|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Oracle® Containers for J2EE Servlet Developer's Guide** **10*g* Release 3 (10.1.3)** B14426-01 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | [IMG_256 Home](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/index.htm) | [IMG_257 Solution Area](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.htm) | [IMG_258 Contents](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/toc.htm) | [IMG_259 Index](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/index.htm) | |

|  |  |  |
| --- | --- | --- |
| [IMG_260 Previous](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/sessions.htm) | [IMG_261 Next](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/listener.htm) |  |

# **5****Understanding and Using Servlet Filters**

When the servlet container calls a method in a servlet on behalf of the client, the HTTP request that the client sent is, by default, passed directly to the servlet. The response that the servlet generates is, by default, passed directly back to the client, with its content unmodified by the container.

Alternatively, you can use servlet filters to preprocess Web application requests and postprocess server responses. Filters were introduced in ["When to Use Filters for Pre-Processing and Post-Processing"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/develop.htm" \l "BABJJBAC), and are described in the following sections:

[Overview of How Filters Work](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFDIHEJ)

[Standard Filter Interfaces](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFJCEFC)

[Implementing and Configuring Filters](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFGEJDJ)

[Simple Filter Example](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFHFIEI)

[Filtering Forward or Include Targets](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFGCJID)

[Using a Filter to Wrap and Alter the Request or Response](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFIAAAH)

[Response Filter Example](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFJHDGA)

## **Overview of How Filters Work**

This section provides an overview of the following topics:

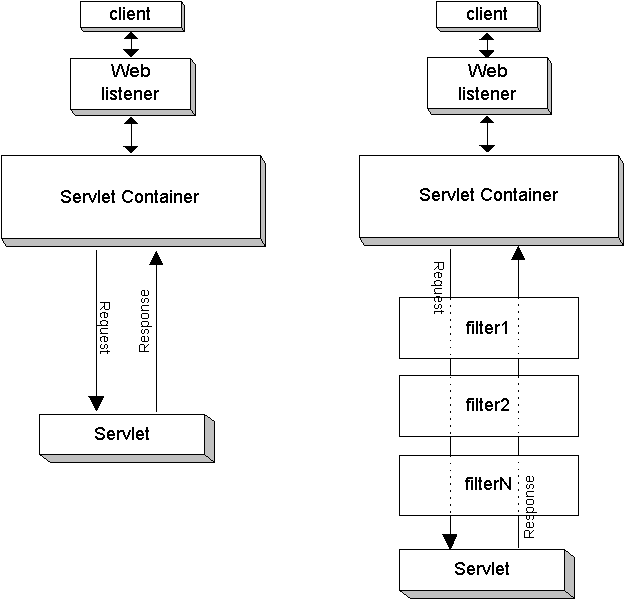
[How the Servlet Container Invokes Filters](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFGAECI)

[Typical Filter Actions](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFCIHAH)

### **How the Se****rvlet Container Invokes Filters**

[Figure 5-1](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "i1000029) shows, on the left, a scenario in which no filters are configured for the servlet being requested. On the right, several filters (1, 2, ..., N) have been configured.

***Figure 5-1 Servlet Invocation with and without Filters***

  
[Description of the illustration invfilt.gif](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/img_text/invfilt.htm)

Each filter implements the javax.servlet.Filter interface, which includes a doFilter() method that takes as input a request and response pair along with a filter chain, which is an instance of a class (provided by the servlet container) that implements the javax.servlet.FilterChain interface. The filter chain reflects the order of the filters. The servlet container, based on the configuration order in the web.xml file, constructs the chain of filters for any servlet or other resource that has filters mapped to it. For each filter in the chain, the filter chain object passed to it represents the remaining filters to be called, in order, followed by the target servlet.

The FilterChain interface also specifies a doFilter() method, which takes a request and response pair as input and is used by each filter to invoke the next entity in the chain.

Also see ["Standard Filter Interfaces"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFJCEFC).

If there are two filters, for example, the key steps of this mechanism would be as follows:

The target servlet is requested. The container detects that there are two filters and creates the filter chain.

The first filter in the chain is invoked by its doFilter() method.

