MyBatis 3 – Schema Migration System

User Guide

⇨ ***NOTE:*** *This guide is not about migrating from older versions of MyBatis. It is a manual about a tool that will change the way you manage changes to your database.*

Introduction

Evolving databases has been one of the major challenges for software development. Often times, regardless of our software development methodology, the database follows a different change management process. Despite our best efforts, few tools and practices have been able to change that. The tools of the past have been GUI centric, proprietary for a particular database and/or carried a steep license cost. Yet, at the end of the day they suffered from the same challenges.

Recently, a few tools arrived and changed all of that. They did so by embracing simplicity and a few simple rules for database evolution to follow. A couple of good examples are Rails Migrations and dbdeploy. Both tools are similar in purpose, but quite different in implementation. The MyBatis Schema Migration System draws from both and seeks to be the best migration tool of its kind.

Goals

To achieve a good database change management practice, we need to identify a few key goals.

Thus, the MyBatis Schema Migration System (or MyBatis Migrations for short) seeks to:

- Work with any database, new or existing
- Leverage the source control system (e.g. Subversion)
- Enable concurrent developers or teams to work independently
- Allow conflicts very visible and easily manageable
- Allow for forward and backward migration (evolve, devolve respectively)
- Make the current status of the database easily accessible and comprehensible
- Enable migrations despite access privileges or bureaucracy
- Work with any methodology
- Encourages good, consistent practices

Installation

Installation is simply a matter of unzipping the package to a directory of your choosing. There are generally two ways to work with this tool:

- Unzip it to a central location. Add MIGRATIONS_HOME to your environment variables, and add MIGRATIONS_HOME to your path. This is a common option, popular among similar tools like Ant or Maven.

- Unzip it into a directory in your workspace for a project that you're currently working on, thus keeping all of the dependencies and the tool version isolated within the project. It's a small framework, and this option has the advantage of portability and zero setup for developers new to the project.

What's Included?

The MyBatis Migrations package is small and simple. The following is the contents of the unzipped package:

```
./lib/mybatis-3-core-3.0.0.188.jar
./migrate
.migrate.cmd
```

The single MyBatis JAR file is the only dependency that MyBatis Migrations has. The two script files do the same thing, but as you can see, one is for *nix shells and the other is for Windows (Note: cygwin users should still call the .cmd version).

The 'migrate' Command

The entire Migrations system can be accessed through this one simple command. You can access the built-in help by typing: `migrate --help`

Calling the migrate command with no options or invalid options also produces the help message. Here's the output of the help command:

```
Usage: migrate command [parameter] [--path=<directory>] [--env=<environment>]
               [--template=<path to template>]

--path=<directory>   Path to repository.  Default current working directory.
--env=<environment>  Environment to configure. Default environment is 'development'.
--force              Forces script to continue even if SQL errors are encountered.
--help               Displays this usage message.
--trace              Shows additional error details (if any).
--template           (Optional) Specify template to be used with 'new'command

Commands:
  init               Creates (if necessary) and initializes a migration path.
  bootstrap          Runs the bootstrap SQL script (see scripts/bootstrap.sql for more).
  new <description>  Creates a new migration with the provided description.
  up                 Run all unapplied migrations.
  down               Undoes the last migration applied to the database.
  version <version>  Migrates the database up or down to the specified version.
  pending            Force executes pending migrations out of order (not recommended).
  status             Prints the changelog from the database if the changelog table exists.
  script <v1> <v2>   Generates a delta migration script from version v1 to v2 (undo if v1 > v2).
```

We'll go through each of these commands in detail, but first, let's talk about lifecycle.

The MyBatis Migrations Lifecycle

Database change management is difficult at the best of times, so to make the situation better, it's important to have a good database evolution strategy. That employed by MyBatis Migrations targets a few key goals:

- Consistent – The schema should be predictable on every machine it's created on.
- Repeatable – The schema can be destroyed and recreated a predictable way.

