ChatScript Database Access

Revision 4/24/2022 cs12.1

Overview

By default, a CS server records the current state of a user in a file stored in the local filesystem under the name topic_username_botname.txt. This works fine for single-server systems. In a world where one uses a second server as a hot backup and transfer users to it while redeploying the primary server, user information gets lost. The solution to this (and the ability to scale CS with an arbitrary number of servers) is to have some kind of networked file server.

CS directly supports use of Mongo, Postgres, MySQL,and MsSQL databases as file servers. Instead of writing the users's state record to the local file system, it writes it as a record in the database. That record will be overwritten each volley with the changed state. There is no accumulation of prior state data. Nor are user logs ever written there. If user logging is on, such logs are written to the machine performing the chat (which may scatter them over many machines). Likewise server logging is kept per machine.

MySQL Fileserver and Database

Fileserver Setup

To run CS MySQL you need two things. First, you need to run an executable that has been compiled with MySQL enabled.
Second, you need to supply parameters for CS startup in the cs_init.txt file or cs_initmore.txt file. A line entry looks like this:

```
mysql="host=localhost port=3306 user=root password=admin db=chatscript"
```

where the host and port refer to the MySQL server access, user and password refer to some userid on the mysql server allowed to access the named db.
The database itself needs to already exist, it is not automatically created by CS. It should be defined with the following fields in the schema/table called chatscript using this statement: The keys (hose, port, user, password, db, appendport) should be in lowercase.

```
create database chatscript character set 'UTF8';
create user chatuser identified by 'password';    or whatever
grant all on chatscript.* to chatuser;
use chatscript;
CREATE TABLE userfiles (userid varchar(100) PRIMARY KEY, file MEDIUMBLOB, when DATETIME DEFA
```

The table name "userfiles" and the field names are fixed. ChatScript assumes them. userid: a limit of 64 is more than enough file: in our world we are going

to have less than 500K bytes of data when: CS never sends or uses this value. It is there to allow an external program to query the database and delete old records since our users do not continue forever with Pearl.

An optional key is `appendport` which takes no value. It means to add the current CS port as a suffix onto the db name. So

```
mysql="host=localhost port=3306 user=root password=admin db=chatscript appendport"
```

will yield a db name of `chatscript1024` for a default CS server.

MySQL Termination:

If at any time a call to utilize MySQL fails, the CS server will abort itself. If you have an automatic restart and retry mechanism, then maybe things will just work next time, or a different CS server will get invoked.

MySQL database

A script may also access MySQL using 3 functions: ^MySQLInit, ^MySQLQuery, ^MySQLClose.

^MySQLInit("information string")

To open a database access, you need to supply a string of information. The data is similar to the cs_init.txt mysql line, e.g.,

```
^mysqlinit("host=localhost port=3306 user=root password=admin db=chatscript")
```

This implies you must have already created the database to be used.

^MySQLClose()

When you are done accessing the database, this will end the connection.

^MySQLQuery("query string" {'^fn})

You may provide a query in the query string. The output, if there is any, will be returned from this function. You have the option to provide a quoted function name to execute on each row of data returned. If your function only takes 1 argument, all output from a row will be sent to it as a single quoted string. If your function takes multiple arguments, the separate columns of a row will be passed to the corresponding arguments.

Sample Bot

There is a sample bot compilable with :build mysql.

MsSQL Fileserver

Currently Microsoft SQL can be used as a file server or as a database. As a file server, this allows infinite scaling of CS.

Fileserver Setup

For Linux you need to install a Microsoft ODBC driver to your machine. Instructions are here: https://docs.microsoft.com/en-us/sql/connect/odbc/linux-mac/installing-the-microsoft-odbc-driver-for-sql-server?view=sql-server-ver15

To run CS MsSQL you need two things. First, you need to run an executable that has been compiled with MySQL enabled.
Second, you need to supply parameters for CS startup in the cs_init.txt file or cs_initmore.txt file. A line entry looks like this:

```
mssql="host=localhost port=3306 user=chatuser password=admin db=chatscript"
```

where the host and port refer to the MsSQL server access, user and password refer to some userid on the mssql server allowed to access the named db.
The database itself needs to already exist, it is not automatically created by CS. It should be defined with the following fields in the schema/table called chatscript using this statement: The keys (host, port, user, password, db) should be in lowercase.

```
create database chatscript character set 'UTF8';
create user chatuser identified by 'password';    or whatever
grant all on chatscript.* to chatuser;
use chatscript;
CREATE TABLE userfiles (userid varchar(100) PRIMARY KEY, [file] VARBINARY(max), stored DATET
```

The table name "userfiles" and the field names are fixed. ChatScript assumes them. userid: a limit of 64 is more than enough file: in our world we are going to have less than 500K bytes of data stored: CS never sends or uses this value. It is there to allow an external program to query the database and delete old records since our users do not continue forever with Pearl.