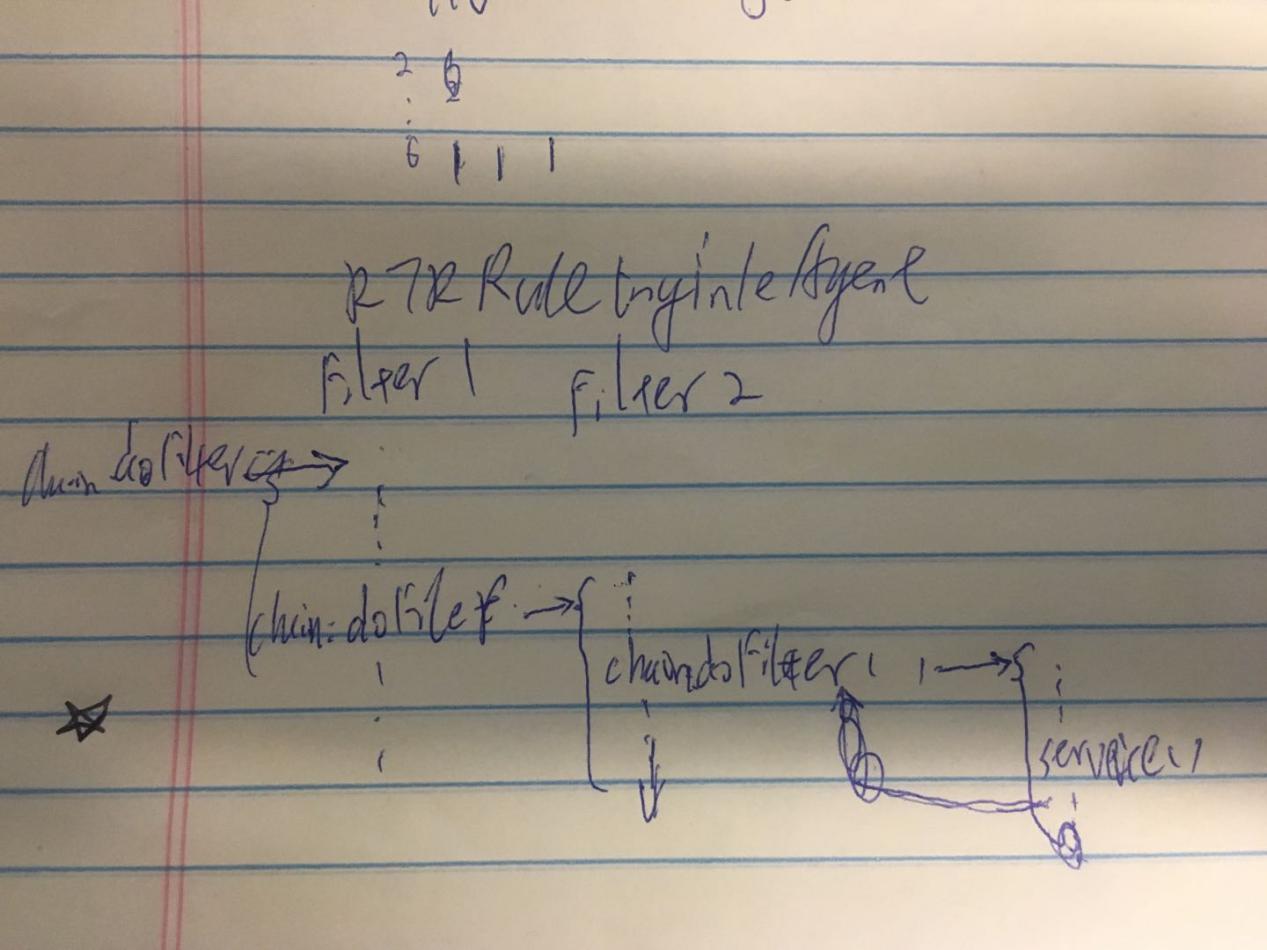
The first filter completes any preprocessing, then calls the doFilter() method of the filter chain. This results in the second filter being invoked by its doFilter() method.

The second filter completes any preprocessing, then calls the doFilter() method of the filter chain. This results in the target servlet being invoked by its service() method.