- Reversible – Changes should be able to be rolled back, or undone.
- Versioned – The version of the schema should be identifiable (via query or tool).
- Auditable – The schema evolution should be auditable and the changes to it logged.
- Automated – The evolution (or devolution) of the schema should be fully automated.
- Serial – The evolution in the database should never branch or evolve conditionally.
- Immutable Changes – No past applied alter or evolution of the database should be modified, instead a new change should be created.
- Concurrently Modifiable – The schema should be safely modifiable by multiple people or teams in a way that encourages teamwork, communication and easy identification of conflicts, without depending on text comparisons (diff) or any particular source control feature (conflicts), but should work very well with source control systems.

Thus, the lifecycle of a schema managed with MyBatis Migrations is as follows:

1. Initialize Repository
2. Bootstrap database schema
3. Create a new migration (or many migrations)
4. Apply migrations

Optional steps include:

- Revert migrations if necessary to resolve conflicts
- Apply pending migrations out of order if it's safe to do so
- Generate migration scripts to be run "offline" in environments that are beyond your control
- Get the status of the system at any time

The following command discussions will provide more detail about how this lifecycle works.

init

The **init** command initializes a new 'migration path', also called a 'repository' (of migration scripts). Regardless of whether your working with a new database or an existing one, you'll run init to create the workspace in which you'll place everything you need to manage database change. Running this command will create the directory specified by the **--path** option (which is the current working directory by default).

Here's an example of running the init command:

```
/$ migrate --path=/home/cbegin/testdb init
```

If I was already in the /home/cbegin/testdb directory, I could simply run:

```
/home/cbegin/testdb$ migrate init
```

When the command is completed, the directory will contain the following sub-directories:

./drivers

Place your JDBC driver .jar or .zip files in this directory. Upon running a migration, the drivers will be dynamically loaded.

./environments

In the environments folder you will find .properties files that represent your database instances. By default a **development.properties** file is created for you to configure your development time database properties. You can also create test.properties and production.properties files. Details about the properties themselves follow later in this document. The environment can be specified when running a migration by using the **--env=<environment>** option (without the path or ".properties" part).

The default environment is **"development"**. The properties file is self documented, but here it is for reference:

```
## Base time zone to ensure times are consistent across machines
time_zone=GMT+0:00

## The character set that scripts are encoded with
# script_char_set=UTF-8

## JDBC connection properties.
driver=
url=
username=
password=

# Name of the table that tracks changes to the database
changelog=CHANGELOG

# If set to true, each statement is isolated
# in its own transaction.  Otherwise the entire
# script is executed in one transaction.
auto_commit=false

# This controls how statements are delimited.
# By default statements are delimited by an
# end of line semicolon.  Some databases may
```

```
# (e.g. MS SQL Server) may require a full line
# delimiter such as GO.
delimiter=;
full_line_delimiter=false

# This ignores the line delimiters and
# simply sends the entire script at once.
# Use with JDBC drivers that can accept large
# blocks of delimited text at once.
send_full_script=false

# Custom driver path to avoid copying your drivers
# driver_path=
```

./scripts

This directory contains your migration SQL files. These are the files that contain your DDL to both upgrade and downgrade your database structure. By default, the directory will contain the script to create the **changelog** table, plus one empty example migration script. To create a new migration script, use the **"new"** command. To run all pending migrations, use the **"up"** command. To undo the last migration applied, use the **"down"** command etc.

bootstrap

If you're working from an existing database, you need to start from a known state. There's no point in trying to rewind time and shoehorn your existing database into a series of migration scripts. It's more practical to just accept the current state of your database schema and identify this as the starting point. The bootstrap script and command exist for this reason. In the scripts directory you'll find **bootstrap.sql**. You can put your existing DDL script in this file. If you don't have a DDL script, you can export your existing database schema and put it in the bootstrap file. You'll want to clean it up so that it doesn't contain anything specific to any one environment, but otherwise almost any script should work. Watch out for DDL that contains conditional elements or branching logic that could generate multiple schemas. While this is sometimes necessary, it's a really good idea to try to eliminate this aspect of your database schema (put such conditional and branching logic in your code or stored procedures instead). If you have multiple DDL files, you'll have to merge them into the single bootstrap file. But worry not, it's the last time you'll ever modify it. One of the rules above was immutable change scripts… the bootstrap is included in that rule.

