Migration Runner Documentation

Release 0.3.4

Andrew Beveridge

Feb 12, 2019

Contents:

CHAPTER 1

SQL Migration Runner

Python script to run SQL migration scripts sequentially from the specified folder, updating latest schema version in the database itself after each migration.

WARNING: this tool was created purely as a solution for the ECS Digital technical test. See PROBLEM for details of the use case and requirements for the task.

It almost certainly should **not** be used for any real-world use case, as mature solutions exist for almost every use case. See "Problem Overview" section of NOTES for further commentary on this topic.

1.1 Requirements

- Python 2.7, or 3.5+
- Existing MySQL or MariaDB database, either running locally or on a remote host.
- Table called `versionTable`, with a single `int(11)` column named "version". See here for schema.
- Directory containing SQL scripts to execute to migrate the database to each version.
  - Each migration / version should have one file.
  - Files should be named to match the pattern `VERSION.brief_description.sql`, where VERSION is an integer representing the database version after executing that script.
- Version numbers should be unique and sequential for consistent results.

1.2 Installation

To install Migration Runner, run this command in your terminal:

```
$ pip install migration_runner
```

This is the preferred method to install Migration Runner, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

For instructions on building from source, see the documentation.

1.3 Usage

Run the `migration_runner` script with `--help` to get usage instructions:

```
$ migration_runner --help

Usage: migration_runner [OPTIONS] SQL_DIRECTORY DB_USER DB_HOST DB_NAME DB_PASSWORD

  A cli tool for executing SQL migrations in sequence.

Options:
  -s, --single-file TEXT  Filename of single SQL script to process.
  -l, --loglevel LVL      Either CRITICAL, ERROR, WARNING, INFO or DEBUG
  -v, --version           Show the version and exit.
  --help                  Show this message and exit.
```

1.4 Examples

1.4.1 Successful usage:

```
$ migration_runner sql-migrations beveradb migration_runner_test.beveradb.tk test_
↪user test_password

2019-02-10 22:16:30,394 - info: Starting with database version: 0
2019-02-10 22:16:30,395 - info: Migrations yet to be processed: 10 (out of 11 in dir)
2019-02-10 22:16:30,721 - info: Successfully upgraded database from version: 0 to 1␣
↪by executing migration in file: 'sql-migrations/001.create_migrations_version_table.
↪sql'
2019-02-10 22:16:31,566 - info: Successfully upgraded database from version: 1 to 2␣
↪by executing migration in file: 'sql-migrations/2.set_current_version_to_1.sql'
2019-02-10 22:16:32,562 - info: Successfully upgraded database from version: 2 to 45␣
↪by executing migration in file: 'sql-migrations/045.createtable.sql'
2019-02-10 22:16:33,236 - info: Successfully upgraded database from version: 45 to 46␣
↪by executing migration in file: 'sql-migrations/046.create_seed_items.sql'
2019-02-10 22:16:34,173 - info: Successfully upgraded database from version: 46 to 48␣
↪by executing migration in file: 'sql-migrations/048.create_rooms.sql'
2019-02-10 22:16:34,849 - info: Successfully upgraded database from version: 48 to 49␣
↪by executing migration in file: 'sql-migrations/049 .rename-object-item.sql'
2019-02-10 22:16:36,258 - info: Successfully upgraded database from version: 49 to 51␣
↪by executing migration in file: 'sql-migrations/051-add-room-relations.sql'
2019-02-10 22:16:37,165 - info: Successfully upgraded database from version: 51 to 52␣
↪by executing migration in file: 'sql-migrations/052.create_customer_order.sql'
2019-02-10 22:16:38,299 - info: Successfully upgraded database from version: 52 to 54␣
↪by executing migration in file: 'sql-migrations/54-fix-customer-address-defaults.sql
↪'
```

(continues on next page)

