



# Tehnologii Java

*Curs -*

Cristian Frăsinaru

`acf@infoiasi.ro`

Facultatea de Informatică

**Universitatea "Al. I. Cuza" Iași**





# Filtrarea comunicării la nivelul Web



# Cuprins

---

- Conceptul de *filtrare*
- Șablonul de proiectare Intercepting Filter
- Crearea filtrelor în Java EE
- Folosirea decoratorilor
- Maparea filtrelor

# Conceptul de *filtrare*

# Problema

La primirea unei cereri pot fi necesare diverse procesări:

- Clientul a fost autentificat ?
- Există o sesiune validă pentru acest client ?
- Este adresa IP dintr-o rețea de încredere ?
- Modul de realizare a cererii respectă constrângerile sistemului ?
- Ce codificare este folosită pentru reprezentarea datelor ?
- Tipul de browser folosit de client este suportat ?

# Exemplu

In componenta de login

```
User user = new User();
user.setName(request.getParameter("userName"));
user.setPassword(request.getParameter("userPassword"));

session.setAttribute("user", user);
```

In fiecare componenta pe care vrem sa o "securizam"

```
User user = (User) session.getAttribute("user");
if (user != null) {
    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/welcome.jsp");
    dispatcher.forward(request, response);
    getServletContext().log("User " + user.getName() + " logged in.");
} else {
    response.sendRedirect("login.jsp");
}
```

# Contextul

Mecanismul de tratare a unei cereri de la nivelul de prezentare trebuie să fie capabil să:

- Direcționeze cererea către componenta responsabilă cu tratarea ei (*handler*)
- Identifice diverse caracteristici ale cererii și să execute **procesări**:
  - verificare
  - modificare
  - compresare
  - codificare/decodificare

# Ce dorim ?

- Să separăm componentele de pre-procesare sau post-procesare de cele dedicate creării răspunsului.
- Să putem adăuga/elimina servicii declarativ, fără a afecta componentele existente.
- Să utilizăm serviciile în diverse combinații, după caz:
  - Autorizare
  - Autenticare
  - Debugging
  - Logging, etc.

