Java Servlet Technology

Shortly after the Web began to be used for delivering services, service providers recognized the need for dynamic content.

Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences.

At the same time, developers also investigated using the server platform for the same purpose.

Initially, Common Gateway Interface (CGI) server-side scripts were the main technology used to generate dynamic content.

Although widely used, CGI scripting technology had many shortcomings, including platform dependence and lack of scalability.

To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

The following topics are addressed here:

- What Is a Servlet?
- Servlet Lifecycle
- Sharing Information
- Creating and Initializing a Servlet
- Writing Service Methods
- Filtering Requests and Responses

- Invoking Other Web Resources
- Accessing the Web Context
- Maintaining Client State
- Finalizing a Servlet
- The mood Example Application
- Further Information about Java Servlet Technology

What Is a Servlet?

A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model.

Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The `javax.servlet` and `javax.servlet.http` packages provide interfaces and classes for writing servlets.

All servlets must implement the `Servlet` interface, which defines lifecycle methods.

When implementing a generic service, you can use or extend the `GenericServlet` class provided with the Java Servlet API.

The `HttpServlet` class provides methods, such as `doGet` and `doPost`, for handling HTTP-specific services.

Servlet Lifecycle

The lifecycle of a servlet is controlled by the container in which the servlet has been deployed.

When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the web container

a. Loads the servlet class.

b. Creates an instance of the servlet class.

c. Initializes the servlet instance by calling the `init` method.

Initialization is covered in Creating and Initializing a Servlet.

2. Invokes the `service` method, passing request and response objects.

Service methods are discussed in Writing Service Methods.

If it needs to remove the servlet, the container finalizes the servlet by calling the servlet's `destroy` method.

For more information, see Finalizing a Servlet.

Handling Servlet Lifecycle Events

You can monitor and react to events in a servlet's lifecycle by defining listener objects whose methods get invoked when lifecycle events occur.

To use these listener objects, you must define and specify the listener class.

Defining the Listener Class

You define a listener class as an implementation of a listener interface.

Table 15-1 lists the events that can be monitored and the corresponding interface that must be implemented.

When a listener method is invoked, it is passed an event that contains information appropriate to the event.

For example, the methods in the `HttpSessionListener` interface are passed an `HttpSessionEvent`, which contains an `HttpSession`.

Table 15-1 Servlet Lifecycle Events

Object	Event	Listener Interface and Event Class
Web context (see Accessing the Web Context)	Initialization and destruction	`javax.servlet.ServletContextListener` and `ServletContextEvent`
	Attribute added, removed, or replaced	`javax.servlet.ServletContextAttributeListener` and `ServletContextAttributeEvent`

Session (See Maintaining Client State)	Creation, invalidation, activation, passivation, and timeout	`javax.servlet.http.HttpSessionListener`, `javax.servlet.http.HttpSessionActivationListener`, and `HttpSessionEvent`
	Attribute added, removed, or replaced	`javax.servlet.http.HttpSessionAttributeListener` and `HttpSessionBindingEvent`

Request	A servlet request has started being processed by web components	`javax.servlet.` `ServletRequestListener` and `ServletRequestEvent`
	Attribute added, removed, or replaced	`javax.servlet.` `ServletRequestAttributeListener` and `ServletRequestAttributeEvent`

Use the `@WebListener` annotation to define a listener to get events for various operations on the particular web application context.

Classes annotated with @WebListener must implement one of the following interfaces:

- javax.servlet.ServletContextListener
- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener

javax.servlet.http.
HttpSessionListener
javax.servlet.http.
HttpSessionAttributeListener

For example, the following code snippet defines a listener that implements two of these interfaces:

```
import javax.servlet.
ServletContextAttributeListener;
```

```
import javax.servlet.
ServletContextListener;
import javax.servlet.annotation.
WebListener;
@WebListener()
public class SimpleServletListener
implements ServletContextListener,
ServletContextAttributeListener {
  ...
```

Handling Servlet Errors

Any number of exceptions can occur when a servlet executes.

