Servlet and JSP Filters

Topics in This Chapter

- Designing basic filters
- Reading request data
- Accessing the servlet context
- Initializing filters
- Blocking the servlet or JSP response
- Modifying the servlet or JSP response
- Using filters for debugging and logging
- Using filters to monitor site access
- Using filters to replace strings
- Using filters to compress the response

Chapter 9

Perhaps the single most important new capability in version 2.3 of the servlet API is the ability to define filters for servlets and JSP pages. Filters provide a powerful and standard alternative to the nonstandard "servlet chaining" supported by some early servers.

A filter is a program that runs on the server before the servlet or JSP page with which it is associated. A filter can be attached to one or more servlets or JSP pages and can examine the request information going into these resources. After doing so, it can choose among the following options.

- Invoke the resource (i.e., the servlet or JSP page) in the normal manner.
- Invoke the resource with modified request information.
- Invoke the resource but modify the response before sending it to the client.
- Prevent the resource from being invoked and instead redirect to a different resource, return a particular status code, or generate replacement output.

This capability provides several important benefits.

First, it lets you encapsulate common behavior in a modular and reusable manner. Do you have 30 different servlets or JSP pages that need to compress their content to decrease download time? No problem: make a single compression filter (Section 9.11) and apply it to all 30 resources.

Second, it lets you separate high-level access decisions from presentation code. This is particularly valuable with JSP, where you usually want to keep the page almost entirely focused on presentation, not business logic. For example, do you want to block access from certain sites without modifying the individual pages to which these access restrictions apply? No problem: create an access restriction filter (Section 9.8) and apply it to as many or few pages as you like.

Finally, filters let you apply wholesale changes to many different resources. Do you have a bunch of existing resources that should remain unchanged except that the company name should be changed? No problem: make a string replacement filter (Section 9.10) and apply it wherever appropriate.

Remember, however, that filters work only in servers that are compliant with version 2.3 of the servlet specification. If your Web application needs to support older servers, you cannot use filters.

Core Warning

Filters fail in servers that are compliant only with version 2.2 or earlier versions of the servlet specification.

9.1 Creating Basic Filters

Creating a filter involves five basic steps:

1. **Create a class that implements the `Filter` interface.** Your class will need three methods: `doFilter`, `init`, and `destroy`. The `doFilter` method contains the main filtering code (see Step 2), the `init` method performs setup operations, and the `destroy` method does cleanup.
2. **Put the filtering behavior in the `doFilter` method.** The first argument to the `doFilter` method is a `ServletRequest` object. This object gives your filter full access to the incoming information, including form data, cookies, and HTTP request headers. The second argument is a `ServletResponse`; it is mostly ignored in simple filters. The final argument is a `FilterChain`; it is used to invoke the servlet or JSP page as described in the next step.
3. **Call the `doFilter` method of the `FilterChain` object.** The `doFilter` method of the `Filter` interface takes a `FilterChain` object as one of its arguments. When you call the `doFilter` method

of that object, the next associated filter is invoked. If no other filter is associated with the servlet or JSP page, then the servlet or page itself is invoked.

4. **Register the filter with the appropriate servlets and JSP pages.** Use the `filter` and `filter-mapping` elements in the deployment descriptor (*web.xml*).
5. **Disable the invoker servlet.** Prevent users from bypassing filter settings by using default servlet URLs.

Details follow.

Create a Class That Implements the Filter Interface

All filters must implement `javax.servlet.Filter`. This interface comprises three methods: `doFilter`, `init`, and `destroy`.

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

The `doFilter` method is executed each time a filter is invoked (i.e., once for each request for a servlet or JSP page with which the filter is associated). It is this method that contains the bulk of the filtering logic.

The first argument is the `ServletRequest` associated with the incoming request. For simple filters, most of your filter logic is based on this object. Cast the object to `HttpServletRequest` if you are dealing with HTTP requests and you need access to methods such as `getHeader` or `getCookies` that are unavailable in `ServletRequest`.

The second argument is the `ServletResponse`. You often ignore this argument, but there are two cases when you use it. First, if you want to completely block access to the associated servlet or JSP page, you can call `response.getWriter` and send a response directly to the client. Section 9.7 gives details; Section 9.8 gives an example. Second, if you want to modify the output of the associated servlet or JSP page, you can wrap the response inside an object that collects all output sent to it. Then, after the servlet or JSP page is invoked, the filter can examine the output, modify it if appropriate, and then send it to the client. See Section 9.9 for details.

The final argument to `doFilter` is a `FilterChain` object. You call `doFilter` on this object to invoke the next filter that is associated with the servlet or JSP page. In no other filters are in effect, then the call to `doFilter` invokes the servlet or JSP page itself.

**public void init(FilterConfig config)
throws ServletException**

The `init` method is executed only when the filter is first initialized. It is not executed each time the filter is invoked. For simple filters you can provide an empty body to this method, but there are two common reasons for using `init`. First, the `FilterConfig` object provides access to the servlet context and to the name of the filter that is assigned in the *web.xml* file. So, it is common to use `init` to store the `FilterConfig` object in a field so that the `doFilter` method can access the servlet context or the filter name. This process is described in Section 9.3. Second, the `FilterConfig` object has a `getInitParameter` method that lets you access filter initialization parameters that are assigned in the deployment descriptor (*web.xml*). Use of initialization parameters is described in Section 9.5.

public void destroy()

This method is called when a server is permanently finished with a given filter object (e.g., when the server is being shut down). Most filters simply provide an empty body for this method, but it can be used for cleanup tasks like closing files or database connection pools that are used by the filter.

Put the Filtering Behavior in the doFilter Method

The `doFilter` method is the key part of most filters. Each time a filter is invoked, `doFilter` is executed. With most filters, the steps that `doFilter` performs are based on the incoming information. So, you will probably make use of the `ServletRequest` that is supplied as the first argument to `doFilter`. This object is frequently typecast to `HttpServletRequest` to provide access to the more specialized methods of that class.

Call the doFilter Method of the FilterChain Object

The `doFilter` method of the `Filter` interface takes a `FilterChain` object as its third argument. When you call the `doFilter` method of that object, the next associated filter is invoked. This process normally continues until the last filter in the chain is invoked. When the final filter calls the `doFilter` method of its `FilterChain` object, the servlet or page itself is invoked.

However, any filter in the chain can interrupt the process by omitting the call to the `doFilter` method of its `FilterChain`. In such a case, the servlet of JSP page is never invoked and the filter is responsible for providing output to the client. For details, see Section 9.7 (Blocking the Response).

Register the Filter with the Appropriate Servlets and JSP Pages

Version 2.3 of the deployment descriptor introduced two elements for use with filters: `filter` and `filter-mapping`. The `filter` element registers a filtering object with the system. The `filter-mapping` element specifies the URLs to which the filtering object applies.

The filter Element

The `filter` element goes near the top of deployment descriptor (*web.xml*), before any `filter-mapping`, `servlet`, or `servlet-mapping` elements. (For more information on the use of the deployment descriptor, see Chapters 4 and 5. For details on the required ordering of elements within the deployment descriptor, see Section 5.2.) The `filter` element contains six possible subelements:

- **`icon`**. This is an optional element that declares an image file that an IDE can use.
- **`filter-name`**. This is a required element that assigns a name of your choosing to the filter.
- **`display-name`**. This is an optional element that provides a short name for use by IDEs.
- **`description`**. This is another optional element that gives information for IDEs. It provides textual documentation.
- **`filter-class`**. This is a required element that specifies the fully qualified name of the filter implementation class.
- **`init-param`**. This is an optional element that defines initialization parameters that can be read with the `getInitParameter` method of `FilterConfig`. A single filter element can contain multiple `init-param` elements.

Remember that filters were first introduced in version 2.3 of the servlet specification. So, your *web.xml* file must use version 2.3 of the DTD. Here is a simple example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>myPackage.FilterClass</filter-class>
  </filter>
  <!-- ... -->
  <filter-mapping>...</filter-mapping>
</web-app>
```

The filter-mapping Element

The `filter-mapping` element goes in the *web.xml* file after the `filter` element but before the `servlet` element. It contains three possible subelements:

- **`filter-name`**. This required element must match the name you gave to the filter when you declared it with the `filter` element.
- **`url-pattern`**. This element declares a pattern starting with a slash (`/`) that designates the URLs to which the filter applies. You must supply `url-pattern` or `servlet-name` in all `filter-mapping` elements. You cannot provide multiple `url-pattern` entries with a single `filter-mapping` element, however. If you want the filter to apply to multiple patterns, repeat the entire `filter-mapping` element.
- **`servlet-name`**. This element gives a name that must match a name given to a servlet or JSP page by means of the `servlet` element. For details on the `servlet` element, see Section 5.3 (Assigning Names and Custom URLs). You cannot provide multiple `servlet-name` elements entries with a single `filter-mapping` element. If you want the filter to apply to multiple servlet names, repeat the entire `filter-mapping` element.

Here is a simple example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>myPackage.FilterClass</filter-class>
  </filter>
  <!-- ... -->
  <filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/someDirectory/SomePage.jsp</url-pattern>
  </filter-mapping>
</web-app>
```

Disable the Invoker Servlet

When you apply filters to resources, you do so by specifying the URL pattern or servlet name to which the filters apply. If you supply a servlet name, that name must match a name given in the `servlet` element of *web.xml*. If you use a URL pattern that applies to a servlet, the pattern must match a pattern that you specified with the `servlet-mapping` *web.xml* element (see Section 5.3, "Assigning Names and Custom URLs"). However, most servers use an "invoker servlet" that provides a default URL for servlets: *http://host/webAppPrefix/servlet/ServletName*. You need to make sure that users don't access servlets with this URL, thus bypassing the filter settings.

