The Servlet API has long been the cornerstone of enterprise application development, but Servlet filters are a relatively new addition to the J2EE family. In this final article in the J2EE pathfinder series, author Kyle Gabhart introduces you to the Servlet filter architecture, defines the many uses of filters, and walks you through the three steps of a typical filter implementation. He also spills the beans on some of the exciting changes you can expect from the just-released Java Servlet 2.4 specification.

What are Servlet filters?

Servlet filters are small Web components that intercept requests and responses to view, extract, or in some way manipulate the data that is being exchanged between client and server. Filters are Web components that typically encapsulate some functionality that, while important, is not central to processing the client request or sending a response. Typical examples include logging data about the request or response, processing security protocols, managing session attributes, and more. Filters provide a modular, object-oriented mechanism for encapsulating common tasks into pluggable components that are declared via a configuration file and processed dynamically.

Many elements combine in Servlet filters to make them unique, powerful, and modular Web components. Namely, Servlet filters are:
• **Declarative:** Filters are declared via XML tags in the Web deployment descriptor (web.xml). This allows filters to be added and removed without touching any application code or JSP pages.

• **Dynamic:** At runtime, filters are invoked by the Servlet container to intercept and process requests and responses.

• **Flexible:** The application of filters to a Web processing environment is broad, covering many of the most common auxiliary's tasks such as logging and security. Filters are also flexible because they can be used to perform pre- and post-processing on calls directly from clients, as well as handling dispatched requests between Web components behind the firewall. Finally, filters can be chained together to provide required functionality.

• **Modular:** By encapsulating application-processing logic into a single class file, filters define modular units that can be easily added to and removed from a request/response chain.

• **Portable:** As with so many aspects of the Java platform, Servlet filters are portable across platforms and containers, further bolstering the modular and reusable qualities of Servlet filters.

• **Reusable:** Thanks to the modular design of a filter's implementation class, and the declarative way that a filter is configured, filters can easily be used across projects and applications.

• **Transparent:** The inclusion of a filter within a request/response chain is designed to supplement, but in no way replace, the core processing provided by a servlet or JSP page. Thus, filters can be added or removed as necessary without breaking the servlet or JSP page.

So, Servlet filters are modular, reusable components that are flexibly declared via a configuration file. Filters process incoming requests and outgoing responses dynamically and can be transparently added and removed without modifying application code. Finally, filters are independent of any platform or Servlet container, allowing them to be easily deployed in any compliant J2EE environment.

In the sections that follow, we will take a closer look at the overall design of the Servlet filter mechanism and the steps involved in implementing, configuring, and deploying filters. We'll also explore some of the practical uses of Servlet filters, and conclude with a brief look at the inclusion of Servlet filters within a Model-View-Controller architecture.

### The Servlet filter architecture

As the name suggests, a Servlet filter is used to intercept an incoming request and/or outgoing response, and monitor, modify, or in some way process the stream of data that is passing through. Filters are self-contained, modular components that can be added to a request/response chain, or removed without affecting the other Web components in the application. Filters simply alter the runtime processing of requests and responses, and thus should not be tied directly into the Web application framework, except through well-defined standard interfaces within the Servlet API.

A Web resource can be configured to have no filters associated with it (default), a single filter (typical), or even a chain of filters. So what does a filter do? Like a servlet, it receives a request and response object. The filter will then inspect the request object and decide to forward the request to the next component in the chain, or stop the request and send a response directly back to the client. If the request is forwarded, it is passed to the next resource in the chain (another filter,
a servlet, or a JSP page). After the request works its way through the chain and is processed by the server, a response is sent back through the chain in reverse order. This gives each filter an opportunity to handle the response object if necessary.

When filters were originally introduced in the Servlet 2.3 spec, they could only filter content between the Web client and the specified Web resource the client was accessing. If that resource was then to dispatch the request to other Web resources, a filter could not be applied to any requests that were delegated behind the scenes. With the 2.4 spec, this restriction has been removed. Servlet filters can now be applied anywhere that a request and response object exist within a J2EE Web environment. So, Servlet filters can be applied between the client and servlet, the servlet and a servlet or JSP page, and between each included JSP page as well. Now that's what I call power and flexibility!

Implementing a Servlet filter

They say that all good things come in threes. I don't know who "they" are, or how much truth there is to this old adage, but there are three steps to implementing a Servlet filter. First, you program the filter implementation class, second add the filter to your Web application (by declaring it within the Web deployment descriptor/web.xml), and, finally, package the application with the filter and deploy it. We'll go over each of these steps in detail.

