Foundations of JSP Design Patterns

ANDREW PATZER

EDITED BY MATTHEW MOODIE

Apress®

Foundations of JSP Design Patterns

ISBN (pbk): 1-59059-411-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: James L. Weaver
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Production Manager: Kari Brooks
Production Editor: Janet Vail
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Greg Teague
Indexer: Brenda Miller
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, LLC, 233 Spring Street, Sixth Floor, New York, NY 10013 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 6

The Decorating Filter Pattern

The first pattern I'll discuss in this book enables you to manipulate the HTTP request and response objects both before and after they're processed by either a servlet or a JSP page. This gives you a great amount of flexibility to customize a base application without requiring modifications to the application code itself. For instance, let's say you have two different applications, each with their own security model. You'd like to avoid making users log into both systems and maintain multiple login IDs. Although this problem has many complicated solutions, filters provide a way to intercept each request and perform the necessary security negotiation between the different systems (see Figure 6-1).

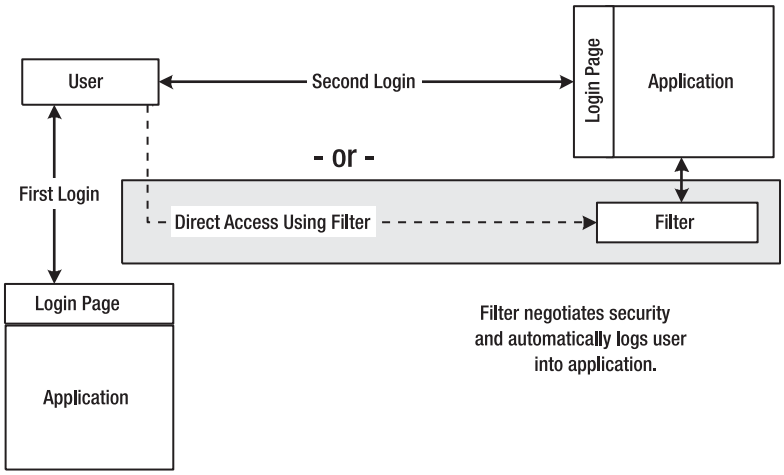

Figure 6-1. *Using filters for integrated security*

This chapter will show you how to create filters and some useful ways to apply them to your JSP and servlet development projects. I'll start each of the pattern chapters with a general definition of the pattern itself. For more detailed pattern definitions, please refer to the J2EE Patterns Catalog at the Sun Java Center (`http://java.sun.com/developer/technicalArticles/J2EE/patterns/`). Next, I'll discuss various strategies for applying the pattern. Finally, you'll apply these strategies to some common functionality you'd like to build into a web application.

Defining the Pattern

The Decorating Filter pattern (also referred to as the Intercepting Filter pattern) involves setting up a series of *filters* that intercept an HTTP request (or an HTTP response), perform a series of operations on the object, and then pass it along to its intended target. This allows for preprocessing and postprocessing of the request and response objects without affecting the core application. Here are some potential uses for filters:

- Logging important information about each request
- Authenticating users
- Transforming input data prior to processing
- Transforming response data for a specific device
- Validating form data and interrupt processing if necessary

The general idea behind using filters is that the request passes through a filter manager. This filter manager associates the request with a specific filter chain and passes the request to the first filter in the chain. Each filter performs its processing and passes the request onto the next filter in the chain. This continues until the last filter in the chain is finished processing the request (see Figure 6-2). The *filtered* request is then forwarded onto its intended target.