For example, suppose that you use `filter` and `filter-mapping` to say that the filter named `SomeFilter` applies to the servlet named `SomeServlet`, as below.

```
<filter>
  <filter-name>SomeFilter</filter-name>
  <filter-class>somePackage.SomeFilterClass</filter-class>
</filter>
<!-- ... -->
<filter-mapping>
  <filter-name>SomeFilter</filter-name>
  <servlet-name>SomeServlet</servlet-name>
</filter-mapping>
```

Next, you use `servlet` and `servlet-mapping` to stipulate that the URL *http://host/webAppPrefix/Blah* should invoke `SomeServlet`, as below.

```
<servlet>
  <servlet-name>SomeServlet</servlet-name>
  <servlet-class>somePackage.SomeServletClass</servlet-class>
</servlet>
<!-- ... -->
<servlet-mapping>
  <servlet-name>SomeServlet</servlet-name>
  <url-pattern>/Blah</url-pattern>
</servlet-mapping>
```

Now, the filter is invoked when clients use the URL *http://host/webAppPrefix/Blah*. No filters apply to *http://host/webAppPrefix/servlet/somePackage.SomeServlet-Class*. Oops.

Section 5.4 (Disabling the Invoker Servlet) discusses server-specific approaches to turning off the invoker. The most portable approach, however, is to simply remap the */servlet* pattern in your Web application so that all requests that include the pattern are sent to the same servlet. To remap the pattern, you first create a simple servlet that prints an error message or redirects users to the top-level page. Then, you use the `servlet` and `servlet-mapping` elements (Section 5.3) to send requests that include the */servlet* pattern to that servlet. Listing 9.1 gives a brief example.

Listing 9.1 *web.xml* (Excerpt that redirects default servlet URLs)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>Error</servlet-name>
    <servlet-class>somePackage.ErrorServlet</servlet-class>
  </servlet>
  <!-- ... -->
  <servlet-mapping>
    <servlet-name>Error</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
  <!-- ... -->
</web-app>
```

9.2 Example: A Reporting Filter

Just to warm up, let's try a simple filter that merely prints a message to standard output whenever the associated servlet or JSP page is invoked. To accomplish this task, the filter has the following capabilities.

1. **A class that implements the `Filter` interface.** This class is called `ReportFilter` and is shown in Listing 9.2. The class provides empty bodies for the `init` and `destroy` methods.

2. **Filtering behavior in the `doFilter` method.** Each time a servlet or JSP page associated with this filter is invoked, the `doFilter` method generates a printout that lists the requesting host and the URL that was invoked. Since the `getRequestURL` method is in `HttpServletRequest`, not `ServletRequest`, I cast the `ServletRequest` object to `HttpServletRequest`.
3. **A call to the `doFilter` method of the `FilterChain`.** After printing the report, the filter calls the `doFilter` method of the `FilterChain` to invoke the servlet or JSP page (or the next filter in the chain if there was one).
4. **Registration with the Web application home page and the servlet that displays the daily special.** First, the `filter` element associates the name `Reporter` with the class `moreservlets.filters.ReportFilter`. Then, the `filter-mapping` element uses a `url-pattern` of `/index.jsp` to associate the filter with the home page. Finally, the `filter-mapping` element uses a `servlet-name` of `TodaysSpecial` to associate the filter with the daily special servlet (the name `TodaysSpecial` is declared in the `servlet` element). See Listing 9.3.
5. **Disablement of the invoker servlet.** First, I created a `RedirectorServlet` (Listing 9.6) that redirects all requests that it receives to the Web application home page. Next, I used the `servlet` and `servlet-mapping` elements (Listing 9.3) to specify that all URLs that begin with *http://host/webAppPrefix/servlet/* should invoke the `RedirectorServlet`.

Given these settings, the filter is invoked each time a client requests the Web application home page (Listing 9.4, Figure 9–1) or the daily special servlet (Listing 9.5, Figure 9–2).

Listing 9.2 *ReportFilter.java*

```
package moreservlets.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*; // For Date class

/** Simple filter that prints a report on the standard output
 *  each time an associated servlet or JSP page is accessed.
 */
```

Listing 9.2 *ReportFilter.java (continued)*

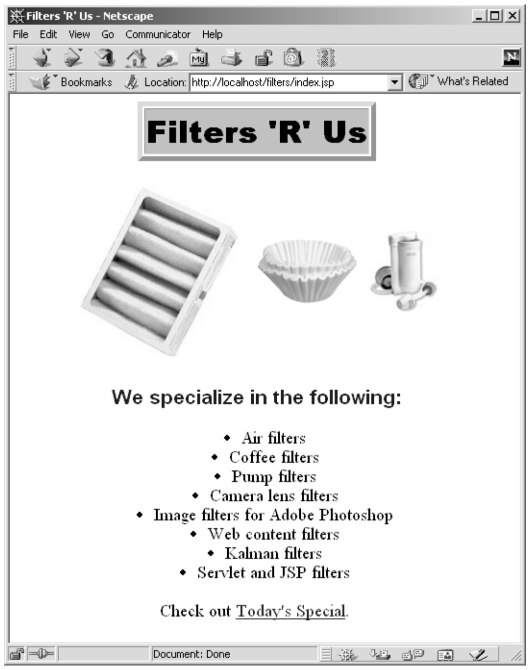
```
public class ReportFilter implements Filter {
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
      throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
    System.out.println(req.getRemoteHost() +
                       " tried to access " +
                       req.getRequestURL() +
                       " on " + new Date() + ".");
    chain.doFilter(request,response);
  }

  public void init(FilterConfig config)
      throws ServletException {
  }

  public void destroy() {}
}
```

Listing 9.3 *web.xml* (Excerpt for reporting filter)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- Register the name "Reporter" for ReportFilter. -->
  <filter>
    <filter-name>Reporter</filter-name>
    <filter-class>
      moreservlets.filters.ReportFilter
    </filter-class>
  </filter>
  <!-- ... -->

  <!-- Apply the Reporter filter to home page. -->
  <filter-mapping>
    <filter-name>Reporter</filter-name>
    <url-pattern>/index.jsp</url-pattern>
  </filter-mapping>
```

Listing 9.3 *web.xml* (Excerpt for reporting filter) *(continued)*

```
  <!-- Also apply the Reporter filter to the servlet named
       "TodaysSpecial".
  -->
  <filter-mapping>
    <filter-name>Reporter</filter-name>
    <servlet-name>TodaysSpecial</servlet-name>
  </filter-mapping>
  <!-- ... -->

  <!-- Give a name to the Today's Special servlet so that filters
       can be applied to it.
  -->
  <servlet>
    <servlet-name>TodaysSpecial</servlet-name>
    <servlet-class>
      moreservlets.TodaysSpecialServlet
    </servlet-class>
  </servlet>
  <!-- ... -->

 <!-- Make /TodaysSpecial invoke the servlet
       named TodaysSpecial (i.e., moreservlets.TodaysSpecial).
  -->
  <servlet-mapping>
    <servlet-name>TodaysSpecial</servlet-name>
    <url-pattern>/TodaysSpecial</url-pattern>
  </servlet-mapping>

  <!-- Turn off invoker. Send requests to index.jsp. -->
  <servlet-mapping>
    <servlet-name>Redirector</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- ... -->
</web-app>
```


Filters 'R' Us - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/filters/index.jsp What's Related
Document: Done

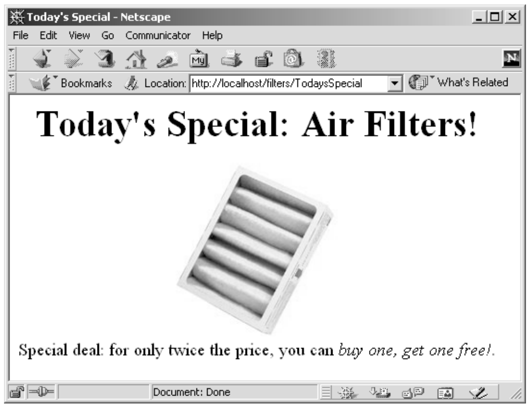
Filters 'R' Us

We specialize in the following:

- Air filters
- Coffee filters
- Pump filters
- Camera lens filters
- Image filters for Adobe Photoshop
- Web content filters
- Kalman filters
- Servlet and JSP filters

Check out Today's Special.

Figure 9–1 Home page for filter company. After the page is deployed on an external server and the reporting filter is attached, each client access results in a printout akin to "purchasing.sun.com tried to access http://www.filtersrus.com/filters/index.jsp on Fri Oct 26 13:19:14 EDT 2001."


Today's Special - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/filters/TodaysSpecial What's Related
Document: Done

Today's Special: Air Filters!

Special deal: for only twice the price, you can *buy one, get one free!*.

Figure 9–2 Page advertising a special sale. After the page is deployed on an external server and the reporting filter is attached, each client access results in a printout akin to "admin.microsoft.com tried to access http://www.filtersrus.com/filters/TodaysSpecial on Fri Oct 26 13:21:56 EDT 2001."

