Apache Tomcat

Windows Authentication How-To

Overview

Integrated Windows authentication is most frequently used within intranet environments since it requires that the server performing the authentication and the user being authenticated are part of the same domain. For the user to be authenticated automatically, the client machine used by the user must also be part of the domain.

There are several options for implementing integrated Windows authentication with Apache Tomcat. They are:

- Built-in Tomcat support.
- Use a third party library such as Waffle.
- Use a reverse proxy that supports Windows authentication to perform the authentication step such as IIS or httpd.

The configuration of each of these options is discussed in the following sections.

Built-in Tomcat support

Kerberos (the basis for integrated Windows authentication) requires careful configuration. If the steps in this guide are followed exactly, then a working configuration will result. It is important that the steps below are followed exactly. There is very little scope for flexibility in the configuration. From the testing to date it is known that:

- The host name used to access the Tomcat server must match the host name in the SPN exactly else authentication will fail. A checksum error may be reported in the debug logs in this case.
- The client must be of the view that the server is part of the local trusted intranet.
- The SPN must be HTTP/<hostname> and it must be exactly the same in all

the places it is used.

- The port number must not be included in the SPN.
- No more than one SPN may be mapped to a domain user.
- Tomcat must run as the domain account with which the SPN has been associated or as domain admin. It is **NOT** recommended to run Tomcat under a domain admin user.
- Convention is that the domain name (dev.local) is always used in lower case. The domain name is typically not case sensitive.
- Convention is that the Kerberos realm name (DEV.LOCAL) is always used in upper case. The realm name **is** case sensitive.
- The domain must be specified when using the ktpass command.

There are four components to the configuration of the built-in Tomcat support for Windows authentication. The domain controller, the server hosting Tomcat, the web application wishing to use Windows authentication and the client machine. The following sections describe the configuration required for each component.

The names of the three machines used in the configuration examples below are win-dc01.dev.local (the domain controller), win-tc01.dev.local (the Tomcat instance) and win-pc01.dev.local (client). All are members of the dev.local domain.

Note: In order to use the passwords in the steps below, the domain password policy had to be relaxed. This is not recommended for production environments.

Domain Controller

These steps assume that the server has already been configured to act as a domain controller. Configuration of a Windows server as a domain controller is outside the scope of this how-to. The steps to configure the domain controller to enable Tomcat to support Windows authentication are as follows:

- Create a domain user that will be mapped to the service name used by the Tomcat server. In this how-to, this user is called tc01 and has a password of tc01pass.

- Map the service principal name (SPN) to the user account. SPNs take the form <service class>/<host>:<port>/<service name>. The SPN used in this how-to is HTTP/win-tc01.dev.local. To map the user to the SPN, run the following:

setspn -A HTTP/win-tc01.dev.local tc01

- Generate the keytab file that the Tomcat server will use to authenticate itself to the domain controller. This file contains the Tomcat private key for the service provider account and should be protected accordingly. To generate the file, run the following command (all on a single line):
- ktpass /out c:\tomcat.keytab /mapuser tc01@DEV.LOCAL
- /princ HTTP/win-tc01.dev.local@DEV.LOCAL

/pass tc01pass /kvno 0

- Create a domain user to be used on the client. In this how-to the domain user is test with a password of testpass.

The above steps have been tested on a domain controller running Windows Server 2019 Standard using the Windows Server 2016 functional level for both the forest and the domain.

Tomcat instance (Windows server)

These steps assume that Tomcat and an appropriate Java JDK/JRE have already been installed and configured and that Tomcat is running as the tc01@dev.local user. The steps to configure the Tomcat instance for Windows authentication are as follows:

- Copy the tomcat.keytab file created on the domain controller to $CATALINA_BASE/conf/tomcat.keytab.
- Create the kerberos configuration file $CATALINA_BASE/conf/krb5.ini. The file used in this how-to contained:
- [libdefaults]
- default_realm = DEV.LOCAL

- default_keytab_name = FILE:c:\apache-tomcat-11.0.x\conf\tomcat.keytab
- default_tkt_enctypes = rc4-hmac,aes256-cts-hmac-sha1-96,aes128-cts-hmac-sha1-96
- default_tgs_enctypes = rc4-hmac,aes256-cts-hmac-sha1-96,aes128-cts-hmac-sha1-96
- forwardable=true
-
- [realms]
- DEV.LOCAL = {
- kdc = win-dc01.dev.local:88
- }
-
- [domain_realm]
- dev.local= DEV.LOCAL

.dev.local= DEV.LOCAL

The location of this file can be changed by setting the java.security.krb5.conf system property.

- Create the JAAS login configuration file $CATALINA_BASE/conf/jaas.conf. The file used in this how-to contained:
- com.sun.security.jgss.krb5.initiate {
- com.sun.security.auth.module.Krb5LoginModule required
- doNotPrompt=true
- principal="HTTP/win-tc01.dev.local@DEV.LOCAL"
- useKeyTab=true

- keyTab="c:/apache-tomcat-11.0.x/conf/tomcat.keytab"
- storeKey=true;
- };
-
- com.sun.security.jgss.krb5.accept {
- com.sun.security.auth.module.Krb5LoginModule required
- doNotPrompt=true
- principal="HTTP/win-tc01.dev.local@DEV.LOCAL"
- useKeyTab=true
- keyTab="c:/apache-tomcat-11.0.x/conf/tomcat.keytab"
- storeKey=true;

};

The location of this file can be changed by setting the java.security.auth.login.config system property. The LoginModule used is a JVM specific one so ensure that the LoginModule specified matches the JVM being used. The name of the login configuration must match the value used by the authentication valve.

The SPNEGO authenticator will work with any Realm but if used with the JNDI Realm, by default the JNDI Realm will use the user's delegated credentials to connect to the Active Directory. If only the authenticated user name is required then the AuthenticatedUserRealm may be used that will simply return a Principal based on the authenticated user name that does not have any roles.

The above steps have been tested on a Tomcat server running Windows Server 2019 Standard with AdoptOpenJDK 8u232-b09 (64-bit).

Tomcat instance (Linux server)

This was tested with:

- Java 1.7.0, update 45, 64-bit

- Ubuntu Server 12.04.3 LTS 64-bit
- Tomcat 8.0.x (r1546570)

It should work with any Tomcat release although it is recommended that the latest stable release is used.

The configuration is the same as for Windows but with the following changes:

- The Linux server does not have to be part of the Windows domain.
- The path to the keytab file in krb5.ini and jaas.conf should be updated to reflect the path to the keytab file on the Linux server using Linux style file paths (e.g. /usr/local/tomcat/...).

Web application

The web application needs to be configured to the use Tomcat specific authentication method of SPNEGO (rather than BASIC etc.) in web.xml. As with the other authenticators, behaviour can be customised by explicitly configuring the authentication valve and setting attributes on the Valve.

Client

The client must be configured to use Kerberos authentication. For Internet Explorer this means making sure that the Tomcat instance is in the "Local intranet" security domain and that it is configured (Tools > Internet Options > Advanced) with integrated Windows authentication enabled. Note that this **will not** work if you use the same machine for the client and the Tomcat instance as Internet Explorer will use the unsupported NTLM protocol.

References

Correctly configuring Kerberos authentication can be tricky. The following references may prove helpful. Advice is also always available from the Tomcat users mailing list.

1. IIS and Kerberos
2. SPNEGO project at SourceForge
3. Oracle Java GSS-API tutorial (Java 7)

4. Oracle Java GSS-API tutorial - Troubleshooting (Java 7)
5. Geronimo configuration for Windows authentication
6. Encryption Selection in Kerberos Exchanges
7. Supported Kerberos Cipher Suites

Third party libraries

Waffle

Full details of this solution can be found through the Waffle web site. The key features are:

- Drop-in solution
- Simple configuration (no JAAS or Kerberos keytab configuration required)
- Uses a native library

Spring Security - Kerberos Extension

Full details of this solution can be found through the Kerberos extension web site. The key features are:

- Extension to Spring Security
- Requires a Kerberos keytab file to be generated
- Pure Java solution

Jespa

Full details of this solution can be found through the project web site. The key features are:

- Pure Java solution
- Advanced Active Directory integration

SPNEGO AD project at SourceForge

Full details of this solution can be found through the project site. The key features are:

- Pure Java solution
- SPNEGO/Kerberos Authenticator
- Active Directory Realm

Reverse proxies

Microsoft IIS

There are three steps to configuring IIS to provide Windows authentication. They are:

1. Configure IIS as a reverse proxy for Tomcat (see the IIS Web Server How-To).
2. Configure IIS to use Windows authentication
3. Configure Tomcat to use the authentication user information from IIS by setting the tomcatAuthentication attribute on the AJP connector to false. Alternatively, set the tomcatAuthorization attribute to true to allow IIS to authenticate, while Tomcat performs the authorization.

