requerimiento/respuesta y retornar

Si un filtro necesita **detener la ejecución de filtros subsiguientes en la cadena**, simplemente retorna desde su método **doFilter()**. Esto causa que la aplicación web retorne al filtro previamente ejecutado (si es que había) y finalice enviando la respuesta al cliente.

```
public class UnFiltro implements javax.servlet.Filter {  
    . . .  
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws ... {  
        HttpServletRequest request=(HttpServletRequest) req;  
        HttpServletResponse response=(HttpServletResponse) res;  
        PrintWriter out = request.getWriter();  
  
        . . .  
        out.println("Hola");  
        out.println("</HTML>");  
        out.close();  
    }  
}
```



### 2 Procesar el par requerimiento/respuesta y pasarlos al próximo filtro en la cadena

Lo más común es que el filtro continúe con la ejecución **del siguiente elemento en la cadena**, entonces el método **doFilter(req, res)** del objeto **FilterChain** debe ser invocado para continuar con la ejecución del próximo recurso.

# Filtros

## Un filtro que computa el tiempo de respuesta de un Servlet

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

Se debe implementar  
esta interface

```
public class FilterTime implements Filter {
    private FilterConfig config;
```

Generalmente se sobrescribe para  
guardar el **FilterConfig**

```
    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 de inicio del request
```

```
        long antes = System.currentTimeMillis();
```

```
        chain.doFilter(req, resp);
```

Esto es lo que indica que el próximo  
filtro o Servlet- sea invocado

```
        // Capturamos el tiempo a la vuelta
```

```
        long despues = System.currentTimeMillis();
```

```
        nomServlet = ((HttpServletRequest)req).getRequestURI();
```

```
        config.getServletContext().log(nomServlet+": " + (despues - antes) + "ms");
```

```
    }
```

```
    public void destroy() {
```

```
        config=null;
```

```
    }
```

```
}
```

Con esta configuración, el  
**FilterTime** será aplicado a  
todos los requerimientos..

```
<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>
```

Este ejemplo esta basado  
en uno de los tantos  
filtros que vienen con

**Tomcat**



# Filtros

## Declaración

La configuración es muy parecida a la de los servlets, con la diferencia que los filtros necesitan de un mapeo para determinar qué filtrar y la ubicación que tendrán en la secuencia de invocación. El elemento `<filter-mapping>`, define a qué recursos de la aplicación web, se le aplicarán los filtros. Asocia un filtro con un conjunto de recursos usando `<url-pattern>` o un filtro con un servlet usando el subelemento `<servlet-name>`

```
<filter>                                     web.xml
  <filter-name>LoginFiltro</filter-name>
  <filter-class>cursoJ2EE.filtros.LoginFiltro</filter-class>
  <init-param>
    <param-name>archivoLog</param-name>
    <param-value>logs.txt</param-value>
  </init-param>
</filter>

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

**Declaración del filtro**

**Declaración de un mapeo para un conjunto de URLs**

Con anotaciones

```
@WebFilter(
    urlPatterns = "/*",
    initParams = @WebInitParam(name = "archivoLog", value = "logs.txt")
)
public class LoginFiltro implements Filter {
    // métodos del filtro
}
```

# Filtros

## Declaración usando Eclipse

La configuración desde eclipse es muy parecida a la de los servlets: se pueden configurar los servlet con XML o con anotaciones. Si el proyecto es versión 2.5 o inferior, por defecto definirá la configuración usando XML, de lo contrario lo hará con anotaciones.

**Create Filter**  
Enter servlet filter deployment descriptor specific information.

Name:

Description:

Initialization parameters:

Name	Value	Description
para1	22	
para2	33	

Filter mappings:

URL Pattern / Servlet Name	Dispatchers
/Servlet1	

☐ Asynchronous Support

< Back   Next >   Finish   Cancel

```
<filter>
  <filter-name>Filter2</filter-name>
  <filter-class>filtros.Filter2</filter-class>
  <init-param>
    <param-name>para1</param-name>
    <param-value>22</param-value>
  </init-param>
  <init-param>
    <param-name>para2</param-name>
    <param-value>33</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>Filter2</filter-name>
  <url-pattern>/Servlet1</url-pattern>
</filter-mapping>
```

```
@WebFilter(
    urlPatterns = { "/Servlet1" },
    initParams = {
        @WebInitParam(name = "para1", value = "22"),
        @WebInitParam(name = "para2", value = "33")
    })
public class Filter2 implements Filter {
```

# Filtros

## Declaración

- Por defecto, una cadena de filtros está definida para manejar solamente requerimientos hechos por los clientes. Si un requerimiento es despachado 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 usando el elemento `<dispatcher>` en el `web.xml`

```
<filter>
  <filter-name>filtroContador</filter-name>
  <filter-class>misFiltros.FiltroContador</filter-class>
</filter>
...
<filter-mapping>
  <filter-name>filtroContador</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Esta configuración indica que el filtro de nombre `FiltroContador`, será aplicado solamente para requerimientos provenientes de clientes (`REQUEST`) y cuando se invoca al método `include()` del `RequestDispatcher` (`INCLUDE`).