Listing 9.4 *index.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Filters 'R' Us</TITLE>
<LINK REL=STYLESHEET
      HREF="filter-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">Filters 'R' Us</TABLE>
<P>
<TABLE>
  <TR>
    <TH><IMG SRC="images/air-filter.jpg" ALT="Air Filter">
    <TH><IMG SRC="images/coffee-filter.gif" ALT="Coffee Filter">
    <TH><IMG SRC="images/pump-filter.jpg" ALT="Pump Filter">
</TABLE>

<H3>We specialize in the following:</H3>
<UL>
  <LI>Air filters
  <LI>Coffee filters
  <LI>Pump filters
  <LI>Camera lens filters
  <LI>Image filters for Adobe Photoshop
  <LI>Web content filters
  <LI>Kalman filters
  <LI>Servlet and JSP filters
</UL>
Check out <A HREF="TodaysSpecial">Today's Special</A>.
</CENTER>
</BODY>
</HTML>
```

Listing 9.5 *TodaysSpecialServlet.java*

```
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Sample servlet used to test the simple filters. */

public class TodaysSpecialServlet extends HttpServlet {
  private String title, picture;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    updateSpecials();
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>Today's Special</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"WHITE\">\n" +
       "<CENTER>\n" +
       "<H1>Today's Special: " + title + "s!</H1>\n" +
       "<IMG SRC=\"images/" + picture + "\"\n" +
       "     ALT=\"" + title + "\">\n" +
       "<BR CLEAR=\"ALL\">\n" +
       "Special deal: for only twice the price, you can\n" +
       "<I>buy one, get one free!</I>.\n" +
       "</BODY></HTML>");
  }

// Rotate among the three available filter images.
```

Listing 9.5 *TodaysSpecialServlet.java (continued)*

```
  private void updateSpecials() {
    double num = Math.random();
    if (num < 0.333) {
      title = "Air Filter";
      picture = "air-filter.jpg";
    } else if (num < 0.666) {
      title = "Coffee Filter";
      picture = "coffee-filter.gif";
    } else {
      title = "Pump Filter";
      picture = "pump-filter.jpg";
    }
  }
}
```

Listing 9.6 *RedirectorServlet.java*

```
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet that simply redirects users to the
 *  Web application home page. Registered with the
 *  default servlet URL to prevent clients from
 *  using http://host/webAppPrefix/servlet/ServletName
 *  to bypass filters or security settings that
 *  are associated with custom URLs.
 */

public class RedirectorServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.sendRedirect(request.getContextPath());
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

9.3 Accessing the Servlet Context from Filters

The ReportFilter of the previous section prints a report on the standard output whenever the designated servlet or JSP page is invoked. A report on the standard output is fine during development—when you run a server on your desktop you typically have a window that displays the standard output. During deployment, however, you are unlikely to have access to this window. So, a natural enhancement is to write the reports into the servlet log file instead of to the standard output.

The servlet API provides two log methods: one that takes a simple String and another that takes a String and a Throwable. These two methods are available from either the GenericServlet or ServletContext classes. Check your server's documentation for the exact location of the log files that these methods use. The problem is that the doFilter method executes *before* the servlet or JSP page with which it is associated. So, you don't have access to the servlet instance and thus can't call the log methods that are inherited from GenericServlet. Furthermore, the API provides no simple way to access the ServletContext from the doFilter method. The only filter-related class that has a method to access the ServletContext is FilterConfig with its getServletContext method. A FilterConfig object is passed to the init method but is not automatically stored in a location that is available to doFilter.

So, you have to store the FilterConfig yourself. Simply create a field of type FilterConfig, then override init to assign its argument to that field. Since you typically use the FilterConfig object only to access the ServletContext and the filter name, you can store the ServletContext and name in fields as well. Here is a simple example:

```
public class SomeFilter implements Filter {
  protected FilterConfig config;
  private ServletContext context;
  private String filterName;

  public void init(FilterConfig config)
      throws ServletException {
    this.config = config; // In case it is needed by subclass.
    context = config.getServletContext();
    filterName = config.getFilterName();
  }

  // doFilter and destroy methods...
}
```

9.4 Example: A Logging Filter

Let's update the `ReportFilter` (Listing 9.2) so that messages go in the log file instead of to the standard output. To accomplish this task, the filter has the following capabilities.

1. **A class that implements the `Filter` interface.** This class is called `LogFilter` and is shown in Listing 9.7. The `init` method of this class stores the `FilterConfig`, `ServletContext`, and filter name in fields of the filter. The class provides an empty body for the `destroy` method.
2. **Filtering behavior in the `doFilter` method.** There are two differences between this behavior and that of the `ReportFilter`: the report is placed in the log file instead of the standard output and the report includes the name of the filter.
3. **A call to the `doFilter` method of the `FilterChain`.** After printing the report, the filter calls the `doFilter` method of the `FilterChain` to invoke the next filter in the chain (or the servlet or JSP page if there are no more filters).
4. **Registration with all URLs.** First, the `filter` element associates the name `LogFilter` with the class `moreservlets.filters.LogFilter`. Next, the `filter-mapping` element uses a `url-pattern` of `/*` to associate the filter with *all* URLs in the Web application. See Listing 9.8.
5. **Disablement of the invoker servlet.** This operation is shown in Section 9.2 and is not repeated here.

After the Web application is deployed on an external server and the logging filter is attached, a client request for the Web application home page results in an entry in the log file like "audits.irs.gov tried to access http://www.filtersrus.com/filters/index.jsp on Fri Oct 26 15:16:15 EDT 2001. (Reported by Logger.)"

Listing 9.7 *LogFilter.java*

```
package moreservlets.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*; // For Date class

/** Simple filter that prints a report in the log file
 *  whenever the associated servlets or JSP pages
 *  are accessed.
 */

public class LogFilter implements Filter {
  protected FilterConfig config;
  private ServletContext context;
  private String filterName;

  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
      throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
    context.log(req.getRemoteHost() +
                " tried to access " +
                req.getRequestURL() +
                " on " + new Date() + ". " +
                "(Reported by " + filterName + ".)");
    chain.doFilter(request,response);
  }

  public void init(FilterConfig config)
      throws ServletException {
    this.config = config; // In case it is needed by subclass.
    context = config.getServletContext();
    filterName = config.getFilterName();
  }

  public void destroy() {}
}
```

Listing 9.8 *web.xml* (Excerpt for logging filter)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->

  <!-- Register the name "Logger" for LogFilter. -->
  <filter>
    <filter-name>Logger</filter-name>
    <filter-class>
      moreservlets.filters.LogFilter
    </filter-class>
  </filter>
  <!-- ... -->

  <!-- Apply the Logger filter to all servlets and
       JSP pages.
  -->
  <filter-mapping>
    <filter-name>Logger</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <!-- ... -->
</web-app>
```

9.5 Using Filter Initialization Parameters

With servlets and JSP pages, you can customize the initialization behavior by supply initialization parameters. For details, see Section 5.5 (Initializing and Preloading Servlets and JSP Pages). The reason this capability is useful is that there are three distinct groups that might want to customize the behavior of servlets or JSP pages:

1. **Developers.** They customize the behavior by changing the code of the servlet or JSP page itself.
2. **End users.** They customize the behavior by entering values in HTML forms.
3. **Deployers.** This third group is the one served by initialization parameters. Members of this group are people who take existing Web applications (or individual servlets or JSP pages) and deploy them in a

customized environment. They are not necessarily developers, so it is not realistic to expect them to modify the servlet and JSP code. Besides, you often omit the source code when distributing servlets. So, developers need a standard way to allow deployers to change servlet and JSP behavior.

If these capabilities are useful for servlets and JSP pages, you would expect them to also be useful for the filters that apply to servlets and JSP page. Indeed they are. However, since filters execute before the servlets or JSP pages to which they are attached, it is not normally possible for end users to customize filter behavior. Nevertheless, it is still useful to permit deployers (not just developers) to customize filter behavior by providing initialization parameters. This behavior is accomplished with the following steps.

1. **Define initialization parameters.** Use the `init-param` subelement of `filter` in *web.xml* along with `param-name` and `param-value` subelements, as follows.

   ```
   <filter>
     <filter-name>SomeFilter</filter-name>
     <filter-class>somePackage.SomeFilterClass</filter-class>
     <init-param>
       <param-name>param1</param-name>
       <param-value>value1</param-value>
     </init-param>
     <init-param>
       <param-name>param2</param-name>
       <param-value>value2</param-value>
     </init-param>
   </filter>
   ```

2. **Read the initialization parameters.** Call the `getInitParameter` method of `FilterConfig` from the `init` method of your filter, as follows.

   ```
   public void init(FilterConfig config)
       throws ServletException {
     String val1 = config.getInitParameter("param1");
     String val2 = config.getInitParameter("param2");
     ...
   }
   ```

3. **Parse the initialization parameters.** Like servlet and JSP initialization parameters, each filter initialization value is of type `String`. So, if you want a value of another type, you have to convert it yourself. For example, you would use `Integer.parseInt` to turn the `String` `"7"` into the `int` 7. When parsing, don't forget to check for missing

and malformed data. Missing initialization parameters result in `null` being returned from `getInitParameter`. Even if the parameters exist, you should consider the possibility that the deployer formatted the value improperly. For example, when converting a `String` to an `int`, you should enclose the `Integer.parseInt` call within a `try`/`catch` block that catches `NumberFormatException`. This handles `null` and incorrectly formatted values in one fell swoop.

9.6 Example: An Access Time Filter

The `LogFilter` of Section 9.4 prints an entry in the log file every time the associated servlet or JSP page is accessed. Suppose you want to modify it so that it only notes accesses that occur at unusual times. Since "unusual" is situation dependent, the servlet should provide default values for the abnormal time ranges and let deployers override these values by supplying initialization parameters. To implement this functionality, the filter has the following capabilities.

1. **A class that implements the `Filter` interface.** This class is called `LateAccessFilter` and is shown in Listing 9.9. The `init` method of this class reads the `startTime` and `endTime` initialization parameters. It attempts to parse these values as type `int`, using default values if the parameters are `null` or not formatted as integers. It then stores the start and end times, the `FilterConfig`, the `ServletContext`, and the filter name in fields of the filter. Finally, `LateAccessFilter` provides an empty body for the `destroy` method.
2. **Filtering behavior in the `doFilter` method.** This method looks up the current time, sees if it is within the range given by the start and end times, and prints a log entry if so.
3. **A call to the `doFilter` method of the `FilterChain`.** After printing the report, the filter calls the `doFilter` method of the `FilterChain` to invoke the next filter in the chain (or the servlet or JSP page if there are no more filters).
4. **Registration with the Web application home page; definition of initialization parameters.** First, the `filter` element associates the name `LateAccessFilter` with the class `moreservlets.filters.LateAccessFilter`. The `filter` element also includes two `init-param` subelements: one that defines the `startTime` parameter and another that defines `endTime`. Since the people that will be accessing the filtersRus home page are programmers, an abnormal range is considered to be between 2:00 a.m. and 10:00 a.m. Finally, the `filter-mapping` element uses a `url-pattern` of

`/index.jsp` to associate the filter with the Web application home page. See Listing 9.10.

5. **Disablement of the invoker servlet.** This operation is shown in Section 9.2 and is not repeated here.

After the Web application is deployed on an external server and the logging filter is attached, a client request for the Web application home page results in an entry in the log file like "WARNING: hacker6.filtersrus.com accessed http://www.filtersrus.com/filters/index.jsp on Oct 30, 2001 9:22:09 AM."

Listing 9.9 *LateAccessFilter.java*

