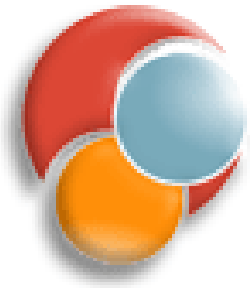


Componentes Web

Sesión 9: Filtros y Wrappers

Puntos a tratar

- 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 que interceptan en caso necesario
- 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

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

Declaración de filtros

- Para poder utilizar un filtro debe ser declarado en el descriptor de despliegue

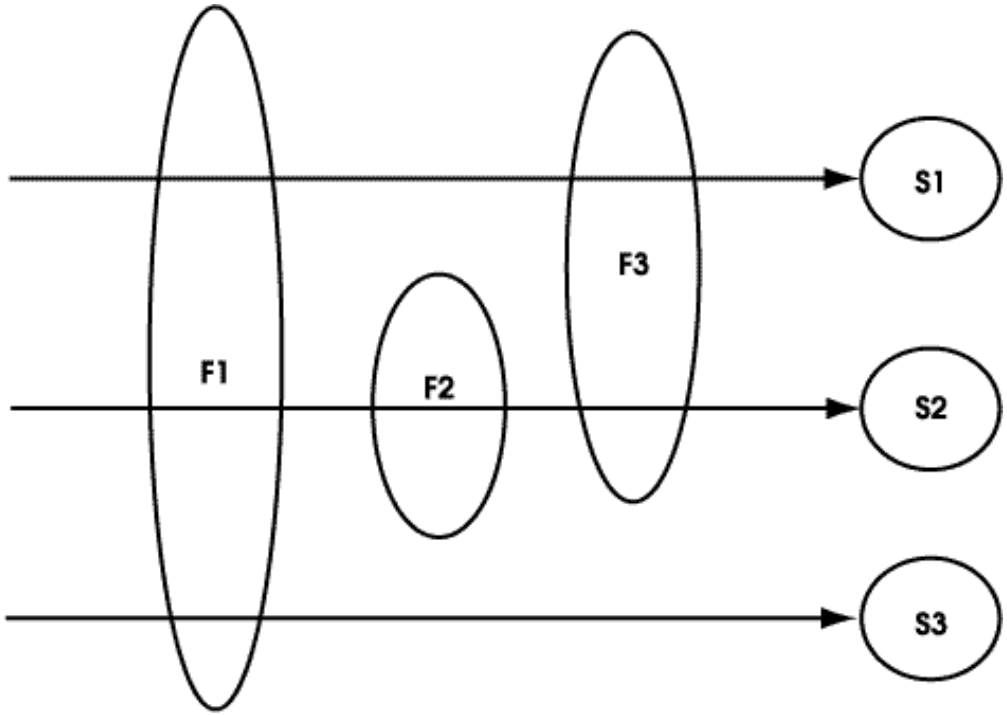
```
<filter>
  <filter-name>Filtro de ejemplo</filter-name>
  <filter-class>es.ua.jtech.filtro.Ejemplo</filter-class>
</filter>
```

- 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



Clase del filtro

- Para implementar un filtro crearemos una clase que implemente la interfaz `Filter`

```
public class MiFiltro implements Filter {
   public void doFilter(ServletRequest request,
                                ServletResponse response,
                                FilterChain chain)
                        throws IOException, ServletException
   {
     // Se ejecuta al interceptarse la petición
   }

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

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

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 peticion

   // En este punto podemos modificar o analizar la peticion

   chain.doFilter(request, response);

   // El servidor ya habra producido
   // la respuesta en response

   // En este punto podremos modificar o analizar
   // la respuesta producida
}
```

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);
```

Un problema concreto

- Vamos a estudiar un ejemplo de aplicación web que podríamos querer desarrollar:
  - Desarrollar un sitio web inteligente que aprenda las preferencias de cada usuario, permitiendo de esta forma sugerir al usuario páginas que puedan ser de su interés, aunque todavía no las haya visitado. Para ello debemos:
    - Interceptar todas las peticiones que hace el usuario
    - Obtener el documento que ha solicitado el usuario en cada petición
    - Analizar el contenido de este documento extrayendo una serie de palabras clave
    - Incrementar en nuestra BD el interés de este usuario en cada una de estas palabras clave
    - Podremos recomendar las páginas que traten las palabras clave más visitadas por el usuario

Implementación

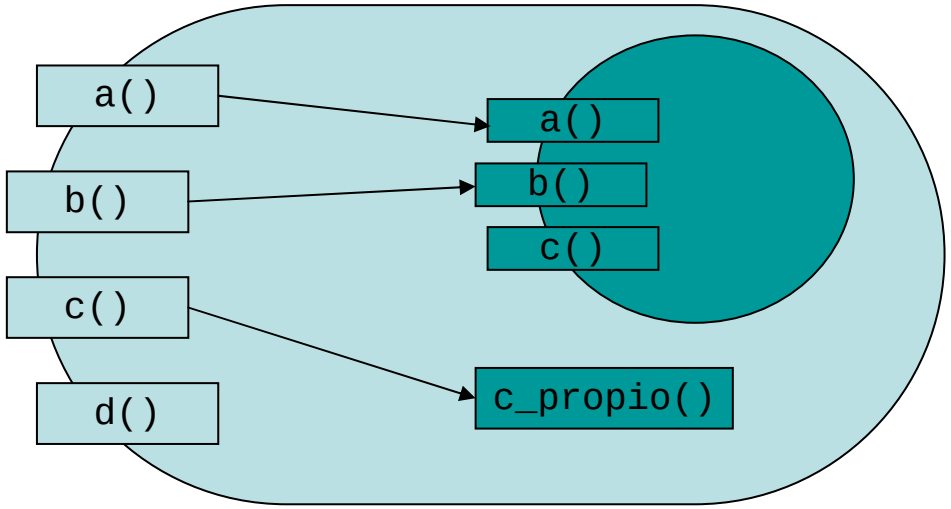
- Utilizar un filtro para interceptar las peticiones
- Una vez interceptada deberemos obtener el contenido devuelto por el recurso solicitado
- Al llamar a

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

- Se escribirá la respuesta del recurso en `response`
- 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 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 queramos 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_res = new GenericReponseWrapper(response);
```

- Llamar a `doFilter` proporcionando como respuesta este objeto

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

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

```
byte [] datos = wrapper_res.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);
```