Apache httpd

Apache httpd does not support Windows authentication out of the box but there are a number of third-party modules that can be used. These include:

1. mod_auth_sspi for use on Windows platforms.
2. mod_auth_ntlm_winbind for non-Windows platforms. Known to work with httpd 2.0.x on 32-bit platforms. Some users have reported stability issues with both httpd 2.2.x builds and 64-bit Linux builds.

There are three steps to configuring httpd to provide Windows authentication. They are:

1. Configure httpd as a reverse proxy for Tomcat (see the Apache httpd Web Server How-To).
2. Configure httpd to use Windows authentication
3. Configure Tomcat to use the authentication user information from httpd

by setting the tomcatAuthentication attribute on the AJP connector to false.

The Tomcat JDBC Connection Pool

Introduction

The **JDBC Connection Pool org.apache.tomcat.jdbc.pool** is a replacement or an alternative to the Apache Commons DBCP connection pool.

So why do we need a new connection pool?

Here are a few of the reasons:

1. Commons DBCP 1.x is single threaded. In order to be thread safe Commons locks the entire pool for short periods during both object allocation and object return. Note that this does not apply to Commons DBCP 2.x.
2. Commons DBCP 1.x can be slow. As the number of logical CPUs grows and the number of concurrent threads attempting to borrow or return objects increases, the performance suffers. For highly concurrent systems the impact can be significant. Note that this does not apply to Commons DBCP 2.x.
3. Commons DBCP is over 60 classes. tomcat-jdbc-pool core is 8 classes, hence modifications for future requirement will require much less changes. This is all you need to run the connection pool itself, the rest is gravy.
4. Commons DBCP uses static interfaces. This means you have to use the right version for a given JRE version or you may see NoSuchMethodException exceptions.
5. It's not worth rewriting over 60 classes, when a connection pool can be accomplished with a much simpler implementation.
6. Tomcat jdbc pool implements the ability retrieve a connection asynchronously, without adding additional threads to the library itself.

7. Tomcat jdbc pool is a Tomcat module, it depends on Tomcat JULI, a simplified logging framework used in Tomcat.
8. Retrieve the underlying connection using the javax.sql.PooledConnection interface.
9. Starvation proof. If a pool is empty, and threads are waiting for a connection, when a connection is returned, the pool will awake the correct thread waiting. Most pools will simply starve.

Features added over other connection pool implementations

1. Support for highly concurrent environments and multi core/cpu systems.
2. Dynamic implementation of interface, will support java.sql and javax.sql interfaces for your runtime environment (as long as your JDBC driver does the same), even when compiled with a lower version of the JDK.
3. Validation intervals - we don't have to validate every single time we use the connection, we can do this when we borrow or return the connection, just not more frequent than an interval we can configure.
4. Run-Once query, a configurable query that will be run only once, when the connection to the database is established. Very useful to setup session settings, that you want to exist during the entire time the connection is established.
5. Ability to configure custom interceptors. This allows you to write custom interceptors to enhance the functionality. You can use interceptors to gather query stats, cache session states, reconnect the connection upon failures, retry queries, cache query results, and so on. Your options are endless and the interceptors are dynamic, not tied to a JDK version of a java.sql/javax.sql interface.
6. High performance - we will show some differences in performance later on
7. Extremely simple, due to the very simplified implementation, the line

count and source file count are very low, compare with c3p0 that has over 200 source files(last time we checked), Tomcat jdbc has a core of 8 files, the connection pool itself is about half that. As bugs may occur, they will be faster to track down, and easier to fix. Complexity reduction has been a focus from inception.

8. Asynchronous connection retrieval - you can queue your request for a connection and receive a Future<Connection> back.
9. Better idle connection handling. Instead of closing connections directly, it can still pool connections and sizes the idle pool with a smarter algorithm.
10. You can decide at what moment connections are considered abandoned, is it when the pool is full, or directly at a timeout by specifying a pool usage threshold.
11. The abandon connection timer will reset upon a statement/query activity. Allowing a connections that is in use for a long time to not timeout. This is achieved using the ResetAbandonedTimer
12. Close connections after they have been connected for a certain time. Age based close upon return to the pool.
13. Get JMX notifications and log entries when connections are suspected for being abandoned. This is similar to the removeAbandonedTimeout but it doesn't take any action, only reports the information. This is achieved using the suspectTimeout attribute.
14. Connections can be retrieved from
    a java.sql.Driver, javax.sql.DataSource or javax.sql.XADataSource This is achieved using the dataSource and dataSourceJNDI attributes.
15. XA connection support

How to use

Usage of the Tomcat connection pool has been made to be as simple as possible, for those of you that are familiar with commons-dbcp, the transition

will be very simple. Moving from other connection pools is also fairly straight forward.

Additional features

The Tomcat connection pool offers a few additional features over what most other pools let you do:

- initSQL - the ability to run an SQL statement exactly once, when the connection is created
- validationInterval - in addition to running validations on connections, avoid running them too frequently.
- jdbcInterceptors - flexible and pluggable interceptors to create any customizations around the pool, the query execution and the result set handling. More on this in the advanced section.
- fairQueue - Set the fair flag to true to achieve thread fairness or to use asynchronous connection retrieval

Inside the Apache Tomcat Container

The Tomcat Connection pool is configured as a resource described in The Tomcat JDBC documentation With the only difference being that you have to specify the factory attribute and set the value to org.apache.tomcat.jdbc.pool.DataSourceFactory

Standalone

The connection pool only has another dependency, and that is on tomcat-juli.jar. To configure the pool in a stand alone project using bean instantiation, the bean to instantiate is org.apache.tomcat.jdbc.pool.DataSource. The same attributes (documented below) as you use to configure a connection pool as a JNDI resource, are used to configure a data source as a bean.

JMX

The connection pool object exposes an MBean that can be registered. In order for the connection pool object to create the MBean, the flag jmxEnabled has to be set to true. This doesn't imply that the pool will be registered with an MBean

server, merely that the MBean is created. In a container like Tomcat, Tomcat itself registers the DataSource with the MBean server,
the org.apache.tomcat.jdbc.pool.DataSource object will then register the actual connection pool MBean. If you're running outside of a container, you can register the DataSource yourself under any object name you specify, and it propagates the registration to the underlying pool. To do this you would
call mBeanServer.registerMBean(dataSource.getPool().getJmxPool(),objectname). Prior to this call, ensure that the pool has been created by
calling dataSource.createPool().

Attributes

To provide a very simple switch to and from commons-dbcp and tomcat-jdbc-pool, Most attributes are the same and have the same meaning.

JNDI Factory and Type

Attribute	Description
factory	factory is required, and the value should be org.apache.tomcat.jdbc.pool.DataSourceFactory
type	Type should always be javax.sql.DataSource or javax.sql.XADataSource Depending on the type a org.apache.tomcat.jdbc.pool.DataSource or a org.apache.tomcat.jdbc.pool.XADataSource will be created.

System Properties

System properties are JVM wide, affect all pools created in the JVM

Attribute	Description
org.apache.tomcat.jdbc.pool.onlyAttemptCurrentClassLoader	(boolean) Controls classloading

	of dynamic classes, such as JDBC drivers, interceptors and validators. If set to false, default value, the pool will first attempt to load using the current loader (i.e. the class loader that loaded the pool classes) and if class loading fails attempt to load using the thread context loader. Set this value to true, if you wish to remain backwards compatible with Apache

	Tomcat 8.0.8 and earlier, and only attempt the current loader. If not set then the default value is false.

Common Attributes

These attributes are shared between commons-dbcp and tomcat-jdbc-pool, in some cases default values are different.

