



# ***Componentes Web***

## *Sesión 7 - Filtros y wrappers*



## Índice

- Introducción a los filtros
- Configuración de los filtros
- Implementación de un filtro
- Introducción a los wrappers
- Aplicaciones de los wrappers
- Implementación de un wrapper
- Utilización de wrappers



## Servlets y filtros

- Los servlets se invocan cuando se hace referencia a una determinada URL a la que están mapeados y devuelven una respuesta
- Puede interesarnos realizar una acción común siempre que se haga una petición a un conjunto de recursos
  - Por ejemplo registrar el número de accesos a nuestras páginas
- Los filtros nos permitirán realizar esto de forma transparente y sin código redundante



## ¿Qué es un filtro?

- Los filtros son componentes del servidor
  - Interceptan la petición del cliente antes de procesarse
  - Interceptan el contenido generado antes de devolverlo al cliente
- Pueden realizar cualquier acción al interceptar estos mensajes
  - Por ejemplo registrar el acceso en una BD
- No suelen generar contenido por si mismos
  - Pueden modificar los mensajes de petición y respuesta
- Actuarán sobre el conjunto de recursos que se especifique en la configuración (`web.xml`)
- Son componentes altamente reutilizables
  - Hacen la aplicación modular



## Aplicaciones de los filtros

- Autenticación de usuarios
- Transformación con hojas XSL-T
- Transformación de imágenes
- Cifrado de datos
- Compresión de datos
- Registro de acceso a recursos
- Log de accesos
- Etc...



## Clase del filtro

- Para implementar un filtro crearemos una clase que implemente la interfaz `Filter` y debe ser mapeado mediante `@WebFilter`

```
@WebFilter("/ruta/*")
public class FiltroEjemplo implements Filter {
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        ...
    }

    public void init(FilterConfig config) throws ServletException {
        // Código de inicialización del filtro
        ...
    }

    public void destroy() {
        // Libera recursos del filtro
        ...
    }
}
```



## Declaración de filtros en web.xml

- En versiones anteriores a servlets 3.0 los filtros deben ser declarados en el descriptor de despliegue

```
<filter>
  <filter-name>Filtro de ejemplo</filter-name>
  <filter-class>org.especialistajee.filtros.FiltroEjemplo</filter-class>
</filter>
```

