GroovySQL - Groovy SQL Client

GroovySQL is a database client designed primarily for batch SQL submission. It is written in Groovy, which compiles to Java, and is compatible with vendor-provided Java database drivers. In particular, it works with the Denodo JDBC virtual database driver for Denodo version 9 as well as the Snowflake, Postgres, MySQL, and SQLite3 JDBC drivers. Support for other Java-based database drivers can be added, though adding additional database driver support requires some minor changes and rebuilding GroovySQL. Beginning with GroovySQL 2.6 the release is built to support Denodo 9, which should be backward-compatible with Denodo 8, as well as Snowflake, Postgres, MySQL, and SQLite3 databases. For best results Java 17 is the recommended JRE version and only requirement; no other libraries, packages, drivers, etc., are needed.

The Denodo Data Virtualization product does not come with a command line interface (CLI) and is typically accessed through tools like DBeaver interactively and Java applications or EAI tools like DataBricks for application service. GroovySQL provides a CLI for use from shell-based batch implementations.

GroovySQL takes input from any one of standard input, command line, or file. SQL statements normally must be terminated with a semicolon although for command-line input the semicolon is optional. In addition to standard input, for example redirected from a file or pipe, GroovySQL also provides an interactive mode with command line editing and history leveraging the jline3 library.

Deploying GroovySQL

The simplest approach to deploying GroovySQL is to download the latest shell and jar file from the Releases and place them in a location within your execution path (`$PATH`). GroovySQL does not require anything to be installed other than Java. All other requirements are self-contained in the GroovySQL jar file. In particular there is no requirement to install Groovy or any database drivers, GroovySQL will locate all those artifacts in its jar file at runtime. The jar file is not extracted or installed anywhere. Recommendation is to copy both files to `/usr/local/bin`, provided it is in the execution path (`$PATH`).

Building GroovySQL

In the event that you need to make changes to GroovySQL, for example to add support for a new database driver, GroovySQL is built using Gradle and can be modified and rebuilt following these steps:

1. Clone the GroovySQL repository to your machine via any of the usual Git methods
2. Change directory into the cloned repo (cd groovysql)
3. Make changes as needed
4. Execute the Gradle wrapper (./gradlew shadowJar)
5. cp ./build/libs/groovysql-x.y-all.jar /usr/local/bin/groovysql.jar
6. cp ./src/main/bin/groovysql /usr/local/bin/groovysql

7. chown 755 /usr/local/bin/groovysql

The Gradle wrapper (gradlew) will take care of downloading and installing the right version of Gradle, compiling everything, and putting together the final jar file as `./build/libs/groovysql-x.y-all.jar`, where x.y represents the current version of GroovySQL.

Running GroovySQL

GroovySQL is deployed as two files, groovysql (shell wrapper) and groovysql.jar (groovysql and all of its dependencies), to a location in the execution path ($PATH). This allows groovysql to be run from the command line and/or batch.

The groovysql shell wrapper accepts all groovysql options and adds the Java options to work around the Java encapsulation introduced with the Java Modular System (JEP261) in Java 9 and increasingly enforced with newer Java versions. The workaround is needed for the various database drivers which continue to rely on reflection to gain access to internal APIs.

By introducing the groovysql shell wrapper, it provides an opportunity for additional control over the groovysql execution environment.

GroovySQL output formats

Output formats supported are:

```
text
csv
html
xml
json
```

The `--width` option, used with text mode, limits the maximum column width for columns. GroovySQL manages text mode output while industry standard packages (insuring standards compliance) handle output formatting for all the other output formats.

The `--jsonstyle` option, used with json mode, allows selecting between all keys and values being quoted (the default) or all keys and string values being quoted (with numeric values unquoted) as JSON allows both approaches.

GroovySQL supports various JSON styles - “quoted”, “standard”, and “spread”. The default is *quoted* and results in all values being quoted while *standard* does not quote integer and floating point numeric values. The *spread* style is a variant of standard that uses the Groovy spread operator to produce the same output as *standard*. JSON keys are always quoted in compliance with JSON standards.

GroovySQL is designed for use in production batch operations and is careful to avoid overwriting any existing files and will abort in the event of a conflict unless the *append* option is in effect in which case it will append the output to an existing file. Following this approach GroovySQL will not result in data loss though data loss can still occur through other mechanisms, e.g. file redirection, etc.

Command Line Options

groovysql [options]

Either the short or long option can be used, with the < arg> supplied as needed.

