Apache Tomcat 8

Version 8.5.100, Mar 19 2024

Realm Configuration How-To

Table of Contents

Quick Start

This document describes how to configure Tomcat to support *container managed security*, by connecting to an existing "database" of usernames, passwords, and user roles. You only need to care about this if you are using a web application that includes one or more `<security-constraint>` elements, and a `<login-config>` element defining how users are required to authenticate themselves. If you are not utilizing these features, you can safely skip this document.

For fundamental background information about container managed security, see the Servlet Specification (Version 2.4), Section 12.

For information about utilizing the *Single Sign On* feature of Tomcat (allowing a user to authenticate themselves once across the entire set of web applications associated with a virtual host), see here.

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:

- JDBCRealm - Accesses authentication information stored in a relational database, accessed via a JDBC driver.
- 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

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 (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.

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>
```

JDBCRealm

Introduction

The JDBC Database Realm has been deprecated and will be removed in Tomcat 10 onwards. Use the DataSourceRealm instead.

JDBCRealm is an implementation of the Tomcat `Realm` interface that looks up users in a relational database accessed via a JDBC driver. 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 JDBCRealm, 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. Place a copy of the JDBC driver you will be using inside the `$CATALINA_HOME/lib` directory. Note that **only** JAR files are recognized!
4. Set up a `<Realm>` element, as described below, in your `$CATALINA_BASE/conf/server.xml` file.
5. Restart Tomcat if it is already running.

Realm Element Attributes

To configure JDBCRealm, you will create a `<Realm>` element and nest it in your `$CATALINA_BASE/conf/server.xml` file, as described above. The attributes for the JDBCRealm 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)
);
```

Example `Realm` elements are included (commented out) in the default `$CATALINA_BASE/conf/server.xml` file. Here's an example for using a MySQL database called "authority", configured with the tables described above, and accessed with username "dbuser" and password "dbpass":

```
<Realm className="org.apache.catalina.realm.JDBCRealm"
      driverName="org.gjt.mm.mysql.Driver"
   connectionURL="jdbc:mysql://localhost/authority?user=dbuser&amp;password=dbpass"
       userTable="users" userNameCol="user_name" userCredCol="user_pass"
   userRoleTable="user_roles" roleNameCol="role_name"/>
```

Additional Notes

JDBCRealm 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 his or her 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.