Using Scripted JDBC connector

jdbc, connector

Configuration

The configuration is done using standard JDBC setting, i.e. setting a proper URL template, JDBC driver etc. We are mainly interested in configuring the CRUD operation scripts. You can either:

- set script path on filesystem (recommended)
- set an inline script (recommended only for deployments without FS)

If you run CzechIdM on appliance, you can place the scripts inside /data/volumes/czechidm/data/ folder, create e.g. a folder /data/volumes/czechidm/data/ourSystemScripts. Then the script path configured in CzechIdM will be e.g. /opt/czechidm/data/ourSystemScripts/search.groovy.

Another useful setting is "Reload script on execution" (reloadScriptOnExecution), which mainly helps during the development stage. This option should be turned off in production, because reloading (recompiling) the Groovy scripts takes time and memory. However, when you need to upgrade the script and this option is turned off, then you have to restart IdM, otherwise the new version of the script wouldn't be loaded. Don't turn on the option "Reload script on execution" only temporarily, because it may not preserve the loaded newer version of the script, after you turn the option off again!

Pooling configuration

We recommed using the connector pool when connectiong systems with this connector. This will make connecting to the datasource more effective. Also, this connector has a suspected memory leak when compiling groovy scripts repeatedly, which is avoided if connector pooling is enabled and "Reload script on execution" is disabled (see above)

Schema attributes

While retrieving account data from target system (the SEARCH operation), the connector requires us to set both __NAME__ and __UID__ attribute unconditionally. Both **must** be Strings. Therefore you are required to do this manually in your SEARCH script. Following is an example of retrieving user data:

```
import groovy.sql.Sql
import groovy.transform.Field

@Field UID_ATTR = "id_column"
@Field TABLE_NAME = "abc_users_table"

def sql = new Sql(connection)
def result = []
String select = "SELECT * FROM $TABLE_NAME ${getWhere(query)}"
```

```
sql.eachRow(select, { row ->
    def res = [:]
    res.put("__UID__", row[UID_ATTR])
    res.put("__NAME__", row[UID_ATTR])
    row.getMetaData().collect({ m -> m.columnName })
            .each({ column -> res.put(column, convertAttribute(column,
row[column])) })
    result.add(res)
})
return result
```

All retrieved attributes must be **a list of maps** (look for `result` and `res` objects in the example above). This is crucial for SEARCH connector method to work properly.

Another 'catch' the connector has is that the __NAME__ is a required input of the CREATE operation. Since it is not expected that you have any column called "__NAME__" in your database schema, this leaves you pretty much two options how to handle the attribute in your installation:

1. ignore the attribute in your script
2. always rename the attribute in the script

The first option expects that you have an additional attribute __NAME__ in your system mapping in CzechIdM, which maps to the exactly same value as your mapping system identifier. Lets say we have a user's table USERS with 1 columns: LOGIN - identifier. In such case we create a new system in CzechIdM and while configuring the system schema, we create **two** attributes:

- LOGIN
- __NAME__

Then in mapping we check the LOGIN attribute as identifier and choose however we want to fill its value. The only difference with __NAME__ attributes is that we will not check the identifier checkbox, otherwise it stays the same.

The second option requires that all of your CRUD scripts handle the __NAME__ attribute in a special way. For example you can rename it to your ID column name. This way of handling the identifier is rather straightforward, but causes high pollution of your JDBC scripts with non-reusable code.

Using groovy SQL

Groovy has a powerful yet simple mechanism of querying databases through JDBC implemented in the `groovy.sql.Sql` class. However one has to be really careful and precise while using groovy's Sql with identity connector framework's (ICF) Attribute objects, because of its loose typing system.

All values of ICF Attribute are of type java.lang.Object. To put it simply, typing does not exist here and the developer must handle it manually. If you only send simple attributes such as Strings, there is probably no need to worry about your queries. However mixing types together may cause unpredictable results.

For example lets say we have a table USERS with columns ID (int) and LOGIN (varchar) and we want

to select a row where ID = 123. So we will do something like this:

```
def id = 123
def retrieved = sql.firstRow("SELECT * FROM USERS WHERE ID = ? LIMIT 1",
[id])
println retrieved
```

Now everything should work just fine and we get our desired row printed out. But what if `id` is a String? Meaning what if we pass `def id = '123'`? Here we get some unpredictable behaviour. When using PostgreSQL, the query will deliberately fail with an exception. But on MySQL everything **works just fine**.

The danger with ICF Attributes is that these contain a *value* field - a list of values the Attribute carries. But if you accidentally call `myString.value` on a String parameter, you get an object of type `class [C`. The funny thing is that your query will still pass on MySQL, but will not return anything (most probably). Therefore always check your types while using groovy Sql!

List of input variables

Following is a list of input variables you can expect in JDBC scripts.

CREATE script

- connection: SQL connection
- action: String correponding to the action ("CREATE" here)
- log: a handler to the Log facility
- objectClass: a String describing the Object class
- id: The entry identifier (OpenICF "Name" atribute. (most often matches the uid)
- attributes: an Attribute Map, containg the `String` attribute name as a key and the `List` attribute value(s) as value.
- password: password string, clear text
- options: a handler to the OperationOptions Map

SEARCH script (GET)

- connection: handler to the SQL connection
- objectClass: a String describing the Object class
- action: a string describing the action ("SEARCH" here)
- log: a handler to the Log facility
- options: a handler to the OperationOptions Map
- query: a handler to the Query Map

UPDATE script

- The connector sends us the following:
- connection : SQL connection

- action: String correponding to the action (UPDATE / ADD_ATTRIBUTE_VALUES / REMOVE_ATTRIBUTE_VALUES)
- UPDATE: For each input attribute, replace all of the current values of that attribute in the target object with the values of that attribute
- ADD_ATTRIBUTE_VALUES: For each attribute that the input set contains, add to the current values of that attribute in the target object all of the values of that attribute in the input set.
- REMOVE_ATTRIBUTE_VALUES: For each attribute that the input set contains, remove from the current values of that attribute in the target object any value that matches one of the values of the attribute from the input set.
- log: a handler to the Log facility
- objectClass: a String describing the Object class
- uid: a String representing the entry uid
- attributes: an Attribute Map, containg the `String` attribute name as a key and the `List` attribute value(s) as value.
- password: password string, clear text (only for UPDATE)
- options: a handler to the OperationOptions Map

DELETE script

- connection: handler to the SQL connection
- action: a string describing the action ("DELETE" here)
- log: a handler to the Log facility
- objectClass: a String describing the Object class
- options: a handler to the OperationOptions Map
- uid: String for the unique id that specifies the object to delete