When an exception occurs, the web container generates a default page containing the following message:

```
A Servlet Exception Has Occurred
```

But you can also specify that the container should return a specific error page for a given exception.

Sharing Information

Web components, like most objects, usually work with other objects to accomplish their tasks.

Web components can do so by

- Using private helper objects (for example, JavaBeans components).

- Sharing objects that are attributes of a public scope.

- Using a database.
- Invoking other web resources.

The Java Servlet technology mechanisms that allow a web component to invoke other web resources are described in Invoking Other Web Resources.

Using Scope Objects

Collaborating web components share information by means of objects that are maintained as attributes of four scope objects.

You access these attributes by using the `getAttribute` and `setAttribute` methods of the class representing the scope.
Table 15-2 lists the scope objects.

Table 15-2 Scope Objects

Scope Object	Class	Accessible from
Web context	`javax.servlet.ServletContext`	Web components within a web context. See Accessing the Web Context.
Session	`javax.servlet.http.HttpSession`	Web components handling a request that belongs to the session. See Maintaining Client State.
Request	Subtype of `javax.servlet.ServletRequest`	Web components handling the request.
Page	`javax.servlet.jsp.JspContext`	The JSP page that creates the object.

Controlling Concurrent Access to Shared Resources

In a multithreaded server, shared resources can be accessed concurrently.

In addition to scope object attributes, shared resources include in-memory data, such as instance or class variables, and external objects, such as files, database connections, and network connections.

Concurrent access can arise in several situations:

- Multiple web components accessing objects stored in the web context.
- Multiple web components accessing objects stored in a session.
- Multiple threads within a web component accessing instance variables.

A web container will typically create a thread to handle each request.

To ensure that a servlet instance handles only one request at a time, a servlet can implement the `SingleThreadModel` interface.

If a servlet implements this interface, no two threads will execute concurrently in the servlet's service method.

A web container can implement this guarantee by synchronizing access to a single instance of the servlet or by maintaining a pool of web component instances and dispatching each new request to a free instance.

This interface does not prevent synchronization problems that result from web components' accessing shared resources, such as static class variables or external objects.

When resources can be accessed concurrently, they can be used in an inconsistent fashion.

You prevent this by controlling the access using the synchronization techniques described in the Threads lesson at http://download.oracle.com/javase/tutorial/essential/concurrency/index.html in *The Java Tutorial, Fourth Edition*, by Sharon Zakhour et al. (Addison-Wesley, 2006).

Creating and Initializing a Servlet

Use the `@WebServlet` annotation to define a servlet component in a web application.

This annotation is specified on a class and contains metadata about the servlet being declared.

The annotated servlet must specify at least one URL pattern.

This is done by using the `urlPatterns` or `value` attribute on the annotation.

All other attributes are optional, with default settings.

Use the `value` attribute when the only attribute on the annotation is the URL pattern; otherwise use the `urlPatterns` attribute when other attributes are also used.

Classes annotated with `@WebServlet` must extend the `javax.servlet.http.HttpServlet` class.

For example, the following code snippet defines a servlet with the URL pattern /report:

```
import javax.servlet.annotation.
WebServlet;
import
javax.servlet.http.HttpServlet;
@WebServlet("/report")
public class MoodServlet extends
HttpServlet { ...
```

The web container initializes a servlet after loading and instantiating the servlet class and before delivering requests from clients.

To customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities, you can either override the `init` method of the `Servlet` interface or specify the `initParams` attribute of the `@WebServlet` annotation.

The `initParams` attribute contains a `@WebInitParam` annotation.

If it cannot complete its initialization process, a servlet throws an `UnavailableException`.

