**MODULE 5**

**Tomcat Valves**

In this module, we will discuss two similar technologies: valves and filters. These technologies were invented for the purpose of intercepting requests for one or more web applications. The first, Tomcat Valve, is a proprietary Tomcat technology. The second, servlet filters, is a server-independent technology that was introduced as part of the Java Servlet specification (since version 2.3). Both of these technologies are used to process HTTP requests and response objects.

We will cover the following:

* Define Tomcat valves and servlet filters, their similarities, differences and usage scenarios
* Describe valve implementation and configuration
* Introduce valves shipped with Apache Tomcat 7 and their configuration
* Implement and configure sample filters
* Discuss useful servlet filters shipped with Tomcat

**Introduction to Valves and Filters**

Since the early adoption of a Java Servlet web application, the need for a HTTP request pre-processor was recognized. Web developers needed a mechanism to pre-process every request before it reaches the web application, and every response before it’s dispatched to the calling client (usually a browser). There are numerous use cases for such a feature. A few examples include the following:

* Logging details about every request in one place, without adding log calls to every implemented servlet.
* For security, allowing access from certain remote IPs.
* Setting required content encoding uniformly for every response.
* Data compression, to make content smaller before dispatching it to the browser.
* Localizing request and response to a particular locale (language and country).

The requirement was simply to implement a required functionality in one place (one Java class), and then apply it to all requests/responses for one or many web applications deployed.

**What Is a Tomcat Valve?**

In 2000, when Tomcat 4 (then called Jakarta Tomcat 4) was developed, it included Tomcat’s proprietary technology to enable the pre-processing of the HTTP request and response on the Tomcat’s container level. At the time, such a feature wasn’t part of the Java Servlet specification 2.2, so Tomcat’s developers decided to invent Tomcat Valve to perform the much-required functionality.

A Tomcat Valve allows you to associate an instance of a Java class with a particular Catalina container. This configuration allows the named class to act as a pre-processor for each request. These classes are called valves, and they must implement the *org.apache.catalina.Valve* interface. To allow easier implementation of valves for web application developers, Tomcat ships with a convenient abstract class that developers can extend to implement various kinds of valves: the *org.apache.catalina.valves.ValveBase* class. Valves were named after the real-world valves that control flow in a pipeline – just as Tomcat’s valves control the request/response flow within the Tomcat server’s processing pipeline. After processing, each valve can give control to the next valve in the chain, or stop processing altogether, effectively blocking the incoming request.

Valves can be configured with any of the Tomcat Catalina containers described in Module 1: Engine, Host, or Context in the CATALINA\_HOME/conf/server.xml configuration file.

If you configure a valve with the Engine container, the configured valve will process all requests to all web applications in all hosts for that engine. For example, if you want to log all request to all web applications deployed to Tomcat instance, you can configure a valve with the Engine container.

Similarly, you can configure a valve with the Host container to process all requests to all web applications belonging to the configured Host.

Finally, if you configure a valve with Context element, it will apply to all requests for that context (the single web application).

As we mentioned before, valves are proprietary to Tomcat and cannot be used in a different servlet container. The Java community noticed that a request preprocessing capability needed to be part of the Java Servlet specification, so it could be used across any servlet container.

**What Is a Servlet Filter?**

The Java Servlet API 2.3 specification was announced in late 2000 and it was confirmed that the Servlet API 2.3 would include the long-anticipated Server filter specification. A filter is a servlet component responsible for intercepting an HTTP request and response, and manipulating them as required.

Similarly to valves, filters can modify the contents of an HTTP request and response (they can change headers, or the actual content), and even decide whether to allow requests to progress to a web application or responses to be rendered in user’s browser (in case of security breach, for example).

Filters are configured in web deployment descriptors for each web application that uses them. This means you can reuse filters for multiple web applications, but each filter and every web application that uses it must be configured separately. On the other hand, you can configure a valve once for use with multiple applications. The configuration includes the URL pattern configuration to which the filter processing should be applied, so you can easily select only relevant request for filter processing.