Attribute	Description
defaultAutoCommit	(boolean) The default auto-commit state of connections created by this pool. If not set, default is JDBC driver default (If not set then the setAutoCommit method will not be called.)
defaultReadOnly	(boolean) The default read-only state of connections created by this pool. If not set then the setReadOnly method will not be called. (Some drivers don't support read only mode, ex: Informix)
defaultTransactionIsolation	(String) The default TransactionIsolation state of connections created by this pool. One of the following: (see javadoc) • NONE • READ_COMMITTED

	• READ_UNCOMMITTED • REPEATABLE_READ • SERIALIZABLE If not set, the method will not be called and it defaults to the JDBC driver.
defaultCatalog	(String) The default catalog of connections created by this pool.
driverClassName	(String) The fully qualified Java class name of the JDBC driver to be used. The driver has to be accessible from the same classloader as tomcat-jdbc.jar
username	(String) The connection username to be passed to our JDBC driver to establish a connection. Note that method DataSource.getConnection(username,password) by default will not use credentials passed into the method, but will use the ones configured here. See alternateUsernameAllowed property for more details.
password	(String) The connection password to be passed to our JDBC driver to establish a connection. Note that method DataSource.getConnection(username,password) by default will not use credentials passed into the method, but will use the ones configured here. See alternateUsernameAllowed property for

	more details.
maxActive	(int) The maximum number of active connections that can be allocated from this pool at the same time. The default value is 100
maxIdle	(int) The maximum number of connections that should be kept in the pool at all times. Default value is maxActive:100 Idle connections are checked periodically (if enabled) and connections that been idle for longer than minEvictableIdleTimeMillis will be released. (also see testWhileIdle)
minIdle	(int) The minimum number of established connections that should be kept in the pool at all times. The connection pool can shrink below this number if validation queries fail. Default value is derived from initialSize:10 (also see testWhileIdle)
initialSize	(int)The initial number of connections that are created when the pool is started. Default value is 10
maxWait	(int) 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 value is 30000 (30 seconds)
testOnBorrow	(boolean) The indication of whether objects will be validated before being borrowed from

	the pool. If the object fails to validate, it will be dropped from the pool, and we will attempt to borrow another. In order to have a more efficient validation, see validationInterval. Default value is false
testOnConnect	(boolean) The indication of whether objects will be validated when a connection is first created. If an object fails to validate, it will be throw SQLException. Default value is false
testOnReturn	(boolean) The indication of whether objects will be validated before being returned to the pool. The default value is false.
testWhileIdle	(boolean) The indication of whether objects will be validated by the idle object evictor (if any). If an object fails to validate, it will be dropped from the pool. The default value is false and this property has to be set in order for the pool cleaner/test thread is to run (also see timeBetweenEvictionRunsMillis)
validationQuery	(String) The SQL query that will be used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw a SQLException. The default value is null. If not specified, connections will be validation by the isValid() method. Example values are SELECT 1(mysql), select 1 from dual(oracle), SELECT 1(MS Sql Server)
validationQueryTimeout	(int) The timeout in seconds before a

	connection validation queries fail. This works by calling java.sql.Statement.setQueryTimeout(seconds) on the statement that executes the validationQuery. The pool itself doesn't timeout the query, it is still up to the JDBC driver to enforce query timeouts. A value less than or equal to zero will disable this feature. The default value is -1.
validatorClassName	(String) The name of a class which implements the org.apache.tomcat.jdbc.pool.Validator interface and provides a no-arg constructor (may be implicit). If specified, the class will be used to create a Validator instance which is then used instead of any validation query to validate connections. The default value is null. An example value is com.mycompany.project.SimpleValidator.
timeBetweenEvictionRunsMillis	(int) The number of milliseconds to sleep between runs of the idle connection validation/cleaner thread. This value should not be set under 1 second. It dictates how often we check for idle, abandoned connections, and how often we validate idle connections. This value will be overridden by maxAge if the latter is non-zero and lower. The default value is 5000 (5 seconds).
numTestsPerEvictionRun	(int) Property not used in tomcat-jdbc-pool.

minEvictableIdleTimeMillis	(int) The minimum amount of time an object may sit idle in the pool before it is eligible for eviction. The default value is 60000 (60 seconds).
accessToUnderlyingConnectionAllowed	(boolean) Property not used. Access can be achieved by calling unwrap on the pooled connection. see javax.sql.DataSource interface, or call getConnection through reflection or cast the object as javax.sql.PooledConnection
removeAbandoned	(boolean) Flag to remove abandoned connections if they exceed the removeAbandonedTimeout. If set to true a connection is considered abandoned and eligible for removal if it has been in use longer than the removeAbandonedTimeout Setting this to true can recover db connections from applications that fail to close a connection. See also logAbandoned The default value is false.
removeAbandonedTimeout	(int) Timeout in seconds before an abandoned(in use) connection can be removed. The default value is 60 (60 seconds). The value should be set to the longest running query your applications might have.
logAbandoned	(boolean) Flag to log stack traces for application code which abandoned a Connection. Logging of abandoned Connections adds overhead for every

	Connection borrow because a stack trace has to be generated. The default value is false.
connectionProperties	(String) The connection properties that will be sent to our JDBC driver when establishing new connections. Format of the string must be [propertyName=property;]* NOTE - The "user" and "password" properties will be passed explicitly, so they do not need to be included here. The default value is null.
poolPreparedStatements	(boolean) Property not used.
maxOpenPreparedStatements	(int) Property not used.

Tomcat JDBC Enhanced Attributes

Attribute	Description
initSQL	(String) A custom query to be run when a connection is first created. The default value is null.
jdbcInterceptors	(String) A semicolon separated list of classnames extending org.apache.tomcat.jdbc.pool.JdbcInterceptor class. See Configuring JDBC interceptors below for more detailed description of syntax and examples. These interceptors will be inserted as an interceptor into the chain of operations on a java.sql.Connection object. The default value is null. Predefined interceptors:

	org.apache.tomcat.jdbc.pool.interceptor.ConnectionState - keeps track of auto commit, read only, catalog and transaction isolation level. org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer - keeps track of opened statements, and closes them when the connection is returned to the pool. More predefined interceptors are described in detail in the JDBC Interceptors section.
validationInterval	(long) avoid excess validation, only run validation at most at this frequency - time in milliseconds. If a connection is due for validation, but has been validated previously within this interval, it will not be validated again. The default value is 3000 (3 seconds).
jmxEnabled	(boolean) Register the pool with JMX or not. The default value is true.
fairQueue	(boolean) Set to true if you wish that calls to getConnection should be treated fairly in a true FIFO fashion. This uses the org.apache.tomcat.jdbc.pool.FairBlockingQueue implementation for the list of the idle connections. The default value is true. This flag is required when you want to use asynchronous connection retrieval. Setting this flag ensures that threads receive connections in the order they arrive. During performance tests, there is a very large difference in how locks and lock waiting is implemented. When fairQueue=true there is a decision making process based on what operating

	system the system is running. If the system is running on Linux (property os.name=Linux. To disable this Linux specific behavior and still use the fair queue, simply add the property org.apache.tomcat.jdbc.pool.FairBlockingQueue.ignoreOS=true to your system properties before the connection pool classes are loaded.
abandonWhenPercentageFull	(int) Connections that have been abandoned (timed out) won't get closed and reported up unless the number of connections in use are above the percentage defined by abandonWhenPercentageFull. The value should be between 0-100. The default value is 0, which implies that connections are eligible for closure as soon as removeAbandonedTimeout has been reached.
maxAge	(long) Time in milliseconds to keep a connection before recreating it. When a connection is borrowed from the pool, the pool will check to see if the now - time-when-connected > maxAge has been reached , and if so, it reconnects before borrow it. When a connection is returned to the pool, the pool will check to see if the now - time-when-connected > maxAge has been reached, and if so, it tries to reconnect. When a connection is idle and timeBetweenEvictionRunsMillis is greater than zero, the pool will periodically check to see if the now - time-when-connected > maxAge has been reached, and if so, it tries to reconnect. Setting maxAge to a value lower than timeBetweenEvictionRunsMillis will override it

