Teiid - Scalable Information Integration

1

Server Extensions Developer Guide

6.2.0

Introduction

This guide introduces to all the user extensions that are supported by the Teiid server. These extensions are provided to let user introduce custom functionality into their deployment of the server. All the extensions are provided though implementing Java based API.

The following are the currently available extensions

1. *Teiid Security*
2. *Command Logging*

Teiid Security

2.1. Teiid Security

2.1.1. Introduction

The Teiid system provides a range of built-in and extensible security features to enable the secure access of data.

For more detailed information see the:

1. Console Guide – for instructions on how to set a range of security settings, on installing Membership Domains, and on setting up roles.
2. Connector Developers Guide – for information on how custom connectors can be developed with advanced security features.
3. Web Services Guide – for information on how to secure web service access to Teiid
4. Client Developers Guide – for information on how to authenticate through JDBC and ODBC.

2.1.2. Authentication

JDBC, ODBC, and web service clients may use simple passwords, a pass-through mechanism called a trusted payload, or a combination of that and related information to authenticate a user.

Typically a user name is required, however user names may be considered optional if the identity of the user can be discerned from the trusted payload or another mechanism. In any case it is up to the installed membership domains to actually determine whether a user can be authenticated.

NOTE: All passwords or security credentials that are passed through the Teiid system will either be encrypted or sent of an encrypted transport by default.

2.1.3. Authorization

Authorization is split into three areas of concern. Admin roles, repository roles, and data roles can be managed in the Console to enforce authorization of administrative tasks, the MetaBase repository, and enterprise data respectively. A role is simply a collection of permissions and a collection of entitled principals. Currently the Teiid security system only allows principals that represent groups to be assigned to a role. The set of available groups are determined by the installed membership domains.

2.2. Membership Domains

Membership domains are at the core of Teiid’s security system and bridge the gap between Teiid and an external security system. A membership domain provides:

- User authentication
- A set of groups
- The groups to which an authenticated user belongs

Access to membership domains is coordinated through the Membership Service. The Membership Service together with the Authorization Service implement the necessary logic to authenticate users, determine role membership, and to enforce roles.

There are multiple types of membership domains that allow for connectivity to different security systems. A Teiid server environment can be configured with multiple membership domains (there can be multiple instances of a given membership domain type). Each membership domain instance must be assigned a unique domain name in the Teiid system. The domain name can be used to fully qualify user names to authenticate only against that domain. The format for a qualified name is *username@domainname.* [mailto:username@domain]

If a user name is not fully qualified, then the installed membership domains will be consulted in order until a domain successfully or unsuccessfully authenticates the user. If a membership domain reports the user does not exist or that the credentials are not recognized, that is not considered an unsuccessful authentication and the next membership domain will be consulted.

If no membership domain can authenticate the user, the logon attempt will fail. For any failed logon attempt the message the users sees will always be the same. It will simply indicate the user could not be authenticated by any membership domain – it will not reveal any further details which could potentially be sensitive information. Details including invalid users, which domains were consulted, etc. will be in the server log with appropriate levels of severity.

2.2.1. Built-in Membership Domains

The Teiid System provides LDAP and File membership domain types.

The LDAP membership domain provides flexible integration with a variety of LDAP servers through JNDI (Java Naming and Directory Interface). Enterprise environments with an LDAP compliant directory server should attempt to utilize the built-in LDAP membership domain before attempting to create a custom solution.

The File membership domain utilizes a simple set of text files to authenticate users and to define their groups. Please note however that the File membership domain is not intended for production use.

Instructions for configuring both of these can be found in the Teiid Console Guide.

Support will be added for additional membership domains with subsequent releases.

2.2.2. Custom Membership Domains

In circumstances not covered by a built-in membership domain, a custom membership domain allows the Teiid security system to interact seamlessly with other enterprise security providers.

Java development is required to implement a custom membership domain. The API and a full example are covered in the following chapters of this document.

Custom Membership Domains

3.1. Creating a Custom Membership Domain

The creation of a custom membership domain based upon the MembershipDomain interface allows for easy extensibility of the Teiid security system. The API and associate classes are focused solely on conveying authentication and group information. At this time other development concerns, such as access to the internal logging facilities, are not documented or generally available to custom membership domain implementors.

Within the development IDE of choice the custom membership domain developer should add the server-api.jar to project's classpath. From there the MembershipDomain interface and the related classes can be extended to create custom membership domain.

Custom membership domains should be implemented to the specific needs of the external security system. For example, in cases where explicit initialization or shutdown is not applicable the implementations of these methods may be left empty.

For membership domains that do require configuration information, the initialization method provides a built-in mechanism for provide a specific set of values. Each instance of a custom membership domain defined through the Console can have a different properties file to drive its initialization. The lookup of a properties file specified through the console will search the classpath, the filesystem, and finally as a URL in that order.

Once the classes that represent the custom membership domain have been implemented they can be made available to a server installation by adding their jar file into the <server install root>/lib/patches directory. If the server is part of a multi-host cluster, the jar must be added to each host running a Membership Service.