Con el elemento `<dispatcher>` se puede indicar si el filtro se aplica:

- Cuando el requerimiento proviene directamente de un cliente, 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`.

```
@WebFilter(
    urlPatterns = "/filtroContador",
    dispatcherTypes = {DispatcherType.REQUEST, DispatcherType.INCLUDE}
)
public class MyFilter implements Filter {
    // métodos del filtro
}
```

# Filtros

## Clases Wrappers

- Otra de las características útiles de los filtros es la habilidad para *wrappear* un requerimiento y/o una respuesta. *Wrappear* significa encapsular un requerimiento o respuesta dentro de otro *customizado* (adaptado).
- Esto es necesario para manipular un objeto request y/o response de una manera no tradicional.
- Para hacer *wrapping*, la **especificación 2.3** incorporó clases especiales:

**HttpServletRequestWrapper** que implementa --> **HttpServletRequest**

**HttpServletResponseWrapper** que implementa ---> **HttpServletResponse**

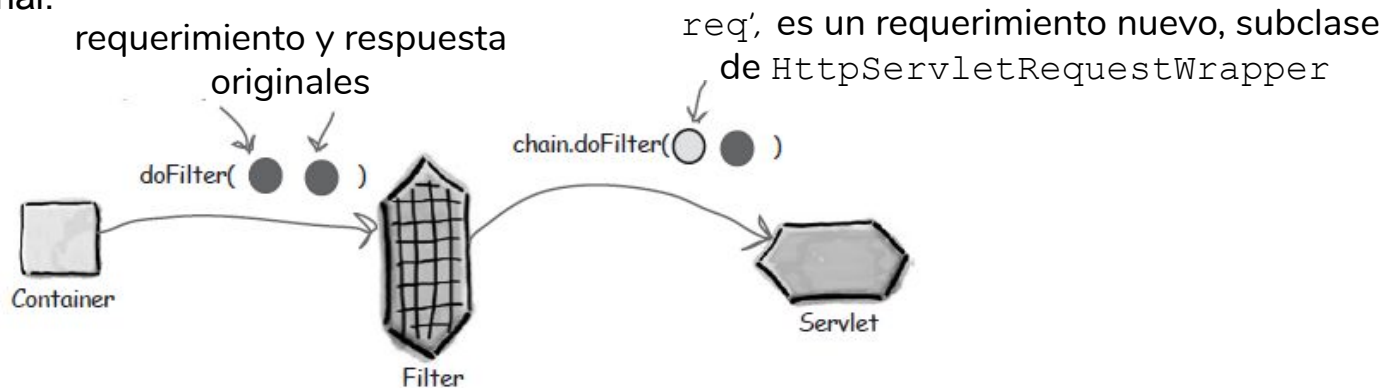
**NOTA:** En realidad, estas dos clases “*wrapean*” un objeto del tipo que ellas implementan! En otras palabras, NO proveen directamente una implementación de los métodos de la interface sino que mantienen una referencia al objeto creado por el Contendor Web subyacente, sobre el que invocan sus métodos/implementaciones.

Crear un *wrapper* personalizado consiste en extender alguna de las clases **HttpServletRequestWrapper** o **HttpServletResponseWrapper** y sobrescribir los métodos necesarios.

