Apache Tomcat

Manager App How-To

Introduction

In many production environments, it is very useful to have the capability to deploy a new web application, or undeploy an existing one, without having to shut down and restart the entire container. In addition, you can request an existing application to reload itself, even if you have not declared it to be reloadable in the Tomcat server configuration file.

To support these capabilities, Tomcat includes a web application (installed by default on context path /manager) that supports the following functions:

- Deploy a new web application from the uploaded contents of a WAR file.
- Deploy a new web application, on a specified context path, from the server file system.
- List the currently deployed web applications, as well as the sessions that are currently active for those web apps.
- Reload an existing web application, to reflect changes in the contents of /WEB-INF/classes or /WEB-INF/lib.
- List the OS and JVM property values.
- List the available global JNDI resources, for use in deployment tools that are preparing <ResourceLink> elements nested in a <Context> deployment description.
- Start a stopped application (thus making it available again).
- Stop an existing application (so that it becomes unavailable), but do not undeploy it.
- Undeploy a deployed web application and delete its document base directory (unless it was deployed from file system).

A default Tomcat installation includes an instance of the Manager application configured for the default virtual host. If you create additional virtual hosts, you may wish to add an instance of the Manager application to one or more of those Hosts. To add an instance of the Manager web application Context to a new host install the manager.xml context configuration file in the $CATALINA_BASE/conf/[enginename]/[hostname] folder. Here is an example:

```
<Context privileged="true" antiResourceLocking="false"
        docBase="${catalina.home}/webapps/manager">
  <CookieProcessor
className="org.apache.tomcat.util.http.Rfc6265CookieProcessor"
                   sameSiteCookies="strict" />
  <Valve className="org.apache.catalina.valves.RemoteAddrValve"
         allow="127\.\d+\.\d+\.\d+|::1|0:0:0:0:0:0:0:1" />
  <Manager
sessionAttributeValueClassNameFilter="java\.lang\.(?:Boolean|Integer|Long|Number|String)|org\.apache\.catalina\.filters\.CsrfPreventionFilter\$LruCache(?:\$1)?|java\.util\.(?:Linked)?HashMap"/>
</Context>
```

There are three ways to use the **Manager** web application.

- As an application with a user interface you use in your browser. Here is an example URL where you can replace localhost with your website host name: http://localhost:8080/manager/html .
- A minimal version using HTTP requests only which is suitable for use by scripts setup by system administrators. Commands are given as part of the request URI, and responses are in the form of simple text that can be easily parsed and processed. See Supported Manager Commands for more information.
- A convenient set of task definitions for the *Ant* (version 1.4 or later) build tool. See Executing Manager Commands With Ant for more information.

Configuring Manager Application Access

The description below uses the variable name $CATALINA_BASE to refer the base directory against which most relative paths are resolved. If you have not configured Tomcat for multiple instances by setting a CATALINA_BASE directory, then $CATALINA_BASE will be set to the value of $CATALINA_HOME, the directory into which you have installed Tomcat.

It would be quite unsafe to ship Tomcat with default settings that allowed anyone on the Internet to execute the Manager application on your server. Therefore, the Manager application is shipped with the requirement that anyone who attempts to use it must authenticate themselves, using a username and password that have one of **manager-xxx** roles associated with them (the role name depends on what functionality is required). Further, there is no username in the default users file ($CATALINA_BASE/conf/tomcat-users.xml) that is assigned to those roles. Therefore, access to the Manager application is completely disabled by default.

You can find the role names in the web.xml file of the Manager web application. The available roles are:

- **manager-gui** — Access to the HTML interface.
- **manager-status** — Access to the "Server Status" page only.
- **manager-script** — Access to the tools-friendly plain text interface that is described in this document, and to the "Server Status" page.
- **manager-jmx** — Access to JMX proxy interface and to the "Server Status" page.

The HTML interface is protected against CSRF (Cross-Site Request Forgery) attacks, but the text and JMX interfaces cannot be protected. It means that users who are allowed access to the text and JMX interfaces have to be cautious when accessing the Manager application with a web browser. To maintain the CSRF protection:

- If you use web browser to access the Manager application using a user that has either **manager-script** or **manager-jmx** roles (for example for testing the plain text or JMX interfaces), you MUST close all windows of

the browser afterwards to terminate the session. If you do not close the browser and visit other sites, you may become victim of a CSRF attack.

- It is recommended to never grant the **manager-script** or **manager-jmx** roles to users that have the **manager-gui** role.

Note that JMX proxy interface is effectively low-level root-like administrative interface of Tomcat. One can do a lot, if one knows what commands to call. You should be cautious when enabling the **manager-jmx** role.

To enable access to the Manager web application, you must either create a new username/password combination and associate one of the **manager-xxx** roles with it, or add a **manager-xxx** role to some existing username/password combination. As the majority of this document describes the using the text interface, this example will use the role name **manager-script**. Exactly how the usernames/passwords are configured depends on which Realm implementation you are using:

- *UserDatabaseRealm* plus *MemoryUserDatabase*, or *MemoryRealm* — The *UserDatabaseRealm* and *MemoryUserDatabase* are configured in the default $CATALINA_BASE/conf/server.xml.
  Both *MemoryUserDatabase* and *MemoryRealm* read an XML-format file by default stored at $CATALINA_BASE/conf/tomcat-users.xml, which can be edited with any text editor. This file contains an XML <user> for each individual user, which might look something like this:

<user username="craigmcc" password="secret" roles="standard,manager-script" />

which defines the username and password used by this individual to log on, and the role names they are associated with. You can add the **manager-script** role to the comma-delimited roles attribute for one or more existing users, and/or create new users with that assigned role.

- *DataSourceRealm* — Your user and role information is stored in a database accessed via JDBC. Add the **manager-script** role to one or more existing users, and/or create one or more new users with this role

assigned, following the standard procedures for your environment.

- *JNDIRealm* — Your user and role information is stored in a directory server accessed via LDAP. Add the **manager-script** role to one or more existing users, and/or create one or more new users with this role assigned, following the standard procedures for your environment.

The first time you attempt to issue one of the Manager commands described in the next section, you will be challenged to log on using BASIC authentication. The username and password you enter do not matter, as long as they identify a valid user in the users database who possesses the role **manager-script**.

In addition to the password restrictions, access to the Manager web application can be restricted by the **remote IP address** or host by adding a RemoteAddrValve or RemoteHostValve. See valves documentation for details. Here is an example of restricting access to the localhost by IP address:

```
<Context privileged="true">
        <Valve className="org.apache.catalina.valves.RemoteAddrValve"
               allow="127\.0\.0\.1"/>
</Context>
```

HTML User-friendly Interface

The user-friendly HTML interface of Manager web application is located at

http://{host}:{port}/manager/html

As has already been mentioned above, you need **manager-gui** role to be allowed to access it. There is a separate document that provides help on this interface. See:

- HTML Manager documentation

The HTML interface is protected against CSRF (Cross-Site Request Forgery) attacks. Each access to the HTML pages generates a random token, which is stored in your session and is included in all links on the page. If your next action does not have correct value of the token, the action will be denied. If the token

has expired you can start again from the main page or *List Applications* page of Manager.

To customize the subtitle of the HTML interface of the Manager web application, you can add any valid xml escaped html code to the htmlSubTitle initialisation parameter of the HTMLManagerServlet

<servlet>

<servlet-name>HTMLManager</servlet-name>

<servlet-class>org.apache.catalina.manager.HTMLManagerServlet</servlet-class>

<init-param>

<param-name>htmlSubTitle</param-name>

<param-value>Company Inc.
<i style='color:red'>Staging</i></param-value>

</init-param>

...

</servlet>

The above string value would unescape and be appended to the title

Company Inc.
Staging

Supported Manager Commands

All commands that the Manager application knows how to process are specified in a single request URI like this:

http://{host}:{port}/manager/text/{command}?{parameters}

where {host} and {port} represent the hostname and port number on which Tomcat is running, {command} represents the Manager command you wish to execute, and {parameters} represents the query parameters that are specific to that command. In the illustrations below, customize the host and port appropriately for your installation.

The commands are usually executed by HTTP GET requests.
The /deploy command has a form that is executed by an HTTP PUT request.

Common Parameters

Most commands accept one or more of the following query parameters:

- **path** - The context path (including the leading slash) of the web application you are dealing with. To select the ROOT web application, specify "/".
  NOTE: It is not possible to perform administrative commands on the Manager application itself.
  NOTE: If the path parameter is not explicitly specified then the path and the version will be derived using the standard Context naming rules from the config parameter or, if the config parameter is not present, the war parameter.
- **version** - The version of this web application as used by the parallel deployment feature. If you use parallel deployment wherever a path is required you must specify a version in addition to the path and it is the combination of path and version that must be unique rather than just the path.
  NOTE: If the path is not explicitly specified, the version parameter is ignored.
- **war** - URL of a web application archive (WAR) file, or pathname of a directory which contains the web application, or a Context configuration ".xml" file. You can use URLs in any of the following formats:
  - **file:/absolute/path/to/a/directory** - The absolute path of a directory that contains the unpacked version of a web application. This directory will be attached to the context path you specify without any changes.
  - **file:/absolute/path/to/a/webapp.war** - The absolute path of a web application archive (WAR) file. This is valid **only** for the /deploy command, and is the only acceptable format to that

command.

- **file:/absolute/path/to/a/context.xml** - The absolute path of a web application Context configuration ".xml" file which contains the Context configuration element.
- **directory** - The directory name for the web application context in the Host's application base directory.
- **webapp.war** - The name of a web application war file located in the Host's application base directory.

Each command will return a response in text/plain format (i.e. plain ASCII with no HTML markup), making it easy for both humans and programs to read). The first line of the response will begin with either OK or FAIL, indicating whether the requested command was successful or not. In the case of failure, the rest of the first line will contain a description of the problem that was encountered. Some commands include additional lines of information as described below.

Internationalization Note - The Manager application looks up its message strings in resource bundles, so it is possible that the strings have been translated for your platform. The examples below show the English version of the messages.

Deploy A New Application Archive (WAR) Remotely

http://localhost:8080/manager/text/deploy?path=/foo

Upload the web application archive (WAR) file that is specified as the request data in this HTTP PUT request, install it into the appBase directory of our corresponding virtual host, and start, deriving the name for the WAR file added to the appBase from the specified path. The application can later be undeployed (and the corresponding WAR file removed) by use of the /undeploy command.

This command is executed by an HTTP PUT request.

The .WAR file may include Tomcat specific deployment configuration, by including a Context configuration XML file in /META-INF/context.xml.

URL parameters include:

- update: When set to true, any existing update will be undeployed first. The

default value is set to false.

- tag: Specifying a tag name, this allows associating the deployed webapp with a tag or label. If the web application is undeployed, it can be later redeployed when needed using only the tag.
- config : URL of a Context configuration ".xml" file in the format **file:/absolute/path/to/a/context.xml**. This must be the absolute path of a web application Context configuration ".xml" file which contains the Context configuration element.

NOTE - This command is the logical opposite of the /undeploy command.

If installation and startup is successful, you will receive a response like this:

OK - Deployed application at context path /foo

Otherwise, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Application already exists at path /foo*

The context paths for all currently running web applications must be unique. Therefore, you must undeploy the existing web application using this context path, or choose a different context path for the new one. The update parameter may be specified as a parameter on the URL, with a value of true to avoid this error. In that case, an undeploy will be performed on an existing application before performing the deployment.

- *Encountered exception*

An exception was encountered trying to start the new web application. Check the Tomcat logs for the details, but likely explanations include problems parsing your /WEB-INF/web.xml file, or missing classes encountered when initializing application event listeners and filters.

Deploy A New Application from a Local Path

Deploy and start a new web application, attached to the specified context path (which must not be in use by any other web application). This command is the logical opposite of the /undeploy command.

This command is executed by an HTTP GET request. There are a number of different ways the deploy command can be used.

Deploy a previously deployed webapp

http://localhost:8080/manager/text/deploy?path=/footoo&tag=footag

This can be used to deploy a previously deployed web application, which has been deployed using the tag attribute. Note that the work directory of the Manager webapp will contain the previously deployed WARs; removing it would make the deployment fail.

Deploy a Directory or WAR by URL

Deploy a web application directory or ".war" file located on the Tomcat server. If no path is specified, the path and version are derived from the directory name or the war file name. The war parameter specifies a URL (including the file: scheme) for either a directory or a web application archive (WAR) file. The supported syntax for a URL referring to a WAR file is described on the Javadocs page for the java.net.JarURLConnection class. Use only URLs that refer to the entire WAR file.

In this example the web application located in the directory /path/to/foo on the Tomcat server is deployed as the web application context named /footoo.

http://localhost:8080/manager/text/deploy?path=/footoo&war=file:/path/to/foo

In this example the ".war" file /path/to/bar.war on the Tomcat server is deployed as the web application context named /bar. Notice that there is no path parameter so the context path defaults to the name of the web application archive file without the ".war" extension.

http://localhost:8080/manager/text/deploy?war=file:/path/to/bar.war

Deploy a Directory or War from the Host appBase

Deploy a web application directory or ".war" file located in your Host appBase directory. The path and optional version are derived from the directory or war file name.

In this example the web application located in a sub directory named foo in the

Host appBase directory of the Tomcat server is deployed as the web application context named /foo. Notice that the context path used is the name of the web application directory.

http://localhost:8080/manager/text/deploy?war=foo

In this example the ".war" file bar.war located in your Host appBase directory on the Tomcat server is deployed as the web application context named /bar.

http://localhost:8080/manager/text/deploy?war=bar.war

Deploy using a Context configuration ".xml" file

If the Host deployXML flag is set to true you can deploy a web application using a Context configuration ".xml" file and an optional ".war" file or web application directory. The context path is not used when deploying a web application using a context ".xml" configuration file.