* *Note. Filters have been available in Tomcat since version 4, which implements Servlet API 2.3. Therefore, the latest Apache Tomcat 7 supports filters as defined in the Servlet API 3.0 specification.*

Servlet filter is defined in the Servlet API via the *javax.servlet.Filter* interface. Every filter class needs to implement this interface, so that the servlet container can invoke it. You will learn more about this interface in the section about filter implementations, later in this chapter.

In addition, you can configure multiple filters for a web application. In such a case, the servlet container chains the filters together, in the order in which they should be executed. Using this chaining mechanism, each filter implementation can have well defined responsibility, decoupled from other filters that are responsible for different tasks, and at the same time decoupled from the servlet implementation, so they can be developed and tested independently.

**Tomcat Valves vs. Servlet Filters**

As you can see, valves and filters have a similar functionality, as they were developed with the same goal: to allow web application developers to implement components that can intercept HTTP requests and responses, and process them independently to the target web application servlet. Because filters are part of the Java Servlet specification, they have gained in popularity – not least because they are platform- independent and can run on any servlet specification compliant server, unlike valves, which are only available on Tomcat. In addition, filter-chaining functionality is very useful when multiple (unrelated) chained processing needs to be applied to requests or responses to a particular web application.

However, being Tomcat-specific technology, valves can be better performing and robust when it comes to web application deployed on Tomcat. This is because they are part of the Tomcat engine API, and are therefore processed on a lower level than filters in the Tomcat’s request-response cycle. In addition, valves can be configured on the engine or host level (unlike filters, which can only be configured on a single web application level). This makes it easier to apply request/response preprocessing for all web applications deployed on Tomcat instance, without having to repeat configuration for each web application. In such a case, web applications do not have to know that their requests or responses are pre-processed by valves, which can be a good thing if you don’t have access to a web application’s configuration files (for example, if you are deploying a third-party web application to your Tomcat server).

**Configuring Tomcat Valves**

Tomcat uses valves internally quite heavily; for example, to manage authentication or to maintain SSL information in a request, or to log request details. Some of these valves are instantiated and configured internally by Tomcat, but some are configurable in Tomcat’s configuration files, server.xml and context.xml. For example, Tomcat’s *BasicAuthenticatorValve* is configured automatically for every context that has BASIC authentication configured. Similarly, *FormAuthenticatorValve* is automatically configured for every context with FORM-based authentication.

In this section, we will demonstrate how we can implement and configure a custom valve, and how we can take advantage of some of the useful valves available with Tomcat out of the box.

**Implementing a Custom Valve**

Let’s see how we can implement a simple valve that logs the Uniform Resource Identifier (URI) of the requested page, as well as the remote IP of the client accessing it.

We mentioned before that every valve has to implement the *org.apache.catalina.Valve* interface, which is quite complex and requires the defining of no less than the seven methods that need to be implemented correctly in order for valve to work as expected. To ease the developer’s life, Tomcat’s API has a convenient abstract class, *org.apache.catalina.valves.ValveBase*, which a developer can extend in order to implement a valve. The abstract ValveBase class has only one method that a developer needs to implement: public abstract void invoke(Request request, Response response). By implementing this method, the developer has access to its arguments, which represent Tomcat’s implementations of Servlet API’s interfaces *HttpServletRequest* and *HttpServletResponse*, which you can modify using Tomcat Valve. It is important to note that this method should give control to the next configured valve for Tomcat after successful invocation by calling getNext().invoke(request, response) method.

In our example, we are going to inspect a few properties of the Request argument, and write them to the system log. Listing 5-1 shows the implementation of our SimpleLoggingValve class.

***Listing 5-1.*** *Valve Implementation That Logs Request Details to System Log*

*public class SimpleLoggingValve extends ValveBase{*

*@Override*

*public void invoke(Request request, Response response) #1 throws IOException, ServletException {*

*String remoteAddress = request.getRemoteAddr(); String requestUri = request.getRequestURI();*

*System.out.println("URI " + requestUri*

*+ " accessed from remote address: "+remoteAddress); #2*

*Valve nextValve = getNext(); #3*

*if(nextValve!=null){*

*nextValve.invoke(request, response); #4*

*}*

*}*

*}*

In this example, we implemented the *invoke(..)* method from the abstract *ValveBase* class (#1), and read *HttpServletRequest.requestURI* property (the URI of the requested page) and the *HttpServletRequest.remoteAddr* property (the IP of the client accessing the server) from the request argument. We then logged the property values to system output console (#2).