	(so idle connection validation/cleaning will run more frequently). The default value is 0, which implies that connections will be left open and no age check will be done upon borrowing from the pool, returning the connection to the pool or when checking idle connections.
useEquals	(boolean) Set to true if you wish the ProxyConnection class to use String.equals and set to false when you wish to use == when comparing method names. This property does not apply to added interceptors as those are configured individually. The default value is true.
suspectTimeout	(int) Timeout value in seconds. Default value is 0. Similar to to the removeAbandonedTimeout value but instead of treating the connection as abandoned, and potentially closing the connection, this simply logs the warning if logAbandoned is set to true. If this value is equal or less than 0, no suspect checking will be performed. Suspect checking only takes place if the timeout value is larger than 0 and the connection was not abandoned or if abandon check is disabled. If a connection is suspect a WARN message gets logged and a JMX notification gets sent once.
rollbackOnReturn	(boolean) If autoCommit==false then the pool can terminate the transaction by calling rollback on the connection as it is returned to the pool Default value is false.
commitOnReturn	(boolean) If autoCommit==false then the pool can complete the transaction by calling commit on the

	connection as it is returned to the pool If rollbackOnReturn==true then this attribute is ignored. Default value is false.
alternateUsernameAllowed	(boolean) By default, the jdbc-pool will ignore the DataSource.getConnection(username,password) call, and simply return a previously pooled connection under the globally configured properties username and password, for performance reasons. The pool can however be configured to allow use of different credentials each time a connection is requested. To enable the functionality described in the DataSource.getConnection(username,password) call, simply set the property alternateUsernameAllowed to true. Should you request a connection with the credentials user1/password1 and the connection was previously connected using different user2/password2, the connection will be closed, and reopened with the requested credentials. This way, the pool size is still managed on a global level, and not on a per schema level. The default value is false. This property was added as an enhancement to bug 50025.
dataSource	(javax.sql.DataSource) Inject a data source to the connection pool, and the pool will use the data source to retrieve connections instead of establishing them using the java.sql.Driver interface. This is useful when you wish to pool XA connections or connections

	established using a data source instead of a connection string. Default value is null
dataSourceJNDI	(String) The JNDI name for a data source to be looked up in JNDI and then used to establish connections to the database. See the dataSource attribute. Default value is null
useDisposableConnectionFacade	(boolean) Set this to true if you wish to put a facade on your connection so that it cannot be reused after it has been closed. This prevents a thread holding on to a reference of a connection it has already called closed on, to execute queries on it. Default value is true.
logValidationErrors	(boolean) Set this to true to log errors during the validation phase to the log file. If set to true, errors will be logged as SEVERE. Default value is false for backwards compatibility.
propagateInterruptState	(boolean) Set this to true to propagate the interrupt state for a thread that has been interrupted (not clearing the interrupt state). Default value is false for backwards compatibility.
ignoreExceptionOnPreLoad	(boolean) Flag whether ignore error of connection creation while initializing the pool. Set to true if you want to ignore error of connection creation while initializing the pool. Set to false if you want to fail the initialization of the pool by throwing exception. The default value is false.
useStatementFacade	(boolean) Set this to true if you wish to wrap statements in order to

	enable equals() and hashCode() methods to be called on the closed statements if any statement proxy is set. Default value is true.

Advanced usage

JDBC interceptors

To see an example of how to use an interceptor, take a look at org.apache.tomcat.jdbc.pool.interceptor.ConnectionState. This simple interceptor is a cache of three attributes, transaction isolation level, auto commit and read only state, in order for the system to avoid not needed roundtrips to the database.

Further interceptors will be added to the core of the pool as the need arises. Contributions are always welcome!

Interceptors are of course not limited to just java.sql.Connection but can be used to wrap any of the results from a method invocation as well. You could build query performance analyzer that provides JMX notifications when a query is running longer than the expected time.

Configuring JDBC interceptors

Configuring JDBC interceptors is done using the **jdbcInterceptors** property. The property contains a list of semicolon separated class names. If the classname is not fully qualified it will be prefixed with the org.apache.tomcat.jdbc.pool.interceptor. prefix.

Example:
jdbcInterceptors="org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;
org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer"
is the same as
jdbcInterceptors="ConnectionState;StatementFinalizer"

Interceptors can have properties as well. Properties for an interceptor are specified within parentheses after the class name. Several properties are separated by commas.

Example:
jdbcInterceptors="ConnectionState;StatementFinalizer(useEquals=true)"

Extra whitespace characters around class names, property names and values are ignored.

org.apache.tomcat.jdbc.pool.JdbcInterceptor

Abstract base class for all interceptors, cannot be instantiated.

Attribute	Description
useEquals	(boolean) Set to true if you wish the ProxyConnection class to use String.equals and set to false when you wish to use == when comparing method names. The default value is true.

org.apache.tomcat.jdbc.pool.interceptor.ConnectionState

Caches the connection for the following attributes autoCommit, readOnly, transactionIsolation and catalog. It is a performance enhancement to avoid roundtrip to the database when getters are called or setters are called with an already set value.

Attribute	Description

org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer

Keeps track of all statements created using createStatement, prepareStatement or prepareCall and closes these statements when the connection is returned to the pool.

Attribute	Description
trace	(boolean as String) Enable tracing of unclosed statements. When enabled and a connection is closed, and statements are not closed, the interceptor will log all stack traces. The default value is false.

org.apache.tomcat.jdbc.pool.interceptor.StatementCache

Caches PreparedStatement and/or CallableStatement instances on a connection.

The statements are cached per connection. The count limit is counted globally for all connections that belong to the same pool. Once the count reaches max, subsequent statements are not returned to the cache and are closed immediately.

Attribute	Description
prepared	(boolean as String) Enable caching of PreparedStatement instances created using prepareStatement calls. The default value is true.
callable	(boolean as String) Enable caching of CallableStatement instances created using prepareCall calls. The default value is false.
max	(int as String) Limit on the count of cached statements across the connection pool. The default value is 50.

org.apache.tomcat.jdbc.pool.interceptor.StatementDecoratorInterceptor

See 48392. Interceptor to wrap statements and result sets in order to prevent access to the actual connection using the methods ResultSet.getStatement().getConnection() and Statement.getConnection()

Attribute	Description

org.apache.tomcat.jdbc.pool.interceptor.QueryTimeoutInterceptor

Automatically calls java.sql.Statement.setQueryTimeout(seconds) when a new statement is created. The pool itself doesn't timeout the query, it is still up to the JDBC driver to enforce query timeouts.

Attribute	Description
queryTimeout	(int as String) The number of seconds to set for the query timeout. A value less than or equal to zero will disable this feature. The default value is 1 seconds.

org.apache.tomcat.jdbc.pool.interceptor.SlowQueryReport

Keeps track of query performance and issues log entries when queries exceed a time threshold of fail. The log level used is WARN

Attribute	Description
threshold	(int as String) The number of milliseconds a query has to exceed before issuing a log alert. The default value is 1000 milliseconds.
maxQueries	(int as String) The maximum number of queries to keep track of in order to preserve memory space. A value less than or equal to 0 will disable this feature. The default value is 1000.
logSlow	(boolean as String) Set to true if you wish to log slow queries. The default value is true.
logFailed	(boolean as String) Set to true if you wish to log failed queries. The default value is false.

org.apache.tomcat.jdbc.pool.interceptor.SlowQueryReportJmx

Extends the SlowQueryReport and in addition to log entries it issues JMX notification for monitoring tools to react to. Inherits all the attributes from its parent class. This class uses Tomcat's JMX engine so it won't work outside of the Tomcat container. By default, JMX notifications are sent through the ConnectionPool mbean if it is enabled. The SlowQueryReportJmx can also register an MBean if notifyPool=false

Attribute	Description
notifyPool	(boolean as String) Set to false if you want JMX notifications to go to the SlowQueryReportJmx MBean The default value is true.
objectName	(String) Define a valid javax.management.ObjectName string that will be used to register this object with the platform mbean server The default value is null and the object will be registered using tomcat.jdbc:type=org.apache.tomcat.jdbc.pool.interceptor.SlowQueryReportJmx,name=the-name-of-the-pool

org.apache.tomcat.jdbc.pool.interceptor.ResetAbandonedTimer

The abandoned timer starts when a connection is checked out from the pool. This means if you have a 30second timeout and run 10x10second queries using the connection it will be marked abandoned and potentially reclaimed depending on the abandonWhenPercentageFull attribute. Using this interceptor it will reset the checkout timer every time you perform an operation on the connection or execute a query successfully.