A Context configuration ".xml" file can contain valid XML for a web application Context just as if it were configured in your Tomcat server.xml configuration file. Here is an example:

<Context path="/foobar" docBase="/path/to/application/foobar">

</Context>

When the optional war parameter is set to the URL for a web application ".war" file or directory it overrides any docBase configured in the context configuration ".xml" file.

Here is an example of deploying an application using a Context configuration ".xml" file.

http://localhost:8080/manager/text/deploy?config=file:/path/context.xml

Here is an example of deploying an application using a Context configuration ".xml" file and a web application ".war" file located on the server.

http://localhost:8080/manager/text/deploy

?config=file:/path/context.xml&war=file:/path/bar.war

Deployment Notes

If the Host is configured with unpackWARs=true and you deploy a war file, the war will be unpacked into a directory in your Host appBase directory.

If the application war or directory is installed in your Host appBase directory and either the Host is configured with autoDeploy=true or the Context path must match the directory name or war file name without the ".war" extension.

For security when untrusted users can manage web applications, the Host deployXML flag can be set to false. This prevents untrusted users from deploying web applications using a configuration XML file and also prevents them from deploying application directories or ".war" files located outside of their Host appBase.

Deploy Response

If installation and startup is successful, you will receive a response like this:

OK - Deployed application at context path /foo

Otherwise, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Application already exists at path /foo*

The context paths for all currently running web applications must be unique. Therefore, you must undeploy the existing web application using this context path, or choose a different context path for the new one. The update parameter may be specified as a parameter on the URL, with a value of true to avoid this error. In that case, an undeploy will be performed on an existing application before performing the deployment.

- *Document base does not exist or is not a readable directory*

The URL specified by the war parameter must identify a directory on this server that contains the "unpacked" version of a web application, or the absolute URL of a web application archive (WAR) file that contains this application. Correct the value specified by the war parameter.

- *Encountered exception*

An exception was encountered trying to start the new web application. Check the Tomcat logs for the details, but likely explanations include problems parsing your /WEB-INF/web.xml file, or missing classes encountered when initializing application event listeners and filters.

- *Invalid application URL was specified*

The URL for the directory or web application that you specified was not valid. Such URLs must start with file:, and URLs for a WAR file must end in ".war".

- *Invalid context path was specified*

The context path must start with a slash character. To reference the ROOT web application use "/".

- *Context path must match the directory or WAR file name:*

If the application war or directory is installed in your Host appBase directory and either the Host is configured with autoDeploy=true the Context path must match the directory name or war file name without the ".war" extension.

- *Only web applications in the Host web application directory can be installed*

If the Host deployXML flag is set to false this error will happen if an attempt is made to deploy a web application directory or ".war" file outside of the Host appBase directory.

List Currently Deployed Applications

http://localhost:8080/manager/text/list

List the context paths, current status (running or stopped), and number of active sessions for all currently deployed web applications. A typical response immediately after starting Tomcat might look like this:

OK - Listed applications for virtual host localhost

/webdav:running:0:webdav

/examples:running:0:examples

/manager:running:0:manager

/:running:0:ROOT

/test:running:0:test##2

/test:running:0:test##1

Reload An Existing Application

http://localhost:8080/manager/text/reload?path=/examples

Signal an existing application to shut itself down and reload. This can be useful when the web application context is not reloadable and you have updated classes or property files in the /WEB-INF/classes directory or when you have added or updated jar files in the /WEB-INF/lib directory.

If this command succeeds, you will see a response like this:

OK - Reloaded application at context path /examples

Otherwise, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Encountered exception*

An exception was encountered trying to restart the web application. Check the Tomcat logs for the details.

- *Invalid context path was specified*

The context path must start with a slash character. To reference the ROOT web application use "/".

- *No context exists for path /foo*

There is no deployed application on the context path that you specified.

- *No context path was specified*

The path parameter is required.

- *Reload not supported on WAR deployed at path /foo*

Currently, application reloading (to pick up changes to the classes

or web.xml file) is not supported when a web application is deployed directly from a WAR file. It only works when the web application is deployed from an unpacked directory. If you are using a WAR file, you should undeploy and then deploy or deploy with the update parameter the application again to pick up your changes.

List OS and JVM Properties

http://localhost:8080/manager/text/serverinfo

Lists information about the Tomcat version, OS, and JVM properties.

If an error occurs, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Encountered exception*

An exception was encountered trying to enumerate the system properties. Check the Tomcat logs for the details.

List Available Global JNDI Resources

http://localhost:8080/manager/text/resources[?type=xxxxx]

List the global JNDI resources that are available for use in resource links for context configuration files. If you specify the type request parameter, the value must be the fully qualified Java class name of the resource type you are interested in (for example, you would specify javax.sql.DataSource to acquire the names of all available JDBC data sources). If you do not specify the type request parameter, resources of all types will be returned.

Depending on whether the type request parameter is specified or not, the first line of a normal response will be:

OK - Listed global resources of all types

or

OK - Listed global resources of type xxxxx

followed by one line for each resource. Each line is composed of fields delimited by colon characters (":"), as follows:

- *Global Resource Name* - The name of this global JNDI resource, which would be used in the global attribute of a <ResourceLink> element.
- *Global Resource Type* - The fully qualified Java class name of this global JNDI resource.

If an error occurs, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Encountered exception*

An exception was encountered trying to enumerate the global JNDI resources. Check the Tomcat logs for the details.

- *No global JNDI resources are available*

The Tomcat server you are running has been configured without global JNDI resources.

Session Statistics

http://localhost:8080/manager/text/sessions?path=/examples

Display the default session timeout for a web application, and the number of currently active sessions that fall within one-minute ranges of their actual timeout times. For example, after restarting Tomcat and then executing one of the JSP samples in the /examples web app, you might get something like this:

OK - Session information for application at context path /examples

Default maximum session inactive interval 30 minutes

<1 minutes: 1 sessions

1 - <2 minutes: 1 sessions

Expire Sessions

http://localhost:8080/manager/text/expire?path=/examples&idle=num

Display the session statistics (like the above /sessions command) and expire sessions that are idle for longer than num minutes. To expire all sessions, use &idle=0 .

OK - Session information for application at context path /examples

Default maximum session inactive interval 30 minutes

1 - <2 minutes: 1 sessions

3 - <4 minutes: 1 sessions

>0 minutes: 2 sessions were expired

Actually /sessions and /expire are synonyms for the same command. The difference is in the presence of idle parameter.

Start an Existing Application

http://localhost:8080/manager/text/start?path=/examples

Signal a stopped application to restart, and make itself available again. Stopping and starting is useful, for example, if the database required by your application becomes temporarily unavailable. It is usually better to stop the web application that relies on this database rather than letting users continuously encounter database exceptions.

If this command succeeds, you will see a response like this:

OK - Started application at context path /examples

Otherwise, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Encountered exception*

An exception was encountered trying to start the web application. Check the Tomcat logs for the details.

- *Invalid context path was specified*

The context path must start with a slash character. To reference the ROOT web application use "/".

- *No context exists for path /foo*

There is no deployed application on the context path that you specified.

- *No context path was specified*

The path parameter is required.

Stop an Existing Application

http://localhost:8080/manager/text/stop?path=/examples

Signal an existing application to make itself unavailable, but leave it deployed. Any request that comes in while an application is stopped will see an HTTP error 404, and this application will show as "stopped" on a list applications command.

If this command succeeds, you will see a response like this:

OK - Stopped application at context path /examples

Otherwise, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Encountered exception*

An exception was encountered trying to stop the web application. Check the Tomcat logs for the details.

- *Invalid context path was specified*

The context path must start with a slash character. To reference the ROOT web application use "/".

- *No context exists for path /foo*

There is no deployed application on the context path that you specified.

- *No context path was specified* The path parameter is required.

Undeploy an Existing Application

http://localhost:8080/manager/text/undeploy?path=/examples

WARNING - This command will delete any web application artifacts that exist within appBase directory (typically "webapps") for this virtual host. This will delete the application .WAR, if present, the application directory resulting either from a deploy in unpacked form or from .WAR expansion as well as the XML Context definition

from $CATALINA_BASE/conf/[enginename]/[hostname]/ directory. If you simply want to take an application out of service, you should use the /stop command instead.

Signal an existing application to gracefully shut itself down, and remove it from Tomcat (which also makes this context path available for reuse later). In addition, the document root directory is removed, if it exists in the appBase directory (typically "webapps") for this virtual host. This command is the logical opposite of the /deploy command.

If this command succeeds, you will see a response like this:

OK - Undeployed application at context path /examples

Otherwise, the response will start with FAIL and include an error message. Possible causes for problems include:

- *Encountered exception*

An exception was encountered trying to undeploy the web application. Check the Tomcat logs for the details.

- *Invalid context path was specified*

The context path must start with a slash character. To reference the ROOT web application use "/".

- *No context exists named /foo*

There is no deployed application with the name that you specified.

- *No context path was specified* The path parameter is required.

Finding memory leaks

http://localhost:8080/manager/text/findleaks[?statusLine=[true|false]]

The find leaks diagnostic triggers a full garbage collection. It should be used with extreme caution on production systems.

The find leaks diagnostic attempts to identify web applications that have caused memory leaks when they were stopped, reloaded or undeployed. Results should

always be confirmed with a profiler. The diagnostic uses additional functionality provided by the StandardHost implementation. It will not work if a custom host is used that does not extend StandardHost.

Explicitly triggering a full garbage collection from Java code is documented to be unreliable. Furthermore, depending on the JVM used, there are options to disable explicit GC triggering, like -XX:+DisableExplicitGC. If you want to make sure, that the diagnostics were successfully running a full GC, you will need to check using tools like GC logging, JConsole or similar.

If this command succeeds, you will see a response like this:

/leaking-webapp

If you wish to see a status line included in the response then include the statusLine query parameter in the request with a value of true.

Each context path for a web application that was stopped, reloaded or undeployed, but which classes from the previous runs are still loaded in memory, thus causing a memory leak, will be listed on a new line. If an application has been reloaded several times, it may be listed several times.

If the command does not succeed, the response will start with FAIL and include an error message.

Connector SSL/TLS cipher information

http://localhost:8080/manager/text/sslConnectorCiphers

The SSL Connector/Ciphers diagnostic lists the SSL/TLS ciphers that are currently configured for each connector.

The response will look something like this:

OK - Connector / SSL Cipher information

Connector[HTTP/1.1-8080]

SSL is not enabled for this connector

Connector[HTTP/1.1-8443]

TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA

TLS_DHE_RSA_WITH_AES_128_CBC_SHA

TLS_ECDH_RSA_WITH_AES_128_CBC_SHA

TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA

...

Connector SSL/TLS certificate chain information

http://localhost:8080/manager/text/sslConnectorCerts

The SSL Connector/Certs diagnostic lists the certificate chain that is currently configured for each virtual host.

The response will look something like this:

OK - Connector / Certificate Chain information

Connector[HTTP/1.1-8080]

SSL is not enabled for this connector

Connector[HTTP/1.1-8443]-_default_-RSA

[

[

Version: V3

Subject: CN=localhost, OU=Apache Tomcat PMC, O=The Apache Software Foundation, L=Wakefield, ST=MA, C=US

Signature Algorithm: SHA256withRSA, OID = 1.2.840.113549.1.1.11

...

Connector SSL/TLS trusted certificate information

http://localhost:8080/manager/text/sslConnectorTrustedCerts

The SSL Connector/Certs diagnostic lists the trusted certificates that are currently configured for each virtual host.

The response will look something like this:

OK - Connector / Trusted Certificate information

Connector[HTTP/1.1-8080]

SSL is not enabled for this connector

Connector[HTTP/1.1-8443]-_default_

[

[

Version: V3

Subject: CN=Apache Tomcat Test CA, OU=Apache Tomcat PMC, O=The Apache Software Foundation, L=Wakefield, ST=MA, C=US

...

Reload TLS configuration

http://localhost:8080/manager/text/sslReload?tlsHostName=name

Reload the TLS configuration files (the certificate and key files, this does not trigger a re-parsing of server.xml). To reload the files for all hosts don't specify the tlsHostName parameter.

OK - Reloaded TLS configuration for [_default_]

Thread Dump

http://localhost:8080/manager/text/threaddump

Write a JVM thread dump.

The response will look something like this:

OK - JVM thread dump

2014-12-08 07:24:40.080

Full thread dump Java HotSpot(TM) Client VM (25.25-b02 mixed mode):

"http-nio-8080-exec-2" Id=26 cpu=46800300 ns usr=46800300 ns blocked 0 for -1 ms waited 0 for -1 ms

java.lang.Thread.State: RUNNABLE

locks java.util.concurrent.ThreadPoolExecutor$Worker@1738ad4

at sun.management.ThreadImpl.dumpThreads0(Native Method)

at sun.management.ThreadImpl.dumpAllThreads(ThreadImpl.java:446)

at org.apache.tomcat.util.Diagnostics.getThreadDump(Diagnostics.java:440)

at org.apache.tomcat.util.Diagnostics.getThreadDump(Diagnostics.java:409)

at org.apache.catalina.manager.ManagerServlet.threadDump(ManagerServlet.java:557)

at org.apache.catalina.manager.ManagerServlet.doGet(ManagerServlet.java:371)

at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:618)

at jakarta.servlet.http.HttpServlet.service(HttpServlet.java:725)

...

VM Info

http://localhost:8080/manager/text/vminfo

Write some diagnostic information about Java Virtual Machine.