Finally, we get to the next valve in the pipeline (#3), and if there is any, we give it control over the request and response objects (#4). Once all valves have been executed, the request will reach the target web application.

* Note. Be sure to include catalina.jar and servlet-api.jar from the CATALINA\_HOME/lib directory to your project CLASSPATH, so that *SimpleLoggingValve* compiles successfully.

As you can see, in a few lines of code we implemented a valve. Now we need to configure Tomcat to use this valve.

The first step in adding our Valve implementation to Tomcat is to make the SimpleLoggingValve class visible to Tomcat’s class loader. In order to achieve that, you need to package the *SimpleLoggingValve* class in a jar file, using the command jar cvf chapter5-valve.jar chapter5- project-compiled-output, where the chapter-5-project-compiled-output is the directory of the project where the compiled classes are located. If you are using Eclipse or any other IDE, you can create the jar file with few mouse clicks; check your IDE documentation for details.

Once you have built the jar file, copy it to the *CATALINA\_HOME/lib* directory, as that directory is on the Tomcat classpath by default. This means that our valve will be on Tomcat’s classpath.

We made our *SimpleLoggingValve* class visible to Tomcat. Now let’s configure our valve to be executed for all requests for all web applications deployed on Tomcat’s default host. In order to do that, all we need to do is place a valve configuration element within the <Host> element in the *CATALINA\_HOME/conf/server.xml* file. You should open the server.xml file, locate the <Host> element, and add the *<Valve>* element to it. Listing 5-2 shows the *<Valve>* element configuration in the server.xml file

***Listing 5-2.*** *SimpleLoggingValve Configuration in server.xml File*

*<Valve className="com.apress.apachetomcat7.chapter5.SimpleLoggingValve" />*

The only required attribute for the *<Valve>* element is the className attribute, where we reference the valve class name to be used – in our case, *SimpleLoggingValve*. All that is left to do is to restart Tomcat (or start it if it wasn’t already running).

We configured our valve, as part of Tomcat’s default host, which means that SimpleLoggingValve should process all requests to all web applications under this host. To prove this, log in to a few of the standard web applications shipped with Tomcat (Examples web application, Manager web application, or any of the sample web applications from previous chapters), and check the Tomcat log file (the Tomcat main log file is located at CATALINA\_HOME/logs/catalina.out). You should see the log messages from our SimpleLoggingValve in the console output, similar to the following output:

*URI /manager/html accessed from remote address: 127.0.0.1*

*URI /manager/html/list accessed from remote address: 127.0.0.1 URI /manager/status accessed from remote address: 127.0.0.1 URI /manager/html/list accessed from remote address: 127.0.0.1*

*URI /examples/jsp/jsp2/el/basic-arithmetic.jsp accessed from remote address: 127.0.0.1*

Because we are accessing Tomcat running on the local machine, you will see the localhost IP address logged for every request (127.0.0.1). If you access the Tomcat server from a remote client over the Internet, you will see the public IP of the client browser accessing the server.

We have implemented a simple valve and demonstrated its configuration in the Tomcat’s server.xml file. Tomcat, however, comes with a number of ready-to-use valves that you can configure to match your needs, without implementing any Java code. Let’s now take a look at few of the useful valves available in Tomcat.

**The Access Log Valve**

Logging access to resources is a common activity for web server administrators. Therefore, Tomcat comes with a valve for logging access to resources—whether that access is at the engine level, the host level, or the context level. The location of a log valve is fairly important from a performance point of view because each log entry requires Tomcat to write data to disk. If you have a logging valve at the engine level of a busy server as well as logging valves for every context, the log will be written to many times and will grow very large. However, you may need to do this to monitor each individual context as well as the whole server. This just illustrates the need for careful planning when using valves.

