Database Migration Plugin - Reference Documentation

Authors: Burt Beckwith

Version: 0.1

Table of Contents

1 Introduction to the Database Migration Plugin 3
2 Getting Started 4
3 Configuration 6
4 General Usage 7
5 Groovy Changes 8
6 Groovy Preconditions 10
7 GORM Support 11

1 Introduction to the Database Migration Plugin

The Database Migration plugin helps you manage database changes while developing Grails applications. The plugin uses the Liquibase library.
Using this plugin (and Liquibase in general) adds some structure and process to managing database changes. It will help avoid inconsistencies, communication issues, and other problems with ad-hoc approaches. TODO remove?
Database migrations are represented in text form, either using a Groovy DSL or native Liquibase XML, in one or more changelog files. This approach makes it natural to maintain the changelog files in source control and also works well with branches. Changelog files can include other changelog files, so often developers create hierarchical files organized with various schemes. One popular approach is to have a root changelog named changlog.groovy (or changelog.xml) and to include a changelog per feature/branch that includes multiple smaller changelogs. Once the feature is finished and merged into the main development tree/trunk the changelog files can either stay as they are or be merged into one large file. Use whatever approach makes sense for your applications, but keep in mind that there are many options available for changelog management.
Individual changes have an ID that should be globally unique, although they also include the username of the user making the change, making the combination of ID and username unique (although technically the ID, username, and changelog location are the "unique key").
As you make changes in your code (typically domain classes) that require changes in the database, you add a new change set to the changelog. Commit the code changes along with the changelog additions, and the other developers on your team will get both when they update from source control. Once they apply the new changes their code and development database will be in sync with your changes. Likewise when you deploy to a QA, a staging server, or production, you'll run the un-run changes that correspond to the code updates to being that environment's database in sync. Liquibase keeps track of previously executed changes so there's no need to think about what has and hasn't been run yet.

Scripts
Your primary interaction with the plugin will be using the provided scripts. For the most part these correspond to the many Liquibase commands that are typically executed directly from the commandline or with its Ant targets, but there are also a few Grails-specific scripts that take advantage of the information available from the GORM mappings.
All of the scripts start with `dbm-` to ensure that they're unique and don't clash with scripts from Grails or other plugins.

2 Getting Started

The first step is to install the plugin:

```
grails install-plugin database-migration
```

Typical initial workflow
Next you'll need to create an initial changelog. You can use Liquibase XML or the plugin's Groovy DSL for individual files. You can even mix and match; Groovy files can include other Groovy files and Liquibase XML files (but XML files can't include Groovy files).
Depending on the state of your database and code, you have two options; either create a changelog from the database or create it from your domain classes. The decision tends to be based on whether you prefer to design the database and adjust the domain classes to work with it, or to design your domain classes and use Hibernate to create the corresponding database structure.
To create a changelog from the database, use the dbm-generate-changelog script:

```
grails dbm-generate-changelog changelog.groovy
```

or

```
grails dbm-generate-changelog changelog.xml
```

depending on whether you prefer the Groovy DSL or XML. The filename is relative to the changelog base folder, which defaults to `grails-app/conf/migrations`.
Since the database is already correct, run the dbm-changelog-sync script to record that the changes have already been applied:

```
grails dbm-changelog-sync
```

Running this script is primarily a no-op except that it records the execution(s) in the Liquibase DATABASECHANGELOG table.
To create a changelog from your domain classes, use the dbm-generate-gorm-changelog script:

```
grails dbm-generate-gorm-changelog changelog.groovy
```

or

```
grails dbm-generate-gorm-changelog changelog.xml
```

If you haven't created the database yet, run the dbm-update script to create the corresponding tables:

```
grails dbm-update
```

or the dbm-changelog-sync script if the database is already in sync with your code:

```
grails dbm-changelog-sync
```

Source control
Now you can commit the changelog and the corresponding application code to source control. Other developers can then update and syncronize their databases, and start doing migrations themselves.

3 Configuration

There are a few configuration options for the plugin:

Property	Default	Meaning
grails.plugin.databasemigration.changelogLocation	`grails-app/conf/migrations`	the folder containir main changelog fil (which can include or more other files)
grails.plugin.databasemigration.changelogFileName	`changelog.groovy`	the name of the ma changelog file
grails.plugin.databasemigration.changelogProperties	none	a map of properties use for property substitution in Groo DSL changelogs
grails.plugin.databasemigration.dbDocLocation	`target/dbdoc`	the directory where output from the dbm-db-doc script written
grails.plugin.databasemigration.updateOnStart	`false`	if `true` then chang from the specified l names will be run a startup
grails.plugin.databasemigration.updateOnStartFileNames	none	one or more file na (relative to `changelogLoca`) to run at startup if `updateOnStart` `true`

4 General Usage

After creating the initial changelog, the typical workflow will be along the lines of:

- make domain class changes that affect the schema
- add changes to the changelog for them
- backup your database in case something goes wrong
- run `grails dbm-update` to update your development environment (or wherever you're applying the changes)
- check the updated domain class(es) and changelog(s) into source control

> When running migration scripts on non-development databases, it's important that you backup the database before running the migration in case anything goes wrong. You could also make a copy of the database and run the script against that, and if there's a problem the real database will be unaffected.