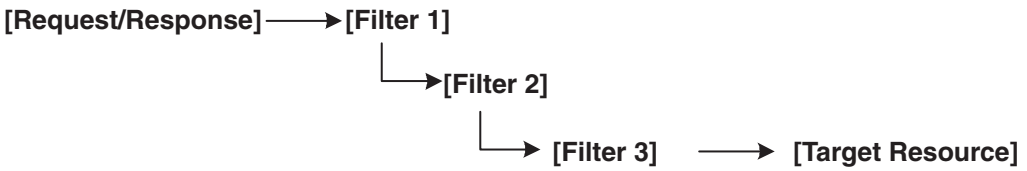

Figure 6-2. *Using filters to preprocess a request and postprocess a response*

Applying Strategies

You have two main strategies for applying the Decorating Filter pattern. The first strategy is to develop a custom filter manager and individual filters. The second strategy is to use the filtering capabilities built into the Java Servlet 2.4 API. It's pretty obvious that you should use standard filtering whenever possible, but sometimes a custom filtering mechanism is necessary. For instance, your application may need to run in an environment that doesn't support the Servlet 2.4 API.

Developing a Custom Filter Strategy

To implement your own filtering mechanism, you could wrap the core request-processing logic in any number of custom filters. These filters would each execute in turn before finally executing the core processing code. Once this filter chain has been completed, the servlet controller would then dispatch the request to the appropriate view. Custom filters are simply

Java classes that make up a sort of linked list. Each filter executes the processing logic of the next filter until each filter in the chain has been executed (see Figure 6-3).

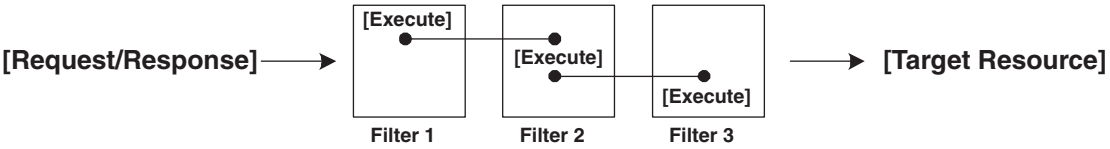

Figure 6-3. *Custom filter strategy*

The problem with using a custom filter strategy is that it must be hard-coded into the servlet controller. This means you must make any changes to the filter chain inside the servlet and then recompile it. Another disadvantage to this approach is that you can't modify the request and response objects within the filters because they'll be dispatched upon completion. Although it may sometimes be necessary to build custom filters, the preferred approach is to use a standard filter strategy, as discussed in the following section.

Using a Standard Filter Strategy

The Servlet 2.4 API allows you to create standard filters and then declare and associate them in `web.xml`. This filter strategy invokes a filter manager with each request, which in turn invokes each of the filters associated with the request URL pattern (see Figure 6-4). Each filter is a "black box" capable of receiving standard request and response objects, processing them, and returning control to the filter manager. The filter manager then invokes the next filter in the chain. This continues until the end of the filter chain is reached and the object is passed onto its intended target.

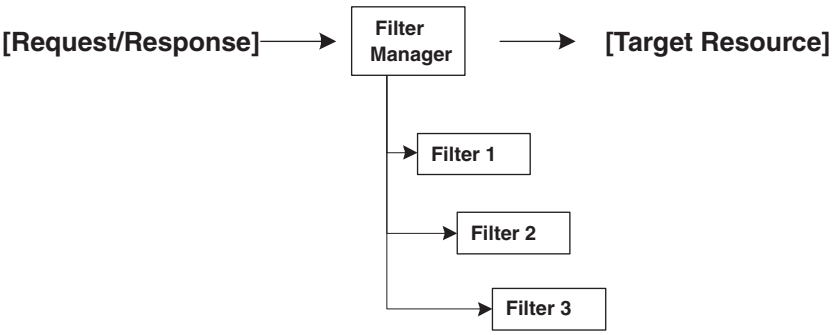

Figure 6-4. *Standard filter strategy*

A key advantage to the standard filter mechanism is that you can add and remove modifiers to the HTTP request and response without affecting the application code. You can modify the request object as well as the output stream because this processing happens outside the

servlet controller. Next, you'll see exactly how to implement a standard filter chain using the filtering mechanism in the 2.4 version of the Servlet API.