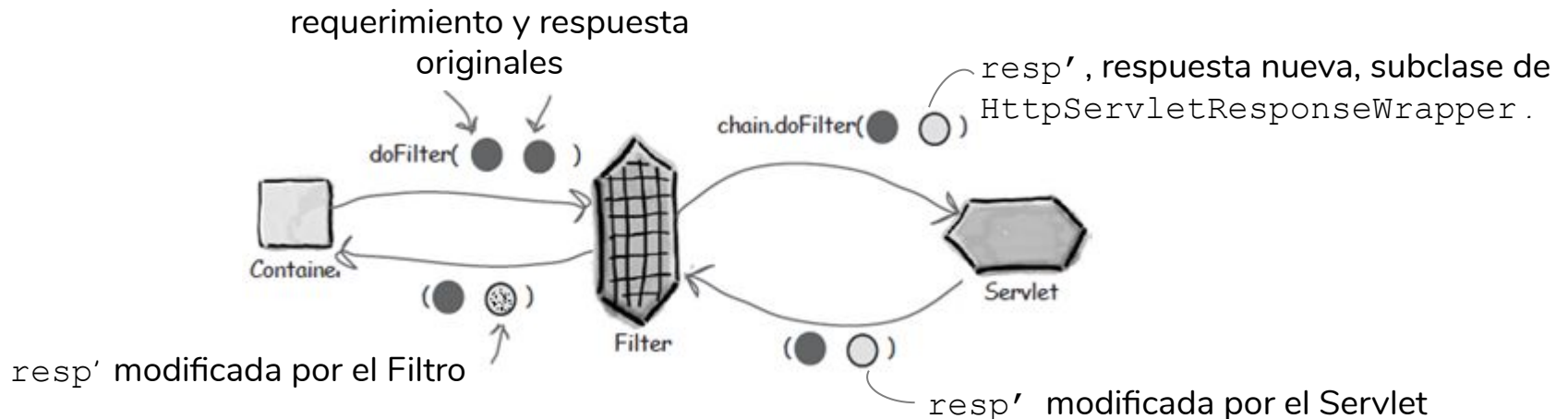
# Filtros

## Programando requerimientos y respuestas *customizados*

(a) Un filtro que **modifica un requerimiento** debe capturar el requerimiento antes de que alcance el recurso web, modificarlo y mandarlo modificado (req') en el método doFilter() para que lo reciba el próximo filtro en la cadena o el recurso final.



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



# Filtros

## Programando requerimientos *customizados*

Para crear un requerimiento *customizado* debemos extender **HttpServletRequestWrapper** y sobrescribir el código de los métodos que se desee del requerimiento entrante, por ejemplo los métodos **getParameter()**, **getHeader()**, ...

Algunos ejemplos de situaciones donde se podría crear un wrapper para un requerimiento:

- Existen casos, en donde la funcionalidad de los métodos de **HttpServletRequest** necesita ser cambiada. Por ejemplo, para proveer información de Auditoria. En este caso, la información acerca de los métodos invocados de **HttpServletRequest** se podrían grabar en un archivo de logs.
- Manipular el requerimiento para detectar ataques con **SQL Injection**. El filtro debe verificar si los parámetros tienen palabras claves de SQL u operadores que cambien la lógica o sintaxis de la sentencia SQL. Si esto ocurre se podría blanquear el parámetro del requerimiento y guardar los datos relacionados al ataque en un archivo de logs.
- Crear y llenar objetos con información que posteriormente será desplegada por JSP's o **servlets**. En este caso, los filtros podrían llenar **JavaBeans** y ligarlos al alcance **request**, de manera tal que la información puede ser mostrada mediante tags estándares de JSP.

# Filtros

## Programando requerimientos *customizados*

Dada la tradicional consulta a una tabla de usuarios:

```
"SELECT * FROM Usuarios WHERE USERNAME='"+user+" ' AND PASSWORD='"+pass+" ' "
```

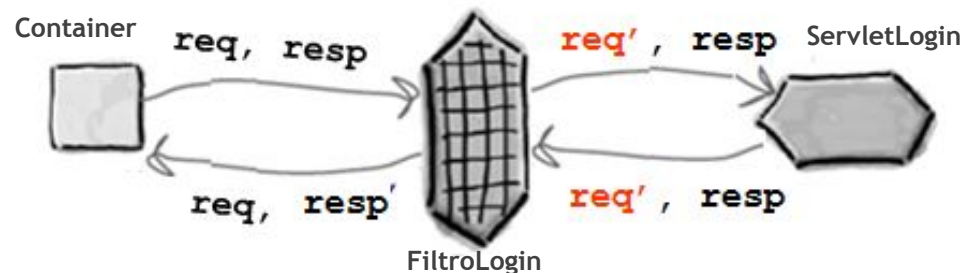
¿Qué pasa si un usuario ingresa en **user**: ' OR 1=1 --

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

El objetivo del FiltroLogin es detectar ataques a la base de datos con SQLInjection.

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

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws ServletException, IOException {
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        if (!username.equals("") || !password.equals("")) {
            SQLInjectionRequestWrapper wrappedReq
            = newSQLInjectionRequestWrapper((HttpServletRequest)request, config.getServletContext());
            chain.doFilter((HttpServletRequest)wrappedReq, response);
        }
        else
            chain.doFilter(request, response);
    }
}
```



# Filtros

## Programando requerimientos *customizados*

Esta clase crea un requerimiento *wrapper* que sobrescribe los métodos `getParameter()`, `getParameterValues()` y `getParameterMap()` de manera que el `servletLogin` reciba parámetros seguros.

