Java Technologies
Web Filters

The Context

- Upon receipt of a request, various processings may be needed:
  - Is the user authenticated?
  - Is there a valid session in progress?
  - Is the IP trusted, is the user's agent supported, ...?
- When sending a response, the result may require various processings:
  - Add some additional design elements.
  - Trim whitespaces, etc.

Example

In the login controller:

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

In <u>every</u> web component that requires a valid user:

```
User user = (User) session.getAttribute("user");
if (user == null) {
  response.sendRedirect("login.jsp");
  return;
}
// ok, we have a user in the session
// ...
```

$\rightarrow$ crosscutting concern

The Concept of Filters

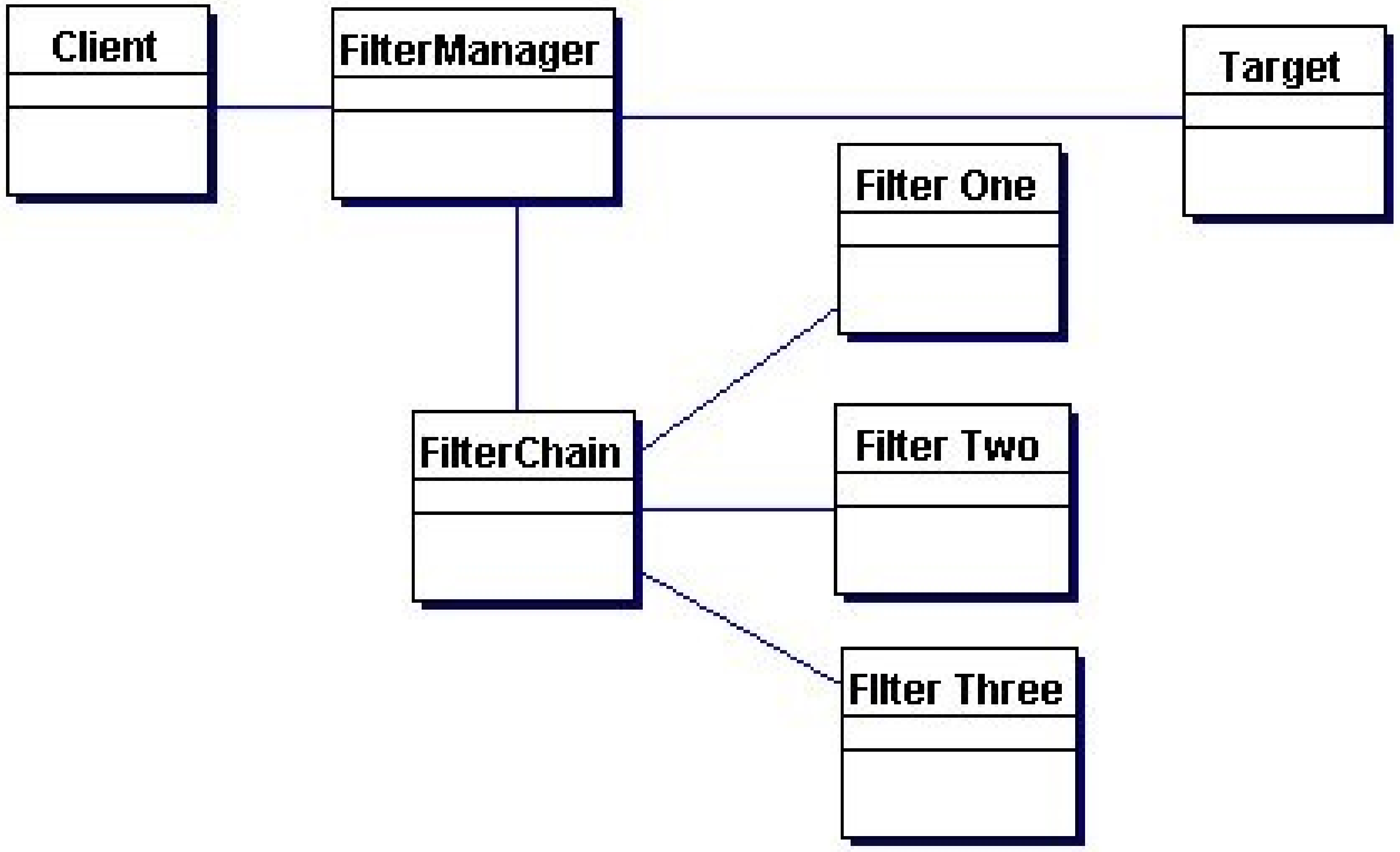
We need a component that:

- Dynamically <u>intercepts</u> requests and responses
  - preprocessing / postprocessing
- Provides <u>reusable functionalities</u> that can be "attached" to any kind of web resource
- Can be used <u>declarative</u>, in a <u>plug-in</u> manner
- Is (usually) <u>independent</u> (does not have any dependencies on other web resource for which it is acting as a filter)

Common Usages

- Authentication
- Logging and auditing
- Image conversion, scaling, etc.
- Data compression, encryption, etc.
- Localization
- Content transformations (for example, XSLT)
- Caching
- ...