An access log valve isn’t a logger, because a logger prints information and errors to a log file so that you can diagnose errors in web applications. For example, if a component encoun- ters problems and a user reports the error message to you, you’d look in the logger’s log, because this is where the error will be reported. However, if you wanted to see how often a client at a certain IP address requests a certain resource, you’d examine the access log valve’s log file.

The logger’s format also depends on the application in question. If it has been written properly, all errors will be written to the error log file in a standard way so that they can be investigated and rectified. If not, you may see nasty Java stack traces with details of the error buried among them.

The typical format for an access log valve is the common log file format, which you can find at [www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format.](http://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format) You may already have an analysis tool that can analyze log files in this format. If not, don’t worry—they’re quite

common. AWStats (http://awstats.sourceforge.net) is a great open source option, though you’ll need Perl to use it. Another option is Webalizer ([www.mrunix.net/webalizer/).](http://www.mrunix.net/webalizer/))

Table 5-2 shows the attributes for the standard access log valve that’s supplied with Tomcat.

In this case, the *className* attribute must be *org.apache.catalina.valves.AccessLogValve*.

**Table 5-2.** *The Attributes for the Standard Access Log Valve*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *className* | The Java class of the valve. This must be org.apache.catalina. valves.AccessLogValve. | Yes |
| *condition* | Turns conditional logging on. If set, the access log valve logs requests only if ServletRequest.getAttribute() is null. For example, if this value is set to userId, a particular request will be logged only if ServletRequest.getAttribute("userId")  == null. | No |
| *directory* | The directory where the log files will be placed. This is usually relative to the CATALINA\_HOME, but you can specify an absolute path instead. The default is logs. | No |
| *prefix* | The prefix added to the name of the log file. | No |
| *resolveHosts* | Determines if the log will contain hostnames via a reverse DNS  lookup. This can take significant time if enabled. The default is false. | No |
| *rotatable* | Determines if log rotation should occur. If false, this file is never rotated, and the *fileDateFormat* attribute is ignored. Use this attribute with caution, because the log file could grow very large indeed. The default is true. | No |
| *suffix* | The extension added to the name of the log file. | No |

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| fileDateFormat | Allows a customized date format in the access log filename. The  date format also decides how often the file is rotated. If you want to rotate every hour, then set this value to yyyy-MM-dd.HH. | No |
| *pattern* | Specifies the format used in the log. You can customize the format, or you can use common or combined as the format (the common format, plus the referrer and user agent are logged). To customize the format, you can use any of the following patterns interspersed with a literal string:  %a: Inserts remote IP address.  %A: Inserts local IP address (of URL resource).  %b: Inserts a bytes sent count, excluding HTTP headers, and shows - if zero.  %B: Inserts a bytes sent count, excluding HTTP headers.  %D: Time taken to process the request in milliseconds.  %h: Inserts remote hostname (or IP address if the resolveHosts  attribute is set to false).  %H: Inserts the request protocol (HTTP).  %l: Inserts remote logical user name (always -).  %m: Inserts request method such as GET and POST.  %p: Inserts the local TCP port where this request is received.  %q: Inserts the query string of this request.  %r: Inserts the first line of the request.  %s: Inserts the HTTP status code of the response.  %S: Inserts the user session ID.  %t: Inserts the date and time in common log file format.  %T: Inserts the time taken to process the request, in seconds.  %u: Inserts the remote user that has been authenticated (if there is none, it’s -).  %U: Inserts the URL path of the request.  %v: Inserts the name of the local virtual host from the request.  %{xxx}i: Use this for incoming headers, where xxx is the header.  %{xxx}c: Use this for a specific cookie, where xxx is the name of  the cookie.  %{xxx}r: Use this for ServletRequest attributes, where xxx is the  attribute.  %{xxx}s: Use this for HttpSession attributes, where xxx is the  attribute.  The default is common, which is %h %l %u %t "%r" %s %b. | No |

**Examining an Example Access Log Valve**

This section contains an example access log valve to demonstrate the attributes listed in Table 5-2. By default, the access log valves in server.xml are commented out, which disables them. This does, however, make it easy to activate them. Open server.xml, and navigate to the localhost *<Host>*

entry. The access log valve is configured after the large comment section (see Listing 5-3).