(continued from previous page)

```
2019-02-10 22:16:39,150 - info: Successfully upgraded database from version: 54 to 55␣
↪by executing migration in file: 'sql-migrations/55exampleorder.sql'
2019-02-10 22:16:39,499 - info: Database version now 55 after processing 10␣
↪migrations. Remaining: 0.
```

1.4.2 Nothing to process:

```
$ migration_runner sql-migrations test_user beveradb.tk migration_runner_test test_
↪password

2019-02-10 22:19:23,252 - info: Starting with database version: 55
2019-02-10 22:19:23,252 - info: Migrations yet to be processed: 0 (out of 11 in dir)
2019-02-10 22:19:23,252 - info: Database version now 55 after processing 0 migrations.
↪ Remaining: 0.
```

1.4.3 Missing argument:

```
$ migration_runner sql-migrations test_user beveradb.tk migration_runner_test

Usage: migration_runner [OPTIONS] SQL_DIRECTORY DB_USER DB_HOST DB_NAME
                        DB_PASSWORD
Try "migration_runner --help" for help.

Error: Missing argument "DB_PASSWORD".
```

1.4.4 Debug output:

```
$ migration_runner -l DEBUG sql-migrations test_user beveradb.tk migration_runner_
↪test fake_password

2019-02-10 22:21:48,074 - debug: CLI execution start
2019-02-10 22:21:48,075 - debug: Migrations found: 11
2019-02-10 22:21:48,075 - debug: Connecting to database with details: user=test_user,␣
↪password=fake_password, host=beveradb.tk, db=migration_runner_test
2019-02-10 22:20:37,731 - error: Database connection error: 1045 (28000): Access␣
↪denied for user 'test_user' (using password: YES)
```

CHAPTER 2

Installation

2.1 Stable release

To install Migration Runner, run this command in your terminal:

```
$ pip install migration_runner
```

This is the preferred method to install Migration Runner, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Migration Runner can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/beveradb/migration_runner
```

Or download the tarball:

```
$ curl  -OL https://github.com/beveradb/migration_runner/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

3.1 Requirements

- Python 2.7, or 3.5+
- Existing MySQL or MariaDB database, either running locally or on a remote host.
- Table called `versionTable`, with a single `int(11)` column named "version". See here for schema.
- Directory containing SQL scripts to execute to migrate the database to each version.
  - Each migration / version should have one file.
  - Files should be named to match the pattern `VERSION.brief_description.sql`, where VERSION is an integer representing the database version after executing that script.
- Version numbers should be unique and sequential for consistent results.

3.2 Basic usage

Run all SQL scripts in the specified directory, skipping any with a version number (numeric filename prefix) lower than the existing version stored in `versionTable`:

```
$ migration_runner ./folder-of-sql-scripts db_user db_hostname db_name db_password
```

3.3 Options

Run the `migration_runner` script with `--help` to get usage instructions:

```
$ migration_runner --help

Usage: migration_runner [OPTIONS] SQL_DIRECTORY DB_USER DB_HOST DB_NAME DB_PASSWORD

  A cli tool for executing SQL migrations in sequence.

Options:
  -s, --single-file TEXT  Filename of single SQL script to process.
  -l, --loglevel LVL      Either CRITICAL, ERROR, WARNING, INFO or DEBUG
  -v, --version           Show the version and exit.
  --help                  Show this message and exit.
```

