sqlalchemy-migrate Documentation

Release

2011, Evan Rosson, Jan Dittberner, Domen Kožar, Chris Withers

October 28, 2011

CONTENTS

Author Evan Rosson

Maintainer Domen Kožar <domenNO@SPAMdev.si>

Issues http://code.google.com/p/sqlalchemy-migrate/issues/list

Source Code http://code.google.com/p/sqlalchemy-migrate/

Continuous Integration http://jenkins.gnuviech-server.de/job/sqlalchemy-migrate-all/

Generated October 28, 2011

License MIT

Version

Overview

Inspired by Ruby on Rails' migrations, SQLAlchemy Migrate provides a way to deal with database schema changes in SQLAlchemy projects.
Migrate was started as part of Google's Summer of Code by Evan Rosson, mentored by Jonathan LaCour.
The project was taken over by a small group of volunteers when Evan had no free time for the project. It is now hosted as a Google Code project. During the hosting change the project was renamed to SQLAlchemy Migrate.
Currently, sqlalchemy-migrate supports Python versions from 2.4 to 2.7. SQLAlchemy Migrate 0.7.0 supports SQLAlchemy 0.5.x, 0.6.x and 0.7.x branches.

Warning: Version **0.6** breaks backward compatibility, please read *changelog* for more info.

CHAPTER

ONE

DOWNLOAD AND DEVELOPMENT

1.1 Download

You can get the latest version of SQLAlchemy Migrate from the project's download page, the cheese shop, pip or via easy_install:

```
easy_install sqlalchemy-migrate
```

or:

```
pip install sqlalchemy-migrate
```

You should now be able to use the *migrate* command from the command line:

```
migrate
```

This should list all available commands. `migrate help COMMAND` will display more information about each command.

If you'd like to be notified when new versions of SQLAlchemy Migrate are released, subscribe to migrate-announce.

1.2 Development

Migrate's Mercurial repository is located at Google Code.

To get the latest trunk:

```
hg clone http://sqlalchemy-migrate.googlecode.com/hg/
```

Patches should be submitted to the issue tracker.

We use hudson Continuous Integration tool to help us run tests on all databases that migrate supports.

CHAPTER

TWO

DIALECT SUPPORT

Operation / Dialect	*sqlite*	*postgres*	*mysql*	*oracle*	*firebird*	mssql
ALTER TABLE RENAME TABLE	yes	yes	yes	yes	no	not supported
ALTER TABLE RENAME COLUMN	yes (workaround) [1]	yes	yes	yes	yes	not supported
ALTER TABLE ADD COLUMN	yes (workaround) [2]	yes	yes	yes	yes	not supported
ALTER TABLE DROP COLUMN	yes (workaround) [5]	yes	yes	yes	yes	not supported
ALTER TABLE ALTER COLUMN	yes (workaround) [5]	yes	yes	yes (with limitations) [3]	yes [4]	not supported
ALTER TABLE ADD CONSTRAINT	partial (workaround) [5]	yes	yes	yes	yes	not supported
ALTER TABLE DROP CONSTRAINT	partial (workaround) [5]	yes	yes	yes	yes	not supported
RENAME INDEX	no	yes	no	yes	yes	not supported

[1] Table is renamed to temporary table, new table is created followed by INSERT statements.

[2] See http://www.sqlite.org/lang_altertable.html for more information. In cases not supported my sqlite, table is renamed to temporary table, new table is created followed by INSERT statements.

[3] You can not change datatype or rename column if table has NOT NULL data, see http://blogs.x2line.com/al/archive/2005/08/30/1231.aspx for more information.

[4] Changing nullable is not supported

CHAPTER

THREE

TUTORIALS

List of useful tutorials:

- Using migrate with Elixir
- Developing with migrations

CHAPTER

FOUR

USER GUIDE

SQLAlchemy Migrate is split into two parts, database schema versioning (`migrate.versioning`) and database migration management (`migrate.changeset`). The versioning API is available as the *migrate* command.

4.1 Database schema versioning workflow

SQLAlchemy migrate provides the `migrate.versioning` API that is also available as the *migrate* command.

Purpose of this package is frontend for migrations. It provides commands to manage migrate repository and database selection aswell as script versioning.

4.1.1 Project setup

Create a change repository

To begin, we'll need to create a *repository* for our project.