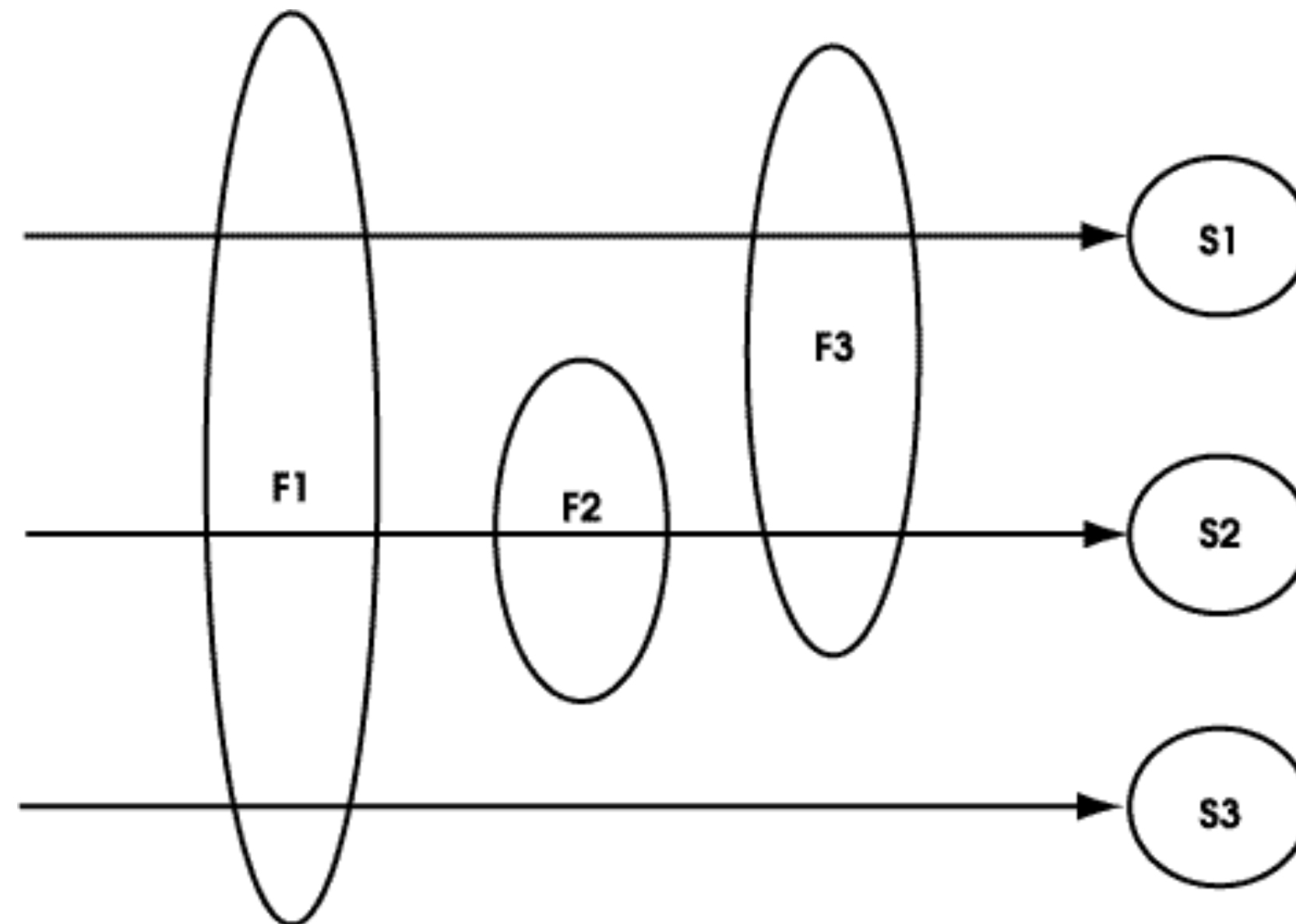
- Deberemos mapear el filtro al conjunto de recursos que serán interceptados por él

```
<filter-mapping>
  <filter-name>Filtro de ejemplo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```



## Cadenas de filtros

- Podemos tener varios filtros mapeados a un mismo recurso
  - Se formará una cadena de filtros







## Encadenamiento

- Al interceptar la petición en `doFilter` será responsabilidad nuestra que esta petición se ejecute
- Para que se ejecute deberemos llamar al método

```
chain.doFilter(request, response);
```

- Si hubiese un siguiente filtro en la cadena con esto haremos que la petición pase a este próximo filtro
  - Será responsabilidad de este nuevo filtro que la petición se ejecute
- Después de llamar a este método el servidor ya habrá generado la respuesta correspondiente



## Filtrado

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

    // Se ha interceptado la petición
    // En este punto podemos modificar o analizar la petición

    chain.doFilter(request, response);

    // En este punto el servidor ya habrá producido
    // la respuesta en response
}
```



## Inicialización y destrucción

- Es conveniente guardar en un campo de nuestro filtro el objeto `FilterConfig` proporcionado en la inicialización del mismo

```
FilterConfig config;

public void init(FilterConfig config) throws ServletException {
    // Código de inicialización del filtro
    this.config = config;
    ...
}

public void destroy() {
    // Libera recursos del filtro
    config = null;
    ...
}
```



## Acceso al contexto

- Para acceder al contexto de la aplicación deberemos utilizar el objeto `FilterConfig`

```
ServletContext context = config.getServletContext();
```

- Este objeto nos permitirá también leer los parámetros de inicialización que hayamos incluido en el fichero `web.xml` para el filtro

```
String valor = config.getInitParameter(nombre_param);
```