Attribute	Description

Code Example

Other examples of Tomcat configuration for JDBC usage can be found in the Tomcat documentation.

Plain Ol' Java

Here is a simple example of how to create and use a data source.

```
import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;
```

```
import org.apache.tomcat.jdbc.pool.DataSource;
import org.apache.tomcat.jdbc.pool.PoolProperties;

public class SimplePOJOExample {

    public static void main(String[] args) throws Exception {
        PoolProperties p = new PoolProperties();
        p.setUrl("jdbc:mysql://localhost:3306/mysql");
        p.setDriverClassName("com.mysql.jdbc.Driver");
        p.setUsername("root");
        p.setPassword("password");
        p.setJmxEnabled(true);
        p.setTestWhileIdle(false);
        p.setTestOnBorrow(true);
        p.setValidationQuery("SELECT 1");
        p.setTestOnReturn(false);
        p.setValidationInterval(30000);
        p.setTimeBetweenEvictionRunsMillis(30000);
        p.setMaxActive(100);
        p.setInitialSize(10);
        p.setMaxWait(10000);
        p.setRemoveAbandonedTimeout(60);
        p.setMinEvictableIdleTimeMillis(30000);
        p.setMinIdle(10);
```

```
            p.setLogAbandoned(true);
            p.setRemoveAbandoned(true);
            p.setJdbcInterceptors(
              "org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;"+
              "org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer");
            DataSource datasource = new DataSource();
            datasource.setPoolProperties(p);

            Connection con = null;
            try {
              con = datasource.getConnection();
              Statement st = con.createStatement();
              ResultSet rs = st.executeQuery("select * from user");
              int cnt = 1;
              while (rs.next()) {
                    System.out.println((cnt++)+". Host:" +rs.getString("Host")+
                      " User:"+rs.getString("User")+"
Password:"+rs.getString("Password"));
              }
              rs.close();
              st.close();
            } finally {
              if (con!=null) try {con.close();}catch (Exception ignore) {}
            }
```

```
        }


  }
```

As a Resource

And here is an example on how to configure a resource for JNDI lookups

```
<Resource name="jdbc/TestDB"
                auth="Container"
                type="javax.sql.DataSource"
                factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
                testWhileIdle="true"
                testOnBorrow="true"
                testOnReturn="false"
                validationQuery="SELECT 1"
                validationInterval="30000"
                timeBetweenEvictionRunsMillis="30000"
                maxActive="100"
                minIdle="10"
                maxWait="10000"
                initialSize="10"
                removeAbandonedTimeout="60"
                removeAbandoned="true"
                logAbandoned="true"
                minEvictableIdleTimeMillis="30000"
                jmxEnabled="true"
```

```
jdbcInterceptors="org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;
                  org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer"
            username="root"
            password="password"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/mysql"/>
```

Asynchronous Connection Retrieval

The Tomcat JDBC connection pool supports asynchronous connection retrieval without adding additional threads to the pool library. It does this by adding a method to the data source called Future<Connection> getConnectionAsync(). In order to use the async retrieval, two conditions must be met:

1. You must configure the fairQueue property to be true.
2. You will have to cast the data source to org.apache.tomcat.jdbc.pool.DataSource

An example of using the async feature is show below.

```
Connection con = null;
try {
    Future<Connection> future = datasource.getConnectionAsync();
    while (!future.isDone()) {
        System.out.println("Connection is not yet available. Do some background work");
        try {
            Thread.sleep(100); //simulate work
        }catch (InterruptedException x) {
            Thread.currentThread().interrupt();
```

```
        }
    }
    con = future.get(); //should return instantly
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("select * from user");
```

Interceptors

Interceptors are a powerful way to enable, disable or modify functionality on a specific connection or its sub components. There are many different use cases for when interceptors are useful. By default, and for performance reasons, the connection pool is stateless. The only state the pool itself inserts are defaultAutoCommit, defaultReadOnly, defaultTransactionIsolation, defaultCatalog if these are set. These 4 properties are only set upon connection creation. Should these properties be modified during the usage of the connection, the pool itself will not reset them.

An interceptor has to extend the org.apache.tomcat.jdbc.pool.JdbcInterceptor class. This class is fairly simple, You will need to have a no arg constructor

```
public JdbcInterceptor() {
}
```

When a connection is borrowed from the pool, the interceptor can initialize or in some other way react to the event by implementing the

```
public abstract void reset(ConnectionPool parent, PooledConnection con);
```

method. This method gets called with two parameters, a reference to the connection pool itself ConnectionPool parent and a reference to the underlying connection PooledConnection con.

When a method on the java.sql.Connection object is invoked, it will cause the

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

method to get invoked. The Method method is the actual method invoked, and Object[] args are the arguments. To look at a very simple example, where we demonstrate how to make the invocation to java.sql.Connection.close() a noop if the connection has been closed

```
if (CLOSE_VAL==method.getName()) {
    if (isClosed()) return null; //noop for already closed.
}
return super.invoke(proxy,method,args);
```

There is an observation being made. It is the comparison of the method name. One way to do this would be to do "close".equals(method.getName()). Above we see a direct reference comparison between the method name and static final String reference. According to the JVM spec, method names and static final String end up in a shared constant pool, so the reference comparison should work. One could of course do this as well:

```
if (compare(CLOSE_VAL,method)) {
    if (isClosed()) return null; //noop for already closed.
}
return super.invoke(proxy,method,args);
```

The compare(String,Method) will use the useEquals flag on an interceptor and do either reference comparison or a string value comparison when the useEquals=true flag is set.

Pool start/stop

When the connection pool is started or closed, you can be notified. You will only be notified once per interceptor class even though it is an instance method. and you will be notified using an interceptor currently not attached to a pool.

```
public void poolStarted(ConnectionPool pool) {
}
```

```
    public void poolClosed(ConnectionPool pool) {

    }
```

When overriding these methods, don't forget to call super if you are extending a class other than JdbcInterceptor

Configuring interceptors
Interceptors are configured using the jdbcInterceptors property or the setJdbcInterceptors method. An interceptor can have properties, and would be configured like this

```
    String jdbcInterceptors=


"org.apache.tomcat.jdbc.pool.interceptor.ConnectionState(useEquals=true,fast=yes)"
```

Interceptor properties
Since interceptors can have properties, you need to be able to read the values of these properties within your interceptor. Taking an example like the one above, you can override the setProperties method.

```
    public void setProperties(Map<String, InterceptorProperty> properties) {
        super.setProperties(properties);
        final String myprop = "myprop";
        InterceptorProperty p1 = properties.get(myprop);
        if (p1!=null) {
            setMyprop(Long.parseLong(p1.getValue()));
        }
    }
```

Getting the actual JDBC connection

Connection pools create wrappers around the actual connection in order to properly pool them. We also create interceptors in these wrappers to be able to

perform certain functions. If there is a need to retrieve the actual connection, one can do so using the javax.sql.PooledConnection interface.

```
Connection con = datasource.getConnection();
Connection actual = ((javax.sql.PooledConnection)con).getConnection();
```

Building

We build the JDBC pool code with 1.6, but it is backwards compatible down to 1.5 for runtime environment. For unit test, we use 1.6 and higher

Other examples of Tomcat configuration for JDBC usage can be found in the Tomcat documentation.

Building from source

Building is pretty simple. The pool has a dependency on tomcat-juli.jar and in case you want the SlowQueryReportJmx

```
javac -classpath tomcat-juli.jar \
        -d . \
        org/apache/tomcat/jdbc/pool/*.java \
        org/apache/tomcat/jdbc/pool/interceptor/*.java \
        org/apache/tomcat/jdbc/pool/jmx/*.java
```

A build file can be found in the Tomcat source repository.

As a convenience, a build file is also included where a simple build command will generate all files needed.

```
ant download    (downloads dependencies)
ant build       (compiles and generates .jar files)
ant dist        (creates a release package)
ant test        (runs tests, expects a test database to be setup)
```

The system is structured for a Maven build, but does generate release artifacts. Just the library itself.

WebSocket How-To

Overview

Tomcat provides support for WebSocket as defined by RFC 6455.