The `--config` option allows the database connection information to be stored in a configuration file and supplied as a single option for convenience. It is shorthand for specifying the `--scheme`, `--node`, `--database`, `--user`, and `--password` options.

short	long option	description
-a	`--append`	allows output to append to an existing file
-A	`--authentication <arg>`	specify authentication via secrets management
-c	`--config <arg>`	specifies a database configuration file
-d	`--database <arg>`	specify database name
-f	`--filein <arg>`	specifies input filename containing SQL
-F	`--format <arg>`	specify desired output format
-H	`--csvheaders`	output CSV headers
-h	`--help`	displays this usage information
-i	`--interactive`	run in interactive mode with editing/history
-j	`--jsonstyle <arg>`	specify JSON style of quoted or standard
-n	`--node <arg>`	specify database node/host name including port
-o	`--fileout <arg>`	specify output filename
-p	`--password <arg>`	specify database password
-s	`--scheme <arg>`	database scheme
-S	`--sql <arg>`	specify SQL statement
-t	`--timestamps`	timestamp output
-T	`--testconnect <arg>`	run a connection test
-u	`--user <arg>`	specify database username
-v	`--verbose <arg>`	specify verbose level
-w	`--width <arg>`	limit maximum text column width

-a|--append

Normally GroovySQL will abort and refuse to overwrite an existing file. With the `--append` option GroovySQL will append output to the existing file instead.

-A|--authentication <arg>

Specify authentication using various secrets management systems. Currently supported authentication stores are:

```
Azure KeyVault ............... azure:key-vault-name:key
Google Secret Manager ......... gcp:secret-name
AWS Secrets Manager ........... aws:secret-id
```

-c|--config <arg>

Specifies a database configuration file, a TOML formatted file containing connection details. See Config File Format below. Use of a configuration file is optional and is intended to encapsulate all the connection details for various endpoints, e.g., development, production, etc. The use of config files has the added advantage that the connection details are not visible through system monitoring commands, e.g. ps(1).

-d|--database <arg>

Specifies the database name to connect to. Can also be specified through a config file as dbName (see `--config` option).

-f|--filein <arg>

Specifies the input filename containing the SQL to be executed. Files can additionally contain control record directives, see Directives.

-F|--format <arg>

Specify desired output format. Valid formats are:

```
text
csv
html
xml
json
```

-H|--csvheaders

Specifies column headers should be generated for CSV output. By default CSV output does not include column headers.

-h|--help

Displays this usage information.

`-i|--interactive`

Run in interactive mode with editing and history support provided by the jline3 library.

`-j|--jsonstyle <arg>`

Specify JSON style of “quoted” or “standard” for numeric values. The JSON standard requires that JSON keys be quoted, however it supports numeric values (which are not quoted). With business data processing it is sometimes preferable to quote all values as well, as that avoids any issues with non-compliant numeric values such as “not a number” (NaN), etc. The default style that GroovySQL uses is to quote all values (`--jsonstyle=quoted`).

`-n|--node <arg>`

Specify database node/host name, optionally including a port specification. Node/host names can be any valid TCP/IP specification - numeric IP address, “localhost”, simple hostnames, or fully qualified domain names. Optionally the node/host name can be followed by a colon and port specification, e.g. “localhost:9999”. Can also be specified through a config file as dbHost (see `--config` option).

`-o|--fileout <arg>`

Specify an output filename which can be a relative or full pathname including file extension. If a file with that name already exists then see the `--append` option for controlling the behavior in those cases.

`-p|--password <arg>`

Specify database password. Can also be specified through a config file as dbPassword (see `--config` option).

`-s|--scheme <arg>`

Database scheme, see Schemes. Can also be specified through a config file as dbScheme (see `--config` option).

`-S|--sql <arg>`

Specify a SQL statement to be executed.

`-t|--timestamps`

Timestamp all output messages, resultSets are excluded (of course).

-T|--testconnect <arg>

Run a connection test, arg is N@W. Diagnostic scenarios can benefit from adding the --timestamp option. See Test Connection Capability.

-u|--user <arg>

Specify database username. Can also be specified through a config file as dbUser (see --config option).

-v|--verbose <arg>

Specify verbose level. See Verbose Levels for details.

-w|--width <arg>

Limit maximum text column width. When using the --format text option, this option limits the displayed column width. The default column width is 30. Column width settings can also be controlled through the .width control record directive (see Directives).

Schemes

GroovySQL uses a URL of jdbc:<scheme>://<node>/<database> to connect to the database. Schemes currently supported by GroovySQL are:

```
vdb
denodo
snowflake
postgresql
mysql
sqlite
```

The dbClass for the connection defaults to the standard DriverManager class based on the scheme as follows:

```
vdb ........................... com.denodo.vdp.jdbc.Driver
denodo ........................ com.denodo.vdp.jdbc.Driver
snowflake ..................... net.snowflake.client.jdbc.SnowflakeDriver
postgresql .................... org.postgres.Driver
mysql ......................... com.mysql.cj.jdbc.Driver
sqlite ........................ org.sqlite.JDBC
```

In nonstandard situations the dbClass can be overridden through the Config file, typically for database driver debugging.