All work with repositories is done using the *migrate* command. Let's create our project's repository:

```
$ migrate create my_repository "Example project"
```

This creates an initially empty repository relative to current directory at my_repository/ named *Example project*.

The repository directory contains a sub directory `versions` that will store the *schema versions*, a configuration file `migrate.cfg` that contains *repository configuration* and a script *manage.py* that has the same functionality as the *migrate* command but is preconfigured with repository specific parameters.

Note: Repositories are associated with a single database schema, and store collections of change scripts to manage that schema. The scripts in a repository may be applied to any number of databases. Each repository has an unique name. This name is used to identify the repository we're working with.

Version control a database

Next we need to declare database to be under version control. Information on a database's version is stored in the database itself; declaring a database to be under version control creates a table named **migrate_version** and associates it with your repository.

The database is specified as a SQLAlchemy database url.

```
$ python my_repository/manage.py version_control sqlite:///project.db
```

We can have any number of databases under this repository's version control.

Each schema has a version that SQLAlchemy Migrate manages. Each change script applied to the database increments this version number. You can see a database's current version:

```
$ python my_repository/manage.py db_version sqlite:///project.db
0
```

A freshly versioned database begins at version 0 by default. This assumes the database is empty. (If this is a bad assumption, you can specify the version at the time the database is declared under version control, with the "version_control" command.) We'll see that creating and applying change scripts changes the database's version number.

Similarly, we can also see the latest version available in a repository with the command:

```
$ python my_repository/manage.py version
0
```

We've entered no changes so far, so our repository cannot upgrade a database past version 0.

Project management script

Many commands need to know our project's database url and repository path - typing them each time is tedious. We can create a script for our project that remembers the database and repository we're using, and use it to perform commands:

```
$ migrate manage manage.py --repository=my_repository --url=sqlite:///project.db
$ python manage.py db_version
0
```

The script manage.py was created. All commands we perform with it are the same as those performed with the *migrate* tool, using the repository and database connection entered above. The difference between the script `manage.py` in the current directory and the script inside the repository is, that the one in the current directory has the database URL preconfigured.

Note: Parameters specified in manage.py should be the same as in *versioning api*. Preconfigured parameter should just be omitted from *migrate* command.

4.1.2 Making schema changes

All changes to a database schema under version control should be done via change scripts - you should avoid schema modifications (creating tables, etc.) outside of change scripts. This allows you to determine what the schema looks like based on the version number alone, and helps ensure multiple databases you're working with are consistent.

Create a change script

Our first change script will create a simple table

```
account = Table('account', meta,
                Column('id', Integer, primary_key=True),
                Column('login', String(40)),
                Column('passwd', String(40)),
)
```

This table should be created in a change script. Let's create one:

```
$ python manage.py script "Add account table"
```

This creates an empty change script at `my_repository/versions/001_Add_account_table.py`. Next, we'll edit this script to create our table.

Edit the change script

Our change script predefines two functions, currently empty: `upgrade()` and `downgrade()`. We'll fill those in

```
from sqlalchemy import *
from migrate import *

meta = MetaData()

account = Table('account', meta,
    Column('id', Integer, primary_key=True),
    Column('login', String(40)),
    Column('passwd', String(40)),
)

def upgrade(migrate_engine):
    meta.bind = migrate_engine
    account.create()

def downgrade(migrate_engine):
    meta.bind = migrate_engine
    account.drop()
```

As you might have guessed, `upgrade()` upgrades the database to the next version. This function should contain the *schema changes* we want to perform (in our example we're creating a table).

`downgrade()` should reverse changes made by `upgrade()`. You'll need to write both functions for every change script. (Well, you don't *have* to write downgrade, but you won't be able to revert to an older version of the database or test your scripts without it.)

Note: As you can see, **migrate_engine** is passed to both functions. You should use this in your change scripts, rather than creating your own engine.

Warning: You should be very careful about importing files from the rest of your application, as your change scripts might break when your application changes. More about writing scripts with consistent behavior.

Test the change script

Change scripts should be tested before they are committed. Testing a script will run its `upgrade()` and `downgrade()` functions on a specified database; you can ensure the script runs without error. You should be testing on a test database - if something goes wrong here, you'll need to correct it by hand. If the test is successful, the database should appear unchanged after `upgrade()` and `downgrade()` run.

