HyperSQL External Authentication

Corrections, criticisms, and comments will be appreciated. Please email them to blaine <dot> simpson <at> admc <dot> com.

We will work though examples of the different methods of external authentication supported by HyperSQL version 2.1.0 and later. I assume familiarity with basic SQL and Java. I will assume here that you are running Java 6 or later. When I say *the main HyperSQL directory* below, I mean the HyperSQL installation directory that contains the subdirectories `build`, `lib`, etc.

The only other setup required is to define the URLID "mem" for a memory-only catalog in the file `sqltool.rc` in your home directory. If you don't have a `sqltool.rc` file in your home directory, just copy the file `sample/sqltool.rc` from your HyperSQL distribution to there. Otherwise, copy or otherwise ensure that you have a "mem" Urlid stanza like that in `sample/sqltool.rc`.

So that I can concentrate on the target matter with minimal setup, and without favoring specific IDE's or frameworks, this tutorial works from an operating system shell. I assume that you know how to get a command shell prompt and run Java commands. To eliminate lengthy explanations of pulldown menus, or the need for dozens of screen shots, I am using the command line tool `SqlTool` in preference to DatabaseManager, SquirrelSQL, etc. If you prefer to use a graphical tool and are bright enough to adjust the given instructions to your tool, you are welcome to do so.

Each section of this documents builds upon previous sections. Therefore, you are advised to follow the document in the given sequence. If you aren't interested in all sections, then stop wherever you want to-- just don't skip around. In particular, if you are not interested in LDAP authentication, then skip the last sections.

Table of Contents

Authentication Functions

Authentication Functions are SQL/JRT Java methods, as described in the Java Language Routines chapter of the User Guide. The backing Java method must satisfy the signature

```
public static java.sql.Array authenticate( String database, String user, String password) throws Exception
```

The throws may be narrowed to any Exception(s) or none. You may obtain the class containing this method from HyperSQL, from somewhere else, or you may write it yourself. For each catalog that you wish to use the authentication function for, you must run the SQL statement

```
SET DATABASE AUTHENTICATION FUNCTION EXTERNAL NAME 'CLASSPATH:full.name.of.static.method'
```

The User Guide explains both the SQL statement and the Java method requirements in this section, but at the time I am writing this, the explanations are wrong in some particulars. I'll update the text here if the User Guide gets corrected.

To save some coding, we'll use a minimalistic auth function method that is in the HyperSQL distribution to support unit testing. If your distribution installation does not contain the file `testsuite.jar`, then run

```
ant -f build/test.xml make.test.suite
```

from the main HyperSQL directory. (Ant will require that you put a 3.x version of the JUnit jar file where it needs it).

1. From the main HyperSQL directory, run
   ```
   java -cp lib/sqltool.jar:lib/testsuite.jar org.hsqldb.cmdline.SqlTool mem
   ```
   On Windows, use semi-colon instead of colon to delimit the classpath elements.
2. Once you have a session, plug in the indicated auth function method like this:
   ```
   SET DATABASE AUTHENTICATION FUNCTION EXTERNAL NAME 'CLASSPATH:org.hsqldb.auth.AuthFunctionUtils.changeAuthFn';
   ```
   As you can see by looking at the implementation of this method in the file `test-src/org/hsqldb/auth/AuthFunctionUtils.java`, it returns the role `CHANGE_AUTHORIZATION`, which is required to use the `CONNECT` command.
3. Prove to yourself that, by default, external authentication is not used to access the SA account.
   ```
   CONNECT USER sa PASSWORD 'wrong';
   ```
   This should fail. It's usually a good idea to leave the SA account set to local-authentication-only. You can use the command `ALTER USER x SET LOCAL true` (or `false`) to specify whether registered AUTHENTICATION FUNCTIONs will be applied to the account. This ALTER USER command is documented in this section of the User Guide.
4. Connect to a non-existence account with a bogus password to verify that the `changeAuthFn` method allows access.
   ```
   CONNECT USER fake PASSWORD 'bogus';
   ```
5. To verify that the function authorized to the `CHANGE_AUTHORIZATION` role, run
   ```
       SELECT * FROM information_schema.enabled_roles;
       COMMIT;
       CONNECT USER SA PASSWORD '';
   ```
   The `COMMIT` statement is to prevent a complication where the database engine doesn't know what we want to do with the active transaction before we switch to the new session.