```
package moreservlets.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.text.*;

/** Filter that keeps track of accesses that occur
 *  at unusual hours.
 */

public class LateAccessFilter implements Filter {
  private FilterConfig config;
  private ServletContext context;
  private int startTime, endTime;
  private DateFormat formatter;

  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
      throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
    GregorianCalendar calendar = new GregorianCalendar();
    int currentTime = calendar.get(calendar.HOUR_OF_DAY);
    if (isUnusualTime(currentTime, startTime, endTime)) {
      context.log("WARNING: " +
                  req.getRemoteHost() +
                  " accessed " +
                  req.getRequestURL() +
                  " on " +
                  formatter.format(calendar.getTime()));
    }
    chain.doFilter(request,response);
  }
```

Listing 9.9 *LateAccessFilter.java (continued)*

```
  public void init(FilterConfig config)
      throws ServletException {
    this.config = config;
    context = config.getServletContext();
    formatter =
      DateFormat.getDateTimeInstance(DateFormat.MEDIUM,
                                     DateFormat.MEDIUM);
    try {
      startTime =
        Integer.parseInt(config.getInitParameter("startTime"));
      endTime =
        Integer.parseInt(config.getInitParameter("endTime"));
    } catch(NumberFormatException nfe) { // Malformed or null
      // Default: access at or after 10 p.m. but before 6 a.m.
      // is considered unusual.
      startTime = 22; // 10:00 p.m.
      endTime = 6;    //  6:00 a.m.
    }
  }

  public void destroy() {}

  // Is the current time between the start and end
  // times that are marked as abnormal access times?

  private boolean isUnusualTime(int currentTime,
                                int startTime,
                                int endTime) {
    // If the start time is less than the end time (i.e.,
    // they are two times on the same day), then the
    // current time is considered unusual if it is
    // between the start and end times.
    if (startTime < endTime) {
      return((currentTime >= startTime) &&
             (currentTime < endTime));
    }
    // If the start time is greater than or equal to the
    // end time (i.e., the start time is on one day and
    // the end time is on the next day), then the current
    // time is considered unusual if it is NOT between
    // the end and start times.
    else {
      return(!isUnusualTime(currentTime, endTime, startTime));
    }
  }
}
```

Listing 9.10 *web.xml* (Excerpt for access time filter)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->

  <!-- Register the name "LateAccessFilter" for
       moreservlets.filter.LateAccessFilter.
       Supply two initialization parameters:
       startTime and endTime.
  -->
  <filter>
    <filter-name>LateAccessFilter</filter-name>
    <filter-class>
      moreservlets.filters.LateAccessFilter
    </filter-class>
    <init-param>
      <param-name>startTime</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>endTime</param-name>
      <param-value>10</param-value>
    </init-param>
  </filter>
  <!-- ... -->

  <!-- Apply LateAccessFilter to the home page. -->
  <filter-mapping>
    <filter-name>LateAccessFilter</filter-name>
    <url-pattern>/index.jsp</url-pattern>
  </filter-mapping>

  <!-- ... -->
</web-app>
```

9.7 Blocking the Response

Up to now, all the filters discussed have concluded their doFilter methods by calling the doFilter method of the FilterChain object. This approach is the normal one—the call to doFilter invokes the next resource in the chain (another filter or the actual servlet or JSP page).

But what if your filter detects an unusual situation and wants to prevent the original resource from being invoked? How can it block the normal response? The answer is quite simple: just omit the call to the doFilter method of the FilterChain object. Instead, the filter can redirect the user to a different page (e.g., with a call to response.sendRedirect) or generate the response itself (e.g., by calling getWriter on the response and sending output, just as with a regular servlet). Just remember that the first two arguments to the filter's main doFilter method are declared to be of type ServletRequest and ServletResponse. So, if you want to use methods specific to HTTP, cast these arguments to HttpServletRequest and HttpServletResponse, respectively. Here is a brief example:

```
public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain)
    throws ServletException, IOException {
  HttpServletRequest req = (HttpServletRequest)request;
  HttpServletResponse res = (HttpServletResponse)response;
  if (isUnusualCondition(req)) {
    res.sendRedirect("http://www.somesite.com");
  } else {
    chain.doFilter(req,res);
  }
}
```

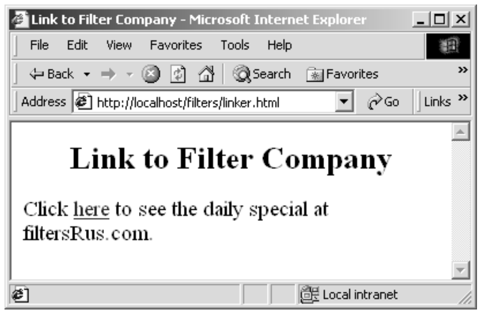
9.8 Example: A Prohibited-Site Filter

Suppose you have a competitor that you want to ban from your site. For example, this competing company might have a service that accesses your site, removes advertisements and information that identify your organization, and displays them to their customers. Or, they might have links to your site that are in framed pages, thus making it appear that your page is part of their site. You'd like to prevent them from accessing certain pages at your site. However, every time their Web hosting company boots them off, they simply change domain names and register with another ISP. So, you want the ability to easily change the domain names that should be banned.

The solution is to make a filter that uses initialization parameters to obtain a list of banned sites. Requests originating or referred from these sites result in a warning message. Other requests proceed normally. To implement this functionality, the filter has the following capabilities.

1. **A class that implements the `Filter` interface.** This class is called `BannedAccessFilter` and is shown in Listing 9.11. The `init` method of this class first obtains a list of sites from an initialization parameter called `bannedSites`. The filter parses the entries in the resultant `String` with a `StringTokenizer` and stores each individual site name in a `HashMap` that is accessible through an instance variable (i.e., field) of the filter. Finally, `BannedAccessFilter` provides an empty body for the `destroy` method.
2. **Filtering behavior in the `doFilter` method.** This method looks up the requesting and referring hosts by using the `getRemoteHost` method of `ServletRequest` and parsing the `Referer` HTTP request header, respectively.
3. **A conditional call to the `doFilter` method of the `FilterChain`.** The filter checks to see if the requesting or referring host is listed in the `HashMap` of banned sites. If so, it calls the `showWarning` method, which sends a custom response to the client. If not, the filter calls `doFilter` on the `FilterChain` object to let the request proceed normally.
4. **Registration with the daily special servlet; definition of initialization parameters.** First, the `filter` element associates the name `BannedAccessFilter` with the class `moreservlets.filters.BannedAccessFilter`. The `filter` element also includes an `init-param` subelement that specifies the prohibited sites (separated by white space). Since the resource that the competing sites abuse is the servlet that shows the daily special, the `filter-mapping` element uses a `servlet-name` of `TodaysSpecial`. The `servlet` element assigns the name `TodaysSpecial` to `moreservlets.TodaysSpecialServlet`. See Listing 9.12.
5. **Disablement of the invoker servlet.** This operation is shown in Section 9.2 and is not repeated here.

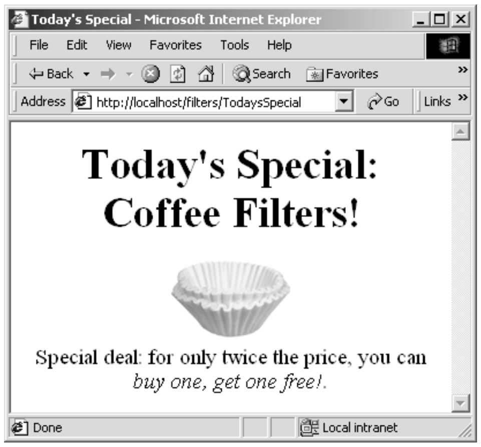
Listing 9.13 shows a very simple page that contains little but a link to the daily special servlet. When that page is hosted on a normal site (Figure 9–3), the link results in the expected output (Figure 9–4). But, when the page that contains the link is hosted on a banned site (Figure 9–5), the link results only in a warning page (Figure 9–6)—access to the real servlet is blocked.