To test the script:

```
$ python manage.py test
Upgrading... done
```

```
Downgrading... done
Success
```

Our script runs on our database (`sqlite:///project.db`, as specified in manage.py) without any errors.

Our repository's version is:

```
$ python manage.py version
1
```

> **Warning:** test command executes actual script, be sure you are NOT doing this on production database.

Upgrade the database

Now, we can apply this change script to our database:

```
$ python manage.py upgrade
0 -> 1... done
```

This upgrades the database (`sqlite:///project.db`, as specified when we created manage.py above) to the latest available version. (We could also specify a version number if we wished, using the `--version` option.) We can see the database's version number has changed, and our table has been created:

```
$ python manage.py db_version
1
$ sqlite3 project.db
sqlite> .tables
account migrate_version
```

Our account table was created - success! As our application evolves, we can create more change scripts using a similar process.

4.1.3 Writing change scripts

By default, change scripts may do anything any other SQLAlchemy program can do.

SQLAlchemy Migrate extends SQLAlchemy with several operations used to change existing schemas - ie. `ALTER TABLE` stuff. See *changeset* documentation for details.

Writing scripts with consistent behavior

Normally, it's important to write change scripts in a way that's independent of your application - the same SQL should be generated every time, despite any changes to your app's source code. You don't want your change scripts' behavior changing when your source code does.

> **Warning: Consider the following example of what NOT to do**

Let's say your application defines a table in the `model.py` file:

```
from sqlalchemy import *

meta = MetaData()
table = Table('mytable', meta,
```

```
    Column('id', Integer, primary_key=True),
)
```

... and uses this file to create a table in a change script:

```
from sqlalchemy import *
from migrate import *
import model

def upgrade(migrate_engine):
    model.meta.bind = migrate_engine

def downgrade(migrate_engine):
    model.meta.bind = migrate_engine
    model.table.drop()
```

This runs successfully the first time. But what happens if we change the table definition in `model.py`?

```
from sqlalchemy import *

meta = MetaData()
table = Table('mytable', meta,
    Column('id', Integer, primary_key=True),
    Column('data', String(42)),
)
```

We'll create a new column with a matching change script

```
from sqlalchemy import *
from migrate import *
import model

def upgrade(migrate_engine):
    model.meta.bind = migrate_engine
    model.table.create()

def downgrade(migrate_engine):
    model.meta.bind = migrate_engine
    model.table.drop()
```

This appears to run fine when upgrading an existing database - but the first script's behavior changed! Running all our change scripts on a new database will result in an error - the first script creates the table based on the new definition, with both columns; the second cannot add the column because it already exists.

To avoid the above problem, you should copy-paste your table definition into each change script rather than importing parts of your application.

Note: Sometimes it is enough to just reflect tables with SQLAlchemy instead of copy-pasting - but remember, explicit is better than implicit!

Writing for a specific database

Sometimes you need to write code for a specific database. Migrate scripts can run under any database, however - the engine you're given might belong to any database. Use engine.name to get the name of the database you're working with

```
>>> from sqlalchemy import *
>>> from migrate import *
>>>
>>> engine = create_engine('sqlite:///:memory:')
>>> engine.name
'sqlite'
```

Writings .sql scripts

You might prefer to write your change scripts in SQL, as .sql files, rather than as Python scripts. SQLAlchemy-migrate can work with that:

```
$ python manage.py version
1
$ python manage.py script_sql postgres
```

This creates two scripts `my_repository/versions/002_postgresql_upgrade.sql` and `my_repository/versions/002_postgresql_downgrade.sql`, one for each *operation*, or function defined in a Python change script - upgrade and downgrade. Both are specified to run with Postgres databases - we can add more for different databases if we like. Any database defined by SQLAlchemy may be used here - ex. sqlite, postgres, oracle, mysql...

4.1.4 Command line usage

migrate command is used for API interface. For list of commands and help use:

```
$ migrate --help
```

migrate command exectues `main()` function. For ease of usage, generate your own *project management script*, which calls `main()` function with keywords arguments. You may want to specify *url* and *repository* arguments which almost all API functions require.