Application development

Tomcat implements the Jakarta WebSocket 2.1 API defined by the Jakarta WebSocket project.

There are several example applications that demonstrate how the WebSocket API can be used. You will need to look at both the client side HTML and the server side code.

Tomcat WebSocket specific configuration

Tomcat provides a number of Tomcat specific configuration options for WebSocket. It is anticipated that these will be absorbed into the WebSocket specification over time.

The write timeout used when sending WebSocket messages in blocking mode defaults to 20000 milliseconds (20 seconds). This may be changed by setting the property org.apache.tomcat.websocket.BLOCKING_SEND_TIMEOUT in the user properties collection attached to the WebSocket session. The value assigned to this property should be a Long and represents the timeout to use in milliseconds. For an infinite timeout, use -1.

The time Tomcat waits for a peer to send a WebSocket session close message after Tomcat has sent a close message to the peer defaults to 30000 milliseconds (30 seconds). This may be changed by setting the property org.apache.tomcat.websocket.SESSION_CLOSE_TIMEOUT in the user properties collection attached to the WebSocket session. The value assigned to this property should be a Long and represents the timeout to use in milliseconds. Values less than or equal to zero will be ignored.

The write timeout Tomcat uses when writing a session close message when the close is abnormal defaults to 50 milliseconds. This may be changed by setting the

property org.apache.tomcat.websocket.ABNORMAL_SESSION_CLOSE_SEND_TIMEOUT in the user properties collection attached to the WebSocket session. The value assigned to this property should be a Long and represents the timeout to use in milliseconds. Values less than or equal to zero will be ignored.

In addition to the Session.setMaxIdleTimeout(long) method which is part of the Jakarta WebSocket API, Tomcat provides greater control of the timing out the session due to lack of activity. Setting the property org.apache.tomcat.websocket.READ_IDLE_TIMEOUT_MS in the user properties collection attached to the WebSocket session will trigger a session timeout if no WebSocket message is received for the specified number of milliseconds. Setting the property org.apache.tomcat.websocket.WRITE_IDLE_TIMEOUT_MS will trigger a session timeout if no WebSocket message is sent for the specified number of milliseconds. These can be used separately or together, with or without Session.setMaxIdleTimeout(long). If the associated property is not specified, the read and/or write idle timeout will be applied.

If the application does not define a MessageHandler.Partial for incoming binary messages, any incoming binary messages must be buffered so the entire message can be delivered in a single call to the registered MessageHandler.Whole for binary messages. The default buffer size for binary messages is 8192 bytes. This may be changed for a web application by setting the servlet context initialization parameter org.apache.tomcat.websocket.binaryBufferSize to the desired value in bytes.

If the application does not define a MessageHandler.Partial for incoming text messages, any incoming text messages must be buffered so the entire message can be delivered in a single call to the registered MessageHandler.Whole for text messages. The default buffer size for text messages is 8192 bytes. This may be changed for a web application by setting the servlet context initialization parameter org.apache.tomcat.websocket.textBufferSize to the desired value in bytes.

When using the WebSocket client to connect to server endpoints, the timeout for

IO operations while establishing the connection is controlled by the userProperties of the provided jakarta.websocket.ClientEndpointConfig. The property is org.apache.tomcat.websocket.IO_TIMEOUT_MS and is the timeout as a String in milliseconds. The default is 5000 (5 seconds).

When using the WebSocket client to connect to server endpoints, the number of HTTP redirects that the client will follow is controlled by the userProperties of the provided jakarta.websocket.ClientEndpointConfig. The property is org.apache.tomcat.websocket.MAX_REDIRECTIONS. The default value is 20. Redirection support can be disabled by configuring a value of zero.

When using the WebSocket client to connect to a server endpoint that requires BASIC or DIGEST authentication, the following user properties must be set:

- org.apache.tomcat.websocket.WS_AUTHENTICATION_USER_NAME
- org.apache.tomcat.websocket.WS_AUTHENTICATION_PASSWORD

Optionally, the WebSocket client can be configured only to send credentials if the server authentication challenge includes a specific realm by defining that realm in the optional user property:

- org.apache.tomcat.websocket.WS_AUTHENTICATION_REALM

When using the WebSocket client to connect to a server endpoint via a forward proxy (also known as a gateway) that requires BASIC or DIGEST authentication, the following user properties must be set:

- org.apache.tomcat.websocket.WS_PROXY_AUTHENTICATION_USER_NAME
- org.apache.tomcat.websocket.WS_PROXY_AUTHENTICATION_PASSWORD

Optionally, the WebSocket client can be configured only to send credentials if the server authentication challenge includes a specific realm by defining that realm in the optional user property:

- org.apache.tomcat.websocket.WS_PROXY_AUTHENTICATION_REALM

The rewrite Valve

Introduction

The rewrite valve implements URL rewrite functionality in a way that is very similar to mod_rewrite from Apache HTTP Server.

Configuration

The rewrite valve is configured as a valve using the org.apache.catalina.valves.rewrite.RewriteValve class name.

The rewrite valve can be configured as a valve added in a Host. See virtual-server documentation for information on how to configure it. It will use a rewrite.config file containing the rewrite directives, it must be placed in the Host configuration folder.

It can also be in the context.xml of a webapp. The valve will then use a rewrite.config file containing the rewrite directives, it must be placed in the WEB-INF folder of the web application

Directives

The rewrite.config file contains a list of directives which closely resemble the directives used by mod_rewrite, in particular the central RewriteRule and RewriteCond directives. Lines that start with a # character are treated as comments and will be ignored.

Note: This section is a modified version of the mod_rewrite documentation, which is Copyright 1995-2006 The Apache Software Foundation, and licensed under the under the Apache License, Version 2.0.

RewriteCond

Syntax: RewriteCond TestString CondPattern

The RewriteCond directive defines a rule condition. One or more RewriteCond can precede a RewriteRule directive. The following rule is then only used if both the current state of the URI matches its pattern, and if these conditions are met.

TestString is a string which can contain the following expanded constructs in

addition to plain text:

- **RewriteRule backreferences**: These are backreferences of the form **$N** (0 <= N <= 9), which provide access to the grouped parts (in parentheses) of the pattern, from the RewriteRule which is subject to the current set of RewriteCond conditions..
- **RewriteCond backreferences**: These are backreferences of the form **%N** (1 <= N <= 9), which provide access to the grouped parts (again, in parentheses) of the pattern, from the last matched RewriteCond in the current set of conditions.
- **RewriteMap expansions**: These are expansions of the form **${mapname:key|default}**. See the documentation for RewriteMap for more details.
- **Server-Variables**: These are variables of the form **%{ *NAME_OF_VARIABLE* }** where *NAME_OF_VARIABLE* can be a string taken from the following list:
    - **HTTP headers:**

HTTP_USER_AGENT
HTTP_REFERER
HTTP_COOKIE
HTTP_FORWARDED
HTTP_HOST
HTTP_PROXY_CONNECTION
HTTP_ACCEPT

    - **connection & request:**

REMOTE_ADDR
REMOTE_HOST
REMOTE_PORT
REMOTE_USER
REMOTE_IDENT
REQUEST_METHOD

SCRIPT_FILENAME

REQUEST_PATH

CONTEXT_PATH

SERVLET_PATH

PATH_INFO

QUERY_STRING

AUTH_TYPE

- **server internals:**

DOCUMENT_ROOT

SERVER_NAME

SERVER_ADDR

SERVER_PORT

SERVER_PROTOCOL

SERVER_SOFTWARE

- **date and time:**

TIME_YEAR

TIME_MON

TIME_DAY

TIME_HOUR

TIME_MIN

TIME_SEC

TIME_WDAY

TIME

- **specials:**

THE_REQUEST

REQUEST_URI

REQUEST_FILENAME

HTTPS

- These variables all correspond to the similarly named HTTP MIME-headers and Servlet API methods. Most are documented elsewhere in the

Manual or in the CGI specification. Those that are special to the rewrite valve include those below.

- REQUEST_PATH
- Corresponds to the full path that is used for mapping.
- CONTEXT_PATH
- Corresponds to the path of the mapped context.
- SERVLET_PATH
- Corresponds to the servlet path.
- THE_REQUEST
- The full HTTP request line sent by the browser to the server (e.g., "GET /index.html HTTP/1.1"). This does not include any additional headers sent by the browser.
- REQUEST_URI
- The resource requested in the HTTP request line. (In the example above, this would be "/index.html".)
- REQUEST_FILENAME
- The full local file system path to the file or script matching the request.
- HTTPS
- Will contain the text "on" if the connection is using SSL/TLS, or "off" otherwise.