Applying the Decorator Filter Pattern

The Servlet API includes a standard mechanism for applying the standard filter strategy. To use filters with your application, simply create the filter classes, declare them in your `web.xml` file, and map them to a specific URL or URL pattern (see Figure 6-5).

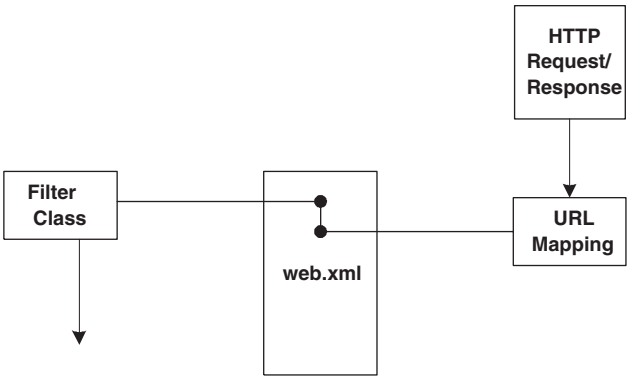

Figure 6-5. *Filtering with J2EE*

Creating the Filter Class

All filters are Java classes that implement the `javax.servlet.Filter` interface. This interface defines the following three methods that must be implemented:

void init(FilterConfig) throws ServletException: Called when the filter is first put into service. The `FilterConfig` must be copied locally within the filter.

void doFilter(ServletRequest, ServletResponse, FilterChain) throws IOException, ServletException: This method is the equivalent of `doPost` or `doGet` in a servlet. Whenever the filter is executed, the `doFilter` method is called. Upon completion of any filter-specific processing, a call to the `doFilter` method of the `FilterChain` object will continue processing any remaining filters.

void destroy(): Called just before the filter is taken out of service. This is where any cleanup tasks would be performed.

It's important to point out that a Java filter operates on `Servlet` objects and not `HTTPServlet` objects. Remember that, although uncommon, a servlet can exist in a non-HTTP environment. With that in mind, to use the request or response objects, they must be cast to their HTTP equivalents, like so:

```
if (request instanceOf HttpServletRequest) {
   HttpServletRequest httpReq = (HttpServletRequest) request;
}
```

Listing 6-1 shows a basic template of a Java filter object.

Listing 6-1. TestFilter.java

```
package jspbook.ch06.filters;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterConfig;
import javax.servlet.FilterChain;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

public class TestFilter implements Filter {

  private FilterConfig config =  null;

  public void init(FilterConfig _config)
    throws ServletException
  {
    this.config = _config;
    // Perform initialization tasks here
  }

  public void doFilter(ServletRequest _req, ServletResponse _res,
    FilterChain _chain) throws IOException, ServletException
  {
    HttpServletRequest httpReq;

    // Cast request object to HttpServletRequest
    if (_req instanceof HttpServletRequest) {
      httpReq = (HttpServletRequest) _req;
    }

    // Perform filter-specific processing here

    // Continue with filter chain
    _chain.doFilter(_req, _res);
  }
```

```
  public void destroy()
  {
    config = null;

    // Perform any cleanup tasks here
  }
}
```

Declaring the Filter

The first task you need to perform to add a filter to your web application is to declare it inside `web.xml`. Inside the `<web-app>` tag, add a `<filter>` tag to describe your filter, before any servlet declarations. This will associate a common name with the filter class you've written. You'll use this name later to map the filter to a specific URL pattern. Inside the `<filter>` tag, add a `<filter-name>` and a `<filter-class>` element, like this:

```
<filter>
  <filter-name>testFilter</filter-name>
  <filter-class>jspbook.ch06.filters.TestFilter</filter-class>
</filter>
```

Mapping the Filter to a URL

Now that you have a filter declared, the next step is to associate the filter with a specific resource. You do this by adding a `<filter-mapping>` tag inside the `<web-app>` tag in `web.xml`. You can map a filter to either a URL pattern or a specific servlet. Here's an example of each type of mapping:

```
<filter-mapping>
  <filter-name>testFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>testFilter</filter-name>
  <servlet-name>MainServlet</servlet-name>
</filter-mapping>
```