3.4 Examples

3.4.1 Successful usage:

```
$ migration_runner sql-migrations test_user beveradb.tk test_db test_password

2019-02-12 13:37:29 - info: Starting with database version: 0
2019-02-12 13:37:29 - info: Migrations yet to be processed: 10 (out of 11 in dir)
2019-02-12 13:37:29 - info: Upgraded DB version from 0 to 1 by executing file: 'sql-
↪migrations/001.create_migrations_version_table.sql'
2019-02-12 13:37:30 - info: Upgraded DB version from 0 to 2 by executing file: 'sql-
↪migrations/2.set_current_version_to_1.sql'
2019-02-12 13:37:31 - info: Upgraded DB version from 2 to 45 by executing file: 'sql-
↪migrations/045.createtable.sql'
2019-02-12 13:37:31 - info: Upgraded DB version from 45 to 46 by executing file: 'sql-
↪migrations/046.create_seed_items.sql'
2019-02-12 13:37:32 - info: Upgraded DB version from 46 to 48 by executing file: 'sql-
↪migrations/048.create_rooms.sql'
2019-02-12 13:37:33 - info: Upgraded DB version from 48 to 49 by executing file: 'sql-
↪migrations/049 .rename-object-item.sql'
2019-02-12 13:37:34 - info: Upgraded DB version from 49 to 51 by executing file: 'sql-
↪migrations/051-add-room-relations.sql'
2019-02-12 13:37:35 - info: Upgraded DB version from 51 to 52 by executing file: 'sql-
↪migrations/052.create_customer_order.sql'
2019-02-12 13:37:36 - info: Upgraded DB version from 52 to 54 by executing file: 'sql-
↪migrations/54-fix-customer-address-defaults.sql'
2019-02-12 13:37:37 - info: Upgraded DB version from 54 to 55 by executing file: 'sql-
↪migrations/55exampleorder.sql'
2019-02-12 13:37:37 - info: Database version now 55 after processing 10 migrations.␣
↪Remaining: 0.
```

3.4.2 Nothing to process:

```
$ migration_runner sql-migrations test_user beveradb.tk test_db test_password

2019-02-10 22:19:23 - info: Starting with database version: 55
2019-02-10 22:19:23 - info: Migrations yet to be processed: 0 (out of 11 in dir)
2019-02-10 22:19:23 - info: Database version now 55 after processing 0 migrations.␣
↪Remaining: 0.
```

3.4.3 Missing argument:

```
$ migration_runner sql-migrations test_user beveradb.tk test_db

Usage: migration_runner [OPTIONS] SQL_DIRECTORY DB_USER DB_HOST DB_NAME
                        DB_PASSWORD
Try "migration_runner --help" for help.

Error: Missing argument "DB_PASSWORD".
```

3.4.4 Debug output:

```
$ migration_runner -l DEBUG sql-migrations test_user beveradb.tk test_db test_password

2019-02-10 22:21:48 - debug: CLI execution start
2019-02-10 22:21:48 - debug: Migrations found: 11
2019-02-10 22:21:48 - debug: Connecting to database with details: user=test_user,␣
↪password=fake_password, host=beveradb.tk, db=migration_runner_test
2019-02-10 22:20:37 - error: Database connection error: 1045 (28000): Access denied␣
↪for user 'test_user' (using password: YES)
```

CHAPTER 4

API Documentation

Information on specific functions, classes, and methods.

4.1 migration_runner package

4.1.1 Submodules

4.1.2 migration_runner.cli module

Console script for migration_runner.

migration_runner.cli.**custom_format**(*self*, *record*)

4.1.3 migration_runner.controller module

class migration_runner.controller.**Controller**(*logger=None*)

Bases: object

process_migrations(*db_params*, *db_version*, *unprocessed_migrations*)

process_migrations_in_directory(*db_params*, *sql_directory*)

process_single_file(*db_params*, *single_file*)

update_current_version(*db_params*, *new_version*)

4.1.4 migration_runner.database_tools module

class migration_runner.database_tools.**DatabaseTools**(*logger=None*)

Bases: object

apply_migration(*db_params*, *sql_filename*)

connect_database(*db_params*)

fetch_current_version(*db_params*)

4.1.5 migration_runner.helpers module

class migration_runner.helpers.**Helpers**(*logger=None*)

Bases: object

append_migration(*migrations*, *filename*)

static extract_sequence_num(*filename*)

find_migrations(*sql_directory*)

static get_unprocessed_migrations(*db_version*, *migrations*)

populate_migrations(*sql_directory*)

static sort_migrations(*migrations*)

4.1.6 Module contents

Top-level package for Migration Runner.

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/beveradb/migration_runner/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

Migration Runner could always use more documentation, whether as part of the official Migration Runner docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/beveradb/migration_runner/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up migration_runner for local development.

