
SQL
CONTEXT

# Contents



# 1 Introduction

Although ConTEXt is a likely candidate for typesetting content that comes from databases it was only in 2011 that I ran into a project where a connection was needed. After all, much document related typesetting happens on files or dedicated storage systems.

Because we run most projects in an infrastructure suitable for TEX, it made sense to add some helper scripts to the ConTEXt core distribution that deal with getting data from (in our case) MySQL databases. That way we can use the already stable infrastructure for installing and updating files that comes with ConTEXt.

As Lua support is nicely integrated in ConTEXt, and as dealing with information from databases involves some kind of programming anyway, there is (at least currently) no TEX interface. The examples shown here work in ConTEXt, but you need to keep in mind that Lua scripts can also use this interface.

*Although this code is under construction the interfaces are unlikely to change, if only because we use it in production.*

# 2 Presets

In order to consult a database you need to provide credentials. You also need to reach the database server, either by using some client program or via a library. More about that later.

Because we don't want to key in all that information again and again, we will collect it in a table. This also permits us to store it in a file and load it on demand. For instance:

```
local presets = {
    database = "test",
    username = "root",
    password = "none",
    host     = "localhost",
    port     = 3306,
}
```

You can put a table in a file `presets.lua` like this:

```
return {
```

```
    database = "test",
    username = "root",
    password = "none",
    host     = "localhost",
    port     = 3306,
}
```

and then load it as follows:

```
local presets = table.load("presets.lua")
```

A `sqlite` database has a much simpler preset. The default suffix of the file is `db`. The other fields are just ignored.

```
return {
    database = "test",
}
```

If you really want, you can use some library to open a connection, execute a query, collect results and close the connection, but here we use just one function that does it all. The presets are used to access the database and the same presets will be used more often it makes sense to keep a connection open as long as possible. That way you can execute much more queries per second, something that makes sense when there are many small ones, as in web related services. A connection is made persistent when the presets have an `id` key, like

```
presets.id = "myproject"
```

# 3 Templates

A query often looks like this:

```
SELECT
    `artist`, `title`
FROM
    `cd`
WHERE
    `artist` = 'archive' ;
```

However, often you want to use the same query for multiple lookups, in which case you can do this:

```
SELECT
    `artist`, `title`
FROM
    `cd`
WHERE
    `artist` = '%artist%' ;
```

In the next section we will see how `%artist%` can be replaced by a more meaningful value. You can a percent sign by entering two in a row: `%%`.

As with any programming language that deals with strings natively, you need a way to escape the characters that fence the string. In sql a field name is fenced by `` and a string by ''. Field names can often be used without `` but you can better play safe.

```
`artist` = 'Chilly Gonzales'
```

Escaping of the ' is simple:

```
`artist` = 'Jasper van''t Hof'
```

When you use templates you often pass a string as variable and you don't want to be bothered with escaping them. In the previous example we used:

```
`artist` = '%artist%'
```

When you expect embedded quotes you can use this:

```
`artist` = '%[artist]%'
```

In this case the variable artist will be escaped. When we reuse a template we store it in a variable:

```
local template = [[
    SELECT
        `artist`, `title`
    FROM
        `cd`
    WHERE
        `artist` = '%artist%' ;
]]
```

# 4 Queries

In order to execute a query you need to pass the previously discussed presets as well as the query itself.

```
local data, keys = utilities.sql.execute {
    presets   = presets,
    template  = template,
    variables = {
        artist = "Porcupine Tree",
    },
}
```

The variables in the presets table can also be passed at the outer level. In fact there are three levels of inheritance: settings, presets and module defaults.

| | |
|---|---|
| presets | a table with values |
| template | a query string |
| templatefile | a file containing a template |
| *resultfile* | a (temporary) file to store the result |
| *queryfile* | a (temporary) file to store a query |
| variables | variables that are subsituted in the template |
| username | used to connect to the database |
| password | used to connect to the database |
| host | the 'machine' where the database server runs on |

| | |
|---|---|
| port | the port where the database server listens to |
| database | the name of the database |

The `resultfile` and `queryfile` parameters are used when a client approach is used. When a library is used all happens in memory.

When the query succeeds two tables are returned: `data` and `keys`. The first is an indexed table where each entry is a hash. So, if we have only one match and that match has only one field, you get something like this:

```
data = {
    {
        key = "value"
    }
}

keys = {
    "key"
}
```

# 5 Converters

All values in the result are strings. Of course we could have provided some automatic type conversion but there are more basetypes in MySQL and some are not even standard sql. Instead the module provides a converter mechanism

```
local converter = utilities.sql.makeconverter {
    { name = "id",      type = "number" },
    { name = "name",    type = "string" },
    { name = "enabled", type = "boolean" },
}
```

You can pass the converter to the execute function:

```
local data, keys = utilities.sql.execute {
    presets   = presets,
    template  = template,
    converter = converter,
    variables = {
        name = "Hans Hagen",
    },
}
```

In addition to numbers, strings and booleans you can also use a function or table:

```
local remap = {
    ["1"] = "info"
    ["2"] = "warning"
    ["3"] = "debug"
    ["4"] = "error"
}
```

```
local converter = utilities.sql.makeconverter {
    { name = "id",     type = "number" },
    { name = "status", type = remap },
}
```

I use this module for managing ConTEXt jobs in web services. In that case we need to store jobtickets and they have some common properties. The definition of the table looks as follows:[1]

```
CREATE TABLE IF NOT EXISTS %basename% (
    `id`        int(11)     NOT NULL AUTO_INCREMENT,
    `token`     varchar(50) NOT NULL,
    `subtoken`  INT(11)     NOT NULL,
    `created`   int(11)     NOT NULL,
    `accessed`  int(11)     NOT NULL,
    `category`  int(11)     NOT NULL,
    `status`    int(11)     NOT NULL,
    `usertoken` varchar(50) NOT NULL,
    `data`      longtext    NOT NULL,
    `comment`   longtext    NOT NULL,

    PRIMARY KEY                    (`id`),
    UNIQUE INDEX `id_unique_index`  (`id` ASC),
    KEY          `token_unique_key` (`token`)
)
DEFAULT CHARSET = utf8 ;
```

We can register a ticket from (for instance) a web service and use an independent watchdog to consult the database for tickets that need to be processed. When the job is finished we register this in the database and the web service can poll for the status.