The response will look something like this:

OK - VM info

2014-12-08 07:27:32.578

Runtime information:

vmName: Java HotSpot(TM) Client VM

vmVersion: 25.25-b02

vmVendor: Oracle Corporation

specName: Java Virtual Machine Specification

specVersion: 1.8

specVendor: Oracle Corporation

managementSpecVersion: 1.2

name: ...

startTime: 1418012458849

uptime: 393855

isBootClassPathSupported: true

OS information:

...

Save Configuration

http://localhost:8080/manager/text/save

If specified without any parameters, this command saves the current configuration of the server to server.xml. The existing file will be renamed as a backup if required.

If specified with a path parameter that matches the path of a deployed web application then the configuration for that web application will be saved to an appropriately named context.xml file in the xmlBase for the current Host.

To use the command a StoreConfig MBean must be present. Typically this is configured using the StoreConfigLifecycleListener.

If the command does not succeed, the response will start with FAIL and include an error message.

Server Status

From the following links you can view Status information about the server. Any one of **manager-xxx** roles allows access to this page.

http://localhost:8080/manager/status

http://localhost:8080/manager/status/all

Displays server status information in HTML format.

http://localhost:8080/manager/status?XML=true

http://localhost:8080/manager/status/all?XML=true

Displays server status information in XML format.

http://localhost:8080/manager/status?JSON=true

http://localhost:8080/manager/status/all?JSON=true

Displays server status information in JSON format.

First, you have the server and JVM version number, JVM provider, OS name and number followed by the architecture type.

Second, there is information about the memory usage of the JVM.

Then, there is information about the Tomcat AJP and HTTP connectors. The same information is available for both of them :

- Threads information : Max threads, min and max spare threads, current thread count and current thread busy.
- Request information : Max processing time and processing time, request and error count, bytes received and sent.
- A table showing Stage, Time, Bytes Sent, Bytes Receive, Client, VHost and Request. All existing threads are listed in the table. Here is the list of the possible thread stages :
    - *"Parse and Prepare Request"* : The request headers are being parsed or the necessary preparation to read the request body (if a transfer encoding has been specified) is taking place.
    - *"Service"* : The thread is processing a request and generating the

response. This stage follows the "Parse and Prepare Request" stage and precedes the "Finishing" stage. There is always at least one thread in this stage (the server-status page).

- *"Finishing"* : The end of the request processing. Any remainder of the response still in the output buffers is sent to the client. This stage is followed by "Keep-Alive" if it is appropriate to keep the connection alive or "Ready" if "Keep-Alive" is not appropriate.
- *"Keep-Alive"* : The thread keeps the connection open to the client in case the client sends another request. If another request is received, the next stage will be "Parse and Prepare Request". If no request is received before the keep alive times out, the connection will be closed and the next stage will be "Ready".
- *"Ready"* : The thread is at rest and ready to be used.

If you are using /status/all command, additional information on each of deployed web applications will be available.

Using the JMX Proxy Servlet

What is JMX Proxy Servlet

The JMX Proxy Servlet is a lightweight proxy to get and set the tomcat internals. (Or any class that has been exposed via an MBean) Its usage is not very user friendly but the UI is extremely helpful for integrating command line scripts for monitoring and changing the internals of tomcat. You can do two things with the proxy: get information and set information. For you to really understand the JMX Proxy Servlet, you should have a general understanding of JMX. If you don't know what JMX is, then prepare to be confused.

JMX Query command

This takes the form:

http://webserver/manager/jmxproxy/?qry=STUFF

Where STUFF is the JMX query you wish to perform. For example, here are some queries you might wish to run:

- qry=*%3Atype%3DRequestProcessor%2C* --> type=RequestProcessor which will locate all workers which can process requests and report their state.
- qry=*%3Aj2eeType=Servlet%2c* --> j2eeType=Servlet which return all loaded servlets.
- qry=Catalina%3Atype%3DEnvironment%2Cresourcetype%3DGlobal%2Cname%3DsimpleValue --> Catalina:type=Environment,resourcetype=Global,name=simpleValue which look for a specific MBean by the given name.

You'll need to experiment with this to really understand its capabilities If you provide no qry parameter, then all of the MBeans will be displayed. We really recommend looking at the tomcat source code and understand the JMX spec to get a better understanding of all the queries you may run.

JMX Get command

The JMXProxyServlet also supports a "get" command that you can use to fetch the value of a specific MBean's attribute. The general form of the get command is:

http://webserver/manager/jmxproxy/?get=BEANNAME&att=MYATTRIBUTE&key=MYKEY

You must provide the following parameters:

1. get: The full bean name
2. att: The attribute you wish to fetch
3. key: (optional) The key into a CompositeData MBean attribute

If all goes well, then it will say OK, otherwise an error message will be shown. For example, let's say we wish to fetch the current heap memory data:

http://webserver/manager/jmxproxy/?get=java.lang:type=Memory&att=HeapMemoryUsage

Or, if you only want the "used" key:

http://webserver/manager/jmxproxy/

?get=java.lang:type=Memory&att=HeapMemoryUsage&key=used

JMX Set command

Now that you can query an MBean, its time to muck with Tomcat's internals! The general form of the set command is :

http://webserver/manager/jmxproxy/?set=BEANNAME&att=MYATTRIBUTE&val=NEWVALUE

So you need to provide 3 request parameters:

1. set: The full bean name
2. att: The attribute you wish to alter
3. val: The new value

If all goes ok, then it will say OK, otherwise an error message will be shown. For example, lets say we wish to turn up debugging on the fly for the ErrorReportValve. The following will set debugging to 10.

http://localhost:8080/manager/jmxproxy/

?set=Catalina%3Atype%3DValve%2Cname%3DErrorReportValve%2Chost%3Dlocalhost

&att=debug&val=10

and my result is (YMMV):

Result: ok

Here is what I see if I pass in a bad value. Here is the URL I used, I try set debugging equal to 'cow':

http://localhost:8080/manager/jmxproxy/

?set=Catalina%3Atype%3DValve%2Cname%3DErrorReportValve%2Chost%3Dlocalhost

&att=debug&val=cow

When I try that, my result is

Error: java.lang.NumberFormatException: For input string: "cow"

JMX Invoke command

The invoke command enables methods to be called on MBeans. The general form of the command is:

http://webserver/manager/jmxproxy/

?invoke=BEANNAME&op=METHODNAME&ps=COMMASEPARATEDPARAMETERS

For example, to call the findConnectors() method of the **Service** use:

http://localhost:8080/manager/jmxproxy/

?invoke=Catalina%3Atype%3DService&op=findConnectors&ps=

Executing Manager Commands With Ant

In addition to the ability to execute Manager commands via HTTP requests, as documented above, Tomcat includes a convenient set of Task definitions for the *Ant* (version 1.4 or later) build tool. In order to use these commands, you must perform the following setup operations:

- Download the binary distribution of Ant from https://ant.apache.org. You must use version **1.4** or later.
- Install the Ant distribution in a convenient directory (called ANT_HOME in the remainder of these instructions).
- Add the $ANT_HOME/bin directory to your PATH environment variable.
- Configure at least one username/password combination in your Tomcat user database that includes the manager-script role.

To use custom tasks within Ant, you must declare them first with an <import> element. Therefore, your build.xml file might look something like this:

<project name="My Application" default="compile" basedir=".">

```
<!-- Configure the directory into which the web application is built -->
<property name="build"        value="${basedir}/build"/>

<!-- Configure the context path for this application -->
<property name="path"         value="/myapp"/>

<!-- Configure properties to access the Manager application -->
<property name="url"          value="http://localhost:8080/manager/text"/>
<property name="username" value="myusername"/>
<property name="password" value="mypassword"/>

<!-- Configure the path to the Tomcat installation -->
<property name="catalina.home" value="/usr/local/apache-tomcat"/>

<!-- Configure the custom Ant tasks for the Manager application -->
<import file="${catalina.home}/bin/catalina-tasks.xml"/>

<!-- Executable Targets -->
<target name="compile" description="Compile web application">
   <!-- ... construct web application in ${build} subdirectory, and
                generated a ${path}.war ... -->
</target>
```

```
  <target name="deploy" description="Install web application"
                depends="compile">
    <deploy url="${url}" username="${username}" password="${password}"
                  path="${path}" war="file:${build}${path}.war"/>
  </target>

  <target name="reload" description="Reload web application"
                depends="compile">
    <reload    url="${url}" username="${username}" password="${password}"
                  path="${path}"/>
  </target>

  <target name="undeploy" description="Remove web application">
    <undeploy url="${url}" username="${username}" password="${password}"
                  path="${path}"/>
  </target>

</project>
```

Note: The definition of the resources task via the import above will override the resources datatype added in Ant 1.7. If you wish to use the resources datatype you will need to use Ant's namespace support to modify catalina-tasks.xml to assign the Tomcat tasks to their own namespace.

Now, you can execute commands like ant deploy to deploy the application to a running instance of Tomcat, or ant reload to tell Tomcat to reload it. Note also that most of the interesting values in this build.xml file are defined as replaceable properties, so you can override their values from the command line.

For example, you might consider it a security risk to include the real manager password in your build.xml file's source code. To avoid this, omit the password property, and specify it from the command line:

ant -Dpassword=secret deploy

Tasks output capture

Using *Ant* version **1.6.2** or later, the Catalina tasks offer the option to capture their output in properties or external files. They support directly the following subset of the <redirector> type attributes:

Attribute	Description	Required
output	Name of a file to which to write the output. If the error stream is not also redirected to a file or property, it will appear in this output.	No
error	The file to which the standard error of the command should be redirected.	No
logError	This attribute is used when you wish to see error output in Ant's log and you are redirecting output to a file/property. The error output will not be included in the output file/property. If you redirect error with the *error* or *errorProperty* attributes, this will have no effect.	No
append	Whether output and error files should be appended to or overwritten. Defaults to false.	No
createemptyfiles	Whether output and error files should be created even when empty. Defaults to true.	No
outputproperty	The name of a property in which the output of the command should be stored. Unless the	No

	error stream is redirected to a separate file or stream, this property will include the error output.	
errorproperty	The name of a property in which the standard error of the command should be stored.	No

A couple of additional attributes can also be specified:

Attribute	Description	Required
alwaysLog	This attribute is used when you wish to see the output you are capturing, appearing also in the Ant's log. It must not be used unless you are capturing task output. Defaults to false. *This attribute will be supported directly by <redirector> in Ant 1.6.3*	No
failonerror	This attribute is used when you wish to avoid that any manager command processing error terminates the ant execution. Defaults to true. It must be set to false, if you want to capture error output, otherwise execution will terminate before anything can be captured. This attribute acts only on manager command execution, any wrong or missing command attribute will still cause Ant execution termination.	No

They also support the embedded <redirector> element in which you can specify its full set of attributes, but input, inputstring and inputencoding that, even if accepted, are not used because they have no meaning in this context. Refer to ant manual for details on <redirector> element attributes.

Here is a sample build file extract that shows how this output redirection support can be used:

```
<target name="manager.deploy"
```

```
        depends="context.status"
        if="context.notInstalled">
        <deploy url="${mgr.url}"
                username="${mgr.username}"
                password="${mgr.password}"
                path="${mgr.context.path}"
                config="${mgr.context.descriptor}"/>
</target>

<target name="manager.deploy.war"
        depends="context.status"
        if="context.deployable">
        <deploy url="${mgr.url}"
                username="${mgr.username}"
                password="${mgr.password}"
                update="${mgr.update}"
                path="${mgr.context.path}"
                war="${mgr.war.file}"/>
</target>

<target name="context.status">
        <property name="running" value="${mgr.context.path}:running"/>
        <property name="stopped" value="${mgr.context.path}:stopped"/>
```

```
<list url="${mgr.url}"
        outputproperty="ctx.status"
        username="${mgr.username}"
        password="${mgr.password}">
</list>

<condition property="context.running">
        <contains string="${ctx.status}" substring="${running}"/>
</condition>
<condition property="context.stopped">
        <contains string="${ctx.status}" substring="${stopped}"/>
</condition>
<condition property="context.notInstalled">
        <and>
                <isfalse value="${context.running}"/>
                <isfalse value="${context.stopped}"/>
        </and>
</condition>
<condition property="context.deployable">
        <or>
                <istrue value="${context.notInstalled}"/>
                <and>
                        <istrue value="${context.running}"/>
                        <istrue value="${mgr.update}"/>
```

```
                </and>
                <and>
                    <istrue value="${context.stopped}"/>
                    <istrue value="${mgr.update}"/>
                </and>
            </or>
        </condition>
        <condition property="context.undeployable">
            <or>
                <istrue value="${context.running}"/>
                <istrue value="${context.stopped}"/>
            </or>
        </condition>
    </target>
```

WARNING: even if it doesn't make many sense, and is always a bad idea, calling a Catalina task more than once, badly set Ant tasks depends chains may cause that a task be called more than once in the same Ant run, even if not intended to. A bit of caution should be exercised when you are capturing output from that task, because this could lead to something unexpected:

- when capturing in a property you will find in it only the output from the *first* call, because Ant properties are immutable and once set they cannot be changed,
- when capturing in a file, each run will overwrite it and you will find in it only the *last* call output, unless you are using the append="true" attribute, in which case you will see the output of each task call appended to the file.

Host Manager App -- Text Interface

Introduction

The **Tomcat Host Manager** application enables you to create, delete, and otherwise manage virtual hosts within Tomcat. This how-to guide is best accompanied by the following pieces of documentation:

- Virtual Hosting How-To for more information about virtual hosting.
- The Host Container for more information about the underlying xml configuration of virtual hosts and description of attributes.