Intercepting Filter Design Pattern



Java EE Filter Architecture

- An API for <u>creating</u> the filters
  - *javax.servlet.Filter* interface

- A method for <u>configuring</u> and <u>plugging-in</u> the filters (mapping them to other resources)
  - *declarative* (in web.xml or using @WebFilter)

- A mechanism for <u>chaining</u> the filters
  - *javax.servlet.FilterChain*

javax.servlet.Filter interface

```
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();
}
```

Example: Logging

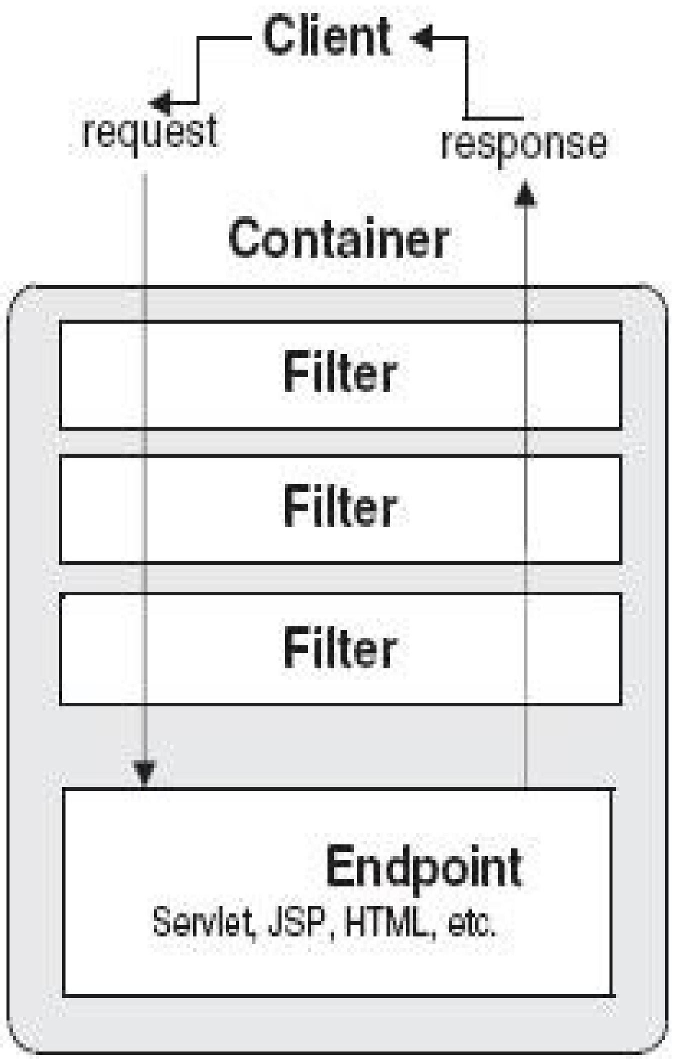
```
@WebFilter(urlPatterns = {"/*"})
public class LogFilter implements Filter {

  public void doFilter(ServletRequest req, ServletResponse res,
                       FilterChain chain)
                       throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;

    // Find the IP of the request
    String ipAddress = request.getRemoteAddr();

    // Write something in the log
    System.out.println(
       "IP: " + ipAddress + ", Time: " + new Date().toString());

    chain.doFilter(req, res);
  }
}
```

Example: Character Encoding

```
public void init(FilterConfig filterConfig) throws ServletException {
   //read the character encoding from a filter initialization parameter
   this.encoding = filterConfig.getInitParameter("encoding");
   // for example: UTF-8 or ISO 8859-16 or Windows-1250 etc.
}

public void doFilter(ServletRequest request,
                     ServletResponse response, FilterChain chain)
                     throws IOException, ServletException {
   if (encoding != null) {
     //useful if the browser does not send character encoding information
     //in the Content-Type header of an HTTP request
     request.setCharacterEncoding(encoding);
   }
   chain.doFilter(request, response);
}
```

You may want to read: “The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)” by Joel Spolsky

javax.servlet.FilterChain interface



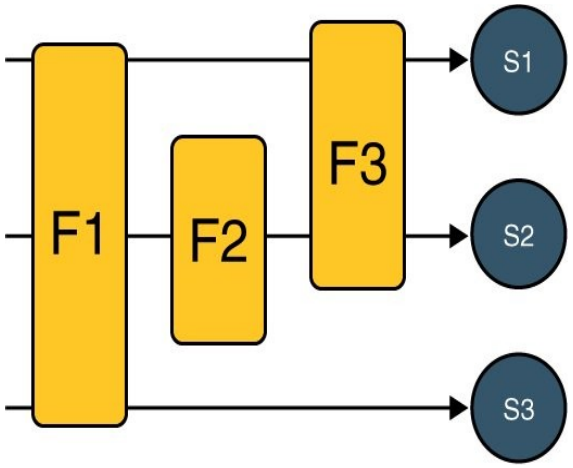
```
public interface FilterChain() {

  void doFilter(
    ServletRequest request,
    ServletResponse response);

}
```