```
public class SQLInjectionRequestWrapper extends HttpServletRequestWrapper {
    private ServletContext _servletContext;

    public SQLInjectionRequestWrapper(HttpServletRequest req, ServletContext ctx) {
        super(req); // para que la superclase tenga una referencia al requerimiento original
        _servletContext = ctx;
    }

    public String getParameter(String p) {
        String param = super.getParameter(p);
        if (isInjected(param)){
            _servletContext.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;
    }
}
```

esta clase implementa la interface `HttpServletRequest`

Se guardan detalles del ataque

También habría que sobrescribir estos dos métodos que leen parámetros

sobrescritura

PI



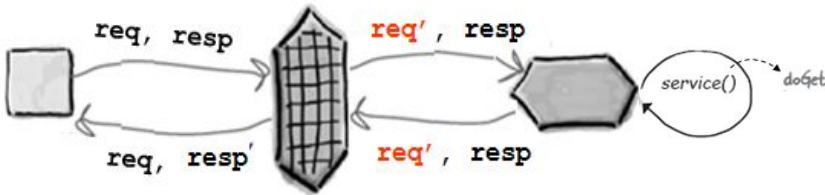
# Filtros

## Programando requerimientos *customizados*

El recurso final en este caso es el [ServletLogin](#), pero podría generalizarse.

```
public class ServletLogin extends HttpServlet implements Servlet {  
  
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws  
        ServletException, IOException {  
  
        String user = request.getParameter("username");  
        String pass = request.getParameter("password");  
  
        // procesamiento para verificar si es un usuario válido  
  
    }  
    . . .  
}
```

**request** es un objeto de tipo  
**SQLInjectionRequestWrapper**



```
<filter>
<filter-name>filtroLogin</filter-name>
<filter-class>misFiltros.FiltroLogin</filter-class>
</filter>
...
<filter-mapping>
    <filter-name>filtroLogin</filter-name>
    <servlet-name>/servletLogin</servlet-name>
</filter-mapping>
```

web.xml

# Filtros

## Programando respuestas *customizadas*

Para crear una respuesta customizada debemos extender `HttpServletResponseWrapper` y sobrescribir los métodos que se desee, del objeto `HttpServletResponse` (por ejemplo `getWriter()`, `getOutputStream()`).

Crear wrappers para las respuestas es más difícil que para los requerimientos. Naturalmente el objeto `HttpServletRequest` es sólo de 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 usarse un objeto `writer` adaptado para modificarla. Muchas veces, esto implica, modificar los HEADERS de la respuesta, por ejemplo `content-length` y `content-type`.

El **post-procesamiento** de la respuesta podría consistir:

- Comprimir dinámicamente el contenido de la respuesta. La idea de este filtro es enviar al cliente menos información. El filtro debe examinar el HEADER del requerimiento 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.

# Filtros

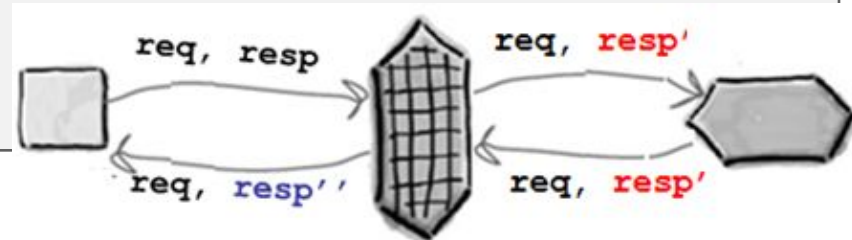
## Programando respuestas *customizadas*

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

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

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain){
        . . .
        Integer counter = (Integer)config.getServletContext().getAttribute("counter");
        counter = counter+1;
        config.getServletContext().setAttribute("counter",counter);
        . . .
        PrintWriter out = resp.getWriter();
        // Le hace un wrap al objeto resp que se recibe como parámetro
        CharResponseWrapper wrapper = new CharResponseWrapper((HttpServletResponse) resp);

        chain.doFilter(req, wrapper);
        CharArrayWriter caw = new CharArrayWriter();
        caw.write(wrapper.toString().substring(0,wrapper.toString().indexOf("</body>")));
        caw.write("<p>\n<center>"+ "Usted es el Visitante:"+counter+"</font></center>");
        caw.write("\n</body></html>");
        // Le configuramos la longitud a la respuesta
        resp.setContentLength(caw.toString().getBytes().length);
        out.write(caw.toString());
        out.close();
    }
}
```