MsSQL Termination:

If at any time a call to utilize MsSQL fails, the CS server will abort itself. If you have an automatic restart and retry mechanism, then maybe things will just work next time, or a different CS server will get invoked. Such a mechanism is done automatically by CS.

From user script one can open a database with `^msqlinit($_params)` and close it with `^msqlclose()`. `^msqlread($_username)` and `^msqlwrite($_username $_value)` perform db operations. Init Params are the same as for using such a db for a filesystem.

Mongo Fileserver and Database

Aside from using Mongo to store data for a chatbot to look up, one can also use Mongo as a replacement for the local file system of the USERS directory. Log files will remain local but the `USERS/topicxxxxx` will be rerouted, as will any `^export` or `^import` request. If you name your file `USERS/ltm-xxxxx` then that will be routed to its own collection, otherwise it will go into the same collection as the topic file.

To set collections, on the command line use:

```
mongo="mongodb://localhost:27017 ChatScript topic:MyCollection"
```

or for user topic and ltm files

```
mongo="mongodb://localhost:27017 ChatScript topic:MyCollection ltm:MyLtm"
```

or for a remote host with no ltm file.

```
mongo=mongodb://127.0.0.1:27017 ChatScript topic:UserTopics
```

Any `^import()` and `^export()` will check the file name for a match against a collection reference to allow for any document to be redirected to the database. For example, with the following definition any file names that cache will be saved in a MyCache collction in the Mongo database.

```
mongo="mongodb://localhost:27017 ChatScript topic:MyCollection ltm:MyLtm cache:MyCache"
```

Read preferences for queries on Mongo collections can be added to each collection definition. To specify the cache documents can be read from a secondary replica, but keeping the topic and ltm read from primary (the default), use definitions like

```
mongo="mongodb://localhost:27017 ChatScript topic:MyCollection ltm:MyLtm cache:MyCache?readP
```

or

```
mongo="mongodb://localhost:27017 ChatScript topic:MyCollection ltm:MyLtm cache:MyCache?readP
```

The additional options follow the standard Mongo style used in URL definitions, see https://www.mongodb.com/docs/manual/reference/connection-string/#read-preference-options

Using a cs_init.txt file to contain that line is most convenient, so no quotes are needed.

Obviously put the correct data for your mongo machine. CS will store user topic files in the Mongo machine and well as `^export` and `^import` data. For user and server logs it will continue to store those on the local machine. You may have the same or different collection active, one or two via filesystem replacement and once via normal script access.

Access A Mongo Database

There are only a couple of functions you need.

^mongoinit(server domain collection)

Once the named collection is open, it becomes the current database the server uses until you close it no matter how many volleys later that is. Currently you can only have one collection open at a time.

By the way, if a call to ^mongoinit fails, the system will both write why to the user log and set it onto the value of $$mongo_error. ^mongonit will fail if the database is already open.

^mongoclose()

closes the currently open collection.

^mongoinsertdocument(key string)

does an up-sert of a json string (which need not have double quotes around it. Key is also a string which does not require double quotes.

^mongodeletedocument(key)

removes the corresponding data.

^mongofinddocument(pattern)

finds documents corresponding to the mongo pattern (see a mongo manual).

```
^mongoinit(mongodb://localhost:27017 ChatScript InputOutput)

^mongoinsertdocument(dog ^"I have a dog")
$_var = ^mongofinddocument(dog)
^mongodeletedocument(dog)
# $_var  ==  I have a dog

^mongoclose()
```

You can include a JSON object's worth of additional attributes on a Mongo upsert using $cs_mongoqueryparams.

You also use a private hook code to alter things on upsert.

Mongo CS variables

$mongo_enable_ssl if set to "true" will then use these variables:

```
$mongosslcafile
$mongosslpemfile
$mongosslpempwd
$mongovalidatessl
$mongo_timeexcess -- record into log file if operation takes more than this ms
```

$$mongo_error

Postgres Fileserver and Databse

While standing in the SRC directory, you can do: `make server` - build ordinary CS server w/o PostgreSQL client `make pgserver` - build CS server w PostgreSQL client (you had to install postgress first)

Usually postgres somes with amazon servers, but if not you may have to do something like

```
sudo apt-get install libpq-dev
```

to build successfully.

On Mac and IOS `#define DISCARDPOSTGRES` is on by default. If you have access to a PostgreSQL server remotely, that's fine. If you need one installed on your machine, see the end for how to install one.