Usage

GroovySQL reads SQL input, submits SQL statements to a connected database, and formats the results in one of the output formats selected.

The SQL input can come from a disk file, the command line through the `--sql` option, or from standard input (keyboard or pipe).

In addition to standard input, GroovySQL also supports an interactive line editing mode with retained history using the jline3 library. History is kept in `$HOME/.groovysql_history`.

Directives

GroovySQL also supports a control record capability. Control records allow directives to be processed during the SQL processing.

Directives supported are:

```
.format <type>
.json <style>
.output <filename>
.remove <filename>
.append <true/false>
.width <max text column width>
```

The directives allow various settings to be specified in the SQL input. For example the output file name can be changed between SQL statements, the maximum column width for text output can be changed, etc.

Verbose Levels

GroovySQL supports various verbose levels as well as a timestamp option for runtime operational feedback.

```
level 0 - no messages (except data of course)
level 1 - basic messages (version info, open/close - default)
level 2 - enhanced messages (adds open/close success, query audit)
level 3 - debug messages (adds input trace, text format field adjustments)
level 4 - debug messages (adds system.properties display)
```

While level 1 is the default, setting `--verbose=0` allows GroovySQL to be used in pipelines. For example, piping output to xmlstarlet(1) or jq(1) for postprocessing.

Config File Format

Config files are optional files containing database connection parameters for a given database. They are written in TOML format and support the following parameters:

```
dbUser      - database username
dbPassword  - database password
dbScheme    - JDBC scheme (see Schemes)
dbHost      - TCP network address (hostname:port)
dbName      - database name
dbOptions   - database options added to the database URL (see Examples)
dbClass     - database driver class name (defaults based on dbScheme)
```

Most of these parameters can also be specified through their own option, e.g. `--user` for dbUser, `--node` for dbHost, etc. None of the parameters is required in a Config file. If an option appears in a Config file and also is specified on the command line then the command line setting overrides, leaving the Config file settings as defaults.

The dbClass parameter is entirely optional as the dbScheme will automatically set a default dbClass. Setting dbClass will override the default. There is no command line option to set dbClass.

A common use case is to use a Config file with dbScheme, dbHost, dbName, and dbOptions specified and leverage the `--authentication` option to handle the authentication aspect.

When using simple user/password directly, the use of Config files keeps passwords off of system monitors and is recommended.

Config files are TOML formatted and therefore support comments (#) as is customary.

Note: Config files are currently the only way to provide JDBC URL connection options via the connection string. For example, if you want to set the `queryTimeout` connection parameter you can add

```
dbOptions = "queryTimeout=1500"
```

to a config file specified via the `--config` option.

Test Connection Capability

Additionally, GroovySQL has a connection testing capability. With the `--testconnect <arg>` option GroovySQL will open a database connection, submit a simple query, read the results, discard the results, and close the connection a requested number of times, pausing between each connection for a requested interval. The `--testconnect` argument is of the form `N@W` where N represents the number of connection iterations and W represents the wait interval between connections measured in seconds. If the interval is not specified it defaults to 1 second. This feature is sometimes useful in diagnosing/investigating intermittent database connectivity issues, e.g., `--testconnect 2880@60` would run over a weekend checking once a minute. Used together with the `--timestamp` option this diagnostic approach can help identify intermittent connection issues.

Examples

File: itemdb.config

```
dbUser = "appuser"
dbPassword = "appword"
dbScheme = "denodo"
dbHost = "dbhost.mydomain.com:9999"
dbName = "itemdb"
dbClass = "com.denodo.vdp.jdbc.Driver"
dbOptions.opt1 = "queryTimeout=1500"
dbOptions.opt2 = "chunkTimeout=10"
dbOptions.opt3 = "chunkSize=500"
```

Specifying the dbClass in the config file is optional. GroovySQL uses a default dbClass based on the dbScheme but if a dbClass is provided then it will override the default. The dbOptions values are concatenated together with ampersands (&) and included in the JDBC URL generated to connect to the database. The opt1/opt2/opt3 identifiers serve to avoid key duplication and control ordering (in case that was important). Any identifiers could be used instead, e.g. a/b/c.

This example generates a connection JDBC URL of:

```
jdbc:denodo://dbhost.mydomain.com:9999/itemdb&queryTimeout=1500&
    chunkTimeout=10&chunkSize=500
```

File: item-extract.sql

```
.format json
.append true
.output ../extract/items.json
SELECT * FROM ITEM;
```

Executing **item-extract.sql** with directives controlling output format and location

```
groovysql --config=itemdb.config --filein=item-extract.sql
```

Executing a SQL statement with output to standard output (the terminal)

```
groovysql --config=itemdb.config --sql "SELECT * FROM ITEM"
```

Or reading the statement from a pipe

```
echo "SELECT * FROM ITEM"| groovysql --config=itemdb.config
```

Either of these examples will run the query and send the output to the screen.