# Filtros

## Programando respuestas *customizadas*

Esta clase crea una respuesta “wrapper” que sobrescribe el método **getWriter()** porque la respuesta que se manejará es texto. Este método retorna un objeto sobre el que se pretende que escriba el Servlet. Un objeto de esta clase sería pasado al método **doFilter()** en la clase **FiltroContador**.

```
public class CharResponseWrapper extends HttpServletResponseWrapper {  
    private CharArrayWriter output = new CharArrayWriter();  
    public CharResponseWrapper(HttpServletResponse resp) {  
        super(resp); //para que la superclase tenga una referencia al req. original  
    }  
  
    //este método se sobrescribe para retornar un output donde el ServletG escriba  
    public PrintWriter getWriter(){  
        return new PrintWriter(output);  
    }  
  
    public String toString(){  
        return this.output.toString();  
    }  
}
```

*esta clase implementa la interface HttpServletResponse*

Si la respuesta no es texto, se podría hacer algo similar sobrescribiendo el método **getOutputStream()**.

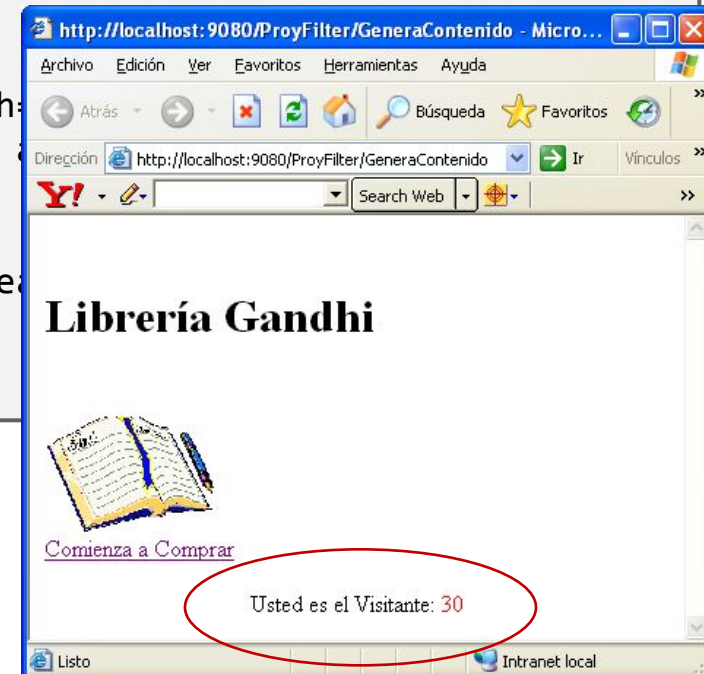
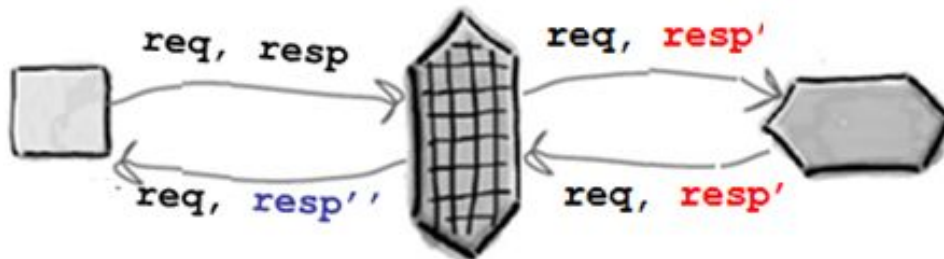
# Filtros

## Programando respuestas *customizadas*

El recurso final podría ser cualquier Servlet, JSP o recurso estático. En este caso el ServletG, es un generador de una página principal de una librería on line.

**resp** es el objeto *wrappeado*

```
public class ServletG extends HttpServlet implements Servlet {  
  
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws  
        ServletException, IOException {  
  
        PrintWriter out = resp.getWriter(); //se obtiene la instancia del Wrapper!!!  
        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=");  
        out.print("<BR><A href=\"/gandhi/Compras\">Comienza a");  
        out.print("</BODY>");  
        out.print("</HTML>");  
        out.close();           // se cierra el stream wrapper  
    }  
}
```



# Bibliografía

- Servlet y JavaServer Pages, Jayson Falkner, Kevin Jones.  
Capítulo 8, *Filters*
- Head First Servlets & JSP, Bryan Basham, Kathy Sierra, Bert Bates. O'Reilly  
Capítulo 13, *The power of Filters*
- Ejemplos que se distribuyen con Tomcat