If api command looks like:

```
$ migrate downgrade URL REPOSITORY VERSION [--preview_sql|--preview_py]
```

and you have a project management script that looks like

```
from migrate.versioning.shell import main

main(url='sqlite://', repository='./project/migrations/')
```

you have first two slots filed, and command line usage would look like:

```
# preview Python script
$ migrate downgrade 2 --preview_py

# downgrade to version 2
$ migrate downgrade 2
```

Changed in version 0.5.4: Command line parsing refactored: positional parameters usage Whole command line parsing was rewriten from scratch with use of OptionParser. Options passed as kwargs to `main()` are now parsed correctly. Options are passed to commands in the following priority (starting from highest):

- optional (given by `--some_option` in commandline)
- positional arguments

- kwargs passed to migrate.versioning.shell.main

4.1.5 Python API

All commands available from the command line are also available for your Python scripts by importing `migrate.versioning.api`. See the `migrate.versioning.api` documentation for a list of functions; function names match equivalent shell commands. You can use this to help integrate SQLAlchemy Migrate with your existing update process.

For example, the following commands are similar:

From the command line:

```
$ migrate help help
/usr/bin/migrate help COMMAND

    Displays help on a given command.
```

From Python

```
import migrate.versioning.api
migrate.versioning.api.help('help')
# Output:
# %prog help COMMAND
#
#     Displays help on a given command.
```

4.1.6 Experimental commands

Some interesting new features to create SQLAlchemy db models from existing databases and vice versa were developed by Christian Simms during the development of SQLAlchemy-migrate 0.4.5. These features are roughly documented in a thread in migrate-users.

Here are the commands' descriptions as given by `migrate help <command>`:

- `compare_model_to_db`: Compare the current model (assumed to be a module level variable of type sqlalchemy.MetaData) against the current database.
- `create_model`: Dump the current database as a Python model to stdout.
- `make_update_script_for_model`: Create a script changing the old Python model to the new (current) Python model, sending to stdout.

As this sections headline says: These features are EXPERIMENTAL. Take the necessary arguments to the commands from the output of `migrate help <command>`.

4.1.7 Repository configuration

SQLAlchemy-migrate repositories can be configured in their migrate.cfg files. The initial configuration is performed by the *migrate create* call explained in *Create a change repository*. The following options are available currently:

- *repository_id* Used to identify which repository this database is versioned under. You can use the name of your project.
- *version_table* The name of the database table used to track the schema version. This name shouldn't already be used by your project. If this is changed once a database is under version control, you'll need to change the table name in each database too.

- *required_dbs* When committing a change script, SQLAlchemy-migrate will attempt to generate the sql for all supported databases; normally, if one of them fails - probably because you don't have that database installed - it is ignored and the commit continues, perhaps ending successfully. Databases in this list MUST compile successfully during a commit, or the entire commit will fail. List the databases your application will actually be using to ensure your updates to that database work properly. This must be a list; example: *['postgres', 'sqlite']*

4.1.8 Customize templates

Users can pass `templates_path` to API functions to provide customized templates path. Path should be a collection of templates, like `migrate.versioning.templates` package directory.

One may also want to specify custom themes. API functions accept `templates_theme` for this purpose (which defaults to *default*)

Example:

```
/home/user/templates/manage $ ls
default.py_tmpl
pylons.py_tmpl

/home/user/templates/manage $ migrate manage manage.py --templates_path=/home/user/templates --templa
```

New in version 0.6.0.

4.2 Database schema migrations

Importing `migrate.changeset` adds some new methods to existing SQLAlchemy objects, as well as creating functions of its own. Most operations can be done either by a method or a function. Methods match SQLAlchemy's existing API and are more intuitive when the object is available; functions allow one to make changes when only the name of an object is available (for example, adding a column to a table in the database without having to load that table into Python).

Changeset operations can be used independently of SQLAlchemy Migrate's *versioning*.

For more information, see the API documentation for `migrate.changeset`. Here are some direct links to the relevent sections of the API documentations:

- `Create a column`
- `Drop a column`
- `Alter a column` (follow a link for list of supported changes)
- `Rename a table`
- `Rename an index`
- `Create primary key constraint`
- `Drop primary key constraint`
- `Create foreign key contraint`
- `Drop foreign key constraint`
- `Create unique key contraint`
- `Drop unique key constraint`
- `Create check key contraint`

- Drop check key constraint

Note: Many of the schema modification methods above take an `alter_metadata` keyword parameter. This parameter defaults to `True`.

The following sections give examples of how to make various kinds of schema changes.