The **Tomcat Host Manager** application is a part of Tomcat installation, by default available using the following context: /host-manager. You can use the host manager in the following ways:

- Utilizing the graphical user interface, accessible at: {server}:{port}/host-manager/html.
- Utilizing a set of minimal HTTP requests suitable for scripting. You can access this mode at: {server}:{port}/host-manager/text.

Both ways enable you to add, remove, start, and stop virtual hosts. Changes may be persisted by using the persist command. This document focuses on the text interface. For further information about the graphical interface, see Host Manager App -- HTML Interface.

Configuring Manager Application Access

The description below uses $CATALINA_HOME to refer the base Tomcat directory. It is the directory in which you installed Tomcat, for example C:\tomcat9, or /usr/share/tomcat9.

The Host Manager application requires a user with one of the following roles:

- admin-gui - use this role for the graphical web interface.
- admin-script - use this role for the scripting web interface.

To enable access to the text interface of the Host Manager application, either

grant your Tomcat user the appropriate role, or create a new one with the correct role. For example, open ${CATALINA_BASE}/conf/tomcat-users.xml and enter the following:

<user username="test" password="chang3m3N#w" roles="admin-script"/>

No further settings is needed. When you now access {server}:{port}/host-manager/text/${COMMAND},you are able to log in with the created credentials. For example:

$ curl -u ${USERNAME}:${PASSWORD} http://localhost:8080/host-manager/text/list

OK - Listed hosts

localhost:

If you are using a different realm you will need to add the necessary role to the appropriate user(s) using the standard user management tools for that realm.

List of Commands

The following commands are supported:

- list
- add
- remove
- start
- stop
- persist

In the following subsections, the username and password is assumed to be **test:test**. For your environment, use credentials created in the previous sections.

List command

Use the **list** command to see the available virtual hosts on your Tomcat instance.

Example command:

curl -u test:test http://localhost:8080/host-manager/text/list

Example response:

OK - Listed hosts

localhost:

Add command

Use the **add** command to add a new virtual host. Parameters used for the **add** command:

- String **name**: Name of the virtual host. **REQUIRED**
- String **aliases**: Aliases for your virtual host.
- String **appBase**: Base path for the application that will be served by this virtual host. Provide relative or absolute path.
- Boolean **manager**: If true, the Manager app is added to the virtual host. You can access it with the */manager* context.
- Boolean **autoDeploy**: If true, Tomcat automatically redeploys applications placed in the appBase directory.
- Boolean **deployOnStartup**: If true, Tomcat automatically deploys applications placed in the appBase directory on startup.
- Boolean **deployXML**: If true, the */META-INF/context.xml* file is read and used by Tomcat.
- Boolean **copyXML**: If true, Tomcat copies */META-INF/context.xml* file and uses the original copy regardless of updates to the application's */META-INF/context.xml* file.

Example command:

curl -u test:test http://localhost:8080/host-manager/text/add?name=www.awesomeserver.com&aliases=awesomeserver.com&appBase/mnt/appDir&deployOnStartup=true

Example response:

add: Adding host [www.awesomeserver.com]

Remove command

Use the **remove** command to remove a virtual host. Parameters used for the **remove** command:

- String **name**: Name of the virtual host to be removed. **REQUIRED**

Example command:

curl -u test:test http://localhost:8080/host-manager/text/remove?name=www.awesomeserver.com

Example response:

remove: Removing host [www.awesomeserver.com]

Start command

Use the **start** command to start a virtual host. Parameters used for the **start** command:

- String **name**: Name of the virtual host to be started. **REQUIRED**

Example command:

curl -u test:test http://localhost:8080/host-manager/text/start?name=www.awesomeserver.com

Example response:

OK - Host www.awesomeserver.com started

Stop command

Use the **stop** command to stop a virtual host. Parameters used for the **stop** command:

- String **name**: Name of the virtual host to be stopped. **REQUIRED**

Example command:

curl -u test:test http://localhost:8080/host-manager/text/stop?name=www.awesomeserver.com

Example response:

OK - Host www.awesomeserver.com stopped

Persist command

Use the **persist** command to persist a virtual host into **server.xml**. Parameters used for the **persist** command:

- String **name**: Name of the virtual host to be persist. **REQUIRED**

This functionality is disabled by default. To enable this option, you must configure the StoreConfigLifecycleListener listener first. To do so, add the following listener to your *server.xml*:

<Listener className="org.apache.catalina.storeconfig.StoreConfigLifecycleListener"/>

Example command:

curl -u test:test http://localhost:8080/host-manager/text/persist?name=www.awesomeserver.com

Example response:

OK - Configuration persisted

Example manual entry:

<Host appBase="www.awesomeserver.com" name="www.awesomeserver.com" deployXML="false" unpackWARs="false">

</Host>

Realm Configuration How-To

Overview

What is a Realm?

A **Realm** is a "database" of usernames and passwords that identify valid users of a web application (or set of web applications), plus an enumeration of the list of *roles* associated with each valid user. You can think of roles as similar to *groups* in Unix-like operating systems, because access to specific web application resources is granted to all users possessing a particular role (rather than enumerating the list of associated usernames). A particular user can have any number of roles associated with their username.

Although the Servlet Specification describes a portable mechanism for applications to *declare* their security requirements (in the web.xml deployment descriptor), there is no portable API defining the interface between a servlet container and the associated user and role information. In many cases, however, it is desirable to "connect" a servlet container to some existing authentication database or mechanism that already exists in the production environment. Therefore, Tomcat defines a Java interface (org.apache.catalina.Realm) that can be implemented by "plug in" components to establish this connection. Six standard plug-ins are provided, supporting connections to various sources of authentication information:

- DataSourceRealm - Accesses authentication information stored in a relational database, accessed via a named JNDI JDBC DataSource.
- JNDIRealm - Accesses authentication information stored in an LDAP based directory server, accessed via a JNDI provider.
- UserDatabaseRealm - Accesses authentication information stored in an UserDatabase JNDI resource, which is typically backed by an XML document (conf/tomcat-users.xml).
- MemoryRealm - Accesses authentication information stored in an in-memory object collection, which is initialized from an XML document (conf/tomcat-users.xml).

- JAASRealm - Accesses authentication information through the Java Authentication & Authorization Service (JAAS) framework.

It is also possible to write your own Realm implementation, and integrate it with Tomcat. To do so, you need to:

- Implement org.apache.catalina.Realm,
- Place your compiled realm in $CATALINA_HOME/lib,
- Declare your realm as described in the "Configuring a Realm" section below,
- Declare your realm to the MBeans Descriptors.

Configuring a Realm

Before getting into the details of the standard Realm implementations, it is important to understand, in general terms, how a Realm is configured. In general, you will be adding an XML element to your conf/server.xml configuration file, that looks something like this:

<Realm className="... class name for this implementation"

... other attributes for this implementation .../>

The <Realm> element can be nested inside any one of of the following Container elements. The location of the Realm element has a direct impact on the "scope" of that Realm (i.e. which web applications will share the same authentication information):

- *Inside an <Engine> element* - This Realm will be shared across ALL web applications on ALL virtual hosts, UNLESS it is overridden by a Realm element nested inside a subordinate <Host> or <Context> element.
- *Inside a <Host> element* - This Realm will be shared across ALL web applications for THIS virtual host, UNLESS it is overridden by a Realm element nested inside a subordinate <Context> element.
- *Inside a <Context> element* - This Realm will be used ONLY for THIS web application.

Common Features

Digested Passwords

For each of the standard Realm implementations, the user's password (by default) is stored in clear text. In many environments, this is undesirable because casual observers of the authentication data can collect enough information to log on successfully, and impersonate other users. To avoid this problem, the standard implementations support the concept of *digesting* user passwords. This allows the stored version of the passwords to be encoded (in a form that is not easily reversible), but that the Realm implementation can still utilize for authentication.

When a standard realm authenticates by retrieving the stored password and comparing it with the value presented by the user, you can select digested passwords by placing a CredentialHandler element inside your <Realm> element. An easy choice to support one of the algorithms SSHA, SHA or MD5 would be the usage of the MessageDigestCredentialHandler. This element must be configured to one of the digest algorithms supported by the java.security.MessageDigest class (SSHA, SHA or MD5). When you select this option, the contents of the password that is stored in the Realm must be the cleartext version of the password, as digested by the specified algorithm.

When the authenticate() method of the Realm is called, the (cleartext) password specified by the user is itself digested by the same algorithm, and the result is compared with the value returned by the Realm. An equal match implies that the cleartext version of the original password is the same as the one presented by the user, so that this user should be authorized.

To calculate the digested value of a cleartext password, two convenience techniques are supported:

- If you are writing an application that needs to calculate digested passwords dynamically, call the static Digest() method of the org.apache.catalina.realm.RealmBase class, passing the cleartext password, the digest algorithm name and the encoding as arguments. This method will return the digested password.

- If you want to execute a command line utility to calculate the digested password, simply execute

CATALINA_HOME/bin/digest.[bat|sh] -a {algorithm} {cleartext-password}

and the digested version of this cleartext password will be returned to standard output.

If using digested passwords with DIGEST authentication, the cleartext used to generate the digest is different and the digest must use one iteration of the MD5 algorithm with no salt. In the examples above {cleartext-password} must be replaced with {username}:{realm}:{cleartext-password}. For example, in a development environment this might take the form testUser:Authentication required:testPassword. The value for {realm} is taken from the <realm-name> element of the web application's <login-config>. If not specified in web.xml, the default value of Authentication required is used.

Usernames and/or passwords using encodings other than the platform default are supported using

CATALINA_HOME/bin/digest.[bat|sh] -a {algorithm} -e {encoding} {input}

but care is required to ensure that the input is correctly passed to the digester. The digester returns {input}:{digest}. If the input appears corrupted in the return, the digest will be invalid.

The output format of the digest is {salt}${iterations}${digest}. If the salt length is zero and the iteration count is one, the output is simplified to {digest}.

The full syntax of CATALINA_HOME/bin/digest.[bat|sh] is:

CATALINA_HOME/bin/digest.[bat|sh] [-a <algorithm>] [-e <encoding>]

[-i <iterations>] [-s <salt-length>] [-k <key-length>]

[-h <handler-class-name>] [-f <password-file> | <credentials>]

- **-a** - The algorithm to use to generate the stored credential. If not specified, the default for the handler will be used. If neither handler nor algorithm is specified then a default of SHA-512 will be used

- **-e** - The encoding to use for any byte to/from character conversion that may be necessary. If not specified, the system encoding (Charset#defaultCharset()) will be used.
- **-i** - The number of iterations to use when generating the stored credential. If not specified, the default for the CredentialHandler will be used.
- **-s** - The length (in bytes) of salt to generate and store as part of the credential. If not specified, the default for the CredentialHandler will be used.
- **-k** - The length (in bits) of the key(s), if any, created while generating the credential. If not specified, the default for the CredentialHandler will be used.
- **-h** - The fully qualified class name of the CredentialHandler to use. If not specified, the built-in handlers will be tested in turn (MessageDigestCredentialHandler then SecretKeyCredentialHandler) and the first one to accept the specified algorithm will be used.
- **-f** - The name of the file that contains passwords to encode. Each line in the file should contain only one password. Using this option ignores other password input.

Example Application

The example application shipped with Tomcat includes an area that is protected by a security constraint, utilizing form-based login. To access it, point your browser at http://localhost:8080/examples/jsp/security/protected/ and log on with one of the usernames and passwords described for the default UserDatabaseRealm.

Manager Application

If you wish to use the Manager Application to deploy and undeploy applications in a running Tomcat installation, you MUST add the "manager-gui" role to at least one username in your selected Realm implementation. This is because the manager web application itself uses a security constraint that requires role "manager-gui" to access ANY request URI within the HTML interface of that

application.

For security reasons, no username in the default Realm (i.e. using conf/tomcat-users.xml is assigned the "manager-gui" role. Therefore, no one will be able to utilize the features of this application until the Tomcat administrator specifically assigns this role to one or more users.

Realm Logging

Debugging and exception messages logged by a Realm will be recorded by the logging configuration associated with the container for the realm: its surrounding Context, Host, or Engine.

Standard Realm Implementations

DataSourceRealm

Introduction

DataSourceRealm is an implementation of the Tomcat Realm interface that looks up users in a relational database accessed via a JNDI named JDBC DataSource. There is substantial configuration flexibility that lets you adapt to existing table and column names, as long as your database structure conforms to the following requirements:

- There must be a table, referenced below as the *users* table, that contains one row for every valid user that this Realm should recognize.
- The *users* table must contain at least two columns (it may contain more if your existing applications required it):
    - Username to be recognized by Tomcat when the user logs in.
    - Password to be recognized by Tomcat when the user logs in. This value may in cleartext or digested - see below for more information.
- There must be a table, referenced below as the *user roles* table, that contains one row for every valid role that is assigned to a particular user. It is legal for a user to have zero, one, or more than one valid role.

- The *user roles* table must contain at least two columns (it may contain more if your existing applications required it):
    - Username to be recognized by Tomcat (same value as is specified in the *users* table).
    - Role name of a valid role associated with this user.

Quick Start

To set up Tomcat to use DataSourceRealm, you will need to follow these steps:

1. If you have not yet done so, create tables and columns in your database that conform to the requirements described above.
2. Configure a database username and password for use by Tomcat, that has at least read only access to the tables described above. (Tomcat will never attempt to write to these tables.)
3. Configure a JNDI named JDBC DataSource for your database. Refer to the JNDI DataSource Example How-To for information on how to configure a JNDI named JDBC DataSource. Be sure to set the Realm's localDataSource attribute appropriately, depending on where the JNDI DataSource is defined.
4. Set up a <Realm> element, as described below, in your $CATALINA_BASE/conf/server.xml file.
5. Restart Tomcat if it is already running.