Link to Filter Company - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/filters/linker.html
Go Links
Local intranet

Link to Filter Company

Click here to see the daily special at filtersRus.com.

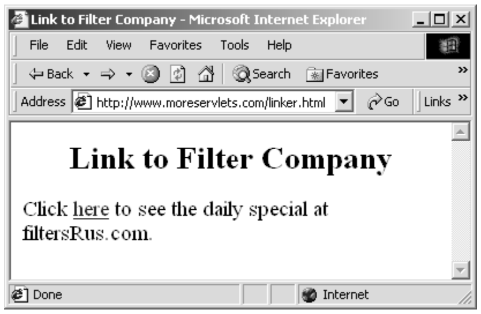
Figure 9–3 A page that links to the daily special servlet. This version is hosted on the desktop development server.


Today's Special - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/filters/TodaysSpecial
Go Links
Done
Local intranet

Today's Special: Coffee Filters!

Special deal: for only twice the price, you can *buy one, get one free!*.

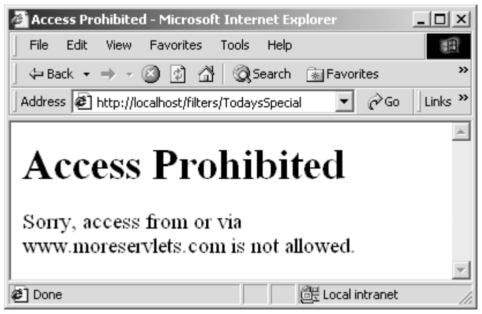
Figure 9–4 You can successfully follow the link from the page of Figure 9–3. The `BannedAccessFilter` does not prohibit access from *localhost*.


Link to Filter Company - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://www.moreservlets.com/linker.html
Go Links
Done
Internet

Link to Filter Company

Click here to see the daily special at filtersRus.com.

Figure 9–5 A page that links to the daily special servlet. This version is hosted on *www.moreservlets.com*.


Access Prohibited - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/filters/TodaysSpecial
Go Links
Done
Local intranet

Access Prohibited

Sorry, access from or via www.moreservlets.com is not allowed.

Figure 9–6 You cannot successfully follow the link from the page of Figure 9–5. The `BannedAccessFilter` prohibits access from *www.moreservlets.com* (an unscrupulous competitor to *filtersRus.com*).

Listing 9.11 *BannedAccessFilter.java*

```
package moreservlets.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.net.*;

/** Filter that refuses access to anyone connecting directly
 *  from or following a link from a banned site.
 */

public class BannedAccessFilter implements Filter {
  private HashSet bannedSiteTable;

  /** Deny access if the request comes from a banned site
   *  or is referred here by a banned site.
   */

  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
      throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
    String requestingHost = req.getRemoteHost();
    String referringHost =
      getReferringHost(req.getHeader("Referer"));
    String bannedSite = null;
    boolean isBanned = false;
```

Listing 9.11 *BannedAccessFilter.java (continued)*

```
    if (bannedSiteTable.contains(requestingHost)) {
      bannedSite = requestingHost;
      isBanned = true;
    } else if (bannedSiteTable.contains(referringHost)) {
      bannedSite = referringHost;
      isBanned = true;
    }
    if (isBanned) {
      showWarning(response, bannedSite);
    } else {
      chain.doFilter(request,response);
    }
  }

  /** Create a table of banned sites based on initialization
   *  parameters. Remember that version 2.3 of the servlet
   *  API mandates the use of the Java 2 Platform. Thus,
   *  it is safe to use HashSet (which determines whether
   *  a given key exists) rather than the clumsier
   *  Hashtable (which has a value for each key).
   */

  public void init(FilterConfig config)
      throws ServletException {
    bannedSiteTable = new HashSet();
    String bannedSites =
      config.getInitParameter("bannedSites");
    // Default token set: white space.
    StringTokenizer tok = new StringTokenizer(bannedSites);
    while(tok.hasMoreTokens()) {
      String bannedSite = tok.nextToken();
      bannedSiteTable.add(bannedSite);
      System.out.println("Banned " + bannedSite);
    }
  }

  public void destroy() {}

  private String getReferringHost(String refererringURLString) {
    try {
      URL referringURL = new URL(refererringURLString);
      return(referringURL.getHost());
    } catch(MalformedURLException mue) { // Malformed or null
      return(null);
    }
  }
```

Listing 9.11 *BannedAccessFilter.java (continued)*

```
  // Replacement response that is returned to users
  // who are from or referred here by a banned site.

  private void showWarning(ServletResponse response,
                           String bannedSite)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>Access Prohibited</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"WHITE\">\n" +
       "<H1>Access Prohibited</H1>\n" +
       "Sorry, access from or via " + bannedSite + "\n" +
       "is not allowed.\n" +
       "</BODY></HTML>");
  }
}
```

Listing 9.12 *web.xml* (Excerpt for prohibited-site filter)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->

  <!-- Register the name "BannedAccessFilter" for
       moreservlets.filter.BannedAccessFilter.
       Supply an initialization parameter:
       bannedSites.
  -->
  <filter>
    <filter-name>BannedAccessFilter</filter-name>
    <filter-class>
      moreservlets.filters.BannedAccessFilter
    </filter-class>
```

Listing 9.12 *web.xml* (Excerpt for prohibited-site filter) *(continued)*

```
    <init-param>
      <param-name>bannedSites</param-name>
      <param-value>
        www.competingsite.com
        www.bettersite.com
        www.moreservlets.com
      </param-value>
    </init-param>
  </filter>
  <!-- ... -->

  <!-- Apply BannedAccessFilter to the servlet named
       "TodaysSpecial".
  -->
  <filter-mapping>
    <filter-name>BannedAccessFilter</filter-name>
    <servlet-name>TodaysSpecial</servlet-name>
  </filter-mapping>
  <!-- ... -->

  <!-- Give a name to the Today's Special servlet so that filters
       can be applied to it.
  -->
  <servlet>
    <servlet-name>TodaysSpecial</servlet-name>
    <servlet-class>
      moreservlets.TodaysSpecialServlet
    </servlet-class>
  </servlet>
  <!-- ... -->

  <!-- Make /TodaysSpecial invoke the servlet
       named TodaysSpecial (i.e., moreservlets.TodaysSpecial).
  -->
  <servlet-mapping>
    <servlet-name>TodaysSpecial</servlet-name>
    <url-pattern>/TodaysSpecial</url-pattern>
  </servlet-mapping>

  <!-- Turn off invoker. Send requests to index.jsp. -->
  <servlet-mapping>
    <servlet-name>Redirector</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- ... -->
</web-app>
```

Listing 9.13 *linker.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Link to Filter Company</TITLE>
</HEAD>
<BODY>
<H2 ALIGN="CENTER">Link to Filter Company</H2>
Click <A HREF="http://localhost/filters/TodaysSpecial">here</A>
to see the daily special at filtersRus.com.
</BODY>
</HTML>
```

9.9 Modifying the Response

OK, so filters can block access to resources or invoke them normally. But what if filters want to change the response that a resource generates? There don't appear to be any methods that provide access to the response that a resource generates. The second argument to `doFilter` (the `ServletResponse`) gives the filter a way to send new output to a client, but it doesn't give the filter access to the output of the servlet or JSP page. How could it? When the `doFilter` method is first invoked, the servlet or JSP page hasn't even executed yet. Once you call the `doFilter` method of the `FilterChain` object, it appears to be too late to modify the response—data has already been sent to the client. Hmm, a quandary.

The solution is to change the response object that is passed to the `doFilter` method of the `FilterChain` object. You typically create a version that buffers up all the output that the servlet or JSP page generates. The servlet 2.3 API provides a useful resource for this purpose: the `HttpServletResponseWrapper` class. Use of this class involves five steps:

1. **Create a response wrapper.** Extend `javax.servlet.http.HttpServletResponseWrapper`.
2. **Provide a `PrintWriter` that buffers output.** Override the `getWriter` method to return a `PrintWriter` that saves everything sent to it and stores that result in a field that can be accessed later.
3. **Pass that wrapper to `doFilter`.** This call is legal because `HttpServletResponseWrapper` implements `HttpServletResponse`.
4. **Extract and modify the output.** After the call to the `doFilter` method of the `FilterChain`, the output of the original resource is available to you through whatever mechanism you provided in Step 2. You can modify or replace it as appropriate for your application.

5. **Send the modified output to the client.** Since the original resource no longer sends output to the client (the output is stored in your response wrapper instead), *you* have to send the output. So, your filter needs to obtain the `PrintWriter` or `OutputStream` from the *original* response object and pass the modified output to that stream.

A Reusable Response Wrapper

Listing 9.14 presents a wrapper that can be used in most applications where you want filters to modify a resource's output. The `CharArrayWrapper` class overrides the `getWriter` method to return a `PrintWriter` that accumulates everything in a big char array. This result is available to the developer through the `toCharArray` (the raw `char[]`) or `toString` (a `String` derived from the `char[]`) method.

Sections 9.10 and 9.11 give two examples of use of this class.