➔ **Note:** The bootstrap.sql is a plain text file and is not a valid "migration" that you'll learn about later. It's meant to be similar to the scripts you probably already use. Therefore you cannot rollback the bootstrap file, nor is it tracked in the audit logs… without exception, whatever you put in the bootstrap file cannot leverage the benefits of the other migration commands. But we have to start somewhere, and it's best to look forward.

To run the bootstrap file, you simply call the **bootstrap** command. You do this once to initialize your database into a known working state. From then on, you'll use migration scripts to evolve the database schema.

The bootstrap command has no parameters. So running it is as simple as:

```
/home/cbegin/testdb$ migrate bootstrap
```

As usual, you can use the --path option to specify the repository path, otherwise the current working directory is assumed to be the root of your migration repository (aka migration path).

If there are environment specific elements in your bootstrap script, you'll learn later that you can use properties to deal with those.

new

Now that you've initialized your repository and bootstrapped your existing database schema, you're ready to start leveraging the power of MyBatis Migrations!

MyBatis Migrations are simple SQL script files (*.sql) that live in the **scripts** directory and follow a very strict convention. Since this convention is so important, we don't want to leave it up to humans to try to get it right every time… so we let automation do what it does best, and keep things consistent.

The **new** command generates the skeleton of a migration script that you can then simply fill in. The command is simply run as follows:

```
/home/cbegin/testdb$ migrate new "create blog table"
```

The parameter that the **new** command takes is a comment describing the migration that you're creating. You don't need quotes around it, but it helps keep the command readable.

When the command is run, it will create a file named something like the following:

```
20090807221754_create_blog_table.sql
```

This format is very important (which is why it's generated). The number at the beginning plays three roles. Fist, it's a *practically unique* identifier, meaning it's highly unlikely that two people will generate the same one at the same time (it's not a big deal to resolve if it does happen). Second, it's a timestamp, indicating when the migration was created. Third, it is an ordinal index, formatted in a way that will keep the migrations sorted in the order in which they were created. The remainder of the filename is the comment you specified in the parameter. Finally, the suffix is .sql, indicating the file type that most editors will recognize.

The contents of the migration script also follows a specific and required pattern. Here's the contents of the file we just generated:

```
--// create blog table
-- Migration SQL that makes the change goes here.


--//@UNDO
-- SQL to undo the change goes here.
```

Notice that your comment once again appears at the top of the file. You can add more comments beneath it and throughout the script if you like.

The section immediately following that comment is where you would put your DDL commands to create the blog table.

Then notice the **--//@UNDO** section. This section demarcates the script file sections, splitting it into two distinct parts. Only the commands above the *undo* section will be executed when *upgrading* a database. Everything beneath the *undo* section will be run when *downgrading* the database. Both sections are kept in the same file for simplicity and clarity. The following is a filled in script:

```
--// create blog table

CREATE TABLE BLOG (
  ID INT,
  NAME VARCHAR(255),
  PRIMARY KEY(ID)
);

--//@UNDO

DROP TABLE BLOG;
```

Notice that the commands are **terminated by a colon**. This is also important, and you will receive a warning and likely a failure if you don't terminate the SQL statements with a colon.

Optionally, you can configure your own template to be consumed by the 'new' command. Configuration requires a file named migrations.properties (in $MIGRATIONS_HOME). This file will contain the location of your template.

```
Example of migration.properties

new_command.template=templates/new_template_migration.sql
```

Alternatively you can manually specify the location of your template as such:

```
migrate new –template=<path to template> "your description"
```

If neither of these are used, or valid, the default template shown on the previous page will be used.