Realm Element Attributes

To configure DataSourceRealm, you will create a <Realm> element and nest it in your $CATALINA_BASE/conf/server.xml file, as described above. The attributes for the DataSourceRealm are defined in the Realm configuration documentation.

Example

An example SQL script to create the needed tables might look something like this (adapt the syntax as required for your particular database):

```
create table users (
```

```
    user_name               varchar(15) not null primary key,
    user_pass               varchar(15) not null
);


create table user_roles (
    user_name               varchar(15) not null,
    role_name               varchar(15) not null,
    primary key (user_name, role_name)
);
```

Here is an example for using a MySQL database called "authority", configured with the tables described above, and accessed with the JNDI JDBC DataSource with name "java:/comp/env/jdbc/authority".

```
<Realm className="org.apache.catalina.realm.DataSourceRealm"
    dataSourceName="jdbc/authority"
    userTable="users" userNameCol="user_name" userCredCol="user_pass"
    userRoleTable="user_roles" roleNameCol="role_name"/>
```

Additional Notes

DataSourceRealm operates according to the following rules:

- When a user attempts to access a protected resource for the first time, Tomcat will call the authenticate() method of this Realm. Thus, any changes you have made to the database directly (new users, changed passwords or roles, etc.) will be immediately reflected.
- Once a user has been authenticated, the user (and their associated roles) are cached within Tomcat for the duration of the user's login. (For FORM-based authentication, that means until the session times out or is invalidated; for BASIC authentication, that means until the user closes their browser). The cached user is **not** saved and restored across

sessions serialisations. Any changes to the database information for an already authenticated user will **not** be reflected until the next time that user logs on again.

- Administering the information in the *users* and *user roles* table is the responsibility of your own applications. Tomcat does not provide any built-in capabilities to maintain users and roles.

JNDIRealm

Introduction

JNDIRealm is an implementation of the Tomcat Realm interface that looks up users in an LDAP directory server accessed by a JNDI provider (typically, the standard LDAP provider that is available with the JNDI API classes). The realm supports a variety of approaches to using a directory for authentication.

Connecting to the directory

The realm's connection to the directory is defined by the **connectionURL** configuration attribute. This is a URL whose format is defined by the JNDI provider. It is usually an LDAP URL that specifies the domain name of the directory server to connect to, and optionally the port number and distinguished name (DN) of the required root naming context.

If you have more than one provider you can configure an **alternateURL**. If a socket connection cannot be made to the provider at the **connectionURL** an attempt will be made to use the **alternateURL**.

When making a connection in order to search the directory and retrieve user and role information, the realm authenticates itself to the directory with the username and password specified by the **connectionName** and **connectionPassword** properties. If these properties are not specified the connection is anonymous. This is sufficient in many cases.

Selecting the user's directory entry

Each user that can be authenticated must be represented in the directory by an individual entry that corresponds to an element in the initial DirContext defined

by the **connectionURL** attribute. This user entry must have an attribute containing the username that is presented for authentication.

Often the distinguished name of the user's entry contains the username presented for authentication but is otherwise the same for all users. In this case the **userPattern** attribute may be used to specify the DN, with "{0}" marking where the username should be substituted.

Otherwise the realm must search the directory to find a unique entry containing the username. The following attributes configure this search:

- **userBase** - the entry that is the base of the subtree containing users. If not specified, the search base is the top-level context.
- **userSubtree** - the search scope. Set to true if you wish to search the entire subtree rooted at the **userBase** entry. The default value of false requests a single-level search including only the top level.
- **userSearch** - pattern specifying the LDAP search filter to use after substitution of the username.

Authenticating the user

- **Bind mode**

By default the realm authenticates a user by binding to the directory with the DN of the entry for that user and the password presented by the user. If this simple bind succeeds the user is considered to be authenticated.

For security reasons a directory may store a digest of the user's password rather than the clear text version (see Digested Passwords for more information). In that case, as part of the simple bind operation the directory automatically computes the correct digest of the plaintext password presented by the user before validating it against the stored value. In bind mode, therefore, the realm is not involved in digest processing. The **digest** attribute is not used, and will be ignored if set.

- **Comparison mode**

Alternatively, the realm may retrieve the stored password from the directory and

compare it explicitly with the value presented by the user. This mode is configured by setting the **userPassword** attribute to the name of a directory attribute in the user's entry that contains the password.

Comparison mode has some disadvantages. First, the **connectionName** and **connectionPassword** attributes must be configured to allow the realm to read users' passwords in the directory. For security reasons this is generally undesirable; indeed many directory implementations will not allow even the directory manager to read these passwords. In addition, the realm must handle password digests itself, including variations in the algorithms used and ways of representing password hashes in the directory. However, the realm may sometimes need access to the stored password, for example to support HTTP Digest Access Authentication (RFC 2069). (Note that HTTP digest authentication is different from the storage of password digests in the repository for user information as discussed above).

Assigning roles to the user

The directory realm supports two approaches to the representation of roles in the directory:

- **Roles as explicit directory entries**

Roles may be represented by explicit directory entries. A role entry is usually an LDAP group entry with one attribute containing the name of the role and another whose values are the distinguished names or usernames of the users in that role. The following attributes configure a directory search to find the names of roles associated with the authenticated user:

    - **roleBase** - the base entry for the role search. If not specified, the search base is the top-level directory context.
    - **roleSubtree** - the search scope. Set to true if you wish to search the entire subtree rooted at the roleBase entry. The default value of false requests a single-level search including the top level only.
    - **roleSearch** - the LDAP search filter for selecting role entries. It optionally includes pattern replacements "{0}" for the

distinguished name and/or "{1}" for the username and/or "{2}" for an attribute from user's directory entry, of the authenticated user. Use **userRoleAttribute** to specify the name of the attribute that provides the value for "{2}".

  - **roleName** - the attribute in a role entry containing the name of that role.
  - **roleNested** - enable nested roles. Set to true if you want to nest roles in roles. If configured, then every newly found roleName and distinguished Name will be recursively tried for a new role search. The default value is false.

- **Roles as an attribute of the user entry**

Role names may also be held as the values of an attribute in the user's directory entry. Use **userRoleName** to specify the name of this attribute.

A combination of both approaches to role representation may be used.

Quick Start

To set up Tomcat to use JNDIRealm, you will need to follow these steps:

1. Make sure your directory server is configured with a schema that matches the requirements listed above.
2. If required, configure a username and password for use by Tomcat, that has read only access to the information described above. (Tomcat will never attempt to modify this information.)
3. Set up a <Realm> element, as described below, in your $CATALINA_BASE/conf/server.xml file.
4. Restart Tomcat if it is already running.

Realm Element Attributes

To configure JNDIRealm, you will create a <Realm> element and nest it in your $CATALINA_BASE/conf/server.xml file, as described above. The attributes for the JNDIRealm are defined in the Realm configuration documentation.

Example

Creation of the appropriate schema in your directory server is beyond the scope of this document, because it is unique to each directory server implementation. In the examples below, we will assume that you are using a distribution of the OpenLDAP directory server (version 2.0.11 or later), which can be downloaded from https://www.openldap.org. Assume that your slapd.conf file contains the following settings (among others):

database ldbm

suffix dc="mycompany",dc="com"

rootdn "cn=Manager,dc=mycompany,dc=com"

rootpw secret

We will assume for connectionURL that the directory server runs on the same machine as Tomcat.
See http://docs.oracle.com/javase/7/docs/technotes/guides/jndi/index.html for more information about configuring and using the JNDI LDAP provider.

Next, assume that this directory server has been populated with elements as shown below (in LDIF format):

Define top-level entry

dn: dc=mycompany,dc=com

objectClass: dcObject

dc:mycompany

Define an entry to contain people

searches for users are based on this entry

dn: ou=people,dc=mycompany,dc=com

objectClass: organizationalUnit

ou: people

```
# Define a user entry for Janet Jones

dn: uid=jjones,ou=people,dc=mycompany,dc=com

objectClass: inetOrgPerson

uid: jjones

sn: jones

cn: janet jones

mail: j.jones@mycompany.com

userPassword: janet


# Define a user entry for Fred Bloggs

dn: uid=fbloggs,ou=people,dc=mycompany,dc=com

objectClass: inetOrgPerson

uid: fbloggs

sn: bloggs

cn: fred bloggs

mail: f.bloggs@mycompany.com

userPassword: fred


# Define an entry to contain LDAP groups

# searches for roles are based on this entry

dn: ou=groups,dc=mycompany,dc=com

objectClass: organizationalUnit

ou: groups
```

```
# Define an entry for the "tomcat" role
dn: cn=tomcat,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: tomcat
uniqueMember: uid=jjones,ou=people,dc=mycompany,dc=com
uniqueMember: uid=fbloggs,ou=people,dc=mycompany,dc=com

# Define an entry for the "role1" role
dn: cn=role1,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: role1
uniqueMember: uid=fbloggs,ou=people,dc=mycompany,dc=com
```

An example Realm element for the OpenLDAP directory server configured as described above might look like this, assuming that users use their uid (e.g. jjones) to login to the application and that an anonymous connection is sufficient to search the directory and retrieve role information:

```
<Realm      className="org.apache.catalina.realm.JNDIRealm"
        connectionURL="ldap://localhost:389"
           userPattern="uid={0},ou=people,dc=mycompany,dc=com"
               roleBase="ou=groups,dc=mycompany,dc=com"
               roleName="cn"
             roleSearch="(uniqueMember={0})"
/>
```

With this configuration, the realm will determine the user's distinguished name

by substituting the username into the userPattern, authenticate by binding to the directory with this DN and the password received from the user, and search the directory to find the user's roles.

Now suppose that users are expected to enter their email address rather than their userid when logging in. In this case the realm must search the directory for the user's entry. (A search is also necessary when user entries are held in multiple subtrees corresponding perhaps to different organizational units or company locations).

Further, suppose that in addition to the group entries you want to use an attribute of the user's entry to hold roles. Now the entry for Janet Jones might read as follows:

dn: uid=jjones,ou=people,dc=mycompany,dc=com

objectClass: inetOrgPerson

uid: jjones

sn: jones

cn: janet jones

mail: j.jones@mycompany.com

memberOf: role2

memberOf: role3

userPassword: janet

This realm configuration would satisfy the new requirements:

```
<Realm      className="org.apache.catalina.realm.JNDIRealm"
        connectionURL="ldap://localhost:389"
              userBase="ou=people,dc=mycompany,dc=com"
            userSearch="(mail={0})"
          userRoleName="memberOf"
```

```
            roleBase="ou=groups,dc=mycompany,dc=com"
            roleName="cn"
          roleSearch="(uniqueMember={0})"
/>
```

Now when Janet Jones logs in as "j.jones@mycompany.com", the realm searches the directory for a unique entry with that value as its mail attribute and attempts to bind to the directory as uid=jjones,ou=people,dc=mycompany,dc=com with the given password. If authentication succeeds, they are assigned three roles: "role2" and "role3", the values of the "memberOf" attribute in their directory entry, and "tomcat", the value of the "cn" attribute in the only group entry of which they are a member.

Finally, to authenticate the user by retrieving the password from the directory and making a local comparison in the realm, you might use a realm configuration like this:

```
<Realm      className="org.apache.catalina.realm.JNDIRealm"
      connectionName="cn=Manager,dc=mycompany,dc=com"
connectionPassword="secret"
        connectionURL="ldap://localhost:389"
          userPassword="userPassword"
            userPattern="uid={0},ou=people,dc=mycompany,dc=com"
            roleBase="ou=groups,dc=mycompany,dc=com"
            roleName="cn"
          roleSearch="(uniqueMember={0})"
/>
```

However, as discussed above, the default bind mode for authentication is usually to be preferred.

Additional Notes

JNDIRealm operates according to the following rules:

- When a user attempts to access a protected resource for the first time, Tomcat will call the authenticate() method of this Realm. Thus, any changes you have made to the directory (new users, changed passwords or roles, etc.) will be immediately reflected.
- Once a user has been authenticated, the user (and their associated roles) are cached within Tomcat for the duration of the user's login. (For FORM-based authentication, that means until the session times out or is invalidated; for BASIC authentication, that means until the user closes their browser). The cached user is **not** saved and restored across sessions serialisations. Any changes to the directory information for an already authenticated user will **not** be reflected until the next time that user logs on again.
- Administering the information in the directory server is the responsibility of your own applications. Tomcat does not provide any built-in capabilities to maintain users and roles.

UserDatabaseRealm

Introduction

UserDatabaseRealm is an implementation of the Tomcat Realm interface that uses a JNDI resource to store user information. By default, the JNDI resource is backed by an XML file. It is not designed for large-scale production use. At startup time, the UserDatabaseRealm loads information about all users, and their corresponding roles, from an XML document (by default, this document is loaded from $CATALINA_BASE/conf/tomcat-users.xml). The users, their passwords and their roles may all be editing dynamically, typically via JMX. Changes may be saved and will be reflected in the XML file.

Realm Element Attributes

To configure UserDatabaseRealm, you will create a <Realm> element and nest it in your $CATALINA_BASE/conf/server.xml file, as described above. The attributes for the UserDatabaseRealm are defined in the Realm configuration

documentation.

User File Format

For the XML file based UserDatabase, the users file uses the same format as the MemoryRealm.