As explained above, the methods for authentication functions are *static* Java methods. If your program uses only one catalog, or you want the same authentication for all catalogs, then you are all set. If you want catalog-specific authentication behavior, then you will need to use a different Java static authentication method for each database catalog, or your Java method implementation will have to switch behavior based upon the value of the database parameter.

The value of the database parameter is not a portion of the JDBC URL, but a strictly 16-character unique catalog identifier that you can determine with the SQL statement `CALL database_name()` and you can change with the SQL statement `SET DATABASE UNIQUE NAME MYDB123456789012`. Database identifiers are special in that double-quoting will not allow you to use anything other than digits and capital letters, or to start with a non-capital-letter. As per normal database object names, unquoted lower-case letters will be raised to caps.

The return value of authentication function methods represents a compromise between simple usage and the need to satisfy SQL/JRT specification requirements. If the method throws (a checked or unchecked Exception) instead of returning a value, then access is denied. Otherwise access is permitted as follows.

- If the method returns the null value, it means to allow access with the initial schema, roles and roles specified for the local account, if any; or to allow access with default initial schema and no roles or privileges if there is no local account with that name.
- If the method returns a `java.sql.Array`, then access will be permitted with roles and initial schema set according to the array elements, and no direct privileges. If there is a local account of the same name, then the local account settings will be ignored. With respect to the individual array elements, if the element is the name of an existing schema, then it will be interpreted as an initial schema setting; if the element is the name of an existing role, then it will be added as a role; otherwise the value will be ignored. Consequently, a list of values, none of which are a valid role or schema, will result in the user being logged in with default initial schema and no roles, regardless of the existence or state of a local account of the same name.

AuthBeanMultiplexer

`AuthBeanMultiplexer` is an authentication function method that serves as a bridge between a static authentication function method, and `AuthFunctionBean` Java beans which are catalog-specific. `AuthBeanMultiplexer` does the work of switching to the bean(s) registered for the catalog and allows for catalog-specific settings which will not collide with settings for other catalogs. It also allows for providing backup authenticators for when some authentication methods fail due to system problems (such as inability to reach the primary remote authentication server). The same exact method, `org.hsqldb.auth.AuthBeanMultiplexer.authenticate` can safely be registered with any number of catalogs in the same JVM.

A primary use case is container-managed HyperSQL database catalogs. An application server with multiple container-managed HyperSQL databases will have HyperSQL classes stored in a single container ClassLoader. If two webapps or enterprise apps happen to use the same authentication class, they will necessarily share the same state and settings. `AuthBeanMultiplexer` is intended to be have cross-application-shareable state and settings, but each catalog gets its own list of authenticators (encapsulated in an `AuthFunctionBean` instance).

See the API spec for `org.hsqldb.auth.AuthFunctionBean`. The method `org.hsqldb.auth.AuthFunctionBean.authenticate()` is basically a more simple and controlled version of the direct authentication functions described in the previous section. Most importantly, there is no database name parameter, since AuthBeanMultiplexer will not invoke the method unless it has already matched the database/catalog name. While throwing `Exception`s still means to not allow access, `AuthFunctionBean.authenticate` differentiates between Runtime and checked Exceptions. The former indicates an authentication *system* failure, so that AuthBeanMultiplexer will try remaining AuthFunctionBeans for the catalog (if any). Checked Exceptions mean that the authenticator has purposefully decided that the attempt should be rejected, and therefore no other AuthFunctionBeans will be tried. Finally, the type returned by `AuthFunctionBean.authenticate()` is the simpler primitive String array, but the elements of the lists are exactly the same as for a direct auth function method (as described fully in the previous section).

`AuthBeanMultiplexer` provides a singleton JavaBean instance which can be used as an equivalent alternative to the static methods. This AuthBeanMultiplexer singleton and the AuthFunctionBeans can therefore be configured and managed declaratively with generic JavaBean facilities such as app server web consoles, app server XML configuration files, or app-bundled Spring bean files.