Specifying Filter Mappings

web.xml

```
<filter>
   <filter-name>HelloFilter</filter-name>
   <filter-class>somepackage.HelloFilterImpl</filter-class>
   <init-param>
      <param-name>greeting</param-name>
      <param-value>Hello World!</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>HelloFilter</filter-name>
   <url-pattern>/hello/*</url-pattern>
</filter-mapping>
```

```
@WebFilter(
  filterName = "HelloFilter",
  urlPatterns = {"/hello/*"},
  initParams = {
   @WebInitParam(greeting = "Hello World!")}
)
public class HelloFilterImpl implements Filter {
 …
}
```



many-to-many

The generic structure of a filter

```
public class GenericFilter 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;
    }

    doAfterProcessing(request, response);
    if (problem != null) {
      processError(problem, response);
    }
  }
...
}
```

Example: Count and Measure

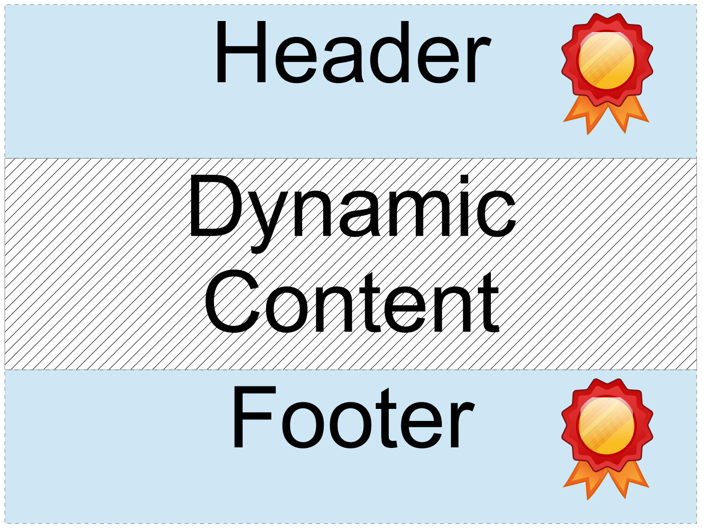
```
@WebFilter(urlPatterns = {"/someComponent"})
public class ResponeTimeFilter implements Filter {
  private AtomicInteger counter = new AtomicInteger();

  public void doFilter(ServletRequest req, ServletResponse res,
                       FilterChain chain)
                       throws IOException, ServletException {
    // Count the requests
    int n = counter.addAndGet(1);

    // Start the timer
    long t0 = System.currentTimeMillis();

    chain.doFilter(req, res);

    // Stop the timer
    long t1 = System.currentTimeMillis();

    app.log("Request " + n + " took " + (t1 - t0) + "ms");
  }
}
```

Filtering the response

The Problem:

<u>Modify the content of the response</u>

- chain.doFilter(
  request, <u>response</u>)
- response
  - getOutputStream
  - getWriter



Decorator Design Pattern

- You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.
- *Decorator Design Pattern*: Attach additional responsibilities to an object dynamically, without altering its structure (class signature).
- *Wrapper*

Decorator example: Java IO

```
public interface Reader {
  int read();
}

public class FileReader implements Reader {
  public int read() { ... }
}

public class BufferedReader implements Reader {
  private FileReader in;
  public BufferedReader(FileReader in} {
    this.in = in;            //receive the original object
  }
  public int read() {
    return in.read();        // inherit old functionality
  }
  public String readLine() { // create new functionality
  ...
  }
}
```

```
Reader original = new FileReader("someFile");

Reader decorated = new BufferedReader(reader);
```

HTTP Wrappers

- Decorating the request
  - **HttpServletRequestWrapper**
  - *implements HttpServletRequest*

```
 ServletRequestWrapper wrapper = new HttpServletRequestWrapper(req) {
      @Override
      public String getLocalName() {
          return "localhost";
      }
  };
  chain.doFilter(wrapper, response);
```

- Decorating the response
  - **HttpServletResponseWrapper**
  - *implements HttpServletResponse*

Creating a Response Wrapper

```
public class SimpleResponseWrapper
                              extends HttpServletResponseWrapper {

    private final StringWriter output;

    public SimpleResponseWrapper(HttpServletResponse response) {
        super(response);
        output = new StringWriter();
    }

    @Override
    public PrintWriter getWriter() {
        // Hide the original writer
        return new PrintWriter(output);
    }

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

Decorating the response

```
@WebFilter(filterName = "ResponseDecorator", urlPatterns = {"/*"})
public class ResponseDecorator implements Filter {

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

    SimpleResponseWrapper wrapper
            = new SimpleResponseWrapper((HttpServletResponse) response);

    //Send the decorated object as a replacement for the original response
    chain.doFilter(request, wrapper);

    //Get the dynamically generated content from the decorator
    String content = wrapper.toString();

    // Modify the content
    content += "<p> Mulțumim!";

    //Send the modified content using the original response
    PrintWriter out = response.getWriter();
    out.write(content);
  }
  ...
}
```

Conclusions

The *filter mechanism* provides a way to encapsulate common functionality in a component that can reused in many different contexts.

Filters are easy to write and configure as well as being portable and reusable.