From there the Console can be used to install and configure the custom membership domain for use in the server environment. See the Console Guide for detailed instructions.

3.1.1. Membership Domain API

The com.metamatrix.platform.security.membership.spi.MembershipDomain interface must be implemented to create a custom membership domain. The implementation class will then be installed and configured in server to enable the custom membership domain.

3.1.2. File Membership Domain Example

The following section contains an example of a membership domain that uses simple text files to determine user authentication and group membership. The users are listed in a properties file containing username=password entries. The group memberships are listed in a properties file containing comma groupname=[username[,username]*] entries.

```
import ...

public class FileMembershipDomain implements MembershipDomain {
    public static final String USERS_FILE = "usersFile";
    public static final String GROUPS_FILE = "groupsFile";
    public static final String CHECK_PASSWORD = "checkPassword";

    private boolean checkPasswords;
    private Properties users;
    private HashMap groups = new HashMap();
    private HashMap userGroups = new HashMap();

    private Properties loadFile(String fileName) throws ServiceStateException {
        Properties result = new Properties();

        //try the classpath
        InputStream is = this.getClass().getResourceAsStream(fileName);

        if (is == null) {
            try {
                //try the filesystem
                is = new FileInputStream(fileName);
            } catch (FileNotFoundException err) {
                try {
                    //try a url
                    is = new URL(fileName).openStream();
                } catch (MalformedURLException err1) {
                              throw new ServiceStateException(err, "Could not load file "+fileName+" for
FileMembershipDomain");
                } catch (IOException err1) {
                             throw new ServiceStateException(err1, "Could not load file "+fileName+" for
FileMembershipDomain");
                }
            }
        }

        try {
            result.load(is);
        } catch (IOException err) {
                        throw new ServiceStateException(err, "Could not load file "+fileName+" for
FileMembershipDomain");
        } finally {
            try {
                is.close();
            } catch (IOException err) {
            }
        }
        return result;
    }

    public void shutdown() throws ServiceStateException {
    }

.... <covered in the next pages> ...

}
```

The above snippet is shows a class that implements the MembershipDomain interface. In this example no meaningful is performed in the shutdown method, so its implementation is left empty. The loadFile method is a simple utility method that will be used to load the users and groups files during initialization (shown in the next snippet).

```
public void initialize(Properties env) throws ServiceStateException {
        checkPasswords = Boolean.valueOf(env.getProperty(CHECK_PASSWORD,
 Boolean.TRUE.toString())).booleanValue();

    String userFile = env.getProperty(USERS_FILE);
    String groupFile = env.getProperty(GROUPS_FILE);

    if (userFile == null) {
        throw new ServiceStateException("Required property " +USERS_FILE+ " was missing.");
    }

    users = loadFile(userFile);

    if (groupFile == null) {
        throw new ServiceStateException("Required property " +GROUPS_FILE+ " was missing.");
    }

    groups.clear();
    groups.putAll(loadFile(groupFile));
    userGroups.clear();
    for (Iterator i = groups.entrySet().iterator(); i.hasNext();) {
        Map.Entry entry = (Map.Entry)i.next();
        String group = (String)entry.getKey();
        String userNames = (String)entry.getValue();
        String[] groupUsers = userNames.split(","); //$NON-NLS-1$

        for (int j = 0; j < groupUsers.length; j++) {
            String user = groupUsers[j].trim();
            Set uGroups = (Set)userGroups.get(user);
            if (uGroups == null) {
                uGroups = new HashSet();
                userGroups.put(user, uGroups);
            }
            uGroups.add(group);
        }
    }
}
```

The initialize method is written to expect two properties “usersFile” and “groupsFile”. Values for these properties should be defined in the properties file specified in the “properties file” connector binding. An optional property “checkPasswords” may be specified that will determine if the membership domain should check the credentials specified as passwords.

```
public SuccessfulAuthenticationToken authenticateUser(String username, Credentials
 credential,Serializable trustedPayload, String applicationName)
                                                                              throws

 {
    if (username == null || credential == null) {
         throw new UnsupportedCredentialException("a username and password must be supplied
 for this domain");
    }

    String password = (String)users.get(username);

    if (password == null) {
        throw new InvalidUserException("user " + username + " is invalid");
    }

                                                    if (!checkPasswords ||
 password.equals(String.valueOf(credential.getCredentialsAsCharArray()))) {
        return new SuccessfulAuthenticationToken(trustedPayload, username);
    }
    throw new LogonException("user " + username + " could not be authenticated");
}
```

The authenticateUser method implementation demonstrates several possible outcomes for an authentication attempt. If a user name and password (in the form of a Credentials object) are not supplied the domain will indicate that it does not support the authentication attempt. In this case for unqualified logons, authentication would proceed to the next membership domain.

If a password cannot be found for the user name cannot be found in the users file, then the domain reports that the user is not valid for the current domain. As with the previous case, unqualified logons would proceed to the next membership domain.

If the domain is not checking passwords or the password value matches the supplied credentials, the membership domain returns a SuccessfulAuthenticationToken. This token may contain an augmented value of the trustedPayload, however in this example the value that was passed in is returned unchanged.