Be aware that the authentication settings are purposefully not stored in the database itself. It is stored in the database whether or not to use the AuthBeanMultiplexer, but the AuthFunctionBeans to be used, i.e. the settings for AuthBeanMultiplexer itself and its AuthFunctionBeans are set at runtime by Java code, or more typically, by JavaBean settings loaded by your container or framework as described in the previous paragraph. We purposefully consider these to be deployment or runtime application settings, in the same way that JDBC URLs for container-managed data sources are managed as runtime settings by application servers. If your app uses application-managed data sources, then your database backup strategy should make sure to also save your AuthBeanMuliplexer configurations. Note that this design does accommodate application recovery even if the AuthBeanMultiplexer settings are lost, because by default the SA account is a *local-auth-only* account. Therefore, even if AuthBeanMultiplexer is totally incapacitated, you can log into the database as user SA and disable external authentication until you restore or fix the AuthBeanMultiplexer settings.

HyperSQL Master/Slave Authentication

HyperSQL Master/Slave Authentication is a situation where one or more slave databases behave as if they have account records matching those in a master database. The master database is a normal HyperSQL catalog which may have been set up for the dedicated purpose of serving as a master catalog, or that may just be one of its purposes. Like all external authentication methods, if the local `information_schema.system_users` record for a given `user_name` has `authentication` value of `LOCAL`, then authentication will occur for that user as if no external authentication had been set up. All we will do in this section is run and look at the *extAuthWithSpring* sample in your HyperSQL distribution. The portion of extAuthWithSpring that we will examine in this section uses Spring bean XML files to configure a `HsqldbSlaveAuthBean` for our application to authenticate to our in-memory application database which acts as an authentication slave to another in-memory authentication master database.

From the directory `integration/extAuthWithSpring`, run

```
ant -Dauthentication.mode=HsqldbSlave
```

The rest of this section will give file paths relative to the `integration/extAuthWithSpring` directory. Unless you happen to have your environment set up right ahead of time, you will get an error message with a suggestion of how to set up your environment. Copy the suggested command for your particular environment and execute it on your command line shell. Then re-run the Ant command above. If the last line of output for the Ant run says "`BUILD SUCCESSFUL`", then the sample program has run successfully.

Now I'll show you what the program did. The ultimate purpose for the sample is to run the JDBC code in method `doJdbcWork()` in the class `JdbcAppClass`.

1. Open up the file `src/org/hsqldb/sample/JdbcAppClass.java` with an editor, web browser, or text file viewer. Notice that the source uses generic JDBC. It is neither HSQLDB-specific nor Spring-specific. The purpose of the `doJdbcWork()` method is to read the table `t1` and verify that the read-in value is `456`. If successful, it will log the message "`Application Success`".
2. Open up the file

   `src/org/hsqldb/sample/SpringExtAuth.java` for viewing. The main method here gets called by Ant after Ant compiles the program. The main method calls `prepMemoryDatabases()` for the obvious purpose, then loads a Spring Framework context, and finally invokes the method `JdbcAppClass.doJdbcWork`.

   The work in `prepMemoryDatabases()` is entirely for setting up this example. In any real app, the master database will exist ahead-of-time. An application with a persistent application database will not need to do any runtime setup of that database, but the `AuthBeanMultiplexer` function must be plugged in ahead of time with the `SET DATABASE AUTHENTICATION...` command (a one-time operation for a persistent database). An application with an in-memory database will do nothing special other than plugging in the `AuthBeanMultiplexer` each time that the database is started and populated. The `SET DATABASE AUTHENTICATION...` command is described in the first section above.

   If your application uses Spring, it will already be wired to automatically instantiate a Spring context, and more likely than not, your container will invoke your application work methods. This sample uses Spring manually so that we can set up external authentication declaratively but without the need for a complicated container or application server. You could use an app server's JavaBean management facility to load the settings. You could also combine the two, for example to have JBoss load a Spring bean factory or context at the server level (as opposed to at the application level). Notice that we have Spring load the file beandefs.xml, plus either "ldapbeans.xml" or "slavebeans.xml". Since for this section we are running with argument "HsqldbSlave", we will load "beandefs.xml" and "slavebeans.xml".