When the target servlet is finished, the chain doFilter() call in the second filter returns, and the second filter can do any postprocessing.

When the second filter is finished, the chain doFilter() call in the first filter returns, and the first filter can do any postprocessing.

When the first filter is finished, execution is complete.



None of the filters are aware of their order. Ordering is handled entirely through the filter chain, according to the order in which filters are configured in web.xml.

### **Typi****cal Filter Actions**

The following are among the possible actions of the doFilter() method of a filter:

Create a wrapper for the request object to allow input filtering. Process the content or headers of the request wrapper as desired.

Create a wrapper for the response object to allow output filtering. Process the content or headers of the response wrapper as desired.

Pass the request and response pair (or wrappers) to the next entity in the chain, using the chain doFilter() method. Alternatively, to block request processing, do *not* call the chain doFilter() method.

Any processing you want to occur before the target resource is invoked must be prior to the chain doFilter() call. Any processing you want to occur after the completion of the target resource must be after the chain doFilter() call. This can include directly setting headers on the response.

Note that if you want to preprocess the request object or postprocess the response object, you cannot directly manipulate the original request or response object. You must use wrappers. When postprocessing a response, for example, the target servlet has already completed and the response could already be committed by the time a filter would have a chance to do anything with the response. You must pass a response wrapper instead of the original response in the chain doFilter() call. See ["Using a Filter to Wrap and Alter the Request or Response"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFIAAAH).

## **Stand****ard Filter Interfaces**

A servlet filter implements the javax.servlet.Filter interface. The main method of this interface, doFilter(), takes a javax.servlet.FilterChain instance, created by the servlet container to represent the entire chain of filters, as input. The initialization method of the Filter interface, init(), takes a filter configuration object, which is an instance of javax.servlet.FilterConfig, as input. This section briefly describes the methods specified in these interfaces.

For additional information about the interfaces and methods discussed here, refer to the Sun Microsystems Javadoc for the javax.servlet package, at:

<http://java.sun.com/j2ee/1.4/docs/api/index.html>

### **Methods of the Filt****er Interface**

The Filter interface specifies the following methods to implement in your filters:

void init(FilterConfig filterConfig)

The servlet container calls init() as a filter is first instantiated and placed into service. This method takes a javax.servlet.FilterConfig instance as input, which the servlet container uses to pass information to the filter during the initialization. Include any special initialization requirements in your implementation. Also see ["Methods of the FilterConfig Interface"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFCDAAA).

void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

This is where your filter processing occurs. Each time a target resource (such as a servlet or JSP page) is requested, where the target resource is mapped to a chain of one or more filters, the servlet container calls the doFilter() method of each filter in the chain, in order according to web.xml filter configurations. (See ["Construction of the Filter Chain"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFHEFFD).) Within the doFilter() processing of a filter, invoke the doFilter() method on the filter chain object that is passed in to the doFilter() method of the filter. (An exception to this is if you want to block request processing.) This is what leads to invocation of the next entity in the chain (either the next filter, or the target servlet if this is the last filter in the chain) after a filter has completed.

destroy(): The servlet container calls destroy() after all execution of the filter has completed (all threads of the doFilter() method have completed, or a timeout has occurred) and the filter is being taken out of service. Include any special cleanup requirements in your implementation.

### **Method of the Filter****Chain Interface**

The FilterChain interface specifies one method:

void doFilter(ServletRequest request, ServletResponse response)

Invoking this method, which you do from the doFilter() method of a filter, causes the next entity in the chain to be invoked—either the next filter, or the target resource (such as a servlet or JSP page) if this method is called from the last filter in the chain.

### **Methods of the Filter****Config Interface**

The FilterConfig interface specifies the following methods, available for your optional use:

java.util.Enumeration getInitParameterNames()

You can set initialization parameters for a filter through <init-param> elements under the <filter> element in the web.xml file. (See ["Configure the Filter"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFHDEGC).) Then, in your filter, you can use the getInitParameterNames() method of the FilterConfig object, which is passed in through the init() method, to retrieve an Enumeration object of Java strings containing the names of the initialization parameters. (The Enumeration object is empty if there are no initialization parameters for the filter.)

String getInitParameter(String paramname)

After retrieving initialization parameter names, use getInitParameter() to retrieve the value of a specified parameter.