Using Filters to Log HTTP Requests

Many times I've needed to examine the HTTP request and response objects as they pass back and forth with each request. For instance, while I was debugging a problem that the browser was having when rendering a dynamically generated Portable Document Format (PDF) document, I needed to obtain an HTTP *sniffer* to examine the contents and headers of the response object. Although this solution worked, it required locating the software, setting it up, and modifying my application to use this sniffer as a proxy server. It would've been much easier if I had been able to create a simple filter that dumped the contents of the HTTP response without having to install and use a third-party application to do so.

You're going to build a simple filter that logs the HTTP request parameters before they reach their intended target. You'll extend this to display the contents of the HTTP response as well. Listing 6-2 shows what the output of this filter may look like after you log into an application and then submit some form data (from the example in Chapter 5).

Listing 6-2. *The Output of a Request-Interception Filter*

```
HTTP Request: Wed Oct 24 11:00:20 CDT 2001:

Remote Address: 127.0.0.1
Remote Host: 127.0.0.1

UID = apatzer
PWD = apress
Submit = Login
action = login

HTTP Request: Wed Oct 24 11:00:32 CDT 2001:

Remote Address: 127.0.0.1
Remote Host: 127.0.0.1

age = 30
lname = Patzer
children = 2
married = Y
sex = M
action = submit
smoker = N
fname = Andrew
```

To implement this filter, you need to declare it and then map it to a URL pattern. In this case, just map it to every request for your particular web application (`jspBook`). Listing 6-3 shows the updated `web.xml` file that describes and maps the `RequestLoggingFilter`.

Listing 6-3. `web.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" version="2.4">

 <!-- Define filters -->
  <filter>
    <filter-name>requestLoggingFilter</filter-name>
    <filter-class>jspbook.ch06.filters.RequestLoggingFilter</filter-class>
  </filter>
```

```
  <filter-mapping>
    <filter-name>requestLoggingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <!-- Servlet mappings -->
  <servlet>
    <servlet-name>
      Main
    </servlet-name>
    <servlet-class>
      jspbook.ch05.Main
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>
      Main
    </servlet-name>
    <url-pattern>
      /ch05/Main
    </url-pattern>
  </servlet-mapping>
...
```

To write a filter class, you need to implement the `javax.servlet.Filter` interface. You then need to implement the `init`, `destroy`, and `doFilter` methods. This filter will first open a log file with a unique time stamp in the filename. You do this in the `init` method so the file can remain open as long as the filter remains in service. You make the file accessible to the rest of the filter by declaring the `FileOutputStream` at the class level, rather than inside the `init` method.

```
private FileOutputStream fos;

private final String LOGGING_FILE = "C:\\jakarta-tomcat-5.0.16\\logs\\tomcat_";

public void init(FilterConfig _config)
  throws ServletException
{
  this.config = _config;
  try {
    /* Timestamp log file */
    File file = new File(LOGGING_FILE
      + new Date().getTime() + ".log");
    fos = new FileOutputStream(file);
  }
  catch (FileNotFoundException e) {
    System.out.println("Error opening log file.");
    System.out.println(e.toString());
  }
}
```

This file is eventually closed in the `destroy` method by closing the `FileOutputStream` object. Inside the `doFilter` method, which executes on each request, the `ServletRequest` object is cast to an `HttpServletRequest` object and passed to a custom method to assemble a log entry containing the HTTP parameters included inside the request. After the log file is written to, the `doFilter` method of the `FilterChain` is executed to continue processing any remaining filters.