Using Postgres as file server

To do this, on the ChatScript command line use:

```
pguser="dbname = mydb host = 129.22.22.24 port = 5432 user = postgres password = somepasswo
```

If host is omitted, it defaults to localhost. Localhost might not work as the name of the host, in which case use 127.0.0.1 .

If you do not specify dbname, the system assumes you have a db named `users`. If it doesn't find it, it will try to open a root database named `postgres`. If it can find that, it will then create the `users` db. CS will automatically create a `users` database and a `userfiles` tables. Each user will get an entry in the userfiles table.

If the dbname is specified, the postgres module will not try to create a new database or a userfiles table.

The postgres code uses the following parameter queries to read, insert, and update a user record:

```
-- read a user
SELECT file FROM userfiles WHERE userid = $1::varchar ;
-- insert a user
INSERT INTO userfiles (file, userid) VALUES ($1::bytea, $2::varchar) ;
-- update a user
UPDATE userfiles SET file = $1::bytea WHERE userid = $2::varchar ;
```

You can override these queries to support alternate schemas. However, the postgres module assumes that any sql used to override the default queries will use the same sequence of arguments. For example, assume you want to store user data in a table named 'randomusertable.' The following parameters can be used to override the default postgres SQL:

```
pguserread=SELECT userdata FROM randomusertable WHERE username=$1::varchar ;
pguserinsert=INSERT INTO randomusertable (userdata,username) VALUES ($1::bytea, $2::varchar)
pguserupdate=UPDATE randomusertable SET userdata = $1::bytea WHERE username = $2::varchar ;
```

Note that the default and override queries use the same arguments in the same order.

Access A PostgreSQL Database (not fileserver)

There are only a couple of functions you need.

^dbinit(xxx)

`xxx` is the parameter string you want to pass PostgreSQL. The default minimal would be something like this:

```
u: (open)
    if (^dbinit( dbname = postgres port = 5432 user = postgres password = mypassword ))
        {db opened}
    else
        {db failed}
```

Once the named database is open, it becomes the current database the server uses until you close it no matter how many volleys later that is. Currently you can only have one database open at a time. By the way, if a call to `^dbinit` fails, the system will both write why to the user log and set it onto the value of `$$db_error`.

`^dbinit(^"null")`

creates a dummy database access, whereby nothing is actually done thereafter but you can pretend to write to the database.

`^dbinit` will fail if the database is already open. If you want it return without failure, having done nothing, just add this at the start of your arguments: `EXISTS`

```
u: (open)
    if (^dbinit(EXISTS dbname = postgres port = 5432 user = postgres password = mypassword )
```

^dbexecute("xxx" 'myfunc)

executes database commands as a blocking call.You may omit the 2nd argument if those commands do not generate answers, otherwise it names a function you have defined whose argument count matches the number of pieces of information coming from the database.

Typically a database query like SELECT will return multiple rows containing multiple columns of values. The number of values per row must match your function, which is called once for each row. The exception is that a one-argument function will accept all the values returned from the query for each row, automatically concatenated with underscore separators.

If any call to your function fails, ^dbexecute stops and fails. When you want to use multiple commands to the database, you must separate them with ; and only answers from the last command are returned (per PostgreSQL standards and behavior).

Note: for any direct calls, the format string ^"xxx"`` is your friend, as it will generally preserve the contents of the string unchanged (except for variable substitutions) to pass to^dbexecute. In addition, don't worry generally about single quotes inside of quoted parameters. ChatScript will automatically double the internal single quotes for you. There will be issues with format strings because ChatScript finds meanings in some things that conflict with postgres command structure. ``` if (^dbexecute(^"INSERT INTO word VALUES '_1');" NULL)) {word added} else {dbexecute failed - $$db_error} ``` The_1has issues because postgre wants quotes around string values, but ChatScript will be trying to decode'_1as the original form of_1. This might require you to do this: ``` $$tmp = '_1 if (^dbexecute(^"INSERT INTO word VALUES '$$tmp');" NULL)) {word added} else {dbexecute failed - $$db_error} ``` Due to conflicts in interpreting‘, *youcan′tusepostgres‘value*$notation inside a quoted string because ChatScript will seek variable values. So, after this, you can read whatever commands PostgreSQL supports because they all funnel through^dbexecute‘.

Note there is a similar issue with text strings coming back from Postgres. They may be multiple words, etc. You should declare arguments to outputmacros that will receive postgres text like this:

```
outputmacro: ^myfunc( ^arg1.HANDLE_QUOTES ^arg2)
```

where ^arg1 will receive a text argument. This causes DBExecute to put double-quotes around the value (and \ on any interior quotes) and then the call to ^myfunc will strip them off after evaluation. You can use :trace sql to see what goes in and what comes out. By the way, if a call to ^dbexecute fails, the system will both write why to the user log and set it onto the value of $$db_error.