So how do we run this script? Well, first it's probably important to understand the current state of the database.

status

The status command will report the current state of the database. The status command takes no parameters and operates on the current working directory or that specified by the **--path** option (as with all other commands).

```
/home/cbegin/testdb$ migrate status
ID              Applied At          Description
=================================================================
20090802210445    ...pending...     create changelog
20090804225328    ...pending...     create blog table
```

Since we've never run a migration, the status of all of the existing migration scripts is **pending**, including the changelog table itself, which is where more detailed status logs are kept. Once we run the **up** command (discussed next), the status will report something like the following:

```
/home/cbegin/testdb$ migrate status
ID              Applied At          Description
=================================================================
20090802210445  2009-08-04 22:51:16  create changelog
20090804225328  2009-08-04 22:51:16  create blog table
```

Thanks to our identifier format, things are in order, and we can see when a migration script was created, as well as when it was applied. The comment helps us read a high level overview of the evolution of this database. As we add migrations this status log will grow. For example:

```
/home/cbegin/testdb$ migrate status
ID              Applied At          Description
=================================================================
20090802210445 2009-08-04 22:51:16 create changelog
```

```
20090804225207 2009-08-04 22:52:51 create author table
20090804225328 2009-08-04 22:54:33 create blog table
20090804225333 2009-08-04 22:54:33 create post table
```

You can also get this information from the **changelog** table by querying it directly in the database. Of course, you won't see any "pending" items, as those are only known to the migration repository until they're applied to the database

up, down

As discussed above, the migration scripts have both a **do** and an **undo** section in them. It's therefore possible to evolve and devolve a database to simplify development and concurrent evolution of the database across development teams. The **up** command runs the **do** section of all pending migrations in order, one after the other. The **down** command runs the **undo** section of the last applied migration only. These commands behave this way because you're most likely to always want the latest revision of the database schema, and if you ever need to roll back, you'll probably only want to do so for the last few versions – and do so in a controlled manner.

Here's a more visual example of how the up and down commands work. We'll use the status command in between to see the effect:

```
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
================================================================
20090802210445    ...pending...    create changelog
20090804225207    ...pending...    create author table
20090804225328    ...pending...    create blog table
20090804225333    ...pending...    create post table

/home/cbegin/testdb$ migrate up
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
================================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
20090804225328 2009-08-04 22:51:17 create blog table
20090804225333 2009-08-04 22:51:17 create post table

/home/cbegin/testdb$ migrate down
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
================================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
```

```
20090804225328 2009-08-04 22:51:17 create blog table
20090804225333    ...pending...    create post table
```

As mentioned, by default up applies all pending migrations, and down undoes just the most recent one. Both up and down commands can take an integer parameter to override these defaults. The integer specifies how many steps should be executed. For example:

```
/home/cbegin/testdb$ migrate down 2
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
=============================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207    ...pending...    create author table
20090804225328    ...pending...    create blog table
20090804225333    ...pending...    create post table
```

There really isn't much more to the up and down commands than that. They let you navigate the evolution of the database schema forward and backward. As usual, they operate on the repository in the current working directory, or the one specified in the optiona **--path** option.

version

The up and down commands are pretty prescriptive in how they work. The **up** command evolves all the way up, and **down** only devolves one step down. Sometimes that might be limiting, so the **version** command exists to allow you to migrate the schema to any specific version of the database you like. You simply call it, specifying the version you'd like to end up at, and the migrations system figures out whether it has to go up or down, and which migrations it needs to run. Here's an example.

```
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
=============================================================
20090802210445    ...pending...    create changelog
20090804225207    ...pending...    create author table
20090804225328    ...pending...    create blog table
20090804225333    ...pending...    create post table

/home/cbegin/testdb$ migrate version 20090804225207
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
=============================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
20090804225328    ...pending...    create blog table
20090804225333    ...pending...    create post table

/home/cbegin/testdb$ migrate up
/home/cbegin/testdb$ migrate status
```

```
ID              Applied At          Description
=================================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
20090804225328 2009-08-04 22:54:32 create blog table
20090804225333 2009-08-04 22:54:32 create post table

/home/cbegin/testdb$ migrate version 20090804225207
/home/cbegin/testdb$ migrate status
ID              Applied At          Description
=================================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
20090804225328    ...pending...    create blog table
20090804225333    ...pending...    create post table
```