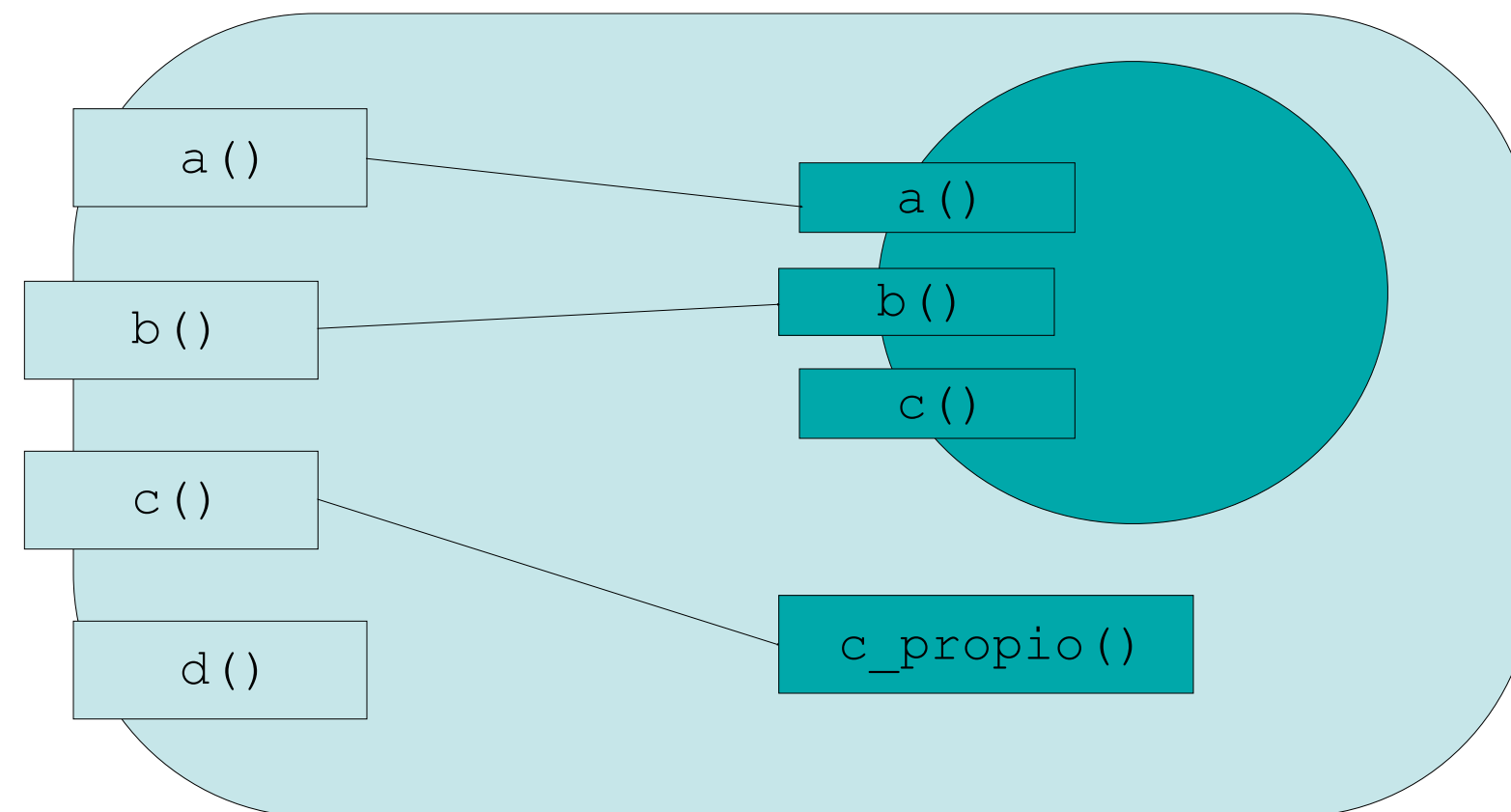
## Acceso a la respuesta

- Problema:
  - La respuesta escrita en `response` se devuelve directamente al cliente
  - No podemos analizarla dentro del filtro porque ya ha sido devuelta al cliente
- Solución:
  - Utilizar un objeto `HttpServletResponse` propio (*wrapper*) que no devuelva la respuesta directamente al cliente (guardar en un *buffer* temporal)



## ¿Qué es un wrapper?

- Objeto que envuelve a otro objeto
  - El objeto interno se guarda en un campo del *wrapper*
- Implementa la misma interfaz que el objeto interno
- Para implementar cada método podemos:
  - Invocar el mismo método en el objeto interno
  - Definir una implementación alternativa