Other things you should be aware of:

1. The variables SCRIPT_FILENAME and REQUEST_FILENAME contain the same value - the value of the filename field of the internal request_rec structure of the Apache server. The first name is the commonly known CGI variable name while the second is the appropriate counterpart of REQUEST_URI (which contains the value of the uri field of request_rec).

2. %{ENV:variable}, where *variable* can be any Java system property, is also available.

3. %{SSL:variable}, where *variable* is the name of an SSL environment variable, are implemented,
   except SSL_SESSION_RESUMED, SSL_SECURE_RENEG, SSL_COMPRESS_METHOD, SSL_TLS_SNI, SSL_SRP_USER, SSL_SRP_USERINFO, SSL_CLIENT_VERIFY, SSL_CLIENT_SAN_OTHER_msUPN_n, SSL_CLIENT_CERT_RFC4523_CEA, SSL_SERVER_SAN_OTHER_dnsSRV_n. When OpenSSL is used, the variables related to the server certificate, prefixed by SSL_SERVER_ are not available.
   Example: %{SSL:SSL_CIPHER_USEKEYSIZE} may expand to 128.

4. %{HTTP:header}, where *header* can be any HTTP MIME-header name, can always be used to obtain the value of a header sent in the HTTP request.
   Example: %{HTTP:Proxy-Connection} is the value of the HTTP header 'Proxy-Connection:'.

CondPattern is the condition pattern, a regular expression which is applied to the current instance of the *TestString*. *TestString* is first evaluated, before being matched against *CondPattern*.

Remember: *CondPattern* is a *perl compatible regular expression* with some additions:

1. You can prefix the pattern string with a '!' character (exclamation mark) to specify a **non**-matching pattern.

2. There are some special variants of *CondPatterns*. Instead of real regular expression strings you can also use one of the following:

   - '**<CondPattern**' (lexicographically precedes)
     Treats the *CondPattern* as a plain string and compares it lexicographically to *TestString*. True if *TestString* lexicographically precedes *CondPattern*.

   - '**>CondPattern**' (lexicographically follows)
     Treats the *CondPattern* as a plain string and compares it

lexicographically to *TestString*. True if *TestString* lexicographically follows *CondPattern*.

- '**=CondPattern**' (lexicographically equal)
  Treats the *CondPattern* as a plain string and compares it lexicographically to *TestString*. True if *TestString* is lexicographically equal to *CondPattern* (the two strings are exactly equal, character for character). If *CondPattern* is "" (two quotation marks) this compares *TestString* to the empty string.
- '**-d**' (is **d**irectory)
  Treats the *TestString* as a pathname and tests whether or not it exists, and is a directory.
- '**-f**' (is regular **f**ile)
  Treats the *TestString* as a pathname and tests whether or not it exists, and is a regular file.
- '**-s**' (is regular file, with **s**ize)
  Treats the *TestString* as a pathname and tests whether or not it exists, and is a regular file with size greater than zero.

Note: All of these tests can also be prefixed by an exclamation mark ('!') to negate their meaning.

3. You can also set special flags for *CondPattern* by appending **[*flags*]** as the third argument to the RewriteCond directive, where *flags* is a comma-separated list of any of the following flags:
   - '**nocase|NC**' (**n**o **c**ase)
     This makes the test case-insensitive - differences between 'A-Z' and 'a-z' are ignored, both in the expanded *TestString* and the *CondPattern*. This flag is effective only for comparisons between *TestString* and *CondPattern*. It has no effect on file system and subrequest checks.
   - '**ornext|OR**' (**or** next condition)
     Use this to combine rule conditions with a local OR instead of the

implicit AND. Typical example:

- RewriteCond %{REMOTE_HOST} ^host1.* [OR]
- RewriteCond %{REMOTE_HOST} ^host2.* [OR]
- RewriteCond %{REMOTE_HOST} ^host3.*

RewriteRule ...some special stuff for any of these hosts...

Without this flag you would have to write the condition/rule pair three times.

Example:

To rewrite the Homepage of a site according to the 'User-Agent:' header of the request, you can use the following:

```
RewriteCond    %{HTTP_USER_AGENT}    ^Mozilla.*
RewriteRule    ^/$                   /homepage.max.html    [L]

RewriteCond    %{HTTP_USER_AGENT}    ^Lynx.*
RewriteRule    ^/$                   /homepage.min.html    [L]

RewriteRule    ^/$                   /homepage.std.html    [L]
```

Explanation: If you use a browser which identifies itself as 'Mozilla' (including Netscape Navigator, Mozilla etc), then you get the max homepage (which could include frames, or other special features). If you use the Lynx browser (which is terminal-based), then you get the min homepage (which could be a version designed for easy, text-only browsing). If neither of these conditions apply (you use any other browser, or your browser identifies itself as something non-standard), you get the std (standard) homepage.

RewriteMap

Syntax: RewriteMap name rewriteMapClassName optionalParameters

The rewriteMapClassName value also allows special values:

- int:toupper: Special map converting passed values to upper case
- int:tolower: Special map converting passed values to lower case
- int:escape: URL escape the passed value
- int:unescape: URL unescape the passed value

The maps are implemented using an interface that users must implement. Its class name is org.apache.catalina.valves.rewrite.RewriteMap, and its code is:

```
package org.apache.catalina.valves.rewrite;


public interface RewriteMap {

    default String setParameters(String params...); // calls setParameters(String) with the first parameter if there is only one

    public String setParameters(String params);

    public String lookup(String key);

}
```

The referenced implementation of such a class – in our example rewriteMapClassName – will be instantiated and initialized with the optional parameter – optionalParameters from above (be careful with whitespace) – by calling setParameters(String). That instance will then be registered under the name given as the first parameter of RewriteMap rule.

Note: You can use more than one parameter. These have to be separated by spaces. Parameters can be quoted with ". This enables space characters inside parameters.

That map instance will be given the the lookup value that is configured in the corresponding RewriteRule by calling lookup(String). Your implementation is free to return null to indicate, that the given default should be used, or to return a replacement value.

Say, you want to implement a rewrite map function that converts all lookup keys

to uppercase. You would start by implementing a class that implements the RewriteMap interface.

```
package example.maps;


import org.apache.catalina.valves.rewrite.RewriteMap;


public class UpperCaseMap implements RewriteMap {


    @Override
    public String setParameters(String params) {
        // nothing to be done here
        return null;
    }


    @Override
    public String lookup(String key) {
        if (key == null) {
            return null;
        }
        return key.toUpperCase(Locale.ENGLISH);
    }


}
```

Compile this class, put it into a jar and place that jar in ${CATALINA_BASE}/lib.

Having done that, you can now define a map with the RewriteMap directive and further on use that map in a RewriteRule.

RewriteMap uc example.maps.UpperCaseMap

RewriteRule ^/(.*)$ ${uc:$1}

With this setup a request to the url path /index.html would get routed to /INDEX.HTML.

RewriteRule

Syntax: RewriteRule Pattern Substitution

The RewriteRule directive is the real rewriting workhorse. The directive can occur more than once, with each instance defining a single rewrite rule. The order in which these rules are defined is important - this is the order in which they will be applied at run-time.

Pattern is a perl compatible regular expression, which is applied to the current URL. 'Current' means the value of the URL when this rule is applied. This may not be the originally requested URL, which may already have matched a previous rule, and have been altered.

Security warning: Due to the way Java's regex matching is done, poorly formed regex patterns are vulnerable to "catastrophic backtracking", also known as "regular expression denial of service" or ReDoS. Therefore, extra caution should be used for RewriteRule patterns. In general it is difficult to automatically detect such vulnerable regex, and so a good defense is to read a bit on the subject of catastrophic backtracking. A good reference is the OWASP ReDoS guide.