Writing Service Methods

The service provided by a servlet is implemented in the `service` method of a `GenericServlet`, in the `do`*`Method`* methods (where *Method* can take the value `Get`, `Delete`, `Options`, `Post`, `Put`, or `Trace`) of an `HttpServlet` object, or in any other protocol-specific methods defined by a class that implements the `Servlet` interface.

The term service method is used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response, based on that information.

For HTTP servlets, the correct procedure for populating the response is to do the following:

1. Retrieve an output stream from the response.
2. Fill in the response headers.
3. Write any body content to the output stream.

Response headers must always be set before the response has been committed.

The web container will ignore any attempt to set or add headers after the response has been committed.

The next two sections describe how to get information from requests and generate responses.

Getting Information from Requests

A request contains data passed between a client and the servlet.

All requests implement the `ServletRequest` interface.

This interface defines methods for accessing the following information:

- Parameters, which are typically used to convey information between clients and servlets
- Object-valued attributes, which are typically used to pass information between the servlet container and a servlet or between collaborating servlets
- Information about the protocol used to communicate the request and about the client and server involved in the request
- Information relevant to localization

You can also retrieve an input stream from the request and manually parse the data.

To read character data, use the `BufferedReader` object returned by the request's `getReader` method.

To read binary data, use the `ServletInputStream` returned by `getInputStream`.

HTTP servlets are passed an HTTP request object, HttpServletRequest, which contains the request URL, HTTP headers, query string, and so on.

An HTTP request URL contains the following parts:

```
http://[host]:[port]
[request-path]?[query-string]
```

The request path is further composed of the following elements:

- **Context path:** A concatenation of a forward slash (/) with the context root of the servlet's web application.

- **Servlet path:** The path section that corresponds to the component alias that activated this request.

This path starts with a forward slash (/).

- **Path info:** The part of the request path that is not part of the context path or the servlet path.

You can use the `getContextPath`, `getServletPath`, and `getPathInfo` methods of the `HttpServletRequest` interface to access this information.

Except for URL encoding differences between the request URI and the path parts, the request URI is always comprised of the context path plus the servlet path plus the path info.

Query strings are composed of a set of parameters and values.

Individual parameters are retrieved from a request by using the `getParameter` method.

There are two ways to generate query strings.

- A query string can explicitly appear in a web page.
- A query string is appended to a URL when a form with a `GET` HTTP method is submitted.

Constructing Responses

A response contains data passed between a server and the client.

All responses implement the `ServletResponse` interface.

This interface defines methods that allow you to

- Retrieve an output stream to use to send data to the client.

To send character data, use the `PrintWriter` returned by the response's `getWriter` method.

To send binary data in a Multipurpose Internet Mail Extensions (MIME) body response, use the `ServletOutputStream` returned by `getOutputStream`.

To mix binary and text data, as in a multipart response, use a `ServletOutputStream` and manage the character sections manually.

- Indicate the content type (for example, `text/html`) being returned by the response with the `setContentType(String)` method.

This method must be called before the response is committed.

A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at http://www.iana.org/assignments/media-types/.

- Indicate whether to buffer output with the `setBufferSize(int)` method.

By default, any content written to the output stream is immediately sent to the client.

Buffering allows content to be written before anything is sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource.

The method must be called before any content is written or before the response is committed.

- Set localization information, such as locale and character encoding.

HTTP response objects, `javax.servlet.http.HttpServletResponse`, have fields representing HTTP headers, such as the following:

- Status codes, which are used to indicate the reason a request is not satisfied or that a request has been redirected.

- Cookies, which are used to store application-specific information at the client.

Sometimes, cookies are used to maintain an identifier for tracking a user's session (see Session Tracking).

Filtering Requests and Responses

A filter is an object that can transform the header and content (or both) of a request or response.

Filters differ from web components in that filters usually do not themselves create a response.

Instead, a filter provides functionality that can be "attached" to any kind of web resource.

Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way, it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.

- Block the request-and-response pair from passing any further.

- Modify the request headers and data.

You do this by providing a customized version of the request.