## Wrappers de petición y respuesta

- Podemos crear *wrappers* para los objetos de petición y respuesta

```
HttpServletRequest
```

```
HttpServletResponse
```

- Redefinir los métodos para los cuales queremos cambiar el comportamiento
  - P.ej., redefinir el método `getOutputStream` de la respuesta para que sea escrita en un buffer
- Utilizaremos estos objetos (*wrappers*) en lugar de los objetos petición y respuesta originales



## Wrappers de la petición

- Envuelven a un objeto `HttpServletRequest`
- Utilizaremos un *wrapper* de la petición para cambiar propiedades de la petición que no puedan ser modificadas utilizando métodos del objeto anterior
- P.ej., añadir parámetros
  - Los parámetros que nos devuelve el objeto petición son los que se enviaron desde el cliente
  - Puede interesarnos añadir parámetros internamente en el servidor para simular una llamada con dichos parámetros
  - Podemos utilizar un *wrapper* para que este nuevo objeto de petición contenga dichos parámetros





## Wrappers de la respuesta

- Envuelven un objeto `HttpServletResponse`
- Utilizaremos un *wrapper* para poder leer desde el lado del servidor las propiedades o el contenido de la respuesta generada, evitando que se devuelva directamente al cliente
  - Comprobar o modificar propiedades (cabeceras) de la respuesta
  - Analizar el contenido devuelto
  - Modificar el contenido devuelto
    - Comprimir la respuesta
    - Codificar la respuesta
    - etc.



## Objetos wrappers

- Disponemos de objetos *wrappers* para la petición y respuesta
  - Petición: `HttpServletRequestWrapper`
  - Respuesta: `HttpServletResponseWrapper`
- Debemos proporcionar el objeto original en el constructor
- Por defecto todos los métodos del *wrapper* invocan el método correspondiente en el objeto interno
  - Deberemos crear subclases de estos *wrappers* para redefinir los métodos necesarios



## Wrapper de respuesta genérico

```
public class GenericResponseWrapper extends HttpServletResponseWrapper {  
  
    private ByteArrayOutputStream output;  
  
    public GenericResponseWrapper(HttpServletResponse response) {  
        super(response);  
        output = new ByteArrayOutputStream();  
    }  
  
    public byte[] getData() {  
        return output.toByteArray();  
    }  
  
    public ServletOutputStream getOutputStream() {  
        return new FilterServletOutputStream(output);  
    }  
  
    public PrintWriter getWriter() {  
        return new PrintWriter(getOutputStream(), true);  
    }  
}
```



## Flujo de salida

- El flujo de salida debe ser de tipo `ServletOutputStream`
- Creamos un subtipo propio de este flujo

```
public class FilterServletOutputStream extends ServletOutputStream {
    private DataOutputStream stream;

    public FilterServletOutputStream(OutputStream output) {
        stream = new DataOutputStream(output);
    }

    public void write(int b) throws IOException {
        stream.write(b);
    }

    public void write(byte[] b) throws IOException {
        stream.write(b);
    }

    public void write(byte[] b, int off, int len) throws IOException {
        stream.write(b, off, len);
    }
}
```



## Uso del wrapper de respuesta genérico

- Crear objeto *wrapper* proporcionando como objeto interno el objeto respuesta original

```
GenericResponseWrapper wrapper = new GenericResponseWrapper(response);
```

- Llamar a `doFilter` proporcionando como respuesta este objeto

```
chain.doFilter(request, wrapper);
```

- La respuesta ya habrá sido escrita en el *wrapper*, podemos consultarla

```
byte [] datos = wrapper.getData();
```

- Si queremos devolver la respuesta al cliente, deberemos volcarla a la respuesta original

```
response.getOutputStream().write(datos);
```



## Otros wrappers

- Si tenemos un *wrapper* de la petición se podrá utilizar de la misma forma

```
MiWrapperPeticion wrapper_req =  
    new MiWrapperPeticion(request);  
chain.doFilter(wrapper_req, wrapper_res);
```

- También podremos utilizar los *wrappers* con los métodos `forward` e `include` de `RequestDispatcher`
  - Podemos consultar el contenido de cualquier recurso

```
rd.include(wrapper_req, wrapper_res);  
rd.forward(wrapper_req, wrapper_res);
```



**¿Preguntas?**