3. Open up the XLM files `resources/beandefs.xml` and `resources/slavebeans.xml` with a good viewer, browser, or editor. See the API spec for `org.hsqldb.auth.AuthBeanMultiplexer` and `org.hsqldb.auth.HsqldbSlaveAuthBean` to see what the used and available methods for these classes do. Spring developers should immediately see what is going on here, and how easy it is to change settings. Non-Spring developers with a basic understanding of JavaBeans should be able to figure it out if they pay attention to the comments and the property and bean names.

JAAS Integration

`JaasAuthBean` provides a bridge between *authentication for a HyperSQL catalog* and *JAAS login module(s)*. The critical property of JaasAuthBean is `applicationKey`. This tells HyperSQL which runtime JAAS configurations to apply. In the case of the Sun and OpenJDK JVMs, at least, the applicationKey is the name of the stanza that will be used from the JAAS config file. The JAAS config file stanza simply lists JAAS login modules and the settings for each module. See the reference guide for details about JAAS.

The goal of this section is to see how to use JaasAuthBean to plug JAAS login modules into HyperSQL. We will run a sample program that does just this, and which proves it by using the JAAS module to access our database catalog and retrieving an expected value. The setup is basically an extension to the setup of the previous section. We have to configure the `JaasAuthBean` with `AuthBeanMultiplexer`, but we also have to configure the JAAS login modules that `JaasAuthBean` will use.

1. View the XML file `resources/jaasbeans.xml`. If you understood the file `resources/slavebeans.xml`, this is entirely analogous but even simpler. See the HyperSQL API Spec for `JaasAuthBean` to find out about the available properties. Note that we specify to use applicationKey value `demo`. That's all the setup needed for the bean.
2. Now we need to configure JAAS. Our sample app will use the JAAS configuration file `resources/jaas.cfg`. (You can see in `build.xml` where we specify the location of the JAAS configuration file with a Java system property). View this file. The syntax for the file is documented at http://download.oracle.com/javase/6/docs/api/javax/security/auth/login/Configuration.html, with the significant omission that they don't say that you can use both "//" and "/*...*/" style comments. All we are interested in here is the {} block after the application key "`demo`". Those who have configured PAM authentication will see that the setup is just like PAM setup. For our application, the config file just says that success of the `StartCharModule` is required for database access, with the specified `StartCharModule` options. The option values here tell `StartCharModule` to allow access to anybody supplying a user name starting with "s" and a password starting with "p". This will work for us because, as you can see in `resources/beandefs.xml` (which we saw in the previous section), our app will try to connect to our application database with user name "straight" and password "pwd".
3. Run
   ```
   ant -Dauthentication.mode=JAAS
   ```
   If everything succeeds and the application successfully authenticates and then retrieves the expected value from the database, the Ant command will complete with the message "`BUILD SUCCESSFUL`".

Our JAAS configuration said to use JAAS login module `StartCharModule`, but you can specify any login modules which provide both a JAAS `NameCallback` and `PasswordCallback`, as described in the API Spec for `JaasAuthBean`. There are open source and commercial JAAS login modules available on the Internet, and you can write your own according to the JAAS Reference Guide. Sun Java 1.6 comes with several login modules, and I'll say something about them in the next section.

LDAP Support by Sun JAAS Module

Sun Java 1.6 comes with several JAAS login modules. Only one of these modules supports JAAS `NameCallback` and `PasswordCallback`, as required by `JaasAuthBean`. That module is `com.sun.security.auth.module.LdapLoginModule`. If you have an LDAP server, you can modify the settings in `resources/jaas.cfg` in the block for application "sunLdap", according to the comments therein, and run

```
ant -Dauthentication.mode=JAAS_LDAP
```

I am not covering this in detail because most users who want to authenticate using LDAP would be better off using the direct (non-JAAS) `AuthFunctionBean` described in the following sections. Sun's `LdapLoginModule` is not very flexible, supports the deprecated LDAPS instead of StartTLS, and can only return a single value that we can use to assign a role or initial schema.