1. Fork the migration_runner repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/migration_runner.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv migration_runner
$ cd migration_runner/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 migration_runner tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/beveradb/migration_runner/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_migration_runner
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Original Use Case

6.1 Context

As mentioned in the README, this tool was originally created purely as a solution for the ECS Digital technical test.

Below are the details of said technical test, as provided by ECS Digital (formerly Forest Technologies).

6.1.1 Use Case:

- A database upgrade requires the execution of numbered scripts stored in a specified folder, e.g. SQL scripts such as `045.createtable.sql`.
- There may be gaps in the numbering and there isn't always a . (dot) after the number.
- The database upgrade is based on looking up the current version in the database and comparing this number to the numbers in the script names.
- If the version number from the db matches the highest number from the script then nothing is executed.
- If the number from the db is lower than the highest number from the scripts, then all scripts that contain a higher number than the db will be executed against the database.
- In addition, the database version table is updated after the install with the highest number.

6.1.2 Requirements:

- Supported Languages: Bash, Python2.7, PHP, Shell, Ruby, Powershell
  - No other languages will be accepted
- The table where the version is stored is called 'versionTable', and the row with the version is 'version'.
  - This table contains only one column with the actual version.
- You will have to use a MySQL database.

- The information about the database and the directory will be passed through arguments, following this format:

```
<directory with .sql scripts> <username for the DB> <DB host> <DB name> <DB␣
↪password>
```

6.1.3 Task:

How would you implement this in order to create an automated solution to the above requirements?

Please send us your script(s) and any associated notes for our review and we will come back to you asap regarding next steps.

Important: the documentation you compile is as important as the quality of the script.

CHAPTER 7

Author Notes

This is where I'll be documenting my own thought process and technical choices while implementing this solution, with review and conclusion once complete.

7.1 Problem Overview

At first glance, this looks like we're implementing a home-baked database migrations solution. This is reinventing the wheel somewhat, as there are a wide variety of existing, mature options for handling database migrations which will be far more robust and future-proof (typically built into ORMs, e.g. SQLAlchemy in Python, Doctrine in PHP or Active Record in Ruby). However, since this is a test with fixed requirements, let's pretend these don't exist and we're inventing the concept of migrations for the first time.

The approach required by the problem definition is to use the intuitive (but naive) approach of running SQL scripts in sequence, keeping track of current database state using a simple version number stored in the database itself to track the most recent script run.

7.2 Approach

7.2.1 Fictional scenario

To make it easier to make sensible assumptions for this use case, I decided to imagine this was being implemented by a developer who was part of a small and inexperienced team working on a web application for a furniture store.

Ths fictional dev team aren't sure of all the requirements, yet as the store owner is still deciding various things! As such, they're pretty much designing the database schema and application architecture on the fly - a situation where handling migrations well is essential!

7.2.2 Language Choice

All of the languages allowed by the test requirements are more than capable for this purpose. However, for a standalone script intended for execution as part of the software development lifecycle, I wanted something both:

1. Feature-rich, so I'm not reinventing the wheel too much and implementing my own SQL client or pattern matching from scratch.
2. Portable, so the script can be useful to devs working from different operating systems.

Now, I've written my fair share of Bash scripts and they have their place, but shell scripting in general doesn't actually meet either of these requirements, so these should be avoided for anything more complex than system-specific automation tasks.

Between the three fully featured languages, Python has a clear advantage in portability, as it is available out of the box in all major Linux distributions and on macOS. It is also fairly common to see Python scripts in SDLC tooling alongside a codebase in another language, whereas choosing PHP or Ruby would be unusual unless the application itself was built in one of those languages.