**Listing 5-3.** *The Access Log Valve in* server.xml

*<Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"*

*prefix="localhost\_access\_log." suffix=".log"*

*pattern="common" resolveHosts="false"/>*

You may not have this entry if you’ve previously used the admin web interface. This means that comments are lost, even if they contain useful default components. The good news is that the old server.xml should have been saved under another name in the conf directory, so you can copy and paste the valve entry from there into the new *server.xml*.

Uncomment this entry, start (or restart) Tomcat, and point your browser to http:// localhost:8080. You should see the default Tomcat welcome page. Now examine the CATALINA\_ HOME/logs directory, and open the localhost\_access\_log.DATE.log file. You’ll see the access log entry for the web page itself, though you’ll also see the entries for the associated image files, all in the common log file format, as shown in Listing 5-4.

**Listing 5-4.** *The* localhost\_access\_log.DATE.log *Log File*

*127.0.0.1 - - [30/Nov/2006:07:39:00 -0800]*

*"GET / HTTP/1.1" 200 7314*

*127.0.0.1 - - [30/Nov/2006:07:39:00 -0800]*

*"GET /asf-logo-wide.gif HTTP/1.1" 200 5866*

*127.0.0.1 - - [30/Nov/2006:07:39:00 -0800]*

*"GET /tomcat-power.gif HTTP/1.1" 200 2324*

*127.0.0.1 - - [30/Nov/2006:07:39:00 -0800]*

*“GET /tomcat.gif HTTP/1.1” 200 1934*

You may want to experiment further with other attributes of the standard access log valve by modifying the previous *<Valve>* entry. You should experiment with other access log valve con- figurations, such as valves at the context level, which you configure in the appropriate context XML file, and valves at the engine level. This way you can use tools to analyze access at various levels of the server.

**Using Request Filter Valves**

As a server administrator, you’ll often find it useful to restrict access to certain resources. You’ve already seen password protection for administration resources, but Tomcat also allows you to use request filter valves to block access so that a user doesn’t even get as far as the password prompt. You can use this facility to block access to admin resources to users who are not on the local machine or an admininstration-only machine. Other options can include blocking denial-of-service (DoS) attacks or denying access to sales data for non-sales personnel, and so on.

Two types of request filter valves exist: the remote address valve and the remote host valve. The first of these filters requests by the client’s IP address, and the second filters by the client’s host. Table 5-3 shows the attributes of the remote address valve.

**Table 5-3.** *The Attributes of the Remote Address Request Filter Valve*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *allow* | A comma-separated list of regular expressions used to match the client’s IP address. If there’s a match, the request is allowed through to its destination. If not, it’s blocked. If this attribute isn’t specified, all requests are allowed except if they match a pattern in the deny attribute. | No |
| *className* | The Java class of the valve. This must be *org.apache.catalina. valves.RemoteAddrValve*. | Yes |
| *deny* | A comma-separated list of regular expressions used to match the client’s IP address. If there’s a match, the request is blocked. If not, it’s allowed. | No |

It’s now possible to see how you can allow access to only those users on a local or admininstration machine. In this example, you’d add the IP address of the local (or administration) machine to the allow list. Listing 5-3 shows the scenario where both conditions are allowed (assuming the administration machine has 192.168.0.73 as its IP address).

**Listing 5-5.** *An Example Remote Address Request Filter Valve*

*<Valve className="org.apache.catalina.valves.RemoteAddrValve" allow="127.0.0.1,192.168.0.73"/>*

Visit a page on the local Tomcat server. You should see the page as usual. Now, remove the portion and restart Tomcat. Visit the same page, and you should be blocked

You can also achieve this effect by denying access to the localhost:

*<Valve className="org.apache.catalina.valves.RemoteAddrValve" deny="127.0.0.1"/>*

Filtering by client host is just as easy. The only difference is the class that implements the valve and the values of the regular expressions. In the case of the remote host request filter valve, the class is org.apache.catalina.valves.RemoteHostValve, and the regular expressions are hostnames instead of IP addresses.