PostgreSQL Commands

The basic commands you most likely need are:

Creating Data

```
CREATE DATABASE name - to make a new database
CREATE TABLE weather - to define a new table
(
city varchar(80),
stateAbbr char[2],
temp_lo int,
temp_hi int,
prcp real,
date date
)
```

```
DROP TABLE weather
```

to delete a table

Note that the weather table, while shown all nicely laid out, has to be a single string to `^dbexecute` which will NOT span multiple lines (but could have been created by `^join(...)` whose arguments could have been neatly laid out. Field names are case insensitive (per std).

```
INSERT INTO weather VALUES ('San Francisco', 'CA', 46, 50, 0.25, '1994-11-27')
```

Note that text values must be encased in 'xxx' (per std). Better style is to name the fields and values, omitting any you don't want to supply:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
VALUES ('1994-11-29', 'Hayward', 54, 37)
```

or with lots of data and a file on the same machine as the database server,

```
COPY weather FROM '/home/user/weather.txt'
```

Retrieving Data

```
SELECT * FROM weather
```

retrieves all fields and all rows of table weather (rows in arbitrary order). You can name the fields and order with this:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather
```

assuming the function of yours named expects 5 arguments. And a constrained query like:

```
SELECT * FROM weather WHERE city = 'San Francisco' AND prcp > 0.0
```

And of course there are other things the database can do, including joins and other searches, but you can go read the PostgreSQL manual for that. The above is enough to do many simple things.

Creating A PostgreSQL Database

It's never as easy as you'd hope. Maybe I've hit all the stumbles for you.

Installation on Windows

While in theory you can install a 64-bit version of PostgreSQL on a 64-bit machine, I found there were problems in compiling ChatScript with it, so I use only the 32-bit version, which works on either 32-bit or 64-bit Windows machines. Note that the 32-bit dll that is needed is at the top level of ChatScript, so the system will automatically find it. For reference, the compilation header files and library for loading are in `src/postgre/WIN32`

First, you need to download the 32-bit Windows PostgreSQL. I used version 9.3.3 from this page: http://www.enterprisedb.com/products-services-training/pgdownload#windows win-x86-32. Download that and run it. You can default everything. ChatScript will just work with it.

Installation on Mac

Sorry. Don't have a mac. You have to figure it out yourself.

Installation on Linux

So many choices. You probably have to read docs for your system. I went wrong in so many directions I lost track of whether I was logged in as me, su, sudo, postgres, etc. If you have problems installing, here is my best knowledge... After that go search the internet or documentation, don't come to me to get postgres installed.

On my amazon AMI server I used

```
sudo yum -y install postgresql postgresql-server postgresql-devel postgresql-contrib
```

My 64-bit machine worked fine, my 32-bit machine had issues. Nominally you need to: set a password for your postgres user (the user that runs the db) and initialize the database in PGDATA via

```
service postgresql initdb
```

or

```
su - postgresql
initdb -D /var/lib/pgsql9/data/
```

Confirm there is now stuff installed in `/var/lib/pgsql9`. You may need to edit `/var/lib/pgsql9/pg_hba.conf` to change the local validation from peer to trust - e.g.

```
local all all trust
```

Sometimes it was already trust, and sometimes it was peer. On a 32-bit AMI I tried to lauch the postgres server but it wanted it to have been installed in `/var/lib/pgsql` for some reason. So I had to create a symbolic link to make it happy:

```
ln -s /var/lib/pgsql9 /var/lib/pgsql
#2 launch the server via
service postgresql start
```

or

```
su - postgres - log in as postgres user
pg_ctl start - starts server in background
#3 make it auto restart on reboot
chkconfig postgresql on
```

You can confirm a server is running by typing

```
psql
```

which is a tool to log into the server. You may have to be logged in as postgres. There's a bunch more about setting up login passwords, etc. RTFM.

WARNINGS about characters from a user:

Be careful about allowing the special characters / and . It will likely break your CS database implementation if you are using Postgres. You can always use ^substitute to alter your input.

The Postgres CS database schema definition for your user data field value is bytea and under the latest version of Postgres, the / and are special characters that will not be saved in this field without escaping it. So, if you save the data and these special characters are within it, it will not save the data. You might not notice it, until you realize that your user data is not being saved. And to fix this, it is a rabbit hole for now. Best to just avoid it altogether, unless you have a solution. You will need to modify the Postgres cs code and database definition. I would stay away from using any reference of / and in anything that might make it to your user files. Actually, I would scan all of your code and get rid of anything that uses these characters.