ServletContext getServletContext()

You can use this method to retrieve the servlet context associated with the requested servlet (which the filter is filtering).

String getFilterName()

You can use this method to retrieve the name of the filter, according to the <filter-name> element in the web.xml file.

## **Implementing and Configuring Filters**

This section shows the basic steps of implementing and configuring a filter. Steps such as these are included in a complete sample in ["Simple Filter Example"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFHFIEI).

There is a subsection describing construction of the filter chain, based on your filter configuration order in web.xml.

### **Imple****ment the Filter Code**

This section lists steps in implementing code for a servlet filter.

Create a class that implements the javax.servlet.Filter interface. For example:

public class TimerFilter implements javax.servlet.Filter { }

For initialization of your filter, implement the init() method, specified in the Filter interface. First, create or retrieve a javax.servlet.FilterConfig object, which init() takes as input. For example:

private FilterConfig filterConfig;

...

public void init(final FilterConfig filterConfig)

{

this.filterConfig = filterConfig;

}

In case you want any special initialization processing, see ["Methods of the FilterConfig Interface"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFCDAAA).

For your filter processing, implement the doFilter() method, specified in the Filter interface. This method takes a request object, a response object, and a javax.servlet.FilterChain object created by the servlet container. Implement whatever processing you want, and (typically) call the doFilter() method of the filter chain object to invoke the next entity in the chain. For example:

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain chain)

throws java.io.IOException, javax.servlet.ServletException

{

long start = System.currentTimeMillis();

System.out.println("Milliseconds in: " + start);

chain.doFilter(request, response);

long end = System.currentTimeMillis();

System.out.println("Milliseconds out: " + end);

}

The first println() call is executed before the rest of the chain is invoked; the second println() call is executed afterward, when chain.doFilter() returns.

Implement the destroy() method, specified in the Filter interface, to clean up resources or do anything special before the filter is taken out of service. For example:

public void destroy()

{

filterConfig = null;

}

|  |
| --- |
| Note:  There are additional considerations in implementing a filter to alter the HTTP request or response. See ["Using a Filter to Wrap and Alter the Request or Response"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFIAAAH). |

### **Config****ure the Filter**

This section lists the steps in configuring a servlet filter. Do the following in web.xml for each filter:

Declare the filter through a <filter> element and its subelements, which maps the filter class (including package) to a filter name. For example:

<filter>

<filter-name>timer</filter-name>

<filter-class>filter.TimerFilter</filter-class>

</filter>

You can optionally specify initialization parameters here, similarly to how you would for a servlet:

<filter>

<filter-name>timer</filter-name>

<filter-class>filter.TimerFilter</filter-class>

<init-param>

<param-name>*name*</param-name>

<param-value>*value*</param-value>

<init-param>

</filter>

Using a <filter-mapping> element and its subelements, map the filter name to a servlet name or URL pattern to associate the filter with the corresponding resource (such as a servlet or JSP page) or resources. For example, to have the filter invoked whenever the servlet of name myservlet is invoked:

<filter-mapping>

<filter-name>timer</filter-name>

<servlet-name>myservlet</servlet-name>

</filter-mapping>

Or, to have the filter invoked whenever sleepy.jsp is requested, according to URL pattern:

<filter-mapping>

<filter-name>timer</filter-name>

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

</filter-mapping>

Note that instead of specifying a particular resource in the <url-pattern> element, you can use wild card characters to match multiple resources, such as in the following example:

<url-pattern>/mypath/\*</url-pattern>

The filter name can be arbitrary, but preferably is meaningful. It is simply used as the linkage in mapping a filter class to a servlet name or URL pattern.

If you configure multiple filters that apply to a resource, they will be entered in the servlet chain according to their declaration order in web.xml, and they will be invoked in that order when the target servlet is requested. See the next section, ["Construction of the Filter Chain"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFHEFFD).

|  |
| --- |
| Note:  There are additional steps to configure a filter for a forward or include target. See ["Filtering Forward or Include Targets"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFGCJID). |

### **Constru****ction of the Filter Chain**

When you declare and map filters in web.xml, the servlet container determines which filters apply to each servlet or other resource (such as a JSP page or static page) in the Web application. Then, for each servlet or resource, the servlet container builds a chain of applicable filters, according to your web.xml configuration order, as follows:

First, any filters that match a servlet or resource according to a <url-pattern> element are placed in the chain, in the order in which the filters are declared in web.xml.