Listing 9.14 *CharArrayWrapper.java*

```
package moreservlets.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


/** A response wrapper that takes everything the client
 *  would normally output and saves it in one big
 *  character array.
 */

public class CharArrayWrapper
             extends HttpServletResponseWrapper {
  private CharArrayWriter charWriter;

  /** Initializes wrapper.
   *  <P>
   *  First, this constructor calls the parent
   *  constructor. That call is crucial so that the response
   *  is stored and thus setHeader, setStatus, addCookie,
   *  and so forth work normally.
   *  <P>
   *  Second, this constructor creates a CharArrayWriter
   *  that will be used to accumulate the response.
   */
```

Listing 9.14 *CharArrayWrapper.java (continued)*

```
  public CharArrayWrapper(HttpServletResponse response) {
    super(response);
    charWriter = new CharArrayWriter();
  }

  /** When servlets or JSP pages ask for the Writer,
   *  don't give them the real one. Instead, give them
   *  a version that writes into the character array.
   *  The filter needs to send the contents of the
   *  array to the client (perhaps after modifying it).
   */

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

  /** Get a String representation of the entire buffer.
   *  <P>
   *  Be sure <B>not</B> to call this method multiple times
   *  on the same wrapper. The API for CharArrayWriter
   *  does not guarantee that it "remembers" the previous
   *  value, so the call is likely to make a new String
   *  every time.
   */

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

  /** Get the underlying character array. */

  public char[] toCharArray() {
    return(charWriter.toCharArray());
  }
}
```

9.10 Example: A Replacement Filter

This section presents one common application of the `CharArrayWrapper` shown in the previous section: a filter that changes all occurrences of a target string to some replacement string.

A Generic Replacement Filter

Listing 9.15 presents a filter that wraps the response in a `CharArrayWrapper`, passes that wrapper to the `doFilter` method of the `FilterChain` object, extracts a `String` that represents all of the resource's output, replaces all occurrences of a target string with a replacement string, and sends that modified result to the client.

There are two things to note about this filter. First, it is an abstract class. To use it, you must create a subclass that provides implementations of the `getTargetString` and `getReplacementString` methods. The next subsection has an example of this process. Second, it uses a small utility class (Listing 9.16) to do the actual string substitution. If you are fortunate enough to be using JDK 1.4, you can use the new regular expression package instead of the low-level and cumbersome methods in the `String` and `StringTokenizer` classes. For details, see *http://java.sun.com/j2se/1.4/docs/api/java/util/regex/Matcher.html* and *http://java.sun.com/j2se/1.4/docs/api/java/util/regex/Pattern.html*. Just remember that use of this package limits portability; the servlet 2.3 specification mandates the Java 2 Platform but does not specify any particular JDK version within that general umbrella.

Listing 9.15 *ReplaceFilter.java*

```
package moreservlets.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Filter that replaces all occurrences of a given
 *  string with a replacement. This is an abstract class:
 *  you <I>must</I> override the getTargetString and
 *  getReplacementString methods in a subclass. The
 *  first of these methods specifies the string in
 *  the response that should be replaced. The second
 *  of these specifies the string that should replace
 *  each occurrence of the target string.
 */

public abstract class ReplaceFilter implements Filter {
  private FilterConfig config;

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

Listing 9.15 *ReplaceFilter.java (continued)*

```
    CharArrayWrapper responseWrapper =
      new CharArrayWrapper((HttpServletResponse)response);
    // Invoke resource, accumulating output in the wrapper.
    chain.doFilter(request,responseWrapper);
    // Turn entire output into one big String.
    String responseString = responseWrapper.toString();
    // In output, replace all occurrences of target string
    // with replacement string.
    responseString =
      FilterUtils.replace(responseString,
                          getTargetString(),
                          getReplacementString());
    // Update the Content-Length header.
    updateHeaders(response, responseString);
    PrintWriter out = response.getWriter();
    out.write(responseString);
  }

  /** Store the FilterConfig object in case subclasses
   *  want it.
   */

  public void init(FilterConfig config)
      throws ServletException {
    this.config = config;
  }

  protected FilterConfig getFilterConfig() {
    return(config);
  }

  public void destroy() {}

  /** The string that needs replacement.
   *  Override this method in your subclass.
   */

  public abstract String getTargetString();

  /** The string that replaces the target.
   *  Override this method in your subclass.
   */

  public abstract String getReplacementString();
```

Listing 9.15 *ReplaceFilter.java (continued)*

```
  /** Updates the response headers. This simple version just sets
   *  the Content-Length header, assuming that we are using a
   *  character set that uses 1 byte per character. For other
   *  character sets, override this method to use different logic
   *  or to give up on persistent HTTP connections. In this latter
   *  case, have this method set the Connection header to "close".
   */

  public void updateHeaders(ServletResponse response,
                            String responseString) {
    response.setContentLength(responseString.length());
  }
}
```

Listing 9.16 *FilterUtils.java*

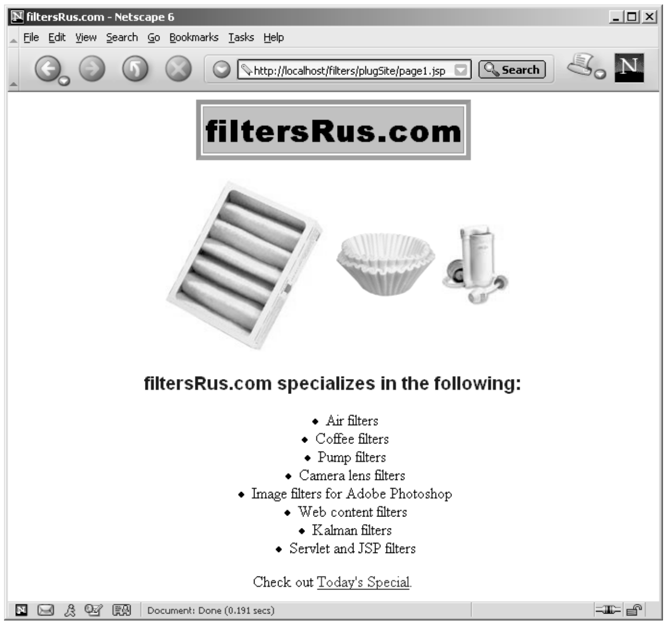
```
package moreservlets.filters;

/** Small utility to assist with response wrappers that
 *  return strings.
 */

public class FilterUtils {

  /** Change all occurrences of orig in mainString to
   *  replacement.
   */

  public static String replace(String mainString,
                               String orig,
                               String replacement) {
    String result = "";
    int oldIndex = 0;
    int index = 0;
    int origLength = orig.length();
    while((index = mainString.indexOf(orig, oldIndex))
          != -1) {
      result = result +
               mainString.substring(oldIndex, index) +
               replacement;
      oldIndex = index + origLength;
    }
    result = result + mainString.substring(oldIndex);
    return(result);
  }
}
```

A Specific Replacement Filter

Oh no! A competitor bought out filtersRus.com. All the Web pages that refer to the company name are now obsolete. But, the developers hate to change all their Web pages since another takeover could occur anytime (this company is a hot commodity, after all). No problem—Listing 9.17 presents a filter that replaces all occurrences of `filtersRus.com` with `weBefilters.com`. Figure 9–7 shows a page (Listing 9.19) that promotes the filtersRus.com site name. Figure 9–8 shows the page after the filter is applied.

To implement this functionality, the filter has the following capabilities.

1. **A class that implements the `Filter` interface.** This class is called `ReplaceSiteNameFilter` and is shown in Listing 9.17. It extends the generic `ReplaceFilter` of Listing 9.15. The inherited `init` method stores the `FilterConfig` object in a field in case subclasses need access to the servlet context or filter name. The parent class also provides an empty body for the `destroy` method.
2. **A wrapped response object.** The `doFilter` method, inherited from `ReplaceFilter`, wraps the `ServletResponse` object in a `CharArrayWrapper` and passes that wrapper to the `doFilter` method of the `FilterChain` object. After this call completes, all other filters and the final resource have executed and the output is inside the wrapper. So, the original `doFilter` extracts a `String` that represents all of the resource's output and replaces all occurrences of the target string with the replacement string. Finally, `doFilter` sends that modified result to the client by supplying the entire `String` to the `write` method of the `PrintWriter` that is associated with the *original* response.
3. **Registration with the JSP page that promotes filtersRus.com.** First, the `filter` element of *web.xml* (Listing 9.18) associates the name `ReplaceSiteNameFilter` with the class `moreservlets.filters.ReplaceSiteNameFilter`. Next, the `filter-mapping` element uses a `url-pattern` of `/plugSite/page2.jsp` (see Listing 9.19) so that the filter fires each time that JSP page is requested.
4. **Disablement of the invoker servlet.** This operation is shown in Section 9.2 and is not repeated here.


filtersRus.com - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://localhost/filters/plugSite/page1.jsp
Search
Document: Done (0.191 secs)

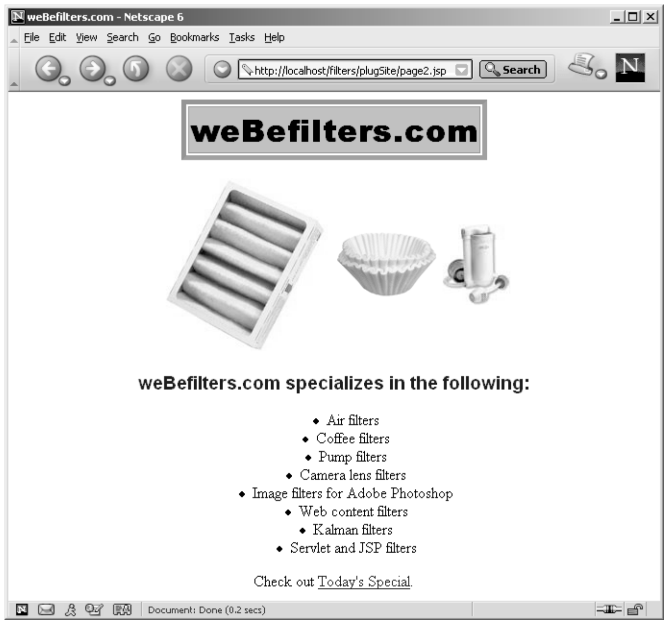
filtersRus.com

filtersRus.com specializes in the following:

- Air filters
- Coffee filters
- Pump filters
- Camera lens filters
- Image filters for Adobe Photoshop
- Web content filters
- Kalman filters
- Servlet and JSP filters

Check out Today's Special.

Figure 9–7 A page that promotes the *filtersRus.com* site.


weBefilters.com - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://localhost/filters/plugSite/page2.jsp
Search
Document: Done (0.2 secs)

weBefilters.com

weBefilters.com specializes in the following:

- Air filters
- Coffee filters
- Pump filters
- Camera lens filters
- Image filters for Adobe Photoshop
- Web content filters
- Kalman filters
- Servlet and JSP filters

Check out Today's Special.

Figure 9–8 The page that promotes the *filtersRus.com* site after its output is modified by the `ReplaceSiteNameFilter`.

Listing 9.17 *ReplaceSiteNameFilter.java*