- Modify the response headers and data.

You do this by providing a customized version of the response.

- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order.

This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

Programming Filters

The filtering API is defined by the `Filter`, `FilterChain`, and `FilterConfig` interfaces in the `javax.servlet` package.

You define a filter by implementing the `Filter` interface.

Use the `@WebFilter` annotation to define a filter in a web application.

This annotation is specified on a class and contains metadata about the filter being declared.

The annotated filter must specify at least one URL pattern.

This is done by using the `urlPatterns` or `value` attribute on the annotation.

All other attributes are optional, with default settings.

Use the `value` attribute when the only attribute on the annotation is the URL pattern; use the `urlPatterns` attribute when other attributes are also used.

Classes annotated with the `@WebFilter` annotation must implement the `javax.servlet.Filter` interface.

To add configuration data to the filter, specify the `initParams` attribute of the `@WebFilter` annotation.

The `initParams` attribute contains a `@WebInitParam` annotation.

The following code snippet defines a filter, specifying an initialization parameter:

```
import javax.servlet.Filter;
import javax.servlet.annotation.
WebFilter;
import javax.servlet.annotation.
WebInitParam;
```

```
@WebFilter(filterName =
"TimeOfDayFilter",
urlPatterns = {"/*"},
initParams = {
@WebInitParam
(name = "mood", value = "awake")})
public class TimeOfDayFilter
implements Filter {
  ....
```

The most important method in the `Filter` interface is `doFilter`, which is passed request, response, and filter chain objects.

This method can perform the following actions:

- Examine the request headers.
- Customize the request object if the filter wishes to modify request headers or data.

- Customize the response object if the filter wishes to modify response headers or data.

- Invoke the next entity in the filter chain.

If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR.

The filter invokes the next entity by calling the `doFilter` method on the chain object, passing in the request and response it was called with or the wrapped versions it may have created.

Alternatively, the filter can choose to block the request by not making the call to invoke the next entity.

In the latter case, the filter is responsible for filling out the response.

- Examine response headers after invoking the next filter in the chain.

- Throw an exception to indicate an error in processing.

In addition to `doFilter`, you must implement the `init` and `destroy` methods.

The `init` method is called by the container when the filter is instantiated.

If you wish to pass initialization parameters to the filter, you retrieve them from the `FilterConfig` object passed to `init`.

Programming Customized Requests and Responses

There are many ways for a filter to modify a request or a response.

For example, a filter can add an attribute to the request or can insert data in the response.

A filter that modifies a response must usually capture the response before it is returned to the client.

To do this, you pass a stand-in stream to the servlet that generates the response.

The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response.

To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the `getWriter` or `getOutputStream` method to return this stand-in stream.

The wrapper is passed to the `doFilter` method of the filter chain.

Wrapper methods default to calling through to the wrapped request or response object.

To override request methods, you wrap the request in an object that extends either `ServletRequestWrapper` or `HttpServletRequestWrapper`.

To override response methods, you wrap the response in an object that extends either `ServletResponseWrapper` or `HttpServletResponseWrapper`.

Specifying Filter Mappings

A web container uses filter mappings to decide how to apply filters to web resources.

A filter mapping matches a filter to a web component by name or to web resources by URL pattern.

The filters are invoked in the order in which filter mappings appear in the filter mapping list of a WAR.