Next, any filters that match a servlet or resource according to a <servlet-name> element are placed in the chain, with the first <servlet-name> match following the last <url-pattern> match.

Finally, the target servlet or other resource is placed at the end of the chain, following the last filter with a <servlet-name> match.

## **Simple Fil****ter Example**

This example shows a filter that is invoked when a JSP page is requested. The JSP page writes a line to the browser. The filter writes two lines to the OC4J console—one line before the JSP page runs, and one after.

### **Write the Simple Filter Code**

Here is the code for the simple filter, TimerFilter. The doFilter() method writes two lines to the OC4J console, one before the target JSP is executed and one after.

package filter;

import javax.servlet.\*;

public class TimerFilter implements javax.servlet.Filter

{

private FilterConfig filterConfig;

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain chain)

throws java.io.IOException, javax.servlet.ServletException

{

long start = System.currentTimeMillis();

System.out.println("Milliseconds in: " + start);

chain.doFilter(request, response);

long end = System.currentTimeMillis();

System.out.println("Milliseconds out: " + end);

}

public void init(final FilterConfig filterConfig)

{

this.filterConfig = filterConfig;

}

public void destroy()

{

filterConfig = null;

}

}

### **Write the Target JSP Page**

Here is the target JSP page, sleepy.jsp, which has a wait interval then outputs the wait time to the browser.

<%

int sleeptime = 1000;

Thread.sleep(sleeptime);

%>

<HTML>

<HEAD>

<TITLE>Sleepy JSP</TITLE>

</HEAD>

<BODY>

<HR>

<P><CENTER>Sleepy JSP slept for <%= sleeptime %> milliseconds!</CENTER></P>

<HR>

</BODY>

</HTML>

### **Configure the Simple Filter**

Here is the configuration in web.xml that declares the simple filter and maps it to requests for sleepy.jsp:

<?xml version="1.0" ?>

<!DOCTYPE web-app (*doctype*...)>

<web-app>

<filter>

<filter-name>timer</filter-name>

<filter-class>filter.TimerFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>timer</filter-name>

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

</filter-mapping>

</web-app>

### **Package the Simple Filter Example**

The WAR file for this example, which we name filter.war, has the following contents and structure:

sleepy.jsp

META-INF/Manifest.mf

WEB-INF/web.xml

WEB-INF/classes/filter/TimerFilter.class

WEB-INF/classes/filter/TimerFilter.java

And the EAR file is as follows:

filter.war

META-INF/Manifest.mf

META-INF/application.xml

(The Manifest.mf files are created automatically by the JAR utility.)

### **Invoke the Simple Filter Example**

For this example, assume that application.xml maps the context path /myfilter to filter.war. In this case, after deployment, you invoke sleepy.jsp as follows, and the filter is executed as a result:

http://*host*:*port*/myfilter/sleepy.jsp

The following is written to the browser:

Sleepy JSP slept for 1000 milliseconds!

In a sample execution, the following is written to the OC4J console (where you started OC4J, for example, if you are running OC4J in a standalone environment):

04/04/30 13:01:19 Milliseconds in: 1083355279136

04/04/30 13:01:20 Milliseconds out: 1083355280152

The "Milliseconds in" line is written before the JSP page is invoked. The "Milliseconds out" line is written after the JSP page is done and execution returns to the filter. In this example, there is a difference of 1016 milliseconds, mostly due to the 1000-millisecond wait in the JSP page.

## **Filte****ring Forward or Include Targets**

You can configure a filter to act on forward or include targets in addition to, or instead of, acting on direct request targets. This is described in the following subsections:

[The web.xml <dispatcher> Element](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFIEDGB)

[Configuring Filters for Forward or Include Targets](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFIHAEA)

|  |
| --- |
| Note:  See ["Dispatching to Other Servlets Through Includes and Forwards"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/develop.htm" \l "BABEIFJE) for general information about including and forwarding. |

### **The web.xml <dispatcher> Element**

Use the <dispatcher> subelement of <filter-mapping> in web.xml if you want to configure filters for forward or include targets. This element has four supported values:

INCLUDE: Use this for the filter to be applied to any include targets matching a specified servlet name or with URLs matching a specified pattern.