If you do want to use `LdapLoginModule`, just work through the rest of this tutorial, pay attention to the comments in `resources/jaasldapbeans.xml`, and `resources/jaas.cfg`, and you will learn all you need to know to get `JaasAuthBean` and `LdapLoginModule` working together.

LDAP Support by LdapAuthBean

The remainder of this document is only of use if you have access to an LDAP server. The class `org.hsqldb.auth.LdapAuthBean` is an implementation of the `org.hsqldb.auth.AuthFunctionBean` interface, exactly like `HsqldbSlaveAuthBean` which was explained in the previous section. But LdapAuthBean also provides a utility program for playing with and testing your LDAP server and your LdapAuthBean settings. The subsequent section will explain how to run the *extAuthWithSpring* sample against your own LDAP server. This section will explain how to set up an LdapAuthBean instance so that HyperSQL will work with your LDAP server.

1. Run
   ```
   java -cp lib/hsqldb.jar org.hsqldb.auth.LdapAuthBean sample/ldap-exerciser.properties myuser mypassword
   ```
   from the main HyperSQL directory. You will get an error message and stack trace, because the program will try to authenticate using the LDAP settings in the file `sample/ldap-exerciser.properties`, and these won't match any LDAP server that you have running... besides the fact that you probably don't have an LDAP record for myuser/mypassword.
2. Make your own LDAP properties file by copying the `ldap-exerciser.properties` file. Something like
   ```
           mkdir tmp
           copy sample/ldap-exerciser.properties tmp/hldap.properties
   ```
   (or right-click the link and download it to there).
3. Open your copy of the properties file with a text editor. If I have achieved my purpose, that file is self-descriptive. Read the comments closely, especially the suggestion to go to the API spec for details about specific settings. You must set the property values in this file to match the access, security, and Directory Information Tree layout of the LDAP server that holds your account records. Follow the commented instructions to enable the settings for PLAIN authentication, StartTLS, DIGEST-MDS5 SASL, or any combination thereof. The examples given should make it clear how to set up any other authentication mechanism. The existence of *Attribute, *Pattern*, and *Template settings allow for LdapAuthBean to accommodate your DIT instead of the other way around. The Patterns and Templates allow you to test for or extract substrings from existing LDAP values, and templates allow you to do LDAP authentication with values derived from the authenticating HyperSQL user name.
4. Leave your editor open and re-run LdapAuthBean with a real LDAP account user name and password, like
   ```
   java -cp lib/hsqldb.jar org.hsqldb.auth.LdapAuthBean tmp/hldap.properties myuser mypassword
   ```
   (Remember that it is generally unsafe to give passwords on your command line. Protect your passwords!) When authentication succeeds, the program will display that fact, or will display the roles and initial schema retrieved (depending on whether you have set property `rolesSchemaAttribute` or not). Keep adjusting your properties file and re-running LdapAuthBean until the program reports the desired results.

LDAP Authentication Example

If you successfully completed the previous section, then you should now be very confident that you know all of the values needed by `LdapAuthBean` for your specific LDAP server. In this section I will once again specify file paths relative to directory `integration/extAuthWithSpring`. We will run the extWithAuthSpring sample again, but this time we'll tell it to not run the authentication master database, but to instead access your LDAP database for authentication decisions.

1. Open up the file `resources/ldapbeans.xml` with a text or XML editor. Set the property values to match those of the properties file that you set up in the previous section. The only anomaly is for property file property `trustStore`, which you can ignore right now because we'll cover that in the next step.
2. From the directory `integration/extAuthWithSpring`, run
   ```
   ant -Dauthentication.mode=LDAP
   ```
   unless you needed to set property `trustStore` in the previous section. In that case you need to tell Ant the path to your trust store too, like this.
   ```
   ant -Dauthentication.mode=LDAP -Dtruststore.path=$HOME/ca/cacert.store
   ```
   If everything succeeds and the application successfully authenticates and then retrieves the expected value from the database, the Ant command will complete with the message "`BUILD SUCCESSFUL`".

Generally, to have LdapAuthBean trust an otherwise untrusted LDAP server's certificate (with StartTLS), you use the standard Java SE mechanism for the purpose by setting Java system property `javax.net.ssl.trustStore`.