4.2.1 Column

Given a standard SQLAlchemy table:

```
table = Table('mytable', meta,
   Column('id', Integer, primary_key=True),
)
table.create()
```

You can create a column with `create()`:

```
col = Column('col1', String, default='foobar')
col.create(table, populate_default=True)

# Column is added to table based on its name
assert col is table.c.col1

# col1 is populated with 'foobar' because of 'populate_default'
```

Note: You can pass *primary_key_name*, *index_name* and *unique_name* to the `create()` method to issue `ALTER TABLE ADD CONSTRAINT` after changing the column.

For multi columns constraints and other advanced configuration, check the *constraint tutorial*. New in version 0.6.0.

You can drop a column with `drop()`:

```
col.drop()
```

You can alter a column with `alter()`:

```
col.alter(name='col2')

# Renaming a column affects how it's accessed by the table object
assert col is table.c.col2

# Other properties can be modified as well
col.alter(type=String(42), default="life, the universe, and everything", nullable=False)

# Given another column object, col1.alter(col2), col1 will be changed to match col2
col.alter(Column('col3', String(77), nullable=True))
assert col.nullable
assert table.c.col3 is col
```

Deprecated since version 0.6.0: Passing a `Column` to `ChangesetColumn.alter()` is deprecated. Pass in explicit parameters, such as *name* for a new column name and *type* for a new column type, instead. Do **not** include any parameters that are not changed.

4.2.2 Table

SQLAlchemy includes support for creating and dropping tables..

Tables can be renamed with `rename()`:

```
table.rename('newtablename')
```

4.2.3 Index

SQLAlchemy supports creating and dropping indexes.

Indexes can be renamed using `rename()`:

```
index.rename('newindexname')
```

4.2.4 Constraint

SQLAlchemy supports creating or dropping constraints at the same time a table is created or dropped. SQLAlchemy Migrate adds support for creating and dropping `PrimaryKeyConstraint`, `ForeignKeyConstraint`, `CheckConstraint` and `UniqueConstraint` constraints independently using `ALTER TABLE` statements.

The following rundowns are true for all constraints classes:

1. Make sure you import the relevent constrain class Migrate and not from SQLAlchemy, for example:

   ```
   from migrate.changeset.constraint import ForeignKeyConstraint
   ```

   The classes in that module have the extra `create()` and `drop()` methods.

2. You can also use Constraints as in SQLAlchemy. In this case passing table argument explicitly is required:

   ```
   cons = PrimaryKeyConstraint('id', 'num', table=self.table)

   # Create the constraint
   cons.create()

   # Drop the constraint
   cons.drop()
   ```

   You can also pass in `Column` objects (and table argument can be left out):

   ```
   cons = PrimaryKeyConstraint(col1, col2)
   ```

3. Some dialects support `CASCADE` option when dropping constraints:

   ```
   cons = PrimaryKeyConstraint(col1, col2)

   # Create the constraint
   cons.create()

   # Drop the constraint
   cons.drop(cascade=True)
   ```

Note: SQLAlchemy Migrate will try to guess the name of the constraints for databases, but if it's something other than the default, you'll need to give its name. Best practice is to always name your constraints. Note that Oracle requires that you state the name of the constraint to be created or dropped.

Examples

Primary key constraints:

```
from migrate.changeset.constraint import PrimaryKeyConstraint

cons = PrimaryKeyConstraint(col1, col2)

# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

Foreign key constraints:

```
from migrate.changeset.constraint import ForeignKeyConstraint

cons = ForeignKeyConstraint([table.c.fkey], [othertable.c.id])

# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

Check constraints:

```
from migrate.changeset.constraint import CheckConstraint

cons = CheckConstraint('id > 3', columns=[table.c.id])

# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

Unique constraints:

```
from migrate.changeset.constraint import UniqueConstraint

cons = UniqueConstraint('id', 'age', table=self.table)

# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

4.3 Repository migration (0.4.5 -> 0.5.4)

migrate_repository.py should be used to migrate your repository from a version before 0.4.5 of SQLAlchemy migrate to the current version. Running **migrate_repository.py** is as easy as:

```
migrate_repository.py repository_directory
```

4.4 FAQ

4.4.1 Q: Adding a nullable=False column

A: Your table probably already contains data. That means if you add column, it's contents will be NULL. Thus adding NOT NULL column restriction will trigger IntegrityError on database level.