FORWARD: Use this for the filter to be applied to any forward targets matching a specified servlet name or with URLs matching a specified pattern.

REQUEST: Use this in addition to an INCLUDE or FORWARD setting (one <dispatcher> element for each setting) for the filter to also be applied to direct request targets matching a specified servlet name or with URLs matching a specified pattern. (It is nonsensical to use the REQUEST value only. If you want the filter to apply only to direct requests, there is no need to use the <dispatcher> element.)

ERROR: Use this for the filter to be applied under the error page mechanism.

See the following section, ["Configuring Filters for Forward or Include Targets"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFIHAEA), for examples.

### **Configuring Filters for Forward or Include Targets**

This section provides a few sample configurations to have a filter act on forward or include targets. We start with the filter declaration, followed by alternative filter mapping configurations:

<filter>

<filter-name>myfilter</filter-name>

<filter-class>mypackage.MyFilter</filter-class>

</filter>

To execute MyFilter to filter an include target named includedservlet:

<filter-mapping>

<filter-name>myfilter</filter-name>

<servlet-name>includedservlet</servlet-name>

<dispatcher>INCLUDE</dispatcher>

</filter-mapping>

Note that the include() call can come from any servlet (or other resource) in the application. Also note that MyFilter would not execute for a direct request of includedservlet, unless you have another <dispatcher> element with the value REQUEST.

To execute MyFilter to filter any servlet directly requested through a URL pattern "/mypath/", or to execute it to filter any forward target that is invoked through a URL pattern starting with "/mypath/":

<filter-mapping>

<filter-name>myfilter</filter-name>

<url-pattern>/mypath/\*</url-pattern>

<dispatcher>FORWARD</dispatcher>

<dispatcher>REQUEST</dispatcher>

</filter-mapping>

## **Using a Filt****er to Wrap and Alter the Request or Response**

Particularly useful functions for a filter are to manipulate a request, or manipulate the response to a request. To manipulate a request or response, you must create a wrapper. You can use the following general steps:

To manipulate requests, create a class that extends the standard javax.servlet.http.HttpServletRequestWrapper class. This class will be your request wrapper, allowing you to modify a request as desired.

To manipulate responses, create a class that extends the standard javax.servlet.http.HttpServletResponseWrapper class. This class will be your response wrapper, allowing you to modify a response after the target servlet or other resource has delivered and possibly committed it.

Optionally create a class that extends the standard javax.servlet.ServletOutputStream class, if you want to add custom functionality to an output stream for the response.

Create a filter that uses instances of your custom classes to alter the request or response as desired.

The next section, ["Response Filter Example"](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFJHDGA), provides an example of a filter that alters the response.

## **Resp****onse Filter Example**

This example employs an HTTP servlet response wrapper that uses a custom servlet output stream. This functionality allows the wrapper to manipulate the response data after the target HTML page is finished writing it out. Without using a wrapper, you cannot change the response data after the servlet output stream has been closed (essentially, after the servlet has committed the response). That is the reason for implementing a filter-specific extension to the ServletOutputStream class in this example.

This example uses the following custom classes:

GenericResponseWrapper: Extends HttpServletResponseWrapper for custom functionality in manipulating an HTTP response.

FilterServletOutputStream: Extends ServletOutputStream to provide custom functionality for use in the response wrapper.

MyGenericFilter: This class is for a generic, empty ("pass-through") filter that is used as a base class.

PrePostFilter: Extends MyGenericFilter and implements doFilter() code to alter the HTTP response, inserting a line before the HTML page output and a line after the HTML page output.