# Soluția

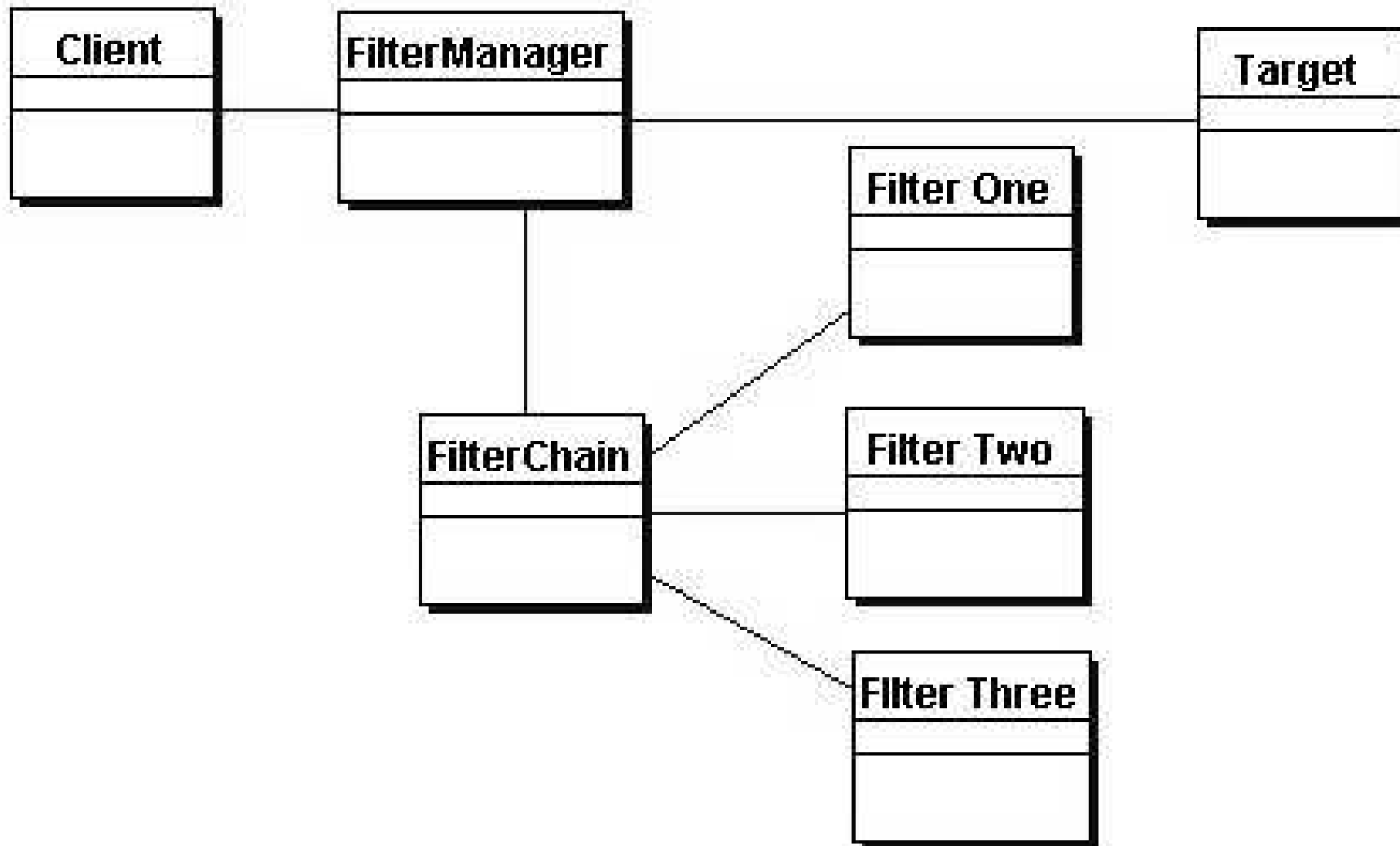


Crearea unei arhitecturi de **filtrare** care să permită:

- Definirea în mod standard a unui filtru
- Declararea în mod standard a aplicabilității filtrelor printr-un mecanism de tip *plug-in*.
- Interceptarea cererilor și aplicarea selectivă a filtrelor
- Interceptarea răspunsurilor și aplicarea selectivă a filtrelor



# Intercepting Filter





# Implementarea filtrelor in Java EE



# Ce este un filtru

Un **filtru** este un obiect care efectuează o acțiune de filtrare asupra:

- **cererii** către o componentă web (servlet, etc.)
- **răspunsului** venit de la o componentă

**Crearea și utilizarea** filtrelor presupun:

- Crearea claselor corespunzătoare filtrelor
- Personalizarea obiectelor de tip cerere/răspuns
- Definirea secvențelor de filtrare pentru fiecare componentă Web.

# Interfața Filter

```
public interface Filter() {
    /**
     * Called by the web container to indicate to a filter
     * that it is being placed into service. */
    void init(FilterConfig filterConfig);

    /**
     * The doFilter method of the Filter is called by the container
     * each time a request/response pair is passed through the chain
     * due to a client request for a resource at the end of the chain */
    void doFilter(ServletRequest request,
                 ServletResponse response,
                 FilterChain chain);

    void destroy();
}
```

# Structura unui filtru

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

        doBeforeProcessing(request, response);
        Throwable problem = null;
        try {
            chain.doFilter(request, response);
        } catch(Throwable t) {
            problem = t;
            t.printStackTrace();
        }
        doAfterProcessing(request, response);
        if (problem != null) {
            precessError(problem, response);
        }
    }
    ...
}
```

# Secvențe de filtre

## Cererea:

*Client* → *Filtrul*<sub>1</sub> → ...*Filtrul*<sub>n</sub> → *Componenta Destinatie*

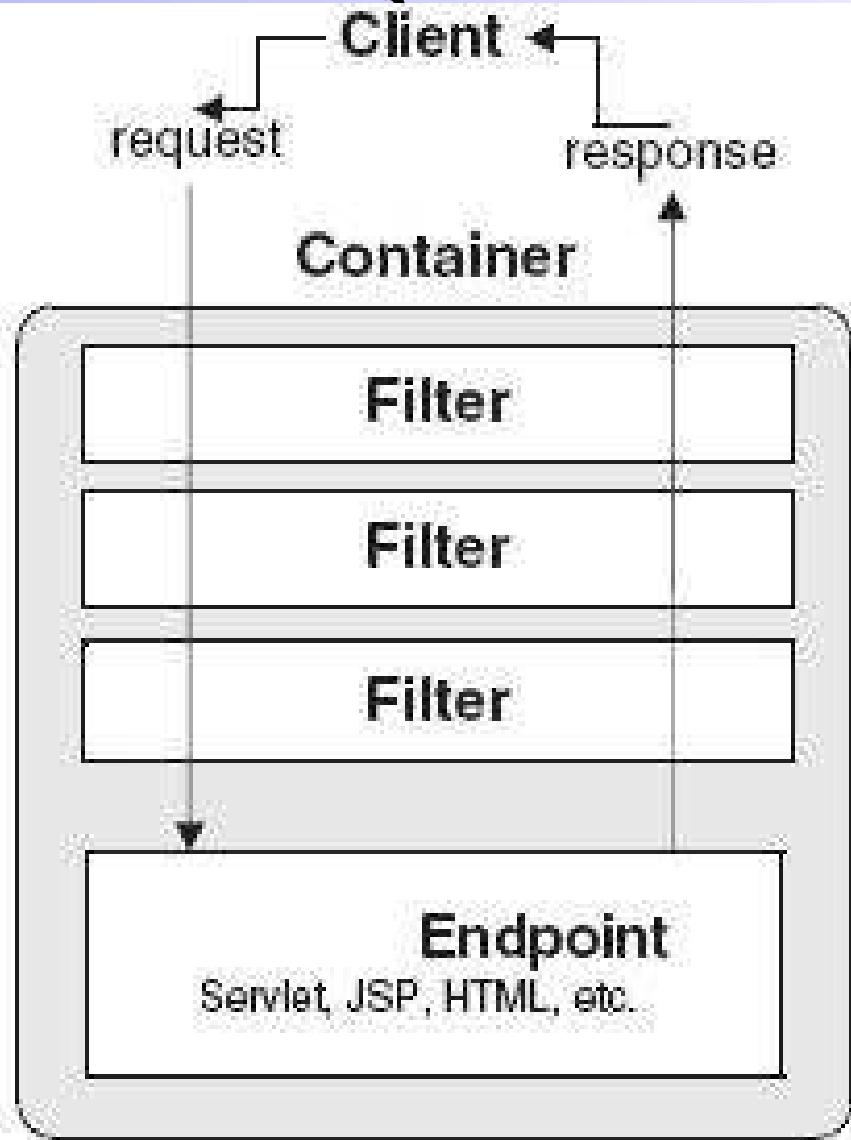
## Răspunsul:

*Client* ← *Filtrul*<sub>1</sub> ← ...*Filtrul*<sub>n</sub> ← *Componenta Destinatie*

## FilterChain:

```
public interface FilterChain() {  
    void doFilter(ServletRequest request,  
                 ServletResponse response);  
}
```

# Secvențe de filtre



# Decoratori (Wrappers)

*Decoratorii* permit modificarea dinamică a comportamentului unui obiect la momentul execuției, prin adăugarea/modificarea funcționalității obiectului original.

## Crearea unui decorator

Considerăm clasa  $C$  de tipul abstract  $I(C)$  (interfață, clasă abstractă).

- Crearea clasei decorator  $D(C)$  pentru clasa  $C$   
Clasa  $D(C)$  va fi de tipul  $I(C)$
- Crearea unui constructor în  $D(C)$  cu argument de tip  $C$  memorat în  $ref(C)$
- În clasa decorator redirectăm toate metodele din  $I(C)$  către  $ref(C)$  sau supradefinim comportamentul original

# Exemplu de decorator

## Fluxuri de Intrare / Ieşire

```
public interface Reader {
    int read();
}
public class FileReader implements Reader {
    public int read() { ... }
}
public class BufferedReader implements Reader {
    private FileReader in;
    BufferedReader(FileReader in) { this.in = in; }
    public int read() { return in.read(); }
    public String readLine() { ... }
}
```

```
Reader original = new FileReader("someFile");
Reader decorated = new BufferedReader(reader);
```

# HTTP Wrappers

## Decorarea obiectului *request*

- prin parametri
- `HttpServletResponseWrapper`
  - decorator pentru `HttpServletResponse`

## Decorarea obiectului *response*

- `HttpServletRequestWrapper`
  - decorator pentru `HttpServletRequest`

# Exemplu de Wrapper

```
public class CharResponseWrapper extends
    HttpServletResponseWrapper {

    private CharArrayWriter output;

    public CharResponseWrapper(HttpServletResponse response) {
        super(response);
        output = new CharArrayWriter();
    }
    public PrintWriter getWriter(){
        return new PrintWriter(output);
    }
    public String toString() {
        return output.toString();
    }
}
```

# Exemplu de Wrapper (cont.)

```
CharResponseWrapper wrapper = new CharResponseWrapper(  
    (HttpServletResponse)response);  
  
chain.doFilter(request, wrapper);  
  
CharArrayWriter caw = new CharArrayWriter();  
caw.write(wrapper.toString().indexOf("</body>")-1));  
caw.write("<h1>Thank you !</h1>");  
caw.write("</body></html>");  
  