You have basically two options:

1. Add the column with a default value and then, after it is created, remove the default value property. This does not work for column types that do not allow default values at all (such as 'text' and 'blob' on MySQL).
2. Add the column without NOT NULL so all rows get a NULL value, UPDATE the column to set a value for all rows, then add the NOT NULL property to the column. This works for all column types.

CHAPTER

FIVE

API DOCUMENTATION

5.1 Module `migrate.changeset` – Schema changes

5.1.1 Module `migrate.changeset` – Schema migration API

5.1.2 Module `ansisql` – Standard SQL implementation

5.1.3 Module `constraint` – Constraint schema migration API

5.1.4 Module `databases` – Database specific schema migration

Module `mysql`

Module `firebird`

Module `oracle`

Module `postgres`

Module `sqlite`

Module `visitor`

5.1.5 Module `schema` – Additional API to SQLAlchemy for migrations

5.2 Module `migrate.versioning` – Database versioning and repository management

5.2.1 Module `api` – Python API commands

5.2.2 Module `genmodel` – ORM Model generator

5.2.3 Module `pathed` – Path utilities

5.2.4 Module `repository` – Repository management

5.2.5 Module `schema` – Migration upgrade/downgrade

5.2.6 Module `schemadiff` – ORM Model differencing

5.2.7 Module `script` – Script actions

CHAPTER

SIX

CHANGELOG

6.1 0.7.1 (2011-05-27)

6.1.1 Fixed Bugs

- docs/_build is excluded from source tarball builds
- use table.append_column() instead of column._set_parent() in ChangesetColumn.add_to_table()
- fix source and issue tracking URLs in documentation

6.2 0.7 (2011-05-27)

6.2.1 Features

- compatibility with SQLAlchemy 0.7
- add migrate.__version__

6.2.2 Fixed bugs

- fix compatibility issues with SQLAlchemy 0.7

6.3 0.6.1 (2011-02-11)

6.3.1 Features

- implemented column adding when foreign keys are present for sqlite
- implemented columns adding with unique constraints for sqlite
- implemented adding unique and foreign key constraints to columns for sqlite
- remove experimental *alter_metadata* parameter

6.3.2 Fixed bugs

- updated tests for Python 2.7
- repository keyword in `api.version_control()` can also be unicode
- added if main condition for manage.py script
- make `migrate.changeset.constraint.ForeignKeyConstraint.autoname()` work with SQLAlchemy 0.5 and 0.6
- fixed case sensitivity in setup.py dependencies
- moved `migrate.changeset.exceptions` and `migrate.versioning.exceptions` to `migrate.exceptions`
- cleared up test output and improved testing of deprecation warnings.
- some documentation fixes
- #107: fixed syntax error in genmodel.py
- #96: fixed bug with column dropping in sqlite
- #94: fixed bug that prevented non-unique indexes being created
- fixed bug with column dropping involving foreign keys
- fixed bug when dropping columns with unique constraints in sqlite
- rewrite of the schema diff internals, now supporting column differences in additon to missing columns and tables.
- fixed bug when passing empty list in `migrate.versioning.shell.main()` failed
- #108: Fixed issues with firebird support.

6.4 0.6 (11.07.2010)

Warning: Backward incompatible changes:
- `api.test()` and schema comparison functions now all accept *url* as first parameter and *repository* as second.
- python upgrade/downgrade scripts do not import *migrate_engine* magically, but recieve engine as the only parameter to function (eg. `def upgrade(migrate_engine):`)
- `Column.alter` does not accept *current_name* anymore, it extracts name from the old column.

6.4.1 Features

- added support for *firebird*
- added option to define custom templates through option `--templates_path` and `--templates_theme`, read more in *tutorial section*
- use Python logging for output, can be shut down by passing `--disable_logging` to `migrate.versioning.shell.main()`
- deprecated *alter_column* comparing of columns. Just use explicit parameter change.
- added support for SQLAlchemy 0.6.x by Michael Bayer