File: txn_audit.sql

```
    select
      100                               -- unnamed integer field
    , 200 as "named field 1"            -- field name longer than field width
    , audit_dt as "audit date"          -- date/time field with named field
       with space
    , audit_reason                      -- standard varchar field
    , item_id                           -- short varchar field (6)
    , price                             -- double

    from
        txn_audit
    join
        txn
    on
        txn.txn_id = txn_audit.txn_id
    ;
```

Executing txn_audit.sql to produce txn_audit.xml in XML format (shown using short options and without `--config` option)

```
$ groovysql -s vdb -n dbhost.mydomain.com:9999 -d itemdb -u appuser -p
   appword -f txn_audit.sql -o txn_audit.xml -F xml
```

Sample executions

Sample text execution

```
$ groovysql --config=itemdb.config --sql "SELECT ITEM_ID, DESCRIPTION FROM
    ITEM LIMIT 3"

item_id     description
----------- ------------------------------
     986461 <!Magnotta> - Bel Paese White
     316882 Beer - Fruli,IPA
     887875 Marlbourough Sauv Blanc!
```

Sample CSV execution (note quoting)

```
$ groovysql --config=itemdb.config --sql "SELECT ITEM_ID, DESCRIPTION FROM
    ITEM LIMIT 3" --format csv

item_id,description
986461,<!Magnotta> - Bel Paese White
316882,"Beer - Fruli,IPA"
887875,Marlbourough Sauv Blanc!
```

Sample HTML execution

```
$ groovysql --config=itemdb.config --sql "SELECT ITEM_ID, DESCRIPTION FROM
    ITEM LIMIT 3" --format html

<table>
  <thead>
    <tr>
      <th>item_id</th>
      <th>description</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>986461</td>
      <td>&lt;!Magnotta&gt; - Bel Paese White</td>
    </tr>
    <tr>
      <td>316882</td>
      <td>Beer - Fruli,IPA</td>
    </tr>
    <tr>
      <td>887875</td>
      <td>Marlbourough Sauv Blanc!</td>
    </tr>
  </tbody>
</table>
```

Sample XML execution

```
$ groovysql --config=itemdb.config --sql "SELECT ITEM_ID, DESCRIPTION FROM
    ITEM LIMIT 3" --format xml

<rows>
  <row><!-- row: 1 -->
    <item_id>986461</item_id>
    <description>&lt;!Magnotta&gt; - Bel Paese White</description>
  </row>
  <row><!-- row: 2 -->
    <item_id>316882</item_id>
    <description>Beer - Fruli,IPA</description>
  </row>
  <row><!-- row: 3 -->
    <item_id>887875</item_id>
    <description>Marlbourough Sauv Blanc!</description>
  </row>
</rows>
```

Sample JSON execution (note: all keys and values are quoted)

```
$ groovysql --config=itemdb.config --sql "SELECT ITEM_ID, DESCRIPTION FROM
    ITEM LIMIT 3" --format json

{
    "rows": [
        {
            "item_id": "986461",
            "description": "<!Magnotta> - Bel Paese White"
        },
        {
            "item_id": "316882",
            "description": "Beer - Fruli,IPA"
        },
        {
            "item_id": "887875",
            "description": "Marlbourough Sauv Blanc!"
        }
    ]
}
```

Sample execution without results (CREATE/INSERT/DELETE/UPDATE)

```
$ groovysql --config=itemdb.config --sql "DELETE FROM ITEM WHERE ITEM_ID
   IS NULL"

updated rowcount: 0
```

Sample execution of Denodo CREATE REMOTE TABLE with jq(1) postprocessing

```
$ cat <<EOF | groovysql --config=itemdb.config -v0 -F json | jq '.rows[]."
   stored procedure result"'
    SELECT *
    FROM CREATE_REMOTE_TABLE()
    WHERE remote_table_name = 'SURVEY_TARGET'
    AND replace_base_view_if_exist = TRUE
    AND replace_remote_table_if_exist = TRUE
    AND datasource_database_name = 'venture3'
    AND datasource_name = 'venture3_ds'
    AND datasource_catalog = 'venture3'
    AND datasource_schema = 'public'
    AND base_view_database_name = 'venture3'
    AND base_view_name = 'SURVEY_TARGET'
    AND base_view_folder = '/base views'
    AND query = 'SELECT * FROM VENTURE3.RAW_SURVEY_DATA';
    EOF
"Step 1 of 3: Created remote table 'survey_target' successfully."
"Step 2 of 3: Inserted 2 rows into remote table 'survey_target'."
"Step 3 of 3: Created base view 'SURVEY_TARGET' successfully in the '
   venture3' database."
```