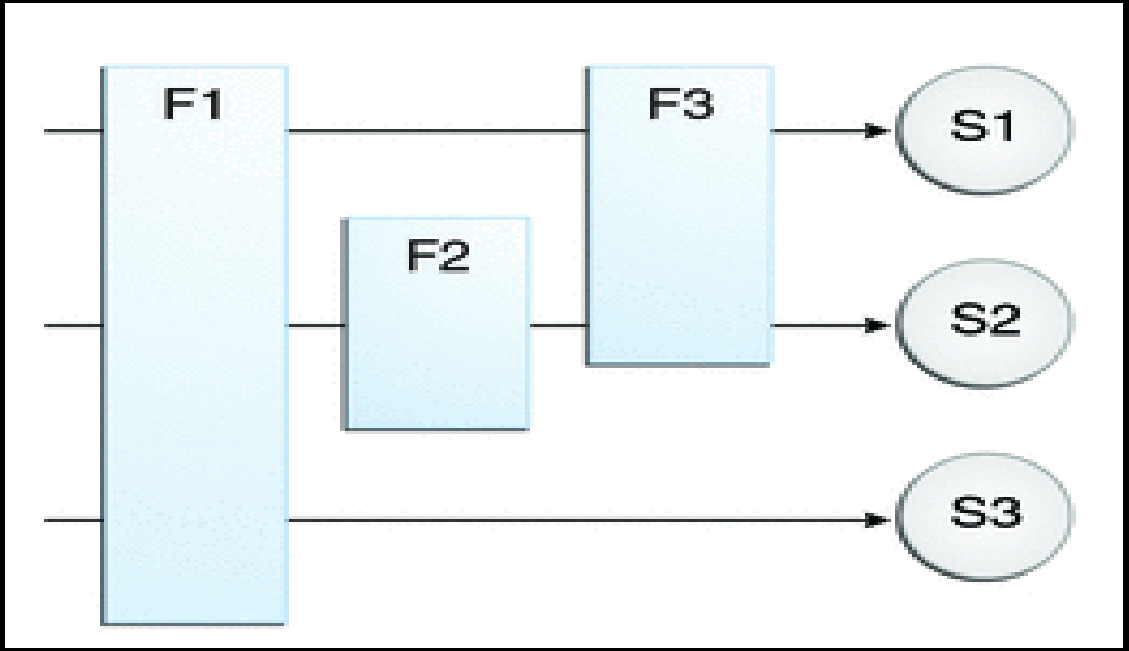
You specify a filter mapping list for a WAR in its deployment descriptor by either using NetBeans IDE or coding the list by hand with XML.

If you want to log every request to a web application, you map the hit counter filter to the URL pattern `/*`.

You can map a filter to one or more web resources, and you can map more than one filter to a web resource.

This is illustrated in Figure 15-1, where filter F1 is mapped to servlets S1, S2, and S3; filter F2 is mapped to servlet S2; and filter F3 is mapped to servlets S1 and S2.

Figure 15-1 Filter-to-Servlet Mapping



Recall that a filter chain is one of the objects passed to the `doFilter` method of a filter.

This chain is formed indirectly by means of filter mappings.

The order of the filters in the chain is the same as the order in which filter mappings appear in the web application deployment descriptor.

When a filter is mapped to servlet S1, the web container invokes the `doFilter` method of F1.

The `doFilter` method of each filter in S1's filter chain is invoked by the preceding filter in the chain by means of the `chain.doFilter` method.

Because S1's filter chain contains filters F1 and F3, F1's call to `chain.doFilter` invokes the `doFilter` method of filter F3.

When F3's `doFilter` method completes, control returns to F1's `doFilter` method.

To Specify Filter Mappings Using NetBeans IDE

1. Expand the application's project node in the Project pane.

2. Expand the Web Pages and WEB-INF nodes under the project node.

3. Double-click `web.xml`.

4. Click Filters at the top of the editor pane.

5. Expand the Servlet Filters node in the editor pane.

6. Click Add Filter Element to map the filter to a web resource by name or by URL pattern.

7. In the Add Servlet Filter dialog, enter the name of the filter in the Filter Name field.

8. Click Browse to locate the servlet class to which the filter applies.

You can include wildcard characters so that you can apply the filter to more than one servlet.