To create the changelog additions, you can either manually create the changes or with the dbm-gorm-diff script (you can also use the dbm-diff script but it's far less convenient and requires a 2nd temporary database).
You have a few options with `dbm-gorm-diff`:

- `dbm-gorm-diff` will dump to the console if no filename is specified, so you can copy/paste from there
- if you include the `--add` parameter when running the script with a filename it will register an include for the the filename in the main changelog for you

Regardless of which approach you use, be sure to inspect generated changes and adjust as necessary.

Autorun on start
Since Liquibase maintains a record of changes that have been applied, you can avoid manually updating the database by taking advantage of the plugin's auto-run feature. By default this is disabled, but you can enable it by adding

```
grails.plugin.databasemigration.updateOnStart = true
```

to Config.groovy. In addition you must specify the file(s) containing changes; specify the name(s) using the `updateOnStartFileNames` property, e.g.:

```
grails.plugin.databasemigration.updateOnStartFileNames = ['changelog.groovy']
```

Since changelogs can contain changelogs you'll most often just specify the root changelog, changelog.groovy by convention. Any changes that haven't been executed (in the specified file(s) or files included by them) will be run in the order specified.

5 Groovy Changes

In addition to the built-in Liquibase changes (see the documentation for what's available) you can also make database changes using Groovy code (as long as you're using the Groovy DSL file format). These changes use the `grailsChange` tag name and are contained in a `changeSet` tag like standard built-in tags.
There are four supported inner tags and two callable methods (to override the default confirmation message and checksum value).

General format
This is the general format of a Groovy-based change; all inner tags and methods are optional:

```
grailsChange {
   init {
       // arbitrary initialization code; note that no database or connection is available
   }
   validate {
      // can call warn(String message) to log a warning or error(String message) to stop pro
   }
   change {
      // arbitrary code; make changes directly and/or return a [SqlStatement|http://www.liqu
      confirm 'change confirmation message'
   }
   rollback {
      // arbitrary code; make rollback changes directly and/or return a [SqlStatement|http:/
      confirm 'rollback confirmation message'
   }
   confirm 'confirmation message'
   checkSum 'override value for checksum'
}
```

Available variables
These variables are available throughout the change closure:

- `changeSet`
  - the current Liquibase `ChangeSet` instance
- `resourceAccessor`
  - the current Liquibase `ResourceAccessor` instance
- `ctx`
  - the Spring `ApplicationContext`
- `application`
  - the `GrailsApplication`

The `change` and `rollback` closures also have the following available:

- `database`
  - the current Liquibase `Database` instance
- `databaseConnection`
  - the current Liquibase `DatabaseConnection` instance, which is a wrapper around the JDBC `Connection` (but doesn't implement the `Connection` interface)
- `connection`
  - the real JDBC `Connection` instance (a shortcut for `database.connection.wrappedConnection`)
- `sql`
  - a `groovy.sql.Sql` instance which uses the current `connection` and can be used for arbitrary queries and updates

init
This is where any optional initialization should happen. You can't access the database from this closure.

validate
If there are any necessary validation checks before executing changes or rollbacks they should be done here. You can log warnings by calling `warn(String message)` and stop processing by calling `error(String message)` . It may make more sense to use one or more `preConditions` instead of directly validating here.

change

All migration changes are done in the `change` closure. You can make changes directly (using the `sql` instance or the `connection`) and/or return one or more `SqlStatement`s. You can call `sqlStatement(SqlStatement statement)` multiple times to register instances to be run. You can also call the `sqlStatements(statements)` method with an array or list of instances to be run.

rollback
All rollback changes are done in the `rollback` closure. You can make changes directly (using the `sql` instance or the `connection`) and/or return one or more `SqlStatement`s. You can call `sqlStatement(SqlStatement statement)` multiple times to register instances to be run. You can also call the `sqlStatements(statements)` method with an array or list of instances to be run.

confirm
The `confirm(String message)` method is used to specify the confirmation message to be shown. The default is "Executed GrailsChange" and it can be overridden in the `change` or `rollback` closures to allow phase-specific messages or outside of both closures to use the same message for the update and rollback phase.

checkSum
The checksum for the change will be generated automatically, but if you want to override the value that gets hashed you can specify it with the `checkSum(String value)` method.