The **version** command is a powerful utility for moving to a specific revision of the database.

pending

Sometimes when working with a team of people, or with multiple teams of people, it's possible that more than one change to the database can be made at a time. Past solutions to this problem have been to centralize the management or responsibility for change to one person or team. But this creates bureaucracy and slows down the development process. It also hurts automation and continuous integration. Therefore we need a better approach. MyBatis Migrations simply allows this situation to occur, but makes it very obvious that it has happened. Then the teams can review the situation, downgrade their schemas and re-run the migrations in order, and re-assess the situation. It allows teams to work autonomously, while encouraging communication, team work and good source control practices. When someone creates migration in another workspace before you, but commits to the source control system later than you, you'll end up with an **orphaned pending** migration. They're easy to spot with the **status** command:

```
/home/cbegin/testdb$ migrate status

ID              Applied At          Description
=================================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
20090804225328    ...pending...    create blog table
20090804225333 2009-08-04 22:51:17 create post table
```

MyBatis Migrations will **not** run this orphaned migration simply by running the **up** command. Instead, you'd have to downgrade to the point just before the orphaned migration, then run **up** to run all of the migrations in order. This is the **safest** and recommended approach.

However, if you and the other team review the change, and decide it's completely isolated and not a conflicting change, then there is a way to run the pending migration(s) out of order. The **pending** command does just that. It runs all pending migrations regardless of their order or position in the status log. So if we were to run the pending command given the situation above, the results would be as we expect:

```
/home/cbegin/testdb$ migrate pending
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
================================================================
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
20090804225328 2009-08-05 24:55:23 create blog table
20090804225333 2009-08-04 22:51:17 create post table
```

Even after the fact, you'll be able to identify any migrations run in this way, as the applied date will give them away. An out-of-order applied date is clear indication that a migration was run out of order. No surprises!

Some commands like **pending** and **down** are highly unlikely to ever be needed in production. By the time you promote migrations to production, you've hopefully decided on your final schema and tested and approved the schema for release. While they won't be used in production, they are highly valuable during the development process. Once you get used to the idea, you won't be able to work without it again.

script

While we developers wish we had unlimited access to every environment, the unfortunate truth is that we don't. Often it's the very production environment that we're targeting that we don't have access to. Someone else, a DBA or Change Management team will need to apply any changes. However, we don't want this to be an excuse to not use a good, automated change management tool.

MyBatis Migrations provides the **script** command to generate a script that can migrate a schema from one version to another. As mentioned above, this will likely always be in an upward direction, but the script command does also support generating undo scripts (just in case you're so unfortunate that you don't even have access to central development databases!).

The script command is quite simple. It takes two version numbers as parameters. Think of it as generating a script to apply all of the identified versions (inclusive) in the order specified. For example, given the following:

```
/home/cbegin/testdb$ migrate status
ID             Applied At          Description
================================================================
```

```
20090802210445 2009-08-04 22:51:17 create changelog
20090804225207 2009-08-04 22:51:17 create author table
20090804225328 2009-08-05 24:55:23 create blog table
20090804225333 2009-08-04 22:51:17 create post table
```

If we need to generate a "do" script to apply the two highlighted versions above, we'd run the following command:

```
/home/cbegin/testdb$ migrate script 20090804225328 20090804225333 > do.sql
```

The script command outputs to stdout, so you can let it print to the console, or pipe it to a file or command.

To generate the corresponding "undo" script, simply specify the version numbers in the opposite order:

```
/home/cbegin/testdb$ migrate script 20090804225333 20090804225328 > undo.sql
```

Command Shortcuts

Any of the commands can be executed with only the first few (unambiguous) characters of their name.

For example, the 'status' command can be shortened to:

```
/home/cbegin/testdb$ migrate sta
```

Or even just:

```
/home/cbegin/testdb$ migrate st
```

If you script migrations using unix shell scripts or Windows batch files, be sure to always use the full name for safety. That way, if the shortcut becomes ambiguous in the future, your script will still function.