*<Valve className="org.apache.catalina.valves.RemoteHostValve" allow="\*.com"/>*

■Note The remote host request filter requires a reverse DNS lookup, so the server must have access to DNS.

**Request Dumper Valve**

The request dumper valve allows you to debug web applications by dumping the headers and cookies of requests and responses to a log. The request dumper valve uses whichever logging mechanism you’ve configured for the component that contains the valve.

You can use it for the following:

* Checking how the scope of a valve affects the requests that are processed
* Debugging other valves and any other request-processing components that are configured on the server

To configure a request dumper valve, modify server.xml, and add the following line to the context, virtual host, or engine that you want to examine:

*<Valve className="org.apache.catalina.valves.RequestDumperValve"/>*

If you add the request dumper valve to the default server.xml file at the engine level, it will use the logger shown in Listing 5-4.

**Listing 5-4.** *An Engine-Level Logger*

*<!-- Global logger unless overridden at lower levels -->*

*<Logger className="org.apache.catalina.logger.FileLogger" prefix="catalina\_log." suffix=".txt" timestamp="true"/>*

For Tomcat 7, you’d use the following logger for logging at the host level; each line of the log has INFO priority:

*log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost]*

If you wanted logging at the context level, assuming that your context is called tomcatBook, you’d use the following:

*log4j.logger.org.apache.catalina.core.ContainerBase. [Catalina].[localhost].[/tomcatBook]*

This means the request dumper valve inherits the logger from a higher-level component, which isn’t always desirable if you’re troubleshooting a specific web application’s request- or response-processing pipeline.

After you’ve set up the valve, visit a web application that will be covered by the valve. Once your request has been processed, open the appropriate log file. You should see something similar to Listing 5-6. The version of Tomcat and the settings of your logger may differ, though the messages will be the same.

***Listing 5-6****. The Output of the Request Dumper Valve*

*REQUEST URI=/tomcatBook/ authType=null characterEncoding=null contentLength=-1 contentType=null contextPath=/tomcatBook*

*cookie=JSESSIONID=7F31F129712D208903FC6F50FD5143EA*

*header=host=localhost:8080 header=user-agent=Mozilla/5.0*

*(Windows; U; Windows NT 5.0; rv:1.7.3) Gecko/20040913 Firefox/0.10.1 header=accept=text/xml,application/xml,application/xhtml+xml, text/html;q=0.9,text/plain;q=0.8,image/png,\*/\*;q=0.5*

*header=accept-language=en-us,en;q=0.5 header=accept-encoding=gzip,deflate*

*header=accept-charset=ISO-8859-1,utf-8;q=0.7,\*;q=0.7*

*header=keep-alive=300 header=connection=keep-alive*

*header=cookie=JSESSIONID=7F31F129712D208903FC6F50FD5143EA*

*locale=en\_US method=GET pathInfo=null protocol=HTTP/1.1 queryString=null remoteAddr=127.0.0.1 remoteHost=127.0.0.1 remoteUser=null*

*requestedSessionId=7F31F129712D208903FC6F50FD5143EA scheme=http*

*serverName=localhost serverPort=8080 servletPath=/index.jsp isSecure=false*

*---------------------------------------------------------------*

*---------------------------------------------------------------*

*authType=null contentLength=-1*

*contentType=text/html;charset=ISO-8859-1 message=null*

*remoteUser=null status=200*

*========================================================*

As you can see, this listing contains a fair amount of information, all of which can be used to analyze a client’s interaction with your server. A word of warning, though: this valve decodes any parameters sent with the request using the platform’s default encoding. This may affect web applications on the server because calls to request.setCharacterEncoding() will have no effect.

**Using Single Sign-on Valves**

Another standard valve that’s frequently used is the single sign-on valve. Conventionally, when- ever users of a web application reach a protected page, they will be required to log in, a process that’s repeated if they browse to another web application on the same server. Using single sign-on, it’s possible to eliminate this repetition, provided that all the web applications on a host use the same Tomcat realm.