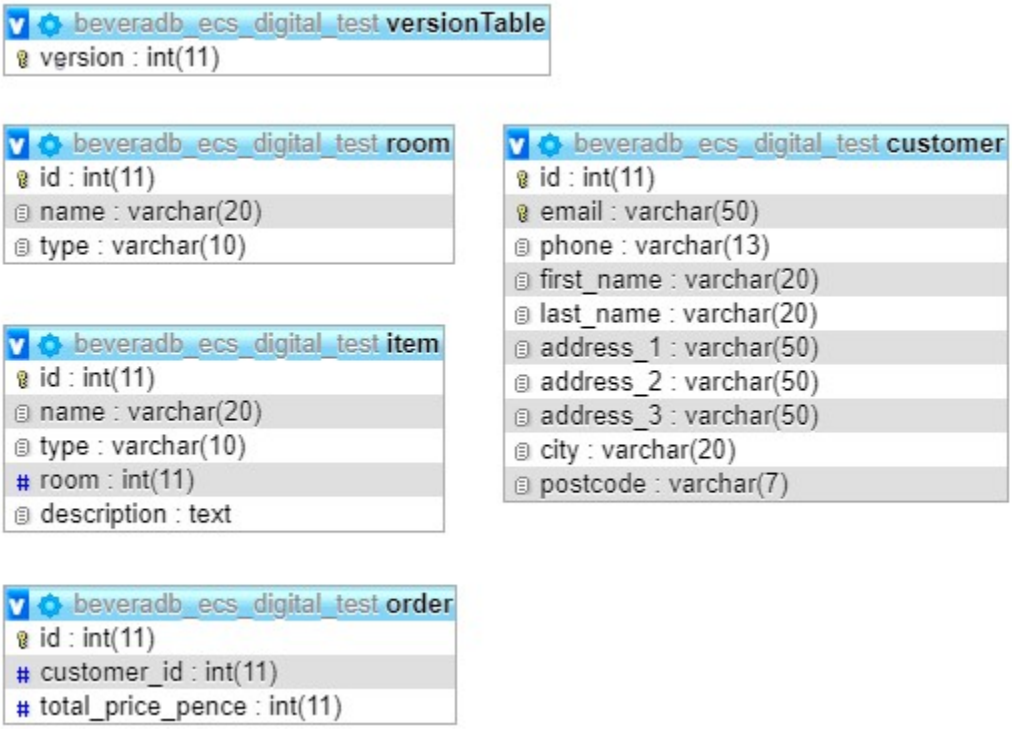
Note: The requirements explicitly reference Python 2.7, which is very bad practice now as the entire community is pushing to migrate away from the 2.x branch since it is being retired in less than 10 months (January 2020). I've written this solution in Python 2.7 compatible code, but anyone still using it really needs to update now!

7.3 Implementation

7.3.1 MySQL database, example SQL scripts

I set up a MySQL database on a remote host to run these on (ignoring SDLC best practices - for now this fictional dev team are deploying changes directly to production!), and began creating SQL scripts to support the fictional scenario described above.

The scripts were then tweaked until there was a working set of migrations simulating the progressive design and creation of a handful of semi-realistic tables:



7.3.2 Filename inconsistency

To meet the test requirements, I introduced a bit of human error / inconsistency in the filenames, as if these were being created by hand by careless developers. They technically all start with a number, but these aren't sequential (which could easily happen if the devs were attempting to collaborate in branches without good communication or any other tooling in their development process). The filename patterns aren't consistent either, with a mix of hyphens, underscores, dots and spaces separating parts.

7.3.3 CLI Interface

As this script is designed to be executed from the command line, I wanted it to have a robust and user-friendly interface. There's a mature and popular Python library called Click which I've used before, so I put together some boilerplate code and tweaked it a bit to expect the parameters defined for this test. There's a "help" option to show usage instructions for the script, and the user is shown an informative error message if they don't enter the correct number of arguments.

7.3.4 Logging

I'm a firm believer in robust logging for even simple scripts, so one of the first things I'll do before implementing functionality is ensure it is easy to access well-formatted debug log entries on demand. As such, I added the click_log library, configured it and added my own custom formatter method to add timestamps to each line.