Example

The default installation of Tomcat is configured with a UserDatabaseRealm nested inside the <Engine> element, so that it applies to all virtual hosts and web applications. The default contents of the conf/tomcat-users.xml file is:

```
<tomcat-users>
  <user username="tomcat" password="tomcat" roles="tomcat" />
  <user username="role1"  password="tomcat" roles="role1"  />
  <user username="both"   password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

Additional Notes

UserDatabaseRealm operates according to the following rules:

- When Tomcat first starts up, it loads all defined users and their associated information from the users file. Changes made to the data in this file will **not** be recognized until Tomcat is restarted. Changes may be made via the UserDatabase resource. Tomcat provides MBeans that may be accessed via JMX for this purpose.
- When a user attempts to access a protected resource for the first time, Tomcat will call the authenticate() method of this Realm.
- Once a user has been authenticated, the user becomes associated within Tomcat for the duration of the user's login. (For FORM-based authentication, that means until the session times out or is invalidated; for BASIC authentication, that means until the user closes their browser). However, the user roles will still reflect the UserDatabase contents, unlike for the other realms. If a user is removed from the database, it will be

considered to have no roles. The useStaticPrincipal attribute of the UserDatabaseRealm can be used to instead cache the user along with all its roles. The cached user is **not** saved and restored across sessions serialisations. When the user's principal object is serialized for any reason, it will also be replaced by a static equivalent object with roles that will no longer reflect the database contents.

MemoryRealm

Introduction

MemoryRealm is a simple demonstration implementation of the Tomcat Realm interface. It is not designed for production use. At startup time, MemoryRealm loads information about all users, and their corresponding roles, from an XML document (by default, this document is loaded from $CATALINA_BASE/conf/tomcat-users.xml). Changes to the data in this file are not recognized until Tomcat is restarted.

Realm Element Attributes

To configure MemoryRealm, you will create a <Realm> element and nest it in your $CATALINA_BASE/conf/server.xml file, as described above. The attributes for the MemoryRealm are defined in the Realm configuration documentation.

User File Format

The users file (by default, conf/tomcat-users.xml must be an XML document, with a root element <tomcat-users>. Nested inside the root element will be a <user> element for each valid user, consisting of the following attributes:

- **name** - Username this user must log on with.
- **password** - Password this user must log on with (in clear text if the digest attribute was not set on the <Realm> element, or digested appropriately as described here otherwise).
- **roles** - Comma-delimited list of the role names associated with this user.

Additional Notes

MemoryRealm operates according to the following rules:

- When Tomcat first starts up, it loads all defined users and their associated information from the users file. Changes to the data in this file will **not** be recognized until Tomcat is restarted.
- When a user attempts to access a protected resource for the first time, Tomcat will call the authenticate() method of this Realm.
- Once a user has been authenticated, the user (and their associated roles) are cached within Tomcat for the duration of the user's login. (For FORM-based authentication, that means until the session times out or is invalidated; for BASIC authentication, that means until the user closes their browser). The cached user is **not** saved and restored across sessions serialisations.
- Administering the information in the users file is the responsibility of your application. Tomcat does not provide any built-in capabilities to maintain users and roles.

JAASRealm

Introduction

JAASRealm is an implementation of the Tomcat Realm interface that authenticates users through the Java Authentication & Authorization Service (JAAS) framework which is now provided as part of the standard Java SE API.

Using JAASRealm gives the developer the ability to combine practically any conceivable security realm with Tomcat's CMA.

JAASRealm is prototype for Tomcat of the JAAS-based J2EE authentication framework for J2EE v1.4, based on the JCP Specification Request 196 to enhance container-managed security and promote 'pluggable' authentication mechanisms whose implementations would be container-independent.

Based on the JAAS login module and principal (see javax.security.auth.spi.LoginModule and javax.security.Principal), you can develop your own security mechanism or wrap another third-party mechanism for integration with the CMA as implemented by Tomcat.

Quick Start

To set up Tomcat to use JAASRealm with your own JAAS login module, you will need to follow these steps:

1. Write your own LoginModule, User and Role classes based on JAAS (see the JAAS Authentication Tutorial and the JAAS Login Module Developer's Guide) to be managed by the JAAS Login Context (javax.security.auth.login.LoginContext) When developing your LoginModule, note that JAASRealm's built-in CallbackHandler only recognizes the NameCallback and PasswordCallback at present.
2. Although not specified in JAAS, you should create separate classes to distinguish between users and roles, extending javax.security.Principal, so that Tomcat can tell which Principals returned from your login module are users and which are roles (see org.apache.catalina.realm.JAASRealm). Regardless, the first Principal returned is *always* treated as the user Principal.
3. Place the compiled classes on Tomcat's classpath
4. Set up a login.config file for Java (see JAAS LoginConfig file) and tell Tomcat where to find it by specifying its location to the JVM, for instance by setting the environment variable: JAVA_OPTS=$JAVA_OPTS -Djava.security.auth.login.config==$CATALINA_BASE/conf/jaas.config
5. Configure your security-constraints in your web.xml for the resources you want to protect
6. Configure the JAASRealm module in your server.xml
7. Restart Tomcat if it is already running.

Realm Element Attributes

To configure JAASRealm as for step 6 above, you create a <Realm> element and nest it in your $CATALINA_BASE/conf/server.xml file within your <Engine> node. The attributes for the JAASRealm are defined in the Realm configuration documentation.

Example

Here is an example of how your server.xml snippet should look.

```
<Realm className="org.apache.catalina.realm.JAASRealm"
                      appName="MyFooRealm"
      userClassNames="org.foobar.realm.FooUser"
       roleClassNames="org.foobar.realm.FooRole"/>
```

It is the responsibility of your login module to create and save User and Role objects representing Principals for the user (javax.security.auth.Subject). If your login module doesn't create a user object but also doesn't throw a login exception, then the Tomcat CMA will break and you will be left at the http://localhost:8080/myapp/j_security_check URI or at some other unspecified location.

The flexibility of the JAAS approach is two-fold:

- you can carry out whatever processing you require behind the scenes in your own login module.
- you can plug in a completely different LoginModule by changing the configuration and restarting the server, without any code changes to your application.

Additional Notes

- When a user attempts to access a protected resource for the first time, Tomcat will call the authenticate() method of this Realm. Thus, any changes you have made in the security mechanism directly (new users, changed passwords or roles, etc.) will be immediately reflected.
- Once a user has been authenticated, the user (and their associated roles) are cached within Tomcat for the duration of the user's login. For FORM-based authentication, that means until the session times out or is invalidated; for BASIC authentication, that means until the user closes their browser. Any changes to the security information for an already authenticated user will **not** be reflected until the next time that user logs

on again.

- As with other Realm implementations, digested passwords are supported if the <Realm> element in server.xml contains a digest attribute; JAASRealm's CallbackHandler will digest the password prior to passing it back to the LoginModule

CombinedRealm

Introduction

CombinedRealm is an implementation of the Tomcat Realm interface that authenticates users through one or more sub-Realms.

Using CombinedRealm gives the developer the ability to combine multiple Realms of the same or different types. This can be used to authenticate against different sources, provide fall back in case one Realm fails or for any other purpose that requires multiple Realms.

Sub-realms are defined by nesting Realm elements inside the Realm element that defines the CombinedRealm. Authentication will be attempted against each Realm in the order they are listed. Authentication against any Realm will be sufficient to authenticate the user.

Realm Element Attributes

To configure a CombinedRealm, you create a <Realm> element and nest it in your $CATALINA_BASE/conf/server.xml file within your <Engine> or <Host>. You can also nest inside a <Context> node in a context.xml file.

Example

Here is an example of how your server.xml snippet should look to use a UserDatabase Realm and a DataSource Realm.

```
<Realm className="org.apache.catalina.realm.CombinedRealm" >
    <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                resourceName="UserDatabase"/>
    <Realm className="org.apache.catalina.realm.DataSourceRealm"
```

```
                dataSourceName="jdbc/authority"

                userTable="users" userNameCol="user_name"
userCredCol="user_pass"

                userRoleTable="user_roles" roleNameCol="role_name"/>

</Realm>
```

LockOutRealm

Introduction

LockOutRealm is an implementation of the Tomcat Realm interface that extends the CombinedRealm to provide lock out functionality to provide a user lock out mechanism if there are too many failed authentication attempts in a given period of time.

To ensure correct operation, there is a reasonable degree of synchronisation in this Realm.

This Realm does not require modification to the underlying Realms or the associated user storage mechanisms. It achieves this by recording all failed logins, including those for users that do not exist. To prevent a DOS by deliberating making requests with invalid users (and hence causing this cache to grow) the size of the list of users that have failed authentication is limited.

Sub-realms are defined by nesting Realm elements inside the Realm element that defines the LockOutRealm. Authentication will be attempted against each Realm in the order they are listed. Authentication against any Realm will be sufficient to authenticate the user.

Realm Element Attributes

To configure a LockOutRealm, you create a <Realm> element and nest it in your $CATALINA_BASE/conf/server.xml file within your <Engine> or <Host>. You can also nest inside a <Context> node in a context.xml file. The attributes for the LockOutRealm are defined in the Realm configuration documentation.

Example

Here is an example of how your server.xml snippet should look to add lock out functionality to a UserDatabase Realm.

```
<Realm className="org.apache.catalina.realm.LockOutRealm" >
    <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                resourceName="UserDatabase"/>
</Realm>
```

JNDI Resources How-To

Introduction

Tomcat provides a JNDI **InitialContext** implementation instance for each web application running under it, in a manner that is compatible with those provided by a Jakarta EE application server. The Jakarta EE standard provides a standard set of elements in the /WEB-INF/web.xml file to reference/define resources.

See the following Specifications for more information about programming APIs for JNDI, and for the features supported by Jakarta EE servers, which Tomcat emulates for the services that it provides:

- Java Naming and Directory Interface (included in JDK 1.4 onwards)
- Jakarta EE Platform Specification (in particular, see Chapter 5 on *Naming*)

web.xml configuration

The following elements may be used in the web application deployment descriptor (/WEB-INF/web.xml) of your web application to define resources:

- **<env-entry>** - Environment entry, a single-value parameter that can be used to configure how the application will operate.
- **<resource-ref>** - Resource reference, which is typically to an object factory for resources such as a JDBC DataSource, a Jakarta Mail Session, or custom object factories configured into Tomcat.
- **<resource-env-ref>** - Resource environment reference, a new variation of resource-ref added in Servlet 2.4 that is simpler to configure for

resources that do not require authentication information.

Providing that Tomcat is able to identify an appropriate resource factory to use to create the resource and that no further configuration information is required, Tomcat will use the information in /WEB-INF/web.xml to create the resource.

Tomcat provides a number of Tomcat specific options for JNDI resources that cannot be specified in web.xml. These include closeMethod that enables faster cleaning-up of JNDI resources when a web application stops and singleton that controls whether or not a new instance of the resource is created for every JNDI lookup. To use these configuration options the resource must be specified in a web application's <Context> element or in the **<GlobalNamingResources>** element of $CATALINA_BASE/conf/server.xml.

context.xml configuration

If Tomcat is unable to identify the appropriate resource factory and/or additional configuration information is required, additional Tomcat specific configuration must be specified before Tomcat can create the resource. Tomcat specific resource configuration is entered in the <Context> elements that can be specified in either $CATALINA_BASE/conf/server.xml or, preferably, the per-web-application context XML file (META-INF/context.xml).

Tomcat specific resource configuration is performed using the following elements in the <Context> element:

- <Environment> - Configure names and values for scalar environment entries that will be exposed to the web application through the JNDI InitialContext (equivalent to the inclusion of an <env-entry> element in the web application deployment descriptor).
- <Resource> - Configure the name and data type of a resource made available to the application (equivalent to the inclusion of a <resource-ref> element in the web application deployment descriptor).
- <ResourceLink> - Add a link to a resource defined in the global JNDI context. Use resource links to give a web application access to a resource defined in the <GlobalNamingResources> child element of

the <Server> element.

- <Transaction> - Add a resource factory for instantiating the UserTransaction object instance that is available at java:comp/UserTransaction.

Any number of these elements may be nested inside a <Context> element and will be associated only with that particular web application.

If a resource has been defined in a <Context> element it is not necessary for that resource to be defined in /WEB-INF/web.xml. However, it is recommended to keep the entry in /WEB-INF/web.xml to document the resource requirements for the web application.

Where the same resource name has been defined for a <env-entry> element included in the web application deployment descriptor (/WEB-INF/web.xml) and in an <Environment> element as part of the <Context> element for the web application, the values in the deployment descriptor will take precedence **only** if allowed by the corresponding <Environment> element (by setting the override attribute to "true").

Global configuration

Tomcat maintains a separate namespace of global resources for the entire server. These are configured in the **<GlobalNamingResources>** element of $CATALINA_BASE/conf/server.xml. You may expose these resources to web applications by using a <ResourceLink> to include it in the per-web-application context.

If a resource has been defined using a <ResourceLink>, it is not necessary for that resource to be defined in /WEB-INF/web.xml. However, it is recommended to keep the entry in /WEB-INF/web.xml to document the resource requirements for the web application.

Using resources

The InitialContext is configured as a web application is initially deployed, and is made available to web application components (for read-only access). All configured entries and resources are placed in the java:comp/env portion of the

JNDI namespace, so a typical access to a resource - in this case, to a JDBC DataSource - would look something like this:

```
// Obtain our environment naming context

Context initCtx = new InitialContext();

Context envCtx = (Context) initCtx.lookup("java:comp/env");


// Look up our data source

DataSource ds = (DataSource)

    envCtx.lookup("jdbc/EmployeeDB");


// Allocate and use a connection from the pool

Connection conn = ds.getConnection();

... use this connection to access the database ...