```
package moreservlets.filters;

public class ReplaceSiteNameFilter extends ReplaceFilter {
  public String getTargetString() {
    return("filtersRus.com");
  }

  public String getReplacementString() {
    return("weBefilters.com");
  }
}
```

Listing 9.18 *web.xml* (Excerpt for site name replacement filter)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->

  <!-- Register the name "ReplaceSiteNameFilter" for
       moreservlets.filters.ReplaceSiteNameFilter.
  -->
  <filter>
    <filter-name>ReplaceSiteNameFilter</filter-name>
    <filter-class>
      moreservlets.filters.ReplaceSiteNameFilter
    </filter-class>
  </filter>
  <!-- ... -->


  <!-- Apply ReplaceSiteNameFilter to page2.jsp page
       in the plugSite directory
  -->
  <filter-mapping>
    <filter-name>ReplaceSiteNameFilter</filter-name>
    <url-pattern>/plugSite/page2.jsp</url-pattern>
  </filter-mapping>

  <!-- ... -->
</web-app>
```

Listing 9.19 *page1.jsp* (Identical to *page2.jsp*)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>filtersRus.com</TITLE>
<LINK REL=STYLESHEET
      HREF="../filter-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">filtersRus.com</TABLE>
<P>
<TABLE>
  <TR>
    <TH><IMG SRC="../images/air-filter.jpg"
             ALT="Air Filter">
    <TH><IMG SRC="../images/coffee-filter.gif"
             ALT="Coffee Filter">
    <TH><IMG SRC="../images/pump-filter.jpg"
             ALT="Pump Filter">
</TABLE>

<H3>filtersRus.com specializes in the following:</H3>
<UL>
  <LI>Air filters
  <LI>Coffee filters
  <LI>Pump filters
  <LI>Camera lens filters
  <LI>Image filters for Adobe Photoshop
  <LI>Web content filters
  <LI>Kalman filters
  <LI>Servlet and JSP filters
</UL>
Check out <A HREF="../TodaysSpecial">Today's Special</A>.
</CENTER>
</BODY>
</HTML>
```

9.11 Example: A Compression Filter

Several recent browsers can handle gzipped content, automatically uncompressing documents that have `gzip` as the value of the `Content-Encoding` response header and then treating the result as though it were the original document. Sending such compressed content can be a real time saver because the time required to compress the document on the server and then uncompress it on the client is typically dwarfed by the savings in download time, especially when dialup connections are used. For example, Listing 9.21 shows a servlet that has very long, repetitive, plain text output: a ripe candidate for compression. If gzip could be applied, it could compress the output by a factor of over 300!

However, although most browsers support this type of encoding, a fair number do not. Sending compressed content to browsers that don't support gzip encoding results in a totally garbled result. Browsers that support content encoding include most versions of Netscape for Unix, most versions of Internet Explorer for Windows, and Netscape 4.7 and later for Windows. So, this compression cannot be done blindly—it is only valid for clients that use the `Accept-Encoding` request header to specify that they support gzip.

A compression filter can use the `CharArrayWrapper` of Section 9.9 to compress content when the browser supports such a capability. Accomplishing this task requires the following:

1. **A class that implements the `Filter` interface.** This class is called `CompressionFilter` and is shown in Listing 9.20. The `init` method stores the `FilterConfig` object in a field in case subclasses need access to the servlet context or filter name. The body of the `destroy` method is left empty.
2. **A wrapped response object.** The `doFilter` method wraps the `ServletResponse` object in a `CharArrayWrapper` and passes that wrapper to the `doFilter` method of the `FilterChain` object. After this call completes, all other filters and the final resource have executed and the output is inside the wrapper. So, the original `doFilter` extracts a character array that represents all of the resource's output. If the client indicates that it supports compression (i.e., has `gzip` as one of the values of its `Accept-Encoding` header), the filter attaches a `GZIPOutputStream` to a `ByteArrayOutputStream`, copies the character array into that stream, and sets the `Content-Encoding` response header to `gzip`. If the client does not support gzip, the unmodified character array is copied to the `ByteArrayOutput-`

Stream. Finally, doFilter sends that result to the client by writing the entire byte array (possibly compressed) to the OutputStream that is associated with the *original* response.

3. **Registration with long servlet.** First, the filter element of *web.xml* (Listing 9.22) associates the name CompressionFilter with the class moreservlets.filters.CompressionFilter. Next, the filter-mapping element uses a servlet-name of LongServlet so that the filter fires each time that long servlet (Listing 9.21) is requested. The servlet and servlet-mapping elements assign the name LongServlet to the servlet and specify the URL that corresponds to the servlet.
4. **Disablement of the invoker servlet.** This operation is shown in Section 9.2 and is not repeated here.

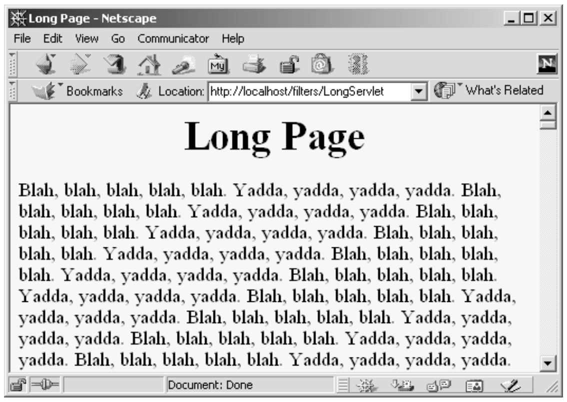
When the filter is attached, the body of the servlet is reduced three *hundred* times and the time to access the servlet on a 28.8K modem is reduced by more than a factor of *ten* (more than 50 seconds uncompressed; less than 5 seconds compressed). A huge savings! However, two small warnings are in order here.

First, there is a saying in the software industry that there are three kinds of lies: lies, darn lies, and benchmarks. The point of this maxim is that people always rig benchmarks to show their point in the most favorable light possible. I did the same thing by using a servlet with long simple output and using a slow modem connection. So, I'm not promising that you will always get a tenfold performance gain. But, it is a simple matter to attach or detach the compression filter. That's the beauty of filters. Try it yourself and see how much it buys you in typical usage conditions.

Second, although the specification does not officially mandate that you set response headers before calling the doFilter method of the FilterChain, some servers (e.g., ServletExec 4.1) require you to do so. This is to prevent you from attempting to set a response header after a resource has sent content to the client. So, for portability, be sure to set response headers before calling chain.doFilter.

Core Warning

If your filter sets response headers, be sure it does so before calling the doFilter method of the FilterChain object.


Long Page - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://localhost/filters/LongServlet
What's Related
Document: Done

Long Page

Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda. Blah, blah, blah, blah, blah. Yadda, yadda, yadda, yadda.

Figure 9–9 The `LongServlet`. The content is more than three hundred times smaller when gzip is used, resulting in more than a tenfold speedup when the servlet is accessed with a 28.8K modem.

Listing 9.20 *CompressionFilter.java*

```
package moreservlets.filters;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.zip.*;

/** Filter that compresses output with gzip
 *  (assuming that browser supports gzip).
 */

public class CompressionFilter implements Filter {
  private FilterConfig config;

  /** If browser does not support gzip, invoke resource
   *  normally. If browser <I>does</I> support gzip,
   *  set the Content-Encoding response header and
   *  invoke resource with a wrapped response that
   *  collects all the output. Extract the output
   *  and write it into a gzipped byte array. Finally,
   *  write that array to the client's output stream.
   */
```

Listing 9.20 *CompressionFilter.java (continued)*

```
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
      throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
    HttpServletResponse res = (HttpServletResponse)response;
    if (!isGzipSupported(req)) {
      // Invoke resource normally.
      chain.doFilter(req,res);
    } else {
      // Tell browser we are sending it gzipped data.
      res.setHeader("Content-Encoding", "gzip");