1. Programming the implementation class

The filter API consists of three simple interfaces (there's the number three again!) nestled snugly within the `javax.servlet` package. Those three interfaces are `Filter`, `FilterChain`, and `FilterConfig`. From a programming standpoint, your filter class will implement the `Filter` interface and then use the `FilterChain` and `FilterConfig` interfaces within your filter class. Your filter class will be passed a reference to a `FilterChain` object to allow the filter to pass control to the next resource in the chain. The `FilterConfig` object will be supplied to the filter by the container to provide access to initialization data for the filter.

In keeping with our pattern of threes, a filter must apply three methods in order to fully implement the `Filter` interface:

- `init()`: This method is called when the filter is instantiated by the container and is designed to prep the filter for processing. The method accepts as input an object of type `FilterConfig`.
- `doFilter()`: In the same way that servlets have a `service()` method (which in turn calls `doPost()` or `doGet()` to handle requests, filters have a single method for processing requests and responses -- `doFilter()`). This method accepts three input parameters: a `ServletRequest`, a `response`, and a `FilterChain` object.
- `destroy()`: As you would expect, this method performs any cleanup operations on the class that may need to take place prior to automatic garbage collection.

Listing 1 demonstrates a very simple filter that tracks the approximate length of time taken to fulfill a client's Web request:

### Listing 1. A filter class implementation

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

public class TimeTrackFilter implements Filter {
    private FilterConfig filterConfig = null;
    
    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
    }
    
    public void destroy() {
        this.filterConfig = null;
    }
    
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
        Date startTime, endTime;
        double totalTime;
        startTime = new Date();
        // Forward the request to the next resource in the chain
        chain.doFilter(request, wrapper);
        // -- Process the response -- \
        // Calculate the difference between the start time and end time
        endTime = new Date();
        totalTime = endTime.getTime() - startTime.getTime();
        totalTime = totalTime / 1000; // Convert from milliseconds to seconds
        StringWriter sw = new StringWriter();
        PrintWriter writer = new PrintWriter(sw);
        writer.println();
        writer.println("=*=*=*=*=*=*=*");
        writer.println("Total elapsed time is: "+totalTime+" seconds.");
        writer.println("=*=*=*=*=*=*=*");
        // Log the resulting string
        writer.flush();
        filterConfig.getServletContext().log(sw.getBuffer().toString());
    }
}

This filter's lifecycle is pretty straightforward, but let's go over it together anyway:

**Initialization**

When the container loads the filter for the first time, the `init()` method is called. The class obtains a reference to a `FilterConfig` object in this method. Our filter doesn't actually need to do this, as no initialization info is being used, but it is here for demonstration purposes.

**Filtering**

This is where the majority of the filter's life is spent. The `doFilter()` method is called by the container, passing in references to the `ServletRequest`, `ServletResponse`, and `FilterChain` objects for this request/response chain. The filter then has the opportunity to process the
request, pass processing on to the next resource in the chain (by calling `doFilter()` on the `FilterChain` object reference), and then process the response once processing control returns to the filter.

**Destruction**

The container calls the `destroy()` method just prior to garbage collection, so that any cleanup code required can be executed.

### 2. Configuring the Servlet filter

Filters are declared via two XML tags within the `web.xml` file. The `<filter>` tag defines a name for the filter and declares the implementation class and `init()` parameters. The `<filter-mapping>` tag associates a filter with a servlet or URL pattern.

Listing 2, a snapshot from a `web.xml` file, shows how to declare the inclusion of a filter:

#### Listing 2. Declaring a filter within `web.xml`

```xml
<filter>
    <filter-name>Page Request Timer</filter-name>
    <filter-class>TimeTrackFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>Page Request Timer</filter-name>
    <servlet-name>Main Servlet</servlet-name>
</filter-mapping>

<servlet>
    <servlet-name>Main Servlet</servlet-name>
    <servlet-class>MainServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Main Servlet</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

In the above code sample, a filter ("Page Request Timer") is declared and mapped to a servlet ("Main Servlet"). A mapping is then defined for the servlet so that every request (indicated by the wildcard) should be sent to that servlet. This is a typical mapping declaration for a controller component. You should note the order of these declarations, as it is imperative that you not deviate from this ordering of elements.

### 3. Deploying a Servlet filter

The truth is, there is absolutely no complexity involved in deploying your filters along with your Web application. Simply include the filter classes alongside your other Web component classes and place the `web.xml` -- complete with filter definitions and filter mapping declarations -- within the Web app structure as you would normally (at the root of the WEB-INF folder), and the servlet container will handle everything from there!

### The many uses of filters

Your ability to utilize filters within your J2EE Web applications is limited only by your own creativity and application design prowess. Anywhere that a decorating filter pattern or interceptor pattern
would be appropriate, you can use a filter. Some of the most common uses for filters are as follows:

- **Logging**: The filter gleans information such as browser type, time of day, forwarding URL, etc. about all requests coming through the system and logs them.

- **Performance**: The filter decompresses content as it comes across the wire before it hits the servlets and JSP pages, and then takes the response content and converts it into a compressed format before sending it on to the client machine.

- **Security**: The filter handles management of authentication tokens and properly restricts access to secure resources, prompting the user for authentication and/or passing them off to a third party for authentication. A filter could even manage an Access Control List to provide authorization in addition to authentication. Placing security logic into a filter rather than a servlet or JSP page provides tremendous flexibility. During development, the filter can be turned off (comment out of web.xml). In production, the filter is turned back on. Also, multiple filters can be added to provide increasing levels of security, encryption, and non-repudiation services as necessary.

- **Session-handling**: Littering your servlets and JSP pages with session-handling code can add up to quite a hassle. Using a filter to manage your session data lets your Web pages focus on displaying content and delegating processing, without worrying about the details of session management.

- **XSLT transformation**: Whether you are working with a mobile client or an XML-based Web service, the ability to translate between XML grammars without embedding the logic into your application is absolutely priceless.

### Fitting filters into an MVC architecture

The Model-View-Controller (MVC) architecture is an effective design that has now been incorporated as the overriding design methodology within the most popular Web application frameworks such as Jakarta Struts and Turbine. Filters serve to augment the request/response processing flow of an MVC architecture. Whether the request/response is between the client and server or between other components on the server, the application of filters in the process flow is the same. From an MVC perspective, the dispatcher component (which is either included in or works in conjunction with the Controller component) forwards requests to the appropriate application component for processing. This makes the Controller layer the optimum location for including Servlet filters. Filters can be applied to all requests by placing them in front of the Controller component itself, or applied to individual Web components by placing it between the controller/dispatcher and the Model and View components.

The MVC architecture is widespread and well-documented. Follow the links in the Related topics section to learn more about MVC and Servlet implementations within MVC architectures.

### Conclusion

While filters have been around for only a couple of years, they have embedded themselves as a critical component of any agile, object-oriented J2EE Web application. In this article, you've been introduced to working with Servlet filters. I've discussed the high-level design of filters, compared
the current (2.4) specification to the earlier (2.3) model, and described the exact steps involved in implementing a filter, declaring it within a Web app, and then deploying it with an application. I've also explained some of the most common uses for Servlet filters and touched on how filters fit into a traditional MVC architecture.

This is the final article in the J2EE pathfinder series. We began our journey at the beginning of the year with a hard look at Enterprise JavaBean components, asking when it truly makes sense to use them and when they are overkill. We then shifted our focus to the Web tier, charting a path through the myriad of options and capabilities within the Servlet, JSP page, JavaBean technology, and Java Servlet APIs. It has been a real pleasure trekking through this series of articles with you. I've enjoyed writing this series and I know from your feedback that it has been a valuable process for you as well. Thank you for being part of the series. I wish you good luck and happy pathfinding!
Related topics

- Sun's J2EE tutorial is always a good place to go for information on core J2EE technologies. To learn about Servlet filters, see the Filtering Requests and Responses section.
- Sing Li's "Taming your Tomcat: Filtering tricks for Tomcat 5" (*developerWorks*, March 2003) is an excellent article about defining Servlet filters in a Tomcat Web environment.
- To learn the basics of Servlet 2.3 filters, read "The Essentials of Filters" on java.sun.com.
- Jason Hunter's "Servlet 2.4: What's in store" (*JavaWorld*, March 2003) is a comprehensive preview of the changes due with the Java Servlet 2.4 specification.
- To go more in-depth with MVC and get a J2EE-specific perspective, study the excerpted guidelines from Sun Microsystems's "Designing Enterprise Applications with the J2EE Platform."