6 Groovy Preconditions

In addition to the built-in Liquibase preconditions (see the documentation for what's available) you can also specify preconditions using Groovy code (as long as you're using the Groovy DSL file format). These changes use the `grailsPrecondition` tag name and are contained in the `databaseChangeLog` tag or in a `changeSet` tag like standard built-in tags.

General format
This is general format of a Groovy-based precondition:

```
grailsPrecondition {
   check {
      // use an assertion
      assert x == x
      // use an assertion with an error message
      assert y == y : 'value cannot be 237'
      // call the fail method
      if (x != x) {
         fail 'x != x'
      }
      // throw an exception (the fail method is preferred)
      if (y != y) {
         throw new RuntimeException('y != y')
      }
   }
}
```

As you can see there are a few ways to indicate that a precondition wasn't met:

- use a simple assertion
- use an assertion with a message
- call the `fail(String message)` method (throws a `PreconditionFailedException`)
- throw an exception (shouldn't be necessary - use `assert` or `fail()` instead)

Available variables

- `database`
  - the current Liquibase `Database` instance
- `databaseConnection`
  - the current Liquibase `DatabaseConnection` instance, which is a wrapper around the JDBC `Connection` (but doesn't implement the `Connection` interface)
- `connection`
  - the real JDBC `Connection` instance (a shortcut for `database.connection.wrappedConnection`)
- `sql`
  - a `groovy.sql.Sql` instance which uses the current `connection` and can be used for arbitrary queries and updates
- `resourceAccessor`
  - the current Liquibase `ResourceAccessor` instance
- `ctx`
  - the Spring `ApplicationContext`
- `application`
  - the `GrailsApplication`
- `changeSet`
  - the current Liquibase `ChangeSet` instance
- `changeLog`
  - the current Liquibase `DatabaseChangeLog` instance

Utility methods

- `createDatabaseSnapshotGenerator()`
  - retrieves the `DatabaseSnapshotGenerator` for the current `Database`
- `createDatabaseSnapshot(String schemaName = null)`
  - creates a `DatabaseSnapshot` for the current `Database` (and schema if specified)

7 GORM Support

The plugin's support for GORM is one feature that differentiates it from using Liquibase directly. Typically when using Liquibase you make changes to a database yourself, and then create changesets manually, or use a diff script to compare your updated database to one that hasn't been updated yet. This is a decent amount of work and is rather error-prone. It's easy to forget some changes that aren't required but help performance, for example creating an index on a foreign key when using MySQL.

create-drop, create, and update
On the other end of the spectrum, Hibernate's `create-drop` mode (or `create`) will create a database that matches your domain model, but it's destructive since all previous data is lost when it runs. This works well in the very early stages of development but gets frustrating quickly. Unfortunately Hibernate's `update` mode seems like a good compromise since it only makes changes to your existing schema, but it's very limited in what it will do. It's very pessimistic and won't make any changes that could lose data. So it will add new tables and columns, but won't drop anything. If you remove a not-null domain class property you'll find you can't insert anymore since the column is still there. And it will create not-null columns as nullable since otherwise existing data would be invalid. It won't even widen a column e.g. from `VARCHAR(100)` to `VARCHAR(200)`.

dbm-gorm-diff
The plugin provides a script that will compare your GORM current domain model with a database that you specify, and the result is a Liquibase changeset - `dbm-gorm-diff`. This is the same changeset you would get if you exported your domain model to a scratch database and diffed it with the other database, but it's more convenient. So a good workflow would be:

- make whatever domain class changes you need (add new ones, delete unneeded ones, add/change/remove properties, etc.)
- once your tests pass and you're ready to commit your changes to source control, run the script to generate the changeset that will bring your database back in line with your code
- add the changeset to an existing changelog file, or use the `include` tag to include the whole file
- run the changeset on your functional test database
- assuming your functional tests pass, check everything in as one commit
- the other members of your team will get both the code and database changes when they next update, and will know to run the update script to sync their database with the latest code
- once you're ready to deploy to QA for testing (or staging or production), you can run all of the un-run changes since the last deployment

dbm-generate-gorm-changelog
The dbm-generate-gorm-changelog script is useful for when you want to switch from `create-drop` mode to doing proper migrations. It's not very useful if you already have a database that's in sync with your code, since you can just use the dbm-generate-changelog script that creates a changelog from your databse.