```
public void doFilter(ServletRequest _req, ServletResponse _res,
  FilterChain _chain) throws IOException, ServletException
{
  /* Log HTTP form parameters */
  if (_req instanceof HttpServletRequest) {
    String log = getParams((HttpServletRequest)_req);
    fos.write(log.getBytes());
  }

  /* Continue with filter chain */
  _chain.doFilter(_req, _res);
}
```

For a complete listing of the `RequestLoggingFilter` class, see Listing 6-4.

Listing 6-4. `RequestLoggingFilter.java`

```
package jspbook.ch06.filters;

import java.io.File;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.util.Date;
import java.util.Enumeration;

import javax.servlet.Filter;
import javax.servlet.FilterConfig;
import javax.servlet.FilterChain;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

public class RequestLoggingFilter implements Filter {

  private FilterConfig config =  null;
  private FileOutputStream fos;

  private final String LOGGING_FILE = "C:\\tomcat_";
```

```
public void init(FilterConfig _config)
  throws ServletException
{
  this.config = _config;
  try {
    /* Timestamp log file */
    File file = new File(LOGGING_FILE
      + new Date().getTime() + ".log");
    fos = new FileOutputStream(file);
  }
  catch (FileNotFoundException e) {
    System.out.println("Error opening log file.");
    System.out.println(e.toString());
  }
}

public void doFilter(ServletRequest _req, ServletResponse _res,
  FilterChain _chain) throws IOException, ServletException
{
  /* Log HTTP form parameters */
  if (_req instanceof HttpServletRequest) {
    String log = getParams((HttpServletRequest)_req);
    fos.write(log.getBytes());
  }

  /* Continue with filter chain */
  _chain.doFilter(_req, _res);
}

public void destroy()
{
  config = null;
  try {
    fos.close();
  }
  catch (IOException e) {
    System.out.println("Error closing log file.");
    System.out.println(e.toString());
  }
}

private String getParams(HttpServletRequest _req)
  throws ServletException
{
  StringBuffer log = new StringBuffer();

  /* Get Http Parms */
  log.append("HTTP Request: ");
  log.append(new Date());
  log.append(":\r\n\r\n");
```

```
    log.append("Remote Address: " + _req.getRemoteAddr() + "\r\n");
    log.append("Remote Host: " + _req.getRemoteHost() + "\r\n\r\n");

    Enumeration paramNames = _req.getParameterNames();
    while (paramNames.hasMoreElements()) {
      String key = (String)paramNames.nextElement();
      String[] values = _req.getParameterValues(key);
      log.append(key + " = ");
      for(int i = 0; i < values.length; i++) {
          log.append(values[i] + " ");
      }
      log.append("\r\n");
    }
    return log.toString();
  }
}
```

Using Filters to Log HTTP Responses

Filters can also be useful for manipulating content after it has been generated, just before it has been rendered by the browser. To see this in action, you'll modify the previous example to write the response content, rather than the request parameters, to your log file. To manipulate the HTTP response, the response object is first wrapped inside a custom wrapper object, the `doFilter` method is called with the wrapped response rather than the original response, and finally the response content is obtained and manipulated as necessary (see Figure 6-6).

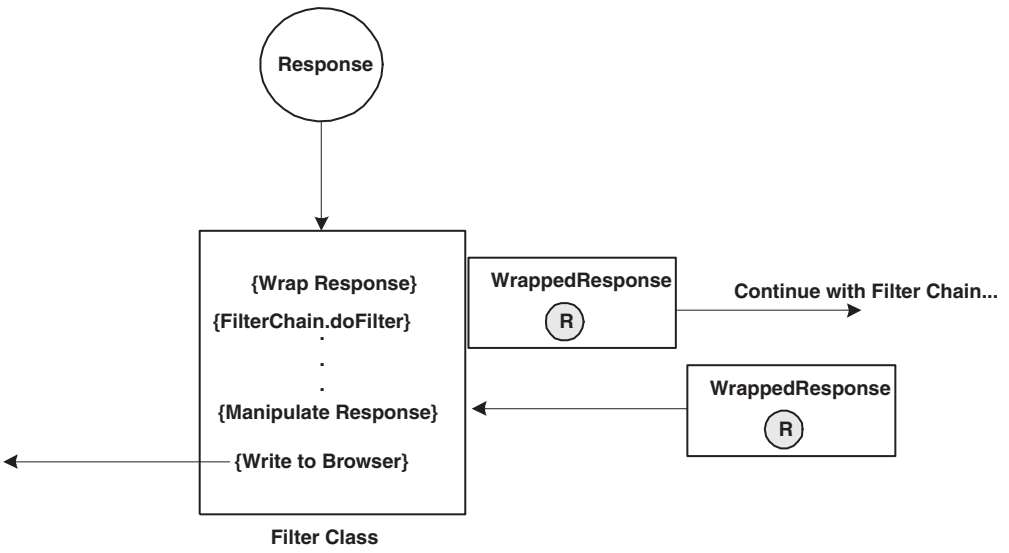

Figure 6-6. *Manipulating response content*

The reason you wrap the response object is that you need to store the response output and be able to reference it locally within your filter. The Servlet API gives you a standard wrapper to extend called `HttpServletResponseWrapper`. To use it, you extend it and override the `getWriter` method to return references to local objects within the filter (see Listing 6-4). Listing 6-5 shows the internal wrapper class and how to wrap the response inside the `doFilter` method.

Listing 6-5. *The* `WrappedResponse` *Wrapper Class*

