MIGRATIONS IN MYSQL

1. Introduction

Everybody who has worked with databases, will be very familiar with the annoyances of database migrations: the fact that you can't edit your database like you would a source code file. You need to write a series of SQL commands which change your database, and once executed, there is no going back. On top of that, there are very few databases which have support for transactional DDL statements. So, when you have a long migration that fails half way, you will have a very big inconsistent mess on your hands.

I normally work with PostgreSQL, which supports transactional DDL statements, but recently I also had to start working with MySQL. Transactional DDL statements is something I had kind of taken for granted, and when I was forced to work MySQL, I decided I needed to make a migration infrastructure that at least allows me to safely try to load long migrations, even if they can fail. I mean, how can you debug otherwise? So, I sat down with my buddy Bigsmoke and we devised something that works reasonably well. We used PHP to write the script, because we already had a lot of PHP code anyway.

The code we made is very much catered to our situation, so you will have to make modifications to it before you can use it. My main goal with this article is to show the general idea, and give you something to work with.

---

2. Theory

We looked at how migrations were handled in Ruby on Rails, for inspiration. Actually, I should say, we looked how we had already modified Ruby on Rails' migration infrastructure... Every migration is stored in it's own numbered *.mysql* file. The database then has a table which keeps track of which migrations have already been executed. All the migration logic needs to do, is find out what the latest executed migration is, and then execute all the new migrations it finds. This is basically what Ruby on Rails does, so you might be wondering what exactly it is we modified. Ruby on Rails uses *.rb* files for the migration commands. We prefer altering our database in SQL, as opposed to obscure Ruby (on Rails) syntax, and therefore all our *.rb* files ever contained, was a directive to execute a *.pgsql* file. We wrote our own migration infrastructure which does away with those*.rb* files, and also with the ability to make a down migration. Database migrations can't be undone. Period. I mean, how are you going to undo "`DROP TABLE customers`"...?

Sounds about complete, doesn't it? Oh, wait, we don't have transactional DDL statements. So, Before we actually run the migration, we need to make sure that it will actually run. The way we chose to do it, is dump the development database, create a new empty temporary database and load the development dump in it. Then, we load the migration into this temporary database. If it succeeds, we run it on the development database itself. Also, we decided it would be a good idea to make a dump of the database, be it production or development, before each migration is run, and commit it to the subversion repository. MySQL dumps are made so that they can be run on a live database, and replace any existing objects. Of course, when you have a migration that failed half way, this will not clean up the mess, because for example, the migration may have created some

tables, and loading the dump will not delete anything. And because it's very likely you won't have permission to drop and create a database on your production server, you will have to clean it manually. But at least you have the dump...

Because we, and probably most people, don't have permission to create a database on the production server, testing the migration on a copy of the production database is not going to be possible. Loading the production dump into a database on our development server is troublesome at best, because of the difference in the "DEFINER" and permission information on the servers used for development and production. The script we made therefore only runs this test when in development mode. However, when removing this condition, it should also work for the production server, provided you have permission to create databases on it. See the comments in the script what you need to edit to make this possible.

As a final note on theory; we would love to have made a dependency based migration system, so you don't have to ask around your development team like: "is anybody working on a migration? Could you finish/wait with it, I need to make one too", but instead just make it dependant on the latest executed migration, but that would have been too much work...

---

3. Implementation

First you need to add a table to the database which is going to store your migration history. This can be done with this very simple script:

```
CREATE TABLE migration_history
(
  migration_nr INTEGER NOT NULL PRIMARY KEY,
```

```
  migrated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL
) ENGINE=INNODB;
```

Be sure to choose the correct table engine (InnoDB in my example).