7.3.5 Finding SQL scripts to execute

To find the SQL scripts in the provided directory, we use a combination of standard library methods to iterate through all files ending with the expected ".sql" suffix and apply a regex pattern to extract the sequence number from the start of the filename (regardless of what character is after the number - we just match all numbers at the start of the string). We then cast the sequence number to an integer value to make any leading zeros irrelevant, and sort the list of SQL scripts by the sequence number to ensure they are processed in the correct order.

7.3.6 Connection to MySQL database

We connect to the MySQL database with parameters specified on the command line, using the official 'mysql-connector-python' library. Oddly, this library wasn't able to function until the legacy 'MySQL-python' library was also installed.

Despite this connector library being the officially endorsed method of interacting with a MySQL/MariaDB database, there are several drawbacks to using this library. For one, it is reliant on a native C++ client implementation, so may require certain packages to be installed on the machine running the script, decreasing portability significantly. Additionally, I encountered a segfault when using this library from the setuptools entry point. I've contributed to the bug report, but this doesn't give me a huge amount of confidence in the library.

Unfortunately, all of the alternatives I could find (e.g. PyMySQL, a pure python library) don't support executing multiple statements at once, meaning I would be required to implement a parser of my own to split statements in the input files and process them individually. This sounds like a highly error-prone thing to implement and would likely have many edge cases, so I wanted to avoid this if possible.

Initially I attempted to re-use a single connection for the lifetime of the script, but found the connection was broken after certain operations were executed, so switched to creating and closing the connection for each migration.

7.3.7 Error handling

As the purpose of this script is to apply schema changes to a database, and in our hypothetical scenario it is likely to be used directly on the production database, I wanted it to fail quickly - if anything goes wrong, stop. For example, you don't want to end up accidentally running one migration before the previous in the sequence, as this could have unintentionally destructive effects on production data. As such; if the database connection fails at any point, we error and exit. If a migration fails to run, we error and exit. If any exceptions are raised at any point, we error and exit, displaying a full and informative error message in the console output.

7.3.8 Identifying unprocessed migrations

Once we have a list of all of the migrations in the SQL scripts folder, we fetch the current version number from the database and filter the list of migrations to only process those with a higher version number. If the 'versionTable' database doesn't exist, or doesn't contain a version row, we set the current version to 0 and assume we're starting out with an empty database.

7.3.9 Applying migrations

There is actually very little complexity here at all - we simply open a connection to the database, get a cursor reference, and read in the whole SQL file into a single execution call. The only thing worth noting here is that in order to execute multiple statements in a single call, the "multi" flag must be passed to the execution call. This is a feature only added to the Python MySQL connector a few years ago though, and there may be some situations where this method of executing all statements in an SQL script doesn't work. There are a variety of alternate methods to choose from though, each with their own pros and cons.

7.3.10 Functional testing

The first thing I did as part of this project was build some example SQL scripts to be executed by this tool, so essentially the functionality was being tested multiple times at every stage of development. To make this more convenient, I added a feature to allow executing an arbitrary specified SQL file, and wrote some SQL to drop all tables which may have been created by subsequent script executions. I kept this an an additional SQL script in the migrations folder with index 0, as in the hypothetical scenario it could be useful for tearing down/recreating functional test environments if the dev team ever decided to create a proper testing pipeline.

CHAPTER 8

Credits

8.1 Development Lead

- Andrew Beveridge andrew@beveridge.uk

8.2 Contributors

None yet. Why not be the first?

CHAPTER 9

History

9.1 0.3.4 (2019-02-12)

- Fixed failing test due to list comparison depending on FS order

9.2 0.3.3 (2019-02-12)

- Added workaround for segfault in mysql-connector library to fix Travis. See bug report https://bugs.mysql.com/bug.php?id=89889 for details.

9.3 0.3.2 (2019-02-12)

- Improved test speed by tweaking Tox config.

9.4 0.3.1 (2019-02-12)

- Significantly refactored and improved docs

9.5 0.3.0 (2019-02-11)

- First release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index