9. Click OK.

10. To constrain how the filter is applied to requests, follow these steps.

a. Expand the Filter Mappings node.

b. Select the filter from the list of filters.

c. Click Add.

d. In the Add Filter Mapping dialog, select one of the following dispatcher types:

- `REQUEST`: Only when the request comes directly from the client

- **ASYNC:** Only when the asynchronous request comes from the client

- **FORWARD:** Only when the request has been forwarded to a component (see Transferring Control to Another Web Component)

- **INCLUDE:** Only when the request is being processed by a component that has been included (see Including Other Resources in the Response)

- **ERROR:** Only when the request is being processed with the error page mechanism (see Handling Servlet Errors)

You can direct the filter to be applied to any combination of the preceding situations by selecting multiple dispatcher types.

If no types are specified, the default option is `REQUEST`.

Invoking Other Web Resources

Web components can invoke other web resources both indirectly and directly.

A web component indirectly invokes another web resource by embedding a URL that points to another web component in content returned to a client.

While it is executing, a web component directly invokes another resource by either including the content of another resource or forwarding a request to another resource.

To invoke a resource available on the server that is running a web component, you must first obtain a `RequestDispatcher` object by using the `getRequestDispatcher("URL")` method.

You can get a `RequestDispatcher` object from either a request or the web context; however, the two methods have slightly different behavior.

The method takes the path to the requested resource as an argument.

A request can take a relative path (that is, one that does not begin with a /), but the web context requires an absolute path.

If the resource is not available or if the server has not implemented a `RequestDispatcher` object for that type of resource, `getRequestDispatcher` will return null.

Your servlet should be prepared to deal with this condition.

Including Other Resources in the Response

It is often useful to include another web resource, such as banner content or copyright information) in the response returned from a web component.

To include another resource, invoke the `include` method of a `RequestDispatcher` object:

```
include(request, response);
```

If the resource is static, the `include` method enables programmatic server-side includes.

If the resource is a web component, the effect of the method is to send the request to the included web component, execute the web component, and then include the result of the execution in the response from the containing servlet.

An included web component has access to the request object but is limited in what it can do with the response object.

- It can write to the body of the response and commit a response.
- It cannot set headers or call any method, such as `setCookie`, that affects the headers of the response.

Transferring Control to Another Web Component

In some applications, you might want to have one web component do preliminary processing of a request and have another component generate the response.

For example, you might want to partially process a request and then transfer to another component, depending on the nature of the request.

To transfer control to another web component, you invoke the `forward` method of a `RequestDispatcher`.

When a request is forwarded, the request URL is set to the path of the forwarded page.

The original URI and its constituent parts are saved as request attributes

`javax.servlet.forward.`
`[request-uri|context-path|`
`servlet-path|path-info|`
`query-string].`

The `forward` method should be used to give another resource responsibility for replying to the user.

If you have already accessed a `ServletOutputStream` or `PrintWriter` object within the servlet, you cannot use this method; doing so throws an `IllegalStateException`.

Accessing the Web Context

The context in which web components execute is an object that implements the `ServletContext` interface.

You retrieve the web context by using the `getServletContext` method.

The web context provides methods for accessing

- Initialization parameters
- Resources associated with the web context
- Object-valued attributes
- Logging capabilities

The counter's access methods are synchronized to prevent incompatible operations by servlets that are running concurrently.

A filter retrieves the counter object by using the context's `getAttribute` method.

The incremented value of the counter is recorded in the log.

Maintaining Client State

Many applications require that a series of requests from a client be associated with one another.

For example, a web application can save the state of a user's shopping cart across requests.

Web-based applications are responsible for maintaining such state, called a session, because HTTP is stateless.

To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

Accessing a Session

Sessions are represented by an `HttpSession` object.

You access a session by calling the `getSession` method of a request object.

This method returns the current session associated with this request; or, if the request does not have a session, this method creates one.

Associating Objects with a Session

You can associate object-valued attributes with a session by name.

Such attributes are accessible by any web component that belongs to the same web context and is handling a request that is part of the same session.