It's easy to imagine more fields, for instance the way ConTEXt is called, what files to use, what results to expect, what extra data to pass, like style directives, etc. Instead of putting that kind of information in fields we store them in a Lua table, serialize that table, and put that in the data field.

The other way around is that we take this data field and convert it back to Lua. For this you can use a helper:

```
local results = utilities.sql.execute { ... }

for i=1,#results do
    local result = results[i]
    result.data = utilities.sql.deserialize(result.data)
end
```

Much more efficient is to use a converter:

```
local converter = utilities.sql.makeconverter {
    ...
    { name = "data", type = "deserialize" },
    ...
```

[1] The tickets manager is part of the ConTEXt distribution.

```
}
```

This way you don't need to loop over the result and deserialize each data field which not only takes less runtime (often neglectable) but also takes less (intermediate) memory. Of course in some cases it can make sense to postpone the deserialization.

A variant is not to store a serialized data table, but to store a key-value list, like:

```
data = [[key_1="value_1" key_2="value_2"]]
```

Such data fields can be converted with:

```
local converter = utilities.sql.makeconverter {
    ...
    { name = "data", type = utilities.parsers.keq_to_hash },
    ...
}
```

You can imagine more converters like this, and if needed you can use them to preprocess data as well.

| | |
|---|---|
| `"boolean"` | This converts a string into the value `true` or `false`. Valid values for `true` are: `1`, `true`, `yes`, `on` and `t` |
| `"number"` | This one does a straightforward `tonumber` on the value. |
| `function` | The given function is applied to value. |
| `table` | The value is resolved via the given table. |
| `"deserialize"` | The value is deserialized into Lua code. |
| `"key"` | The value is used as key which makes the result table is now hashed instead of indexed. |
| `"entry"` | An entry is added with the given name and optionally with a default value. |

# 6 Typesetting

For good reason a ConTEXt job often involves multiple passes. Although the database related code is quite efficient it can be considered a waste of time and bandwidth to fetch the data several times. For this reason there is another function:

```
local data, keys = utilities.sql.prepare {
    tag = "table-1",
    ...
}

-- do something useful with the result

local data, keys = utilities.sql.prepare {
    tag = "table-2",
    ...
}

-- do something useful with the result
```

The `prepare` alternative stores the result in a file and reuses it in successive runs.

# 7 Methods

Currently we have several methods for accessing a database:

| | |
|---|---|
| client | use the command line tool, pass arguments and use files |
| library | use the standard library (somewhat tricky in LuaTEX as we need to work around bugs) |
| lmxsql | use the library with a Lua based pseudo client (stay in the Lua domain) |
| swiglib | use the (still experimental) library that comes with LuaTEX |

All methods use the same interface (`execute`) and hide the dirty details for the user. All return the data and keys tables and all take care of the proper escaping and parsing.

# 8 Helpers

There are some helper functions and extra modules that will be described when they are stable.

There is an 'extra' option to the `context` command that can be used to produce an overview of a database. You can get more information about this with the command:

```
context --extra=sql-tables --help
```

# 9 Example

The distribution has a few examples, for instance a logger. The following code shows a bit of this (we assume that the swiglib sqlite module is present):

```
require("util-sql")
utilities.sql.setmethod("sqlite")
require("util-sql-loggers")

local loggers = utilities.sql.loggers

local presets = {
 -- method    = "sqlite",
    database  = "loggertest",
    datatable = "loggers",
    id        = "loggers",
}

os.remove("loggertest.db") -- start fresh

local db = loggers.createdb(presets)

loggers.save(db, { -- variant 1: data subtable
    type   = "error",
    action = "process",
    data   = { filename = "test-1", message = "whatever a" }
} )

loggers.save(db, { -- variant 2: flat table
```

```
    type     = "warning",
    action   = "process",
    filename = "test-2",
    message  = "whatever b"
} )

local result = loggers.collect(db, {
    start = {
        day   = 1,
        month = 1,
        year  = 2016,
    },
    stop = {
        day   = 31,
        month = 12,
        year  = 2116,
    },
    limit  = 1000000,
 -- type   = "error",
    action = "process"
})

context.starttabulate { "|||||||" }
for i=1,#result do
    local r = result[i]
    context.NC() context(r.time)
    context.NC() context(r.type)
    context.NC() context(r.action)
    if r.data then
        context.NC() context(r.data.filename)
        context.NC() context(r.data.message)
    else
        context.NC()
        context.NC()
    end
    context.NC() context.NR()
end
context.stoptabulate()

-- local result = loggers.cleanup(db, {
--     before = {
--         day   = 1,
--         month = 1,
--         year  = 2117,
--     },
-- })
```

In this example we typeset the (small) table):

| | | | | |
|---|---|---|---|---|
| 1510074565 | error | process | test-1 | whatever a |
| 1510074565 | warning | process | test-2 | whatever b |

# 10 Colofon

**author** Hans Hagen, PRAGMA ADE, Hasselt NL
**version** November 7, 2017
**website** www.pragma-ade.nl – www.contextgarden.net