conn.close();
```

Tomcat Standard Resource Factories

Tomcat includes a series of standard resource factories that can provide services to your web applications, but give you configuration flexibility (via the <Context> element) without modifying the web application or the deployment descriptor. Each subsection below details the configuration and usage of the standard resource factories.

See Adding Custom Resource Factories for information about how to create, install, configure, and use your own custom resource factory classes with Tomcat.

NOTE - Of the standard resource factories, only the "JDBC Data Source" and "User Transaction" factories are mandated to be available on other platforms, and then they are required only if the platform implements the Jakarta EE specs. All other standard resource factories, plus custom resource factories that you

write yourself, are specific to Tomcat and cannot be assumed to be available on other containers.

Generic JavaBean Resources

0. Introduction

This resource factory can be used to create objects of *any* Java class that conforms to standard JavaBeans naming conventions (i.e. it has a zero-arguments constructor, and has property setters that conform to the setFoo() naming pattern. The resource factory will only create a new instance of the appropriate bean class every time a lookup() for this entry is made if the singleton attribute of the factory is set to false.

The steps required to use this facility are described below.

1. Create Your JavaBean Class

Create the JavaBean class which will be instantiated each time that the resource factory is looked up. For this example, assume you create a class com.mycompany.MyBean, which looks like this:

```
package com.mycompany;


public class MyBean {


    private String foo = "Default Foo";


    public String getFoo() {
        return (this.foo);
    }


    public void setFoo(String foo) {
```

```
        this.foo = foo;
    }

    private int bar = 0;

    public int getBar() {
        return (this.bar);
    }

    public void setBar(int bar) {
        this.bar = bar;
    }

}
```

2. Declare Your Resource Requirements

Next, modify your web application deployment descriptor (/WEB-INF/web.xml) to declare the JNDI name under which you will request new instances of this bean. The simplest approach is to use a <resource-env-ref> element, like this:

```
<resource-env-ref>
    <description>
        Object factory for MyBean instances.
    </description>
    <resource-env-ref-name>
        bean/MyBeanFactory
```

```
    </resource-env-ref-name>
    <resource-env-ref-type>
        com.mycompany.MyBean
    </resource-env-ref-type>
</resource-env-ref>
```

WARNING - Be sure you respect the element ordering that is required by the DTD for web application deployment descriptors! See the Servlet Specification for details.

3. Code Your Application's Use Of This Resource

A typical use of this resource environment reference might look like this:

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
MyBean bean = (MyBean) envCtx.lookup("bean/MyBeanFactory");


writer.println("foo = " + bean.getFoo() + ", bar = " +
                   bean.getBar());
```

4. Configure Tomcat's Resource Factory

To configure Tomcat's resource factory, add an element like this to the <Context> element for this web application.

```
<Context ...>
    ...
    <Resource name="bean/MyBeanFactory" auth="Container"
              type="com.mycompany.MyBean"
              factory="org.apache.naming.factory.BeanFactory"
              bar="23"/>
```

...

</Context>

Note that the resource name (here, bean/MyBeanFactory must match the value specified in the web application deployment descriptor. We are also initializing the value of the bar property, which will cause setBar(23) to be called before the new bean is returned. Because we are not initializing the foo property (although we could have), the bean will contain whatever default value is set up by its constructor.

If the bean property is of type String, the BeanFactory will call the property setter using the provided property value. If the bean property type is a primitive or a primitive wrapper, the the BeanFactory will convert the value to the appropriate primitive or primitive wrapper and then use that value when calling the setter. Some beans have properties with types that cannot automatically be converted from String. If the bean provides an alternative setter with the same name that does take a String, the BeanFactory will attempt to use that setter. If the BeanFactory cannot use the value or perform an appropriate conversion, setting the property will fail with a NamingException.

The forceString property available in earlier Tomcat releases has been removed as a security hardening measure.

Memory UserDatabase Resources

0. Introduction

UserDatabase resources are typically configured as global resources for use by a UserDatabase realm. Tomcat includes a UserDatabaseFactory that creates UserDatabase resources backed by an XML file - usually tomcat-users.xml.

The steps required to set up a global UserDatabase resource are described below.

1. Create/edit the XML file

The XML file is typically located at $CATALINA_BASE/conf/tomcat-users.xml however, you are free to locate the file anywhere on the file system. It

is recommended that the XML files are placed in $CATALINA_BASE/conf. A typical XML would look like:

```
<?xml version="1.0" encoding="UTF-8"?>

<tomcat-users>

  <role rolename="tomcat"/>

  <role rolename="role1"/>

  <user username="tomcat" password="tomcat" roles="tomcat"/>

  <user username="both" password="tomcat" roles="tomcat,role1"/>

  <user username="role1" password="tomcat" roles="role1"/>

</tomcat-users>
```

2. Declare Your Resource

Next, modify $CATALINA_BASE/conf/server.xml to create the UserDatabase resource based on your XML file. It should look something like this:

```
<Resource name="UserDatabase"

          auth="Container"

          type="org.apache.catalina.UserDatabase"

          description="User database that can be updated and saved"

          factory="org.apache.catalina.users.MemoryUserDatabaseFactory"

          pathname="conf/tomcat-users.xml"

          readonly="false" />
```

The pathname attribute can be a URL, an absolute path or a relative path. If relative, it is relative to $CATALINA_BASE.

The readonly attribute is optional and defaults to true if not supplied. If the XML is writable then it will be written to when Tomcat starts. **WARNING:** When the file is written it will inherit the default file permissions for the user Tomcat is running as. Ensure that these are appropriate to maintain the security of your installation.

If referenced in a Realm, the UserDatabase will, by default, monitor pathname for changes and reload the file if a change in the last modified time is observed. This can be disabled by setting the watchSource attribute to false.

3. Configure the Realm

Configure a UserDatabase Realm to use this resource as described in the Realm configuration documentation.

DataSource UserDatabase Resources

0. Introduction

Tomcat also include a UserDatabase that uses a DataSource resource as the backend. The backend resource must be declared in the same JNDI context as the user database that will use it.

The steps required to set up a global UserDatabase resource are described below.

1. Database schema

The database shema for the user database is flexible. It can be the same as the schema used for the DataSourceRealm, with only a table for users (user name, password), and another one listing the roles associated with each user. To support the full UserDatabase features, it must include additional tables for groups, and is compatible with referential integrity between users, groups and roles.

The full featured schema with groups and referential integrity could be:

```
create table users (
    user_name               varchar(32) not null primary key,
    user_pass               varchar(64) not null,
    user_fullname           varchar(128)
    -- Add more attributes as needed
```

```
);

create table roles (
    role_name               varchar(32) not null primary key,
    role_description    varchar(128)
);

create table groups (
    group_name              varchar(32) not null primary key,
    group_description varchar(128)
);

create table user_roles (
    user_name               varchar(32) references users(user_name),
    role_name               varchar(32) references roles(role_name),
    primary key (user_name, role_name)
);

create table user_groups (
    user_name               varchar(32) references users(user_name),
    group_name              varchar(32) references groups(group_name),
    primary key (user_name, group_name)
);
```

```
create table group_roles (
   group_name              varchar(32) references groups(group_name),
   role_name               varchar(32) references roles(role_name),
   primary key (group_name, role_name)
);
```

The minimal schema without the ability to use groups will be (it is the same as for the DataSourceRealm):

```
create table users (
   user_name               varchar(32) not null primary key,
   user_pass               varchar(64) not null,
   -- Add more attributes as needed
);


create table user_roles (
   user_name               varchar(32),
   role_name               varchar(32),
   primary key (user_name, role_name)
);
```

2. Declare Your Resource

Next, modify $CATALINA_BASE/conf/server.xml to create the UserDatabase resource based on your DataSource and its schema. It should look something like this:

```
<Resource name="UserDatabase" auth="Container"
                  type="org.apache.catalina.UserDatabase"
                  description="User database that can be updated and saved"
```

```
factory="org.apache.catalina.users.DataSourceUserDatabaseFactory"
                dataSourceName="jdbc/authority" readonly="false"
                userTable="users" userNameCol="user_name"
userCredCol="user_pass"
                userRoleTable="user_roles" roleNameCol="role_name"
                roleTable="roles" groupTable="groups"
userGroupTable="user_groups"
                groupRoleTable="group_roles" groupNameCol="group_name" />
```

The dataSourceName attribute is the JNDI name of the DataSource that will be the backend for the UserDatabase. It must be declared in the same JNDI Context as the UserDatabase. Please refer to the DataSource resources documentation for further instructions.

The readonly attribute is optional and defaults to true if not supplied. If the database is writable then changes made through the Tomcat management to the UserDatabase can be persisted to the database using the save operation.

Alternately, changes can also be made directly to the backend database.

3. Resource configuration

Attribute	Description
dataSourceName	The name of the JNDI JDBC DataSource for this UserDatabase.
groupNameCol	Name of the column, in the "groups", "group roles" and "user groups" tables, that contains the group's name.
groupRoleTable	Name of the "group roles" table, which must contain columns named by the groupNameCol and roleNameCol attribute

	s.
groupTable	Name of the "groups" table, which must contain columns named by the groupNameCol attribute.
readonly	If this is set to true, then changes to the UserDatabase can be persisted to the DataSource by using the save method. The default value is true.
roleAndGroupDescriptionCol	Name of the column, in the "roles" and "groups" tables, that contains the description for the roles and groups.
roleNameCol	Name of the column, in the "roles", "user roles" and "group roles" tables, which contains a role name assigned to the corresponding user. This attribute is **required** in majority of configurations. See **allRolesMode** attribute of the associated realm for a rare case when it can be omitted.
roleTable	Name of the "roles" table, which must contain columns named by the roleNameCol attribute.
userCredCol	Name of the column, in the "users" table, which contains the user's credentials (i.e. password). If a CredentialHandler is specified, this component will assume that the passwords have been encoded with the specified algorithm. Otherwise, they will be assumed to be in clear text.

userGroupTable	Name of the "user groups" table, which must contain columns named by the userNameCol and groupNameCol attributes.
userNameCol	Name of the column, in the "users", "user groups" and "user roles" tables, that contains the user's username.
userFullNameCol	Name of the column, in the "users" table, that contains the user's full name.
userRoleTable	Name of the "user roles" table, which must contain columns named by the userNameCol and roleNameCol attributes. This attribute is **required** in majority of configurations. See **allRolesMode** attribute of the associated realm for a rare case when it can be omitted.
userTable	Name of the "users" table, which must contain columns named by the userNameCol and userCredCol attributes.

4. Configure the Realm

Configure a UserDatabase Realm to use this resource as described in the Realm configuration documentation.

Jakarta Mail Sessions

0. Introduction

In many web applications, sending electronic mail messages is a required part of the system's functionality. The Jakarta Mail API makes this process relatively straightforward, but requires many configuration details that the client

application must be aware of (including the name of the SMTP host to be used for message sending).

Tomcat includes a standard resource factory that will create jakarta.mail.Session session instances for you, already configured to connect to an SMTP server. In this way, the application is totally insulated from changes in the email server configuration environment - it simply asks for, and receives, a preconfigured session whenever needed.

The steps required for this are outlined below.

1. Declare Your Resource Requirements

The first thing you should do is modify the web application deployment descriptor (/WEB-INF/web.xml) to declare the JNDI name under which you will look up preconfigured sessions. By convention, all such names should resolve to the mail subcontext (relative to the standard java:comp/env naming context that is the root of all provided resource factories. A typical web.xml entry might look like this:

```
<resource-ref>
  <description>
    Resource reference to a factory for jakarta.mail.Session
    instances that may be used for sending electronic mail
    messages, preconfigured to connect to the appropriate
    SMTP server.
  </description>
  <res-ref-name>
    mail/Session
  </res-ref-name>
  <res-type>
    jakarta.mail.Session
```

```
    </res-type>
    <res-auth>
        Container
    </res-auth>
</resource-ref>
```

WARNING - Be sure you respect the element ordering that is required by the DTD for web application deployment descriptors! See the Servlet Specification for details.

2. Code Your Application's Use Of This Resource

A typical use of this resource reference might look like this:

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
Session session = (Session) envCtx.lookup("mail/Session");

Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(request.getParameter("from")));
InternetAddress to[] = new InternetAddress[1];
to[0] = new InternetAddress(request.getParameter("to"));
message.setRecipients(Message.RecipientType.TO, to);
message.setSubject(request.getParameter("subject"));
message.setContent(request.getParameter("content"), "text/plain");
Transport.send(message);
```

Note that the application uses the same resource reference name that was declared in the web application deployment descriptor. This is matched up against the resource factory that is configured in the <Context> element for the

web application as described below.

3. Configure Tomcat's Resource Factory

To configure Tomcat's resource factory, add an elements like this to the <Context> element for this web application.

```
<Context ...>

  ...

  <Resource name="mail/Session" auth="Container"

            type="jakarta.mail.Session"

            mail.smtp.host="localhost"/>

  ...