The single sign-on valve caches users’ credentials on the server and will authenticate users as they move among web applications on a host. The credentials are cached in the client’s ses- sion, which means that a single sign-on will be effective throughout a session. The user’s browser will send a cookie with a value that uniquely identifies this user as a user who has signed in. The valve then associates the new request with the existing user credentials and allows the user to visit protected

resources. This is one of the main reasons for having a common realm for the host.

Table 5-4 describes the attributes of the single sign-on valve.

**Table 5-4.** *The Attributes of the Single Sign-on Valve*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *className* | The Java class of the valve. This must be *org.apache. catalina.authenticator.SingleSignOn*. | Yes |
| *requireReauthentication* | Determines whether the valve should use the authentication realm to authenticate the user every time authentication is required. If false, the valve uses the cookie sent by the client and automatically authenticates the user without rechecking the realm. *The default is false.* | No |

**Configuring a Single Sign-on Valve**

Before seeing what single sign-on does, you should first experience the problem that sometimes makes it necessary to configure single sign-on. You’ll need two separate, protected web applications. Luckily Tomcat comes with two such web applications: the manager application and the admin application. If you’re using Tomcat 6, you’ll have to download the admin application or protect another application for this example to work.

You should already have a user who has the manager role required for access to the manager application and a user who has the admin role required for access to the admin application. You may even have a user who has both. If not, you must create one in tomcat-users.xml now, as shown in Listing 5-6.

**Listing 5-6.** *A User with Manager and Admin Roles Defined in tomcat-users.xml*

*<role rolename="manager"/>*

*<role rolename="admin"/>*

*<user username="tomcat" password="tomcat" roles="tomcat,manager,admin"/>*

Here, the tomcat user has three roles: tomcat, manager, and admin. Now, start (or restart) Tomcat, and navigate to http://localhost:8080/manager/html/. You’ll be asked for your user details as usual. Sign in as the user with both roles. Once you’ve done so, you should see the web interface of the manager application.

The next step is to navigate to http://localhost:8080/admin/. You’ll be presented with the form for logging into the admin application, which means your login for the manager application hasn’t carried over into the admin application despite the details being valid for both. This is where single sign-on comes in.

Open server.xml, and navigate to the valve as shown in Listing 5-7. It’s the first valve in the localhost host, after the large commented-out section.

**Listing 5-7**. The Single Sign-on Valve in server.xml

*<!-- Normally, users must authenticate themselves to each web app individually. Uncomment the following entry if you would like a user to be authenticated the first time they encounter a*

*resource protected by a security constraint, and then have that user identity maintained across \*all\* web applications contained in this virtual host. -->*

*<!--*

*<Valve className="org.apache.catalina.authenticator.SingleSignOn" debug="0"/>*

*-->*

Uncomment the valve, and restart Tomcat. Make sure you’ve closed your browser windows to start a new session, and navigate to http://localhost:8080/manager/html/ as before. Log in as the user with both roles again. Once you’ve logged in successfully, navigate to http:// localhost:8080/admin/. This time, you won’t be asked to log in again, because the valve will recognize you from the cookie sent by your browser and will authenticate you.

**Configuring User Sessions**

Sessions can play an important part in a server’s performance and its ability to service client requests. When you shut down Tomcat, all session information is usually lost, and sessions that are idle take up valuable working memory until the session timeout—which is typically a long period, since some users may leave their computers. Therefore, it would be useful to save session information across restarts so that users don’t experience a loss of service. Equally, it may be useful to remove idle sessions from memory and store them elsewhere to improve performance.

To solve these problems, Tomcat comes with session managers, which are works in progress, with features and configuration that are subject to change.

You can do the following with the session managers:

* You can swap inactive sessions onto disk, thereby releasing the memory consumed by them and making memory available for active sessions.
* You can save current sessions to disk when you shut down Tomcat; upon restart, the saved sessions are restored.
* You can save sessions lasting beyond a specified threshold period to disk, enabling the system to survive an unexpected crash.
* The last two features enable you to give a reliable service to users despite minor server failures or restarts.
* Tomcat also allows you to configure clustering so that you can replicate a users’ session across more than one server, thus minimizing the risk of losing their information when one server crashes or becomes unavailable.