### **Write the Custom Output Stream Code**

This class, FilterServletOutputStream, extends the standard ServletOutputStream class to implement special functionality that the response wrapper will use.

package mypkg;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

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);

}

}

### **Write the Response Wrapper Code**

This class, GenericResponseWrapper, extends the standard HttpServletResponseWrapper class to implement custom functionality to manipulate an HTTP response.

package mypkg;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class GenericResponseWrapper extends HttpServletResponseWrapper {

private ByteArrayOutputStream output;

private int contentLength;

private String contentType;

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);

}

public void setContentLength(int length) {

this.contentLength = length;

super.setContentLength(length);

}

public int getContentLength() {

return contentLength;

}

public void setContentType(String type) {

this.contentType = type;

super.setContentType(type);

}

public String getContentType() {

return contentType;

}

}

### **Write the Base Filter Code**

This class, MyGenericFilter, is an empty filter, providing a template that is extended by the response filter of this example.

package mypkg;

import javax.servlet.\*;

public class MyGenericFilter implements javax.servlet.Filter {

public FilterConfig filterConfig;

public void doFilter(final ServletRequest request,

final ServletResponse response,

FilterChain chain)

throws java.io.IOException, javax.servlet.ServletException {

chain.doFilter(request,response);

}

public void init(final FilterConfig filterConfig) {

this.filterConfig = filterConfig;

}

public void destroy() {

}

}

### **Write the Response Filter Code**

This class, PrePostFilter, which extends MyGenericFilter, is a filter that alters the response of the target HTML page, prepending a line of output before the HTML page output, and appending a line of output after the HTML page output.

package mypkg;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class PrePostFilter extends MyGenericFilter {

public void doFilter(final ServletRequest request,

final ServletResponse response,

FilterChain chain)

throws IOException, ServletException {

OutputStream out = response.getOutputStream();

out.write(new String("<HR>PRE<HR>").getBytes());

GenericResponseWrapper wrapper =

new GenericResponseWrapper((HttpServletResponse) response);

chain.doFilter(request,wrapper);

out.write(wrapper.getData());

out.write(new String("<HR>POST<HR>").getBytes());

out.close();

}

}

### **Write the Target HTML Page**

This HTML page, prepostfilter.html, is requested by the user in this example. The filter inserts output before and after the output of this page.

<HTML>

<HEAD>

<TITLE>PrePostFilter Example</TITLE>

</HEAD>

<BODY>

This is a testpage. You should see<br>

this text when you invoke prepostfilter.html, <br>

as well as the additional material added<br>

by the PrePostFilter class.

<br>

</BODY>

</HTML>

### **Configure the Response Filter**

Here is the configuration in web.xml that declares the response filter and maps it to requests for prepostfilter.html:

<?xml version="1.0" ?>

<!DOCTYPE web-app (*doctype*...)>

<web-app>

<filter>

<filter-name>prepost</filter-name>

<filter-class>mypkg.PrePostFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>prepost</filter-name>

<url-pattern>/prepostfilter.html</url-pattern>

</filter-mapping>

</web-app>

### **Package the Response Filter Example**

The WAR file for this example, which we name myresponsewrapper.war, has the following contents and structure:

prepostfilter.html

META-INF/Manifest.mf

WEB-INF/web.xml

WEB-INF/classes/mypkg/FilterServletOutputStream.class

WEB-INF/classes/mypkg/FilterServletOutputStream.java

WEB-INF/classes/mypkg/GenericResponseWrapper.class

WEB-INF/classes/mypkg/GenericResponseWrapper.java

WEB-INF/classes/mypkg/MyGenericFilter.class

WEB-INF/classes/mypkg/MyGenericFilter.java

WEB-INF/classes/mypkg/PrePostFilter.class

WEB-INF/classes/mypkg/PrePostFilter.java

And the EAR file is as follows:

myresponsewrapper.war

META-INF/application.xml

META-INF/Manifest.mf

(The Manifest.mf files are created automatically by the JAR utility.)

### **Invoke the Response Filter Example**

For this example, assume that application.xml maps the context path /mywrapper to myresponsewrapper.war. In this case, after deployment, you invoke prepostfilter.html as follows, and the filter is executed as a result:

http://*host*:*port*/mywrapper/prepostfilter.html

The following is output to the browser, with "PRE", "POST", and the horizontal rules coming from the filter:

[http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221\_03/web.1013/b14426/filters.htm#BCFGCJID](http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/web.1013/b14426/filters.htm" \l "BCFGCJID)