</Context>
```

Note that the resource name (here, mail/Session) must match the value specified in the web application deployment descriptor. Customize the value of the mail.smtp.host parameter to point at the server that provides SMTP service for your network.

Additional resource attributes and values will be converted to properties and values and passed to jakarta.mail.Session.getInstance(java.util.Properties) as part of the java.util.Properties collection. In addition to the properties defined in Appendix A of the Jakarta Mail specification, individual providers may also support additional properties.

If the resource is configured with a password attribute and either a mail.smtp.user or mail.user attribute then Tomcat's resource factory will configure and add a jakarta.mail.Authenticator to the mail session.

4. Install the Jakarta Mail API

Download the Jakarta Mail API.

Unpackage the distribution and place jakarta.mail-api-2.1.0.jar into $CATALINA_HOME/lib so that it is available to Tomcat during the initialization of

the mail Session Resource. **Note:** placing this jar in both $CATALINA_HOME/lib and a web application's lib folder will cause an error, so ensure you have it in the $CATALINA_HOME/lib location only.

5. Install a compatible implementaion

Select and download a compatible implementation.

Unpackage the implementation and place the jar file(s) into $CATALINA_HOME/lib.

Note: Other implementations may be available

6. Restart Tomcat

For the additional JAR to be visible to Tomcat, it is necessary for the Tomcat instance to be restarted.

Example Application

The /examples application included with Tomcat contains an example of utilizing this resource factory. It is accessed via the "JSP Examples" link. The source code for the servlet that actually sends the mail message is in /WEB-INF/classes/SendMailServlet.java.

WARNING - The default configuration assumes that there is an SMTP server listing on port 25 on localhost. If this is not the case, edit the <Context> element for this web application and modify the parameter value for the mail.smtp.host parameter to be the host name of an SMTP server on your network.

JDBC Data Sources

0. Introduction

Many web applications need to access a database via a JDBC driver, to support the functionality required by that application. The Jakarta EE Platform Specification requires Jakarta EE Application Servers to make available a *DataSource* implementation (that is, a connection pool for JDBC connections) for this purpose. Tomcat offers exactly the same support, so that database-based applications you develop on Tomcat using this service will run unchanged

on any Jakarta EE server.

For information about JDBC, you should consult the following:

- http://www.oracle.com/technetwork/java/javase/jdbc/index.html - Home page for information about Java Database Connectivity.
- http://java.sun.com/j2se/1.3/docs/guide/jdbc/spec2/jdbc2.1.frame.html - The JDBC 2.1 API Specification.
- http://java.sun.com/products/jdbc/jdbc20.stdext.pdf - The JDBC 2.0 Standard Extension API (including the javax.sql.DataSource API). This package is now known as the "JDBC Optional Package".
- https://jakarta.ee/specifications/platform/9/ - The Jakarta EE Platform Specification (covers the JDBC facilities that all Jakarta EE platforms must provide to applications).

NOTE - The default data source support in Tomcat is based on the **DBCP 2** connection pool from the Commons project. However, it is possible to use any other connection pool that implements javax.sql.DataSource, by writing your own custom resource factory, as described below.

1. Install Your JDBC Driver

Use of the *JDBC Data Sources* JNDI Resource Factory requires that you make an appropriate JDBC driver available to both Tomcat internal classes and to your web application. This is most easily accomplished by installing the driver's JAR file(s) into the $CATALINA_HOME/lib directory, which makes the driver available both to the resource factory and to your application.

2. Declare Your Resource Requirements

Next, modify the web application deployment descriptor (/WEB-INF/web.xml) to declare the JNDI name under which you will look up preconfigured data source. By convention, all such names should resolve to the jdbc subcontext (relative to the standard java:comp/env naming context that is the root of all provided resource factories. A typical web.xml entry might look like this:

<resource-ref>

```
  <description>
    Resource reference to a factory for java.sql.Connection
    instances that may be used for talking to a particular
    database that is configured in the <Context>
    configuration for the web application.
  </description>
  <res-ref-name>
    jdbc/EmployeeDB
  </res-ref-name>
  <res-type>
    javax.sql.DataSource
  </res-type>
  <res-auth>
    Container
  </res-auth>
</resource-ref>
```

WARNING - Be sure you respect the element ordering that is required by the DTD for web application deployment descriptors! See the Servlet Specification for details.

3. Code Your Application's Use Of This Resource

A typical use of this resource reference might look like this:

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
DataSource ds = (DataSource)
  envCtx.lookup("jdbc/EmployeeDB");
```

```
Connection conn = ds.getConnection();

... use this connection to access the database ...

conn.close();
```

Note that the application uses the same resource reference name that was declared in the web application deployment descriptor. This is matched up against the resource factory that is configured in the <Context> element for the web application as described below.

4. Configure Tomcat's Resource Factory

To configure Tomcat's resource factory, add an element like this to the <Context> element for the web application.

```
<Context ...>

   ...

   <Resource name="jdbc/EmployeeDB"
                  auth="Container"
                  type="javax.sql.DataSource"
                  username="dbusername"
                  password="dbpassword"
                  driverClassName="org.hsql.jdbcDriver"
                  url="jdbc:HypersonicSQL:database"
                  maxTotal="8"
                  maxIdle="4"/>

   ...

</Context>
```

Note that the resource name (here, jdbc/EmployeeDB) must match the value

specified in the web application deployment descriptor.

This example assumes that you are using the HypersonicSQL database JDBC driver. Customize the driverClassName and driverName parameters to match your actual database's JDBC driver and connection URL.

The configuration properties for Tomcat's standard data source resource factory (org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory) are as follows:

- **driverClassName** - Fully qualified Java class name of the JDBC driver to be used.
- **username** - Database username to be passed to our JDBC driver.
- **password** - Database password to be passed to our JDBC driver.
- **url** - Connection URL to be passed to our JDBC driver. (For backwards compatibility, the property driverName is also recognized.)
- **initialSize** - The initial number of connections that will be created in the pool during pool initialization. Default: 0
- **maxTotal** - The maximum number of connections that can be allocated from this pool at the same time. Default: 8
- **minIdle** - The minimum number of connections that will sit idle in this pool at the same time. Default: 0
- **maxIdle** - The maximum number of connections that can sit idle in this pool at the same time. Default: 8
- **maxWaitMillis** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. Default: -1 (infinite)

Some additional properties handle connection validation:

- **validationQuery** - SQL query that can be used by the pool to validate connections before they are returned to the application. If specified, this query MUST be an SQL SELECT statement that returns at least one row.
- **validationQueryTimeout** - Timeout in seconds for the validation query to

return. Default: -1 (infinite)

- **testOnBorrow** - true or false: whether a connection should be validated using the validation query each time it is borrowed from the pool. Default: true
- **testOnReturn** - true or false: whether a connection should be validated using the validation query each time it is returned to the pool. Default: false

The optional evictor thread is responsible for shrinking the pool by removing any connections which are idle for a long time. The evictor does not respect minIdle. Note that you do not need to activate the evictor thread if you only want the pool to shrink according to the configured maxIdle property.

The evictor is disabled by default and can be configured using the following properties:

- **timeBetweenEvictionRunsMillis** - The number of milliseconds between consecutive runs of the evictor. Default: -1 (disabled)
- **numTestsPerEvictionRun** - The number of connections that will be checked for idleness by the evictor during each run of the evictor. Default: 3
- **minEvictableIdleTimeMillis** - The idle time in milliseconds after which a connection can be removed from the pool by the evictor. Default: 30*60*1000 (30 minutes)
- **testWhileIdle** - true or false: whether a connection should be validated by the evictor thread using the validation query while sitting idle in the pool. Default: false

Another optional feature is the removal of abandoned connections. A connection is called abandoned if the application does not return it to the pool for a long time. The pool can close such connections automatically and remove them from the pool. This is a workaround for applications leaking connections.

The abandoning feature is disabled by default and can be configured using the

following properties:

- **removeAbandonedOnBorrow** - true or false: whether to remove abandoned connections from the pool when a connection is borrowed. Default: false
- **removeAbandonedOnMaintenance** - true or false: whether to remove abandoned connections from the pool during pool maintenance. Default: false
- **removeAbandonedTimeout** - The number of seconds after which a borrowed connection is assumed to be abandoned. Default: 300
- **logAbandoned** - true or false: whether to log stack traces for application code which abandoned a statement or connection. This adds serious overhead. Default: false

Finally there are various properties that allow further fine tuning of the pool behaviour:

- **defaultAutoCommit** - true or false: default auto-commit state of the connections created by this pool. Default: true
- **defaultReadOnly** - true or false: default read-only state of the connections created by this pool. Default: false
- **defaultTransactionIsolation** - This sets the default transaction isolation level. Can be one
  of NONE, READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_REA
  D, SERIALIZABLE. Default: no default set
- **poolPreparedStatements** - true or false: whether to pool PreparedStatements and CallableStatements. Default: false
- **maxOpenPreparedStatements** - The maximum number of open statements that can be allocated from the statement pool at the same time. Default: -1 (unlimited)
- **defaultCatalog** - The name of the default catalog. Default: not set
- **connectionInitSqls** - A list of SQL statements run once after a

Connection is created. Separate multiple statements by semicolons (;). Default: no statement

- **connectionProperties** - A list of driver specific properties passed to the driver for creating connections. Each property is given as name=value, multiple properties are separated by semicolons (;). Default: no properties
- **accessToUnderlyingConnectionAllowed** - true or false: whether accessing the underlying connections is allowed. Default: false

For more details, please refer to the Commons DBCP 2 documentation.

Adding Custom Resource Factories

If none of the standard resource factories meet your needs, you can write your own factory and integrate it into Tomcat, and then configure the use of this factory in the <Context> element for the web application. In the example below, we will create a factory that only knows how to create com.mycompany.MyBean beans from the Generic JavaBean Resources example above.

1. Write A Resource Factory Class

You must write a class that implements the JNDI service provider javax.naming.spi.ObjectFactory interface. Every time your web application calls lookup() on a context entry that is bound to this factory (assuming that the factory is configured with singleton="false"), the getObjectInstance() method is called, with the following arguments:

- **Object obj** - The (possibly null) object containing location or reference information that can be used in creating an object. For Tomcat, this will always be an object of type javax.naming.Reference, which contains the class name of this factory class, as well as the configuration properties (from the <Context> for the web application) to use in creating objects to be returned.
- **Name name** - The name to which this factory is bound relative to nameCtx, or null if no name is specified.

- **Context nameCtx** - The context relative to which the name parameter is specified, or null if name is relative to the default initial context.
- **Hashtable environment** - The (possibly null) environment that is used in creating this object. This is generally ignored in Tomcat object factories.

To create a resource factory that knows how to produce MyBean instances, you might create a class like this:

```
package com.mycompany;


import java.util.Enumeration;

import java.util.Hashtable;

import javax.naming.Context;

import javax.naming.Name;

import javax.naming.NamingException;

import javax.naming.RefAddr;

import javax.naming.Reference;

import javax.naming.spi.ObjectFactory;


public class MyBeanFactory implements ObjectFactory {


   public Object getObjectInstance(Object obj,

        Name name2, Context nameCtx, Hashtable environment)

        throws NamingException {


        // Acquire an instance of our specified bean class

        MyBean bean = new MyBean();
```

```
        // Customize the bean properties from our attributes
        Reference ref = (Reference) obj;
        Enumeration addrs = ref.getAll();
        while (addrs.hasMoreElements()) {
            RefAddr addr = (RefAddr) addrs.nextElement();
            String name = addr.getType();
            String value = (String) addr.getContent();
            if (name.equals("foo")) {
                bean.setFoo(value);
            } else if (name.equals("bar")) {
                try {
                    bean.setBar(Integer.parseInt(value));
                } catch (NumberFormatException e) {
                    throw new NamingException("Invalid 'bar' value " + value);
                }
            }
        }

        // Return the customized instance
        return (bean);

    }
```

}

In this example, we are unconditionally creating a new instance of the com.mycompany.MyBean class, and populating its properties based on the parameters included in the <ResourceParams> element that configures this factory (see below). You should note that any parameter named factory should be skipped - that parameter is used to specify the name of the factory class itself (in this case, com.mycompany.MyBeanFactory) rather than a property of the bean being configured.

For more information about ObjectFactory, see the JNDI Service Provider Interface (SPI) Specification.

You will need to compile this class against a class path that includes all of the JAR files in the $CATALINA_HOME/lib directory. When you are through, place the factory class (and the corresponding bean class) unpacked under $CATALINA_HOME/lib, or in a JAR file inside $CATALINA_HOME/lib. In this way, the required class files are visible to both Catalina internal resources and your web application.

2. Declare Your Resource Requirements

Next, modify your web application deployment descriptor (/WEB-INF/web.xml) to declare the JNDI name under which you will request new instances of this bean. The simplest approach is to use a <resource-env-ref> element, like this:

```
<resource-env-ref>
  <description>
    Object factory for MyBean instances.
  </description>
  <resource-env-ref-name>
    bean/MyBeanFactory
  </resource-env-ref-name>
  <resource-env-ref-type>
```

```
    com.mycompany.MyBean
  </resource-env-ref-type>
</resource-env-ref>
```

WARNING - Be sure you respect the element ordering that is required by the DTD for web application deployment descriptors! See the Servlet Specification for details.

3. Code Your Application's Use Of This Resource

A typical use of this resource environment reference might look like this:

```
Context initCtx = new InitialContext();

Context envCtx = (Context) initCtx.lookup("java:comp/env");

MyBean bean = (MyBean) envCtx.lookup("bean/MyBeanFactory");


writer.println("foo = " + bean.getFoo() + ", bar = " +
                   bean.getBar());
```

4. Configure Tomcat's Resource Factory

To configure Tomcat's resource factory, add an elements like this to the <Context> element for this web application.

```
<Context ...>
  ...
  <Resource name="bean/MyBeanFactory" auth="Container"
            type="com.mycompany.MyBean"
            factory="com.mycompany.MyBeanFactory"
            singleton="false"
            bar="23"/>
  ...
```

</Context>

Note that the resource name (here, bean/MyBeanFactory must match the value specified in the web application deployment descriptor. We are also initializing the value of the bar property, which will cause setBar(23) to be called before the new bean is returned. Because we are not initializing the foo property (although we could have), the bean will contain whatever default value is set up by its constructor.

You will also note that, from the application developer's perspective, the declaration of the resource environment reference, and the programming used to request new instances, is identical to the approach used for the *Generic JavaBean Resources* example. This illustrates one of the advantages of using JNDI resources to encapsulate functionality - you can change the underlying implementation without necessarily having to modify applications using the resources, as long as you maintain compatible APIs.