The main script is migrate.php. It needs three other files: mymysql-functions.php for easy database access, db-connection.php for making the database connection and shell-output-helpers.php for outputting in nice colors. The *migrate.php* script looks as follows:

```
<?php

////////////////////////////////////////////////////////////////////////
//
// Copyright 2007, 2008, Wiebe Cazemier (http://www.halfgaar.net), Rowan Rodrik
van der Molen
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
//
////////////////////////////////////////////////////////////////////////

// FIXME: test if safe mode is active. It won't run in safe mode.

require_once(dirname(__FILE__).'/db-connection.php');
require_once(dirname(__FILE__).'/mymysql-functions.php');
require_once(dirname(__FILE__).'/shell-output-helpers.php');
```

```
echo_yellow("Using environment ".WEB_ENV);

echo "\n";

$migration_test_db = "migration_test";

$previous_version = intval(mymysql_select_value("SELECT MAX(migration_nr) FROM migration_history"));

chdir(dirname(__FILE__));
$handle = opendir(dirname(__FILE__));
if ( ! $handle ) trigger_error("Opening current dir failed.", E_USER_ERROR);

$migration_files = array();

while (false !== ($file = readdir($handle))) {
  if (preg_match("/([0-9]+)_([_a-zA-Z0-9]*).mysql/", $file) )
  {
    $migration_files[] = $file;
  }
}

closedir($handle);

sort(&$migration_files);

foreach($migration_files as $filename)
{
  $current_version = intval(mymysql_select_value("SELECT MAX(migration_nr) FROM migration_history"));

  $matches = array();
  preg_match("/([0-9]+)_([_a-zA-Z0-9]*).mysql$/", $filename, &$matches);
  $file_version = intval($matches[1]); // file_version is the number of the current migration file

  if ( $file_version == $previous_version )
```

```
    die_red("Duplicate versioned migration found, fool...");

  if ( $file_version > $current_version )
  {
    echo_yellow("Making dump of revsion $previous_version.");

    $dump_status = null;
    $dump_command = sprintf("mysqldump --routines --user=%s --password=%s --host=%s %s > pre_migration_%s_dump.mysql", MYMYSQL_USER, MYMYSQL_PASSWORD, MYMYSQL_HOST, MYMYSQL_DB, WEB_ENV);
    echo_green($dump_command);
    system($dump_command, &$dump_status);
    if ($dump_status != 0) die_red("Dumping pre-migration dump file failed abysmally!");

    echo "\n";

    // If you have DB create permissions on your production server, it should be possible to remove this condition. The generated commands should adjust accordingly. But, test it to be sure.
    if ( WEB_ENV != "production")
    {
      echo_yellow("Testing the migration on a test database. Note: this test is only done in development, because you often can't create databases on your production server because of lack of permissions. So, always test your migrations on your development DB first.");

      // drop a possibly existing test db
      $drop_test_db_query = "DROP DATABASE IF EXISTS $migration_test_db;";
      echo_green("$drop_test_db_query");
      if (! mysql_query($drop_test_db_query)) die_red("Could not delete $migration_test_db db.");

      // create a new test db, so you can be sure it's empty
      $create_test_db_query = "CREATE DATABASE $migration_test_db;";
      echo_green("$create_test_db_query");
      if (! mysql_query($create_test_db_query)) die_red("Could not create $migration_test_db db.");
```

```
        // load the dump of development into the test db
        $load_into_test_command = sprintf("mysql --user=%s --password=%s --host=%s $migration_test_db < pre_migration_%s_dump.mysql", MYMYSQL_USER, MYMYSQL_PASSWORD, MYMYSQL_HOST, WEB_ENV);
        echo_green("$load_into_test_command");
        $load_into_test_db_status = null;
        system($load_into_test_command, &$load_into_test_db_status);
        if ($load_into_test_db_status != 0) die_red("Loading dump into test DB failed.");

        // Run the migration on the test db
        $test_migration_status = null;
        $migration_test_command = sprintf("mysql --user=%s --password=%s --host=%s $migration_test_db < $filename", MYMYSQL_USER, MYMYSQL_PASSWORD, MYMYSQL_HOST);
        echo_green($migration_test_command);
        system($migration_test_command, &$test_migration_status);
        if ($test_migration_status != 0 ) die_red("Migration $file_version has errors in it. Aborting.");

        // At this point, we know the test succeeded, so we can continue safely.
        echo_yellow("Testing of migration $file_version successful, there don't appear to be any errors. Continueing...");
    }

    echo "\n";

    echo_yellow("Committing pre-migration dump of ".WEB_ENV." version $previous_version.");
    $svn_commit_command = sprintf("svn commit -m 'Updated pre-migration dump of %s to $previous_version' pre_migration_%s_dump.mysql", WEB_ENV, WEB_ENV);
    echo_green("$svn_commit_command");
    $svn_commit_result = null;
    system($svn_commit_command, &$svn_commit_result);
    if ($svn_commit_result != 0) die_red("Commiting backup dump into svn failed.");

    echo "\n";
```

```
    // do stuff here that executes the migration.
    echo_yellow("Migrating from version $previous_version to $file_version.");
    $migration_status = null;
    $migration_command = sprintf("mysql --user=%s --password=%s --host=%s %s < $filename", MYMYSQL_USER, MYMYSQL_PASSWORD, MYMYSQL_HOST, MYMYSQL_DB);
    echo_green($migration_command);
    system($migration_command, &$migration_status);

    // If loading the migration failed.
    if ($migration_status != 0)
    {
      echo "\n";

      die_red("Migration failed! You now have a possible inconsistent database, because it did pass the initial test, but failed when the migration was actually run. You need to find out what statements of the migration did work, and bring it back to the state of the previous migration. For the record, migration $file_version is the one that failed. \n");

    }

    echo "\n";

    // Insert the version number of the migration into the migration_history table.
    echo_yellow("Inserting version $file_version into migration_history table");
    mymysql_insert("migration_history", array("migration_nr" => $file_version));

    echo "\n\n\n";
  }

  $previous_version = $file_version;
}

?>
```

The migration files should start with one or more digit, followed by an underscore and end with .mysql. Expressed in a regular expression, they should be named "*/[0-9]+_[_a-zA-Z0-9]*.mysql$/*". So, for example, "*005_add_orders_table.mysql*". They should be placed in the same directory as the *migrate.php* script. The way the *require* statements are written now, the supporting *.php* files need to be in the same directory as well, but you are welcome to change it, of course. Personally, I have them somewhere else, but for the purpose of demonstrating this script, I made it as straight forward as possible.

One of the things that will likely be different in your setup, is the prescence of the WEB_ENV environment variable. We use that to set the database connection properties to either our development or production database. You will have to find a way to detect your environment yourself.

---

4. Running the script

To run the script, simply execute "`php migrate.php`". When executed this way, it will run on the development environment (see the db-connection.php file why this is so). To run it on production, run it like "`WEB_ENV=production php migrate.php`".

---

5. Final words

As I said in the introduction, this code is catered to our situation, so you will have to change and check it, before you actually use it. A good way would be to comment out all the system commands, and leave only the echo's, so you know

what commands will be run. Or, you can only use this article for inspiration to write your own migration routines.

Source : http://www.halfgaar.net/mysql-migrations