Recall that your application can notify web context and session listener objects of servlet lifecycle events (Handling Servlet Lifecycle Events).

You can also notify objects of certain events related to their association with a session such as the following:

- When the object is added to or removed from a session.

To receive this notification, your object must implement the `javax.servlet.http.HttpSessionBindingListener` interface.

- When the session to which the object is attached will be passivated or activated.

A session will be passivated or activated when it is moved between virtual machines or saved to and restored from persistent storage.

To receive this notification, your object must implement the `javax.servlet.http.HttpSessionActivationListener` interface.

Session Management

Because an HTTP client has no way to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed.

The timeout period can be accessed by using a session's `getMaxInactiveInterval` and `setMaxInactiveInterval` methods.

- To ensure that an active session is not timed out, you should periodically access the session by using service methods because this resets the session's time-to-live counter.

- When a particular client interaction is finished, you use the session's `invalidate` method to invalidate a session on the server side and remove any session data.

To Set the Timeout Period Using NetBeans IDE

To set the timeout period in the deployment descriptor using NetBeans IDE, follow these steps.

1. Open the project if you haven't already.

2. Expand the project's node in the Projects pane.

3. Expand the Web Pages node and then the WEB-INF node.

4. Double-click `web.xml`.

5. Click General at the top of the editor.

6. In the Session Timeout field, type an integer value.

The integer value represents the number of minutes of inactivity that must pass before the session times out.

Session Tracking

To associate a session with a user, a web container can use several methods, all of which involve passing an identifier between the client and the server.

The identifier can be maintained on the client as a cookie, or the web component can include the identifier in every URL that is returned to the client.

If your application uses session objects, you must ensure that session tracking is enabled by having the application rewrite URLs whenever the client turns off cookies.

You do this by calling the response's `encodeURL(URL)` method on all URLs returned by a servlet.

This method includes the session ID in the URL only if cookies are disabled; otherwise, the method returns the URL unchanged.

Finalizing a Servlet

A servlet container may determine that a servlet should be removed from service (for example, when a container wants to reclaim memory resources or when it is being shut down).

In such a case, the container calls the `destroy` method of the `Servlet` interface.

In this method, you release any resources the servlet is using and save any persistent state.

The `destroy` method releases the database object created in the `init` method .

A servlet's service methods should all be complete when a servlet is removed.

The server tries to ensure this by calling the `destroy` method only after all service requests have returned or after a server-specific grace period, whichever comes first.

If your servlet has operations that may run longer than the server's grace period, the operations could still be running when `destroy` is called.

You must make sure that any threads still handling client requests complete.

The remainder of this section explains how to do the following:

- Keep track of how many threads are currently running the `service` method.

- Provide a clean shutdown by having the `destroy` method notify long-running threads of the shutdown and wait for them to complete.

- Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return.

Tracking Service Requests

To track service requests, include in your servlet class a field that counts the number of service methods that are running.

The field should have synchronized access methods to increment, decrement, and return its value:

```
public class ShutdownExample
extends HttpServlet {
private int serviceCounter = 0; ...
// Access methods for
// serviceCounter
protected synchronized void
enteringServiceMethod()
{ serviceCounter++; }
protected synchronized void
leavingServiceMethod()
{ serviceCounter--; }
```

```
protected synchronized int
numServices()
{ return serviceCounter; }
}
```

The `service` method should increment the service counter each time the method is entered and should decrement the counter each time the method returns.

This is one of the few times that your `HttpServlet` subclass should override the `service` method.

The new method should call `super.service` to preserve the functionality of the original `service` method:

```
protected void
service(HttpServletRequest req,
HttpServletResponse resp)throws
ServletException,IOException{
enteringServiceMethod();
try { super.service(req, resp); }
finally { leavingServiceMethod(); }
}
```

Notifying Methods to Shut Down

To ensure a clean shutdown, your `destroy` method should not release any shared resources until all the service requests have completed.

One part of doing this is to check the service counter.