```
private static class WrappedResponse extends HttpServletResponseWrapper
{
  private CharArrayWriter buffer;

  public WrappedResponse(HttpServletResponse response)
  {
    super(response);
    buffer = new CharArrayWriter();
  }

  public PrintWriter getWriter()
  {
    return new PrintWriter(buffer);
  }

  public String toString() {
    return buffer.toString();
  }
}

public void doFilter(ServletRequest _req, ServletResponse _res,
  FilterChain _chain) throws IOException, ServletException
{
  /* Wrap the response object */
  WrappedResponse wrappedRes = new WrappedResponse((HttpServletResponse) _res);

  /* Continue Processing */
  _chain.doFilter(_req, wrappedRes);
```

The remainder of the `doFilter` method retrieves the contents as a String object, writes them to the log file, and continues to write the content to the original response object. This is necessary because you passed a wrapped object through the filter chain rather than the original response. Here's the code that executes after the filter chain returns from its processing:

```
  /* Log the response content */
  StringBuffer log = new StringBuffer();
  log.append("*** HTTP Response: ").append(new Date()).append("***\r\n\r\n");
  String output = wrappedRes.toString();
  log.append(output).append("\r\n\r\n");
  fos.write(log.toString().getBytes());
```

```
  /* Write content to browser */
  _res.setContentLength(output.length());
  _res.getWriter().print(output);
  _res.getWriter().close();
}
```

To deploy this filter, add the appropriate entries to your `web.xml` file like you did in the previous example. For a complete listing of the filter code, see Listing 6-6.

Listing 6-6. `ResponseLoggingFilter.java`

```
package jspbook.ch06.filters;

import java.io.File;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.CharArrayWriter;
import java.io.IOException;

import java.util.Date;

import javax.servlet.Filter;
import javax.servlet.FilterConfig;
import javax.servlet.FilterChain;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletResponseWrapper;

public class ResponseLoggingFilter implements Filter {

  private FilterConfig config =  null;
  private FileOutputStream fos;

  private final String LOGGING_FILE = "C:\\tomcat_";

  private static class WrappedResponse extends HttpServletResponseWrapper
  {
    private CharArrayWriter buffer;

    public WrappedResponse(HttpServletResponse response)
    {
      super(response);
      buffer = new CharArrayWriter();
    }
```

```
    public PrintWriter getWriter()
    {
      return new PrintWriter(buffer);
    }

    public String toString() {
      return buffer.toString();
    }
  }

  public void init(FilterConfig _config)
    throws ServletException
  {
    this.config = _config;
    try {
      /* Timestamp log file */
      File file = new File(LOGGING_FILE
        + new Date().getTime() + ".log");
      fos = new FileOutputStream(file);
    }
    catch (FileNotFoundException e) {
      System.out.println("Error opening log file.");
      System.out.println(e.toString());
    }
  }

  public void doFilter(ServletRequest _req, ServletResponse _res,
    FilterChain _chain) throws IOException, ServletException
  {