Finally if authentication was not successful, a LogonException is thrown to indicate the user has failed authentication. Authentication failure due to a LogonException will immediately fail the overall user authentication even with an unqualified logon.

NOTE: The message text, and any chained exceptions, in an UnsupportedCredentialException, InvalidUserException, or LogonException will appear in server log.

```
public Set getGroupNames() throws MembershipSourceException {
  Set resultNames = new HashSet(groups.keySet());
    return resultNames;
}

public Set getGroupNamesForUser(String username) throws InvalidUserException,
                                                MembershipSourceException {
  // See if this user is in the domain
    if (!users.containsKey(username)) {
        throw new InvalidUserException("user " + username + " is invalid");
    }

    Set usersGroups = (Set)userGroups.get(username);
    if (usersGroups == null) {
        return Collections.EMPTY_SET;
    }
    return usersGroups;
}
```

The last two methods needed to implement the MembershipDomain interface are shown above. The getGroupNames method returns all known group names from the groups file. The getGroupNamesForUser method returns all groups for the given user. The mapping from users to groups was established in the initialize method.

NOTE: It is important that the return values from all of the MembershipDomain methods are Serializable

NOTE: The preceding example is case sensitive with respect to user names. The Teiid system does not require logons to be case insensitive. That is up to the implementation of the member domain.

Appendix A. Advanced Topics

A.1. Authentication Interceptor

In some circumstances a trusted payload may be applicable to authenticating in several membership domains or may be a secondary authentication method for a built-in LDAP membership domain. In these situations a custom membership domain (intercepting domain) may be introduced at the beginning of the domain list to authenticate into other domains. To achieve this, the SuccessfulAuthenticationToken returned by the intercepting domain should contain a fully qualified user name to a target domain.

The interceptor domain may be solely focused on authentication. If it only authenticates into other domains, then the interceptor domain may provide dummy implementations of the getGroupNames and getGroupNamesforUser methods.

A.2. Deployment Considerations

Membership domains are not individual services that can be independently configured within a cluster, rather they are dependent upon Membership Service instances. Each installed membership domain instance will be active on each MMProcess with a Membership Service.

NOTE: Since the Membership Service cannot be restarted changes to membership domain configurations require bouncing the Teiid Server before taking effect. It follows also that if a custom membership is not written to recover gracefully from connectivity or other environmental issues a server restart is required to re-initialize the membership domain.

A.3. Migration From Previous Versions

It is recommended that customers who have utilized the internal JDBC membership domain from releases prior to MetaMatrix 5.5 migrate those users and groups to an LDAP compliant directory server. Several free and open source directory servers can be used including:

The Fedora Directory Server *http://directory.fedoraproject.org/*

Open LDAP *http://www.openldap.org/*

Apache Directory Server *http://directory.apache.org/*

Implementations of the MembershipDomainInterface interface from releases prior to MetaMatrix 5.5 will need to be manually migrated to the new MembershipDomain interface.

If there are additional questions or the need for guidance in the migration process, please contact technical support.

Command Logging

Commaand logger lets the user capture the commands that are being executed in the Teiid System at the user level as well the at the data source level. The user commands are commands that have been submitted to the system and data source commands are those that are being executed by the connectors

Users have the ability to capture command logging information in a log file, saved to a user-specified location. If a custom command logger is in use, command logging information will no longer be logged to the repository tables. Users and administrators must choose between the default logging to the Teiid Repository and custom command logging.

4.1. Command Logging API

Command Logging is defined by interface com.metamatrix.dqp.spi.CommandLoggerSPI administrators and users can write their own Java implementation of this interface and plug it into Teiid by supplying the fully-qualified classname as the value of the metamatrix.server.commandLoggerClassname system property.

Users must also add the implementation class file to the classpath of the Teiid System. This can be done by creating a jar file, adding it to the /lib/patches directory section, and doing a full stop and restart of the server.

A user’s CommandLogger implementation may require its own property names and values. Add these name/value pairs as a semicolon-delimited list after the command logger classname property in the Teiid Console. For example, a user might need to place a value such as this into the Console system property:

```
com.myCode.MyClass;metamatrix.transaction.log.storeMMCMD=true;metamatrix.transaction.log.storeSRCCMD=tru
```

The first part of this string tells Teiid what class it needs to load and instantiate. The other two parts of the string (delimited by semicolons), are a couple of name/value pairs that will be passed to the newly-instantiated Command Logger.

The system properties "metamatrix.transaction.log.storeMMCMD" and "metamatrix.transaction.log.storeSRCCMD" control whether to send the custom logger user level command and source level commands respectively.

Note

Teiid has provided a sample implementation of the SPI that logs to a file. Supply the following text for the "metamatrix.server.commandLoggerClassname" property in the Console to enable it

```
com.metamatrix.dqp.spi.basic.FileCommandLogger;dqp.commandLogger.fileName=commandLogF
```

This means that an instance of the class FileCommandLogger will be used by the new Tracking Service implementation. The FileCommandLogger is expecting a filename property called dqp.commandLogger.fileName to be passed to it, and the value of that property is commandLogFile.txt.