Some hints on the syntax of regular expressions:

Text:

.	Any single character
[chars**]**	Character class: Any character of the class 'chars'
[^chars**]**	Character class: Not a character of the class 'chars'

text1**|**text2 Alternative: text1 or text2

Quantifiers:

?	0 or 1 occurrences of the preceding text
*****	0 or N occurrences of the preceding text (N > 0)
+	1 or N occurrences of the preceding text (N > 1)

Grouping:

(text**)**	Grouping of text (used either to set the borders of an alternative as above, or to make backreferences, where the **N**th group can be referred to on the RHS of a RewriteRule as $**N**)

Anchors:

^	Start-of-line anchor
$	End-of-line anchor

Escaping:

****char	escape the given char (for instance, to specify the chars ".[]()" *etc.*)

For more information about regular expressions, have a look at the perl regular expression manpage ("perldoc perlre"). If you are interested in more detailed information about regular expressions and their variants (POSIX regex etc.) the following book is dedicated to this topic:

Mastering Regular Expressions, 2nd Edition

Jeffrey E.F. Friedl

O'Reilly & Associates, Inc. 2002

ISBN 978-0-596-00289-3

In the rules, the NOT character ('!') is also available as a possible pattern prefix. This enables you to negate a pattern; to say, for instance: '*if the current URL does **NOT** match this pattern*'. This can be used for exceptional cases, where it is easier to match the negative pattern, or as a last default rule.

Note: When using the NOT character to negate a pattern, you cannot include grouped wildcard parts in that pattern. This is because, when the pattern does NOT match (i.e., the negation matches), there are no contents for the groups. Thus, if negated patterns are used, you cannot use $N in the substitution string!

The *substitution* of a rewrite rule is the string which is substituted for (or replaces) the original URL which *Pattern* matched. In addition to plain text, it can include

1. back-references ($N) to the RewriteRule pattern
2. back-references (%N) to the last matched RewriteCond pattern
3. server-variables as in rule condition test-strings (%{VARNAME})
4. mapping-function calls (${mapname:key|default})

Back-references are identifiers of the form $**N** (**N**=0..9), which will be replaced by the contents of the **N**th group of the matched *Pattern*. The server-variables are the same as for the *TestString* of a RewriteCond directive. The mapping-functions come from the RewriteMap directive and are explained there. These three types of variables are expanded in the order above.

As already mentioned, all rewrite rules are applied to the *Substitution* (in the order in which they are defined in the config file). The URL is **completely replaced** by the *Substitution* and the rewriting process continues until all rules have been applied, or it is explicitly terminated by a **L** flag.

The special characters $ and % can be quoted by prepending them with a backslash character \.

There is a special substitution string named '-' which means: **NO substitution**! This is useful in providing rewriting rules which **only** match URLs but do not substitute anything for them. It is commonly used in conjunction with the **C** (chain) flag, in order to apply more than one pattern before substitution occurs.

Unlike newer mod_rewrite versions, the Tomcat rewrite valve does not automatically support absolute URLs (the specific redirect flag must be used to be able to specify an absolute URLs, see below) or direct file serving.

Additionally you can set special flags for *Substitution* by appending **[*flags*]** as the third argument to the RewriteRule directive. *Flags* is a comma-separated list of any of the following flags:

- '**chain|C**' (**c**hained with next rule)
  This flag chains the current rule with the next rule (which itself can be chained with the following rule, and so on). This has the following effect: if a rule matches, then processing continues as usual - the flag has no effect. If the rule does **not** match, then all following chained rules are skipped. For instance, it can be used to remove the '.www' part, inside a per-directory rule set, when you let an external redirect happen (where the '.www' part should not occur!).

- '**cookie|CO=***NAME*:*VAL*:*domain*[:*lifetime*[:*path*]]' (set **co**okie)
  This sets a cookie in the client's browser. The cookie's name is specified by *NAME* and the value is *VAL*. The *domain* field is the domain of the cookie, such as '.apache.org', the optional *lifetime* is the lifetime of the cookie in minutes, and the optional *path* is the path of the cookie

- '**env|E=***VAR*:*VAL*' (set **e**nvironment variable)
  This forces a request attribute named *VAR* to be set to the value *VAL*, where *VAL* can contain regexp backreferences ($N and %N) which will be expanded. You can use this flag more than once, to set more than one variable.

- '**forbidden|F**' (force URL to be **f**orbidden)
  This forces the current URL to be forbidden - it immediately sends back an

HTTP response of 403 (FORBIDDEN). Use this flag in conjunction with appropriate RewriteConds to conditionally block some URLs.

- '**gone|G**' (force URL to be **g**one)
  This forces the current URL to be gone - it immediately sends back an HTTP response of 410 (GONE). Use this flag to mark pages which no longer exist as gone.

- '**host|H**=*Host*' (apply rewriting to **h**ost)
  Rather that rewrite the URL, the virtual host will be rewritten.

- '**last|L**' (**l**ast rule)
  Stop the rewriting process here and don't apply any more rewrite rules. This corresponds to the Perl last command or the break command in C. Use this flag to prevent the currently rewritten URL from being rewritten further by following rules. For example, use it to rewrite the root-path URL ('/') to a real one, *e.g.*, '/e/www/'.

- '**next|N**' (**n**ext round)
  Re-run the rewriting process (starting again with the first rewriting rule). This time, the URL to match is no longer the original URL, but rather the URL returned by the last rewriting rule. This corresponds to the Perl next command or the continue command in C. Use this flag to restart the rewriting process - to immediately go to the top of the loop.
  Be careful not to create an infinite loop!

- '**nocase|NC**' (**n**o **c**ase)
  This makes the *Pattern* case-insensitive, ignoring difference between 'A-Z' and 'a-z' when *Pattern* is matched against the current URL.

- '**noescape|NE**' (**n**o URI **e**scaping of output)
  This flag prevents the rewrite valve from applying the usual URI escaping rules to the result of a rewrite. Ordinarily, special characters (such as '%', '$', ';', and so on) will be escaped into their hexcode equivalents ('%25', '%24', and '%3B', respectively); this flag prevents this from happening. This allows percent symbols to appear in the output, as in

RewriteRule /foo/(.*) /bar?arg=P1\%3d$1 [R,NE]

which would turn '/foo/zed' into a safe request for '/bar?arg=P1=zed'.

- '**qsappend|QSA**' (**q**uery **s**tring **a**ppend)
  This flag forces the rewrite engine to append a query string part of the substitution string to the existing string, instead of replacing it. Use this when you want to add more data to the query string via a rewrite rule.

- '**redirect|R [=*code*]**' (force **r**edirect)
  Prefix *Substitution* with http://thishost[:thisport]/ (which makes the new URL a URI) to force an external redirection. If no *code* is given, an HTTP response of 302 (FOUND, previously MOVED TEMPORARILY) will be returned. If you want to use other response codes in the range 300-399, simply specify the appropriate number or use one of the following symbolic names: temp (default), permanent, seeother. Use this for rules to canonicalize the URL and return it to the client - to translate '/~' into '/u/', or to always append a slash to /u/*user*, etc.
  Note: When you use this flag, make sure that the substitution field is a valid URL! Otherwise, you will be redirecting to an invalid location. Remember that this flag on its own will only
  prepend http://thishost[:thisport]/ to the URL, and rewriting will continue. Usually, you will want to stop rewriting at this point, and redirect immediately. To stop rewriting, you should add the 'L' flag.

- '**skip|S**=*num*' (**s**kip next rule(s))
  This flag forces the rewriting engine to skip the next *num* rules in sequence, if the current rule matches. Use this to make pseudo if-then-else constructs: The last rule of the then-clause becomes skip=N, where N is the number of rules in the else-clause. (This is **not** the same as the 'chain|C' flag!)

- '**type|T**=*MIME-type*' (force MIME **t**ype)
  Force the MIME-type of the target file to be *MIME-type*. This can be used to set up the content-type based on some conditions. For example, the following snippet allows .php files to be *displayed* by mod_php if they are

called with the .phps extension:

```
RewriteRule ^(.+\.php)s$ $1 [T=application/x-httpd-php-source]
```

- '**valveSkip|VS**' (skip valve)
  This flag can be used to setup conditional execution of valves. When the flag is set and the rule matches, the rewrite valve will skip the next valve in the Catalina pipeline. If the rewrite valve is the last of the pipeline, then the flag will be ignored and the container basic valve will be invoked. If rewrite occurred, then the flag will not have any effect.