    /* Wrap the response object */
    WrappedResponse wrappedRes = new WrappedResponse((HttpServletResponse) _res);

    /* Continue Processing */
    _chain.doFilter(_req, wrappedRes);

    /* Log the response content */
    StringBuffer log = new StringBuffer();
    log.append("*** HTTP Response: ").append(new Date()).append("***\r\n\r\n");
    //String output = bapw.getStream().toString();
    String output = wrappedRes.toString();
    log.append(output).append("\r\n\r\n");
    fos.write(log.toString().getBytes());

    /* Write content to browser */
    _res.setContentLength(output.length());
    _res.getWriter().print(output);
    _res.getWriter().close();
  }
```

```
  public void destroy()
  {
    config = null;
    try {
      fos.close();
    }
    catch (IOException e) {
      System.out.println("Error closing log file.");
      System.out.println(e.toString());
    }
  }
}
```

Using the `RequestLoggingFilter` and the `ResponseLoggingFilter`, each HTTP request made to the server is written to a log file, and each response is written to a different log file. To see this in action, open the MVC example from Chapter 5. Access the `login.jsp` page, and enter a valid username and password. Enter a set of data into the next page. Click the submit button to complete the survey.

The previous set of actions produced three HTTP requests. The first request was to display the `login.jsp` page. The second request was to submit the login data and display the survey form. The final request was to submit the survey data and display a confirmation page. Listing 6-7 shows the log file for the request filter.

Listing 6-7. *Log File for Request Filter*

```
HTTP Request: Thu May 27 12:56:02 BST 2004:

Remote Address: 127.0.0.1
Remote Host: 127.0.0.1

HTTP Request: Thu May 27 12:56:08 BST 2004:

Remote Address: 127.0.0.1
Remote Host: 127.0.0.1

UID = apatzer
PWD = apress
Submit = Login
action = login
HTTP Request: Thu May 27 12:56:38 BST 2004:

Remote Address: 127.0.0.1
Remote Host: 127.0.0.1
```

```
age = 39
lname = Glyzewski
children = 2
married = y
sex = M
action = submit
smoker = n
fname = Dave
```

The `ResponseLoggingFilter` logged three separate responses to go along with the three requests logged by the `RequestLoggingFilter`. The first response generated was the HTML to display the login page. The second response was the HTML to display the survey form. The final response was the HTML to display the confirmation page. Listing 6-8 shows an abbreviated log file containing these three responses.

Listing 6-8. *Abbreviated Log File for Response Filter*

```
*** HTTP Response: Thu May 27 13:39:45 BST 2004***

<html>
  <head>
    <title>Quoting System Login</title>
  </head>

  <body bgcolor="#FFFF99">

==> HTML code to display login page (omitted)

</body>
</html>

*** HTTP Response: Thu May 27 13:40:19 BST 2004***

<html>
  <head>
    <title>Insurance Quoting System</title>
  </head>

  <body bgcolor="#FFFF99">

==> HTML code to display survey page (omitted)

</body>
</html>


*** HTTP Response: Thu May 27 13:40:40 BST 2004***
```

```
<html>
  <head>
    <title>Insurance Quoting System</title>
  </head>

  <body bgcolor="#FFFF99">

==> HTML code to display confirmation page (omitted)

</body>
</html>
```

Summary

This chapter showed a great way to perform the preprocessing and postprocessing of an HTTP request and response using the Decorating Filter pattern from the J2EE Patterns Catalog. Fortunately, the Servlet API provides a standard filtering mechanism that enables you to create filters and declaratively add and remove them from filter chains associated with specific URL patterns. As you learn about the remaining presentation patterns, you'll begin to see how filters can play a big part in the design of your web applications. In the last part of this book, you'll use filters again as you put everything together into a web application framework.