Another part is to notify the long-running methods that it is time to shut down.

For this notification, another field is required.

The field should have the usual access methods:

```
public class ShutdownExample
extends HttpServlet {
private boolean shuttingDown; ...
//Access methods for shuttingDown
```

```
protected synchronized void
setShuttingDown(boolean flag)
{ shuttingDown = flag; }
protected synchronized boolean
isShuttingDown()
{ return shuttingDown; }
}
```

Here is an example of the `destroy` method using these fields to provide a clean shutdown:

```
public void destroy() {
// Check to see whether there are
// still service methods
// running, and if there are,
// tell them to stop.
if (numServices() > 0)
{ setShuttingDown(true); }
/* Wait for
* the service methods to stop.
*/
```

```
while(numServices() > 0) {
try {
Thread.sleep(interval);
} catch (InterruptedException e) {}
}
}
```

Creating Polite Long-Running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely.

Methods that might run for a long time should check the value of the field that notifies them of shutdowns and should interrupt their work, if necessary:

```
public void doPost(...) {...
for(i = 0; ((i < lotsOfStuffToDo)
&& !isShuttingDown()); i++) {
try
{ partOfLongRunningOperation(i); }
catch (InterruptedException e)
{ ... }
}
}
```

The mood Example Application

The `mood` example application, located in *tut-install*`/examples/web/mood`, is a simple example that displays Duke's moods at different times during the day.

The example shows how to develop a simple application by using the `@WebServlet`, `@WebFilter`, and `@WebListener` annotations to create a servlet, a listener, and a filter.

Components of the `mood` Example Application

The `mood` example application is comprised of three components: `mood.web.MoodServlet`, `mood.web.TimeOfDayFilter`, and `mood.web.SimpleServletListener`.

MoodServlet, the presentation layer of the application, displays Duke's mood in a graphic, based on the time of day.

The @WebServlet annotation specifies the URL pattern:

```
@WebServlet("/report")
public class MoodServlet extends
HttpServlet {   ...
```

TimeOfDayFilter sets an initialization parameter indicating that Duke is awake:

```
@WebFilter(
filterName = "TimeOfDayFilter",
urlPatterns = {"/*"},
initParams = {
@WebInitParam(name = "mood",
value = "awake")})
public class TimeOfDayFilter
implements Filter { ...
```

The filter calls the `doFilter` method, which contains a `switch` statement that sets Duke's mood based on the current time.

`SimpleServletListener` logs changes in the servlet's lifecycle.

The log entries appear in the server log.

Building, Packaging, Deploying, and Running the mood Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the mood example.

To Build, Package, Deploy, and Run the `mood` Example Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/web/`

3. Select the mood folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the mood project and select Build.

7. Right-click the project and select Deploy.

8. In a web browser, open the URL
`http://localhost:8080/mood/report`.

The URL specifies the context root, followed by the URL pattern specified for the servlet.

A web page appears with the title “Servlet MoodServlet at /mood” a text string describing Duke’s mood, and an illustrative graphic.

To Build, Package, Deploy, and Run the `mood` Example Using Ant

1. In a terminal window, go to:

   `tut-install/examples/web/mood/`

2. Type the following command:

   `ant`

This target builds the WAR file and copies it to the *tut-install*`/examples/web/mood/dist/` directory.

3. Type `ant deploy`.

Ignore the URL shown in the deploy target output.

4. In a web browser, open the URL `http://localhost:8080/mood/report`.

The URL specifies the context root, followed by the URL pattern.

A web page appears with the title "Servlet MoodServlet at /mood" a text string describing Duke's mood, and an illustrative graphic.

Further Information about Java Servlet Technology

For more information on Java Servlet technology, see

- Java Servlet 3.0 specification:

  http://jcp.org/en/jsr/detail?id=315

- Java Servlet web site:

  http://www.oracle.com/technetwork/java/index-jsp-135475.html