PrintWriter out = response.getWriter();  
response.setContentLength(caw.toString().getBytes().length);  
out.write(caw.toString());  
out.close();
```

# Maparea filtrelor

**Servlet Filters**

**SimpleFilter** → /\*

Filter Name:

Description:

Filter Class:   [Go to Source](#)

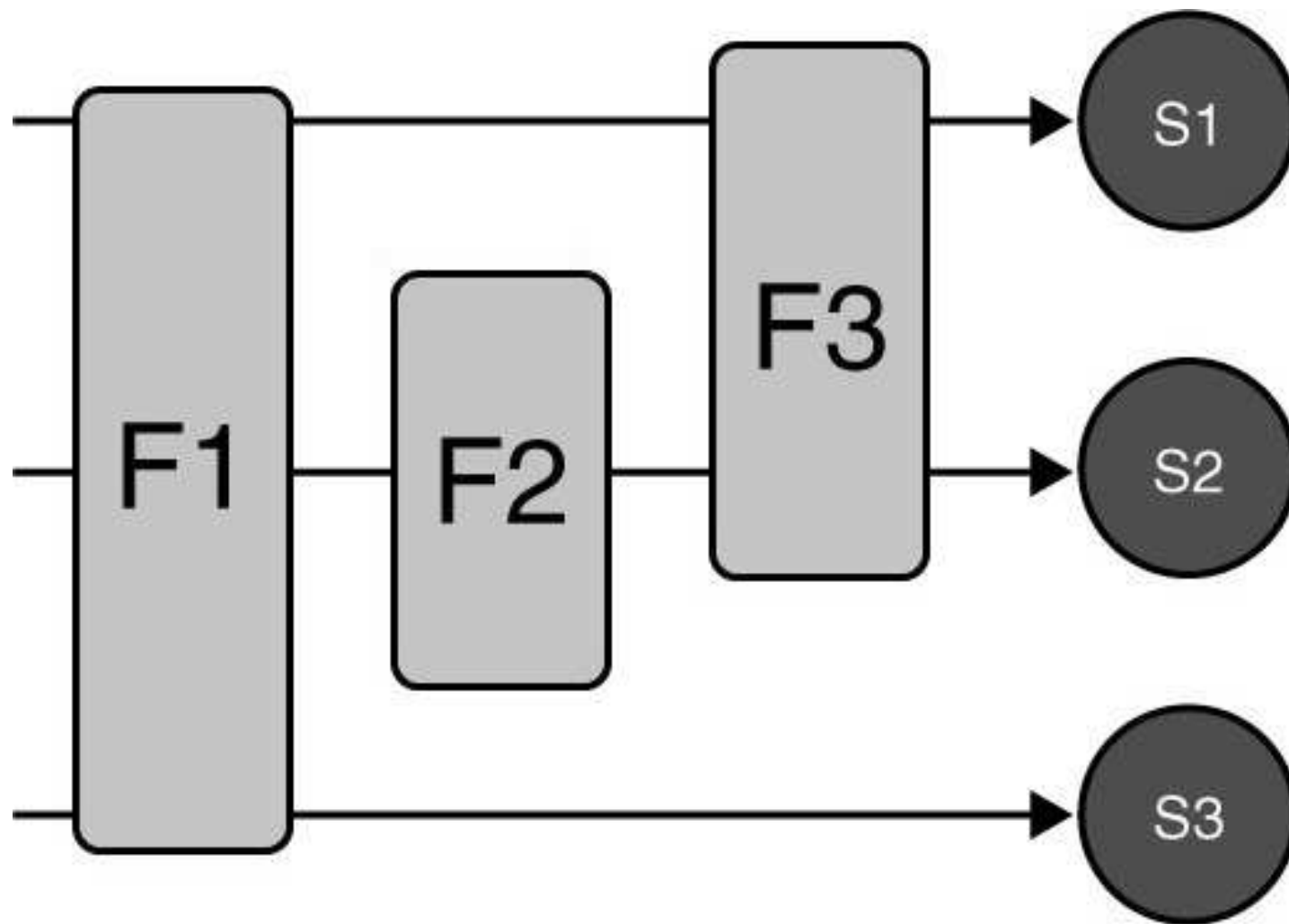
Initialization Parameters:

| Param Name | Param Value | Description |
|------------|-------------|-------------|
| max        | 100         |             |

**Filter Mappings**

| Filter Name  | Applies To | Dispatcher Types |
|--------------|------------|------------------|
| SimpleFilter | /* (URL)   |                  |

# Many-to-Many



# Concluzii



- Componente independente, slab-legate care oferă servicii adiacente celor furnizate de servleturi.
- Promovează reutilizarea codului
- Configurarea este declarativă și flexibilă
- Folosirea lor în situația în care partajează informații este inefficientă și nerecomandată.



# web.xml:*prelude-coda*

```
<jsp-config>
  <jsp-property-group>

    <url-pattern>*.jsp</url-pattern>

    <include-prelude>
      /welcome.jsp
    </include-prelude>

    <include-coda>
      /thank-you.jsp
    </include-coda>

  </jsp-property-group>
</jsp-config>
```