      // Invoke resource, accumulating output in the wrapper.
      CharArrayWrapper responseWrapper =
        new CharArrayWrapper(res);
      chain.doFilter(req,responseWrapper);

      // Get character array representing output.
      char[] responseChars = responseWrapper.toCharArray();

      // Make a writer that compresses data and puts
      // it into a byte array.
      ByteArrayOutputStream byteStream =
        new ByteArrayOutputStream();
      GZIPOutputStream zipOut =
        new GZIPOutputStream(byteStream);
      OutputStreamWriter tempOut =
        new OutputStreamWriter(zipOut);

      // Compress original output and put it into byte array.
      tempOut.write(responseChars);

      // Gzip streams must be explicitly closed.
      tempOut.close();

      // Update the Content-Length header.
      res.setContentLength(byteStream.size());

      // Send compressed result to client.
      OutputStream realOut = res.getOutputStream();
      byteStream.writeTo(realOut);
    }
  }
```

Listing 9.20 *CompressionFilter.java (continued)*

```
  /** Store the FilterConfig object in case subclasses
   *  want it.
   */

  public void init(FilterConfig config)
      throws ServletException {
    this.config = config;
  }

  protected FilterConfig getFilterConfig() {
    return(config);
  }

  public void destroy() {}

  private boolean isGzipSupported(HttpServletRequest req) {
    String browserEncodings =
      req.getHeader("Accept-Encoding");
    return((browserEncodings != null) &&
           (browserEncodings.indexOf("gzip") != -1));
  }
}
```

Listing 9.21 *LongServlet.java*

```
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet with <B>long</B> output. Used to test
 *  the effect of the compression filter of Chapter 9.
 */

public class LongServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```

Listing 9.21 *LongServlet.java (continued)*

```
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Long Page";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n");
    String line = "Blah, blah, blah, blah, blah. " +
                  "Yadda, yadda, yadda, yadda.";
    for(int i=0; i<10000; i++) {
      out.println(line);
    }
    out.println("</BODY></HTML>");
  }
}
```

Listing 9.22 *web.xml* (Excerpt for compression filter)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->

  <!-- Register the name "CompressionFilter" for
       moreservlets.filters.CompressionFilter.
  -->
  <filter>
    <filter-name>CompressionFilter</filter-name>
    <filter-class>
      moreservlets.filters.CompressionFilter
    </filter-class>
  </filter>
  <!-- ... -->
```

Listing 9.22 *web.xml* (Excerpt for compression filter) *(continued)*

```
  <!-- Apply CompressionFilter to the servlet named
       "LongServlet".
  -->
  <filter-mapping>
    <filter-name>CompressionFilter</filter-name>
    <servlet-name>LongServlet</servlet-name>
  </filter-mapping>
  <!-- ... -->

  <!-- Give a name to the servlet that generates long
       (but very exciting!) output.
  -->
  <servlet>
    <servlet-name>LongServlet</servlet-name>
    <servlet-class>moreservlets.LongServlet</servlet-class>
  </servlet>
  <!-- ... -->

  <!-- Make /LongServlet invoke the servlet
       named LongServlet (i.e., moreservlets.LongServlet).
  -->
  <servlet-mapping>
    <servlet-name>LongServlet</servlet-name>
    <url-pattern>/LongServlet</url-pattern>
  </servlet-mapping>

  <!-- Turn off invoker. Send requests to index.jsp. -->
  <servlet-mapping>
    <servlet-name>Redirector</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- ... -->
</web-app>
```

9.12 The Complete Filter Deployment Descriptor

The previous sections showed various excerpts of the *web.xml* file for filtersRus.com. This section shows the file in its entirety.

Listing 9.23 *web.xml* (Complete version for filter examples)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- Order matters in web.xml! For the elements
       used in this example, this order is required:
          filter
          filter-mapping
          servlet
          servlet-mapping
          welcome-file-list
  -->

  <!-- Register the name "Reporter" for ReportFilter. -->
  <filter>
    <filter-name>Reporter</filter-name>
    <filter-class>
      moreservlets.filters.ReportFilter
    </filter-class>
  </filter>

  <!-- Register the name "Logger" for LogFilter. -->
  <filter>
    <filter-name>Logger</filter-name>
    <filter-class>
      moreservlets.filters.LogFilter
    </filter-class>
  </filter>

  <!-- Register the name "LateAccessFilter" for
       moreservlets.filter.LateAccessFilter.
       Supply two initialization parameters:
       startTime and endTime.
  -->
  <filter>
    <filter-name>LateAccessFilter</filter-name>
    <filter-class>
      moreservlets.filters.LateAccessFilter
    </filter-class>
    <init-param>
      <param-name>startTime</param-name>
      <param-value>2</param-value>
    </init-param>
```

Listing 9.23 *web.xml* (Complete version for filter examples) *(continued)*

```
    <init-param>
      <param-name>endTime</param-name>
      <param-value>10</param-value>
    </init-param>
  </filter>

  <!-- Register the name "BannedAccessFilter" for
       moreservlets.filter.BannedAccessFilter.
       Supply an initialization parameter:
       bannedSites.
  -->
  <filter>
    <filter-name>BannedAccessFilter</filter-name>
    <filter-class>
      moreservlets.filters.BannedAccessFilter
    </filter-class>
    <init-param>
      <param-name>bannedSites</param-name>
      <param-value>
        www.competingsite.com
        www.bettersite.com
        www.moreservlets.com
      </param-value>
    </init-param>
  </filter>

  <!-- Register the name "ReplaceSiteNameFilter" for
       moreservlets.filters.ReplaceSiteNameFilter.
  -->
  <filter>
    <filter-name>ReplaceSiteNameFilter</filter-name>
    <filter-class>
      moreservlets.filters.ReplaceSiteNameFilter
    </filter-class>
  </filter>

  <!-- Register the name "CompressionFilter" for
       moreservlets.filters.CompressionFilter.
  -->
  <filter>
    <filter-name>CompressionFilter</filter-name>
    <filter-class>
      moreservlets.filters.CompressionFilter
    </filter-class>
  </filter>
```

Listing 9.23 *web.xml* (Complete version for filter examples) *(continued)*

```
<!-- Apply the Reporter filter to the servlet named
    "TodaysSpecial".
-->
<filter-mapping>
  <filter-name>Reporter</filter-name>
  <servlet-name>TodaysSpecial</servlet-name>
</filter-mapping>

<!-- Also apply the Reporter filter to home page. -->
<filter-mapping>
  <filter-name>Reporter</filter-name>
  <url-pattern>/index.jsp</url-pattern>
</filter-mapping>

<!-- Apply the Logger filter to all servlets and
     JSP pages.
-->
<filter-mapping>
  <filter-name>Logger</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- Apply LateAccessFilter to the home page. -->
<filter-mapping>
  <filter-name>LateAccessFilter</filter-name>
  <url-pattern>/index.jsp</url-pattern>
</filter-mapping>

<!-- Apply BannedAccessFilter to the servlet named
    "TodaysSpecial".
-->
<filter-mapping>
  <filter-name>BannedAccessFilter</filter-name>
  <servlet-name>TodaysSpecial</servlet-name>
</filter-mapping>

<!-- Apply ReplaceSiteNameFilter to page2.jsp page
     in the plugSite directory
-->
<filter-mapping>
  <filter-name>ReplaceSiteNameFilter</filter-name>
  <url-pattern>/plugSite/page2.jsp</url-pattern>
</filter-mapping>
```

Listing 9.23 *web.xml* (Complete version for filter examples) *(continued)*

```
<!-- Apply CompressionFilter to the servlet named
    "LongServlet".
-->
<filter-mapping>
  <filter-name>CompressionFilter</filter-name>
  <servlet-name>LongServlet</servlet-name>
</filter-mapping>

<!-- Give a name to the Today's Special servlet so that filters
     can be applied to it.
-->
<servlet>
  <servlet-name>TodaysSpecial</servlet-name>
  <servlet-class>
    moreservlets.TodaysSpecialServlet
  </servlet-class>
</servlet>

<!-- Give a name to the servlet that redirects users
     to the home page.
-->
<servlet>
  <servlet-name>Redirector</servlet-name>
  <servlet-class>moreservlets.RedirectorServlet</servlet-class>
</servlet>

<!-- Give a name to the servlet that generates long
     (but very exciting!) output.
-->
<servlet>
  <servlet-name>LongServlet</servlet-name>
  <servlet-class>moreservlets.LongServlet</servlet-class>
</servlet>

<!-- Make /TodaysSpecial invoke the servlet
     named TodaysSpecial (i.e., moreservlets.TodaysSpecial).
-->
<servlet-mapping>
  <servlet-name>TodaysSpecial</servlet-name>
  <url-pattern>/TodaysSpecial</url-pattern>
</servlet-mapping>
```

Listing 9.23 *web.xml* (Complete version for filter examples) *(continued)*

```
  <!-- Make /LongServlet invoke the servlet
       named LongServlet (i.e., moreservlets.LongServlet).
  -->
  <servlet-mapping>
    <servlet-name>LongServlet</servlet-name>
    <url-pattern>/LongServlet</url-pattern>
  </servlet-mapping>

  <!-- Turn off invoker. Send requests to index.jsp. -->
  <servlet-mapping>
    <servlet-name>Redirector</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- If URL gives a directory but no filename, try index.jsp
       first and index.html second. If neither is found,
       the result is server specific (e.g., a directory
       listing).  Order of elements in web.xml matters.
       welcome-file-list needs to come after servlet but
       before error-page.
  -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

</web-app>
```