- Constraint classes have *cascade=True* keyword argument to issue `DROP CASCADE` where supported
- added `UniqueConstraint`/`CheckConstraint` and corresponding create/drop methods
- API *url* parameter can also be an `Engine` instance (this usage is discouraged though sometimes necessary)
- code coverage is up to 80% with more than 100 tests
- alter, create, drop column / rename table / rename index constructs now accept *alter_metadata* parameter. If True, it will modify Column/Table objects according to changes. Otherwise, everything will be untouched.
- added *populate_default* bool argument to `Column.create` which issues corresponding UPDATE statements to set defaults after column creation
- `Column.create` accepts *primary_key_name*, *unique_name* and *index_name* as string value which is used as contraint name when adding a column

6.4.2 Fixed bugs

- ORM methods now accept *connection* parameter commonly used for transactions
- *server_defaults* passed to `Column.create` are now issued correctly
- use SQLAlchemy quoting system to avoid name conflicts (for issue 32)
- complete refactoring of `ColumnDelta` (fixes issue 23)
- partial refactoring of `changeset` package
- fixed bug when `Column.alter`(server_default='string') was not properly set
- constraints passed to `Column.create` are correctly interpreted (`ALTER TABLE ADD CONSTRAINT` is issued after `ATLER TABLE ADD COLUMN`)
- script names don't break with dot in the name

6.4.3 Documentation

- *dialect support* table was added to documentation
- majoy update to documentation

6.5 0.5.4

- fixed preview_sql parameter for downgrade/upgrade. Now it prints SQL if the step is SQL script and runs step with mocked engine to only print SQL statements if ORM is used. [Domen Kozar]
- use entrypoints terminology to specify dotted model names (module.model:User) [Domen Kozar]
- added engine_dict and engine_arg_* parameters to all api functions (deprecated echo) [Domen Kozar]
- make –echo parameter a bit more forgivable (better Python API support) [Domen Kozar]
- apply patch to refactor cmd line parsing for Issue 54 by Domen Kozar

6.6 0.5.3

- apply patch for Issue 29 by Jonathan Ellis
- fix Issue 52 by removing needless parameters from object.__new__ calls

6.7 0.5.2

- move sphinx and nose dependencies to extras_require and tests_require
- integrate patch for Issue 36 by Kumar McMillan
- fix unit tests
- mark ALTER TABLE ADD COLUMN with FOREIGN KEY as not supported by SQLite

6.8 0.5.1.2

- corrected build

6.9 0.5.1.1

- add documentation in tarball
- add a MANIFEST.in

6.10 0.5.1

- SA 0.5.x support. SQLAlchemy < 0.5.1 not supported anymore.
- use nose instead of py.test for testing
- Added –echo=True option for all commands, which will make the sqlalchemy connection echo SQL statements.
- Better PostgreSQL support, especially for schemas.
- modification to the downgrade command to simplify the calling (old way still works just fine)
- improved support for SQLite
- add support for check constraints (EXPERIMENTAL)
- print statements removed from APIs
- improved sphinx based documentation
- removal of old commented code
- PEP-8 clean code

6.11 0.4.5

- work by Christian Simms to compare metadata against databases
- new repository format
- a repository format migration tool is in migrate/versioning/migrate_repository.py
- support for default SQL scripts
- EXPERIMENTAL support for dumping database to model

6.12 0.4.4

- patch by pwannygoodness for Issue #15
- fixed unit tests to work with py.test 0.9.1
- fix for a SQLAlchemy deprecation warning

6.13 0.4.3

- patch by Kevin Dangoor to handle database versions as packages and ignore their __init__.py files in version.py
- fixed unit tests and Oracle changeset support by Christian Simms

6.14 0.4.2

- package name is sqlalchemy-migrate again to make pypi work
- make import of sqlalchemy's SchemaGenerator work regardless of previous imports

6.15 0.4.1

- setuptools patch by Kevin Dangoor
- re-rename module to migrate

6.16 0.4.0

- SA 0.4.0 compatibility thanks to Christian Simms
- all unit tests are working now (with sqlalchemy >= 0.3.10)

6.17 0.3

- SA 0.3.10 compatibility

6.18 0.2.3

- Removed lots of SA monkeypatching in Migrate's internals
- SA 0.3.3 compatibility
- Removed logsql (#75)
- Updated py.test version from 0.8 to 0.9; added a download link to setup.py
- Fixed incorrect "function not defined" error (#88)
- Fixed SQLite and .sql scripts (#87)

6.19 0.2.2

- Deprecated driver(engine) in favor of engine.name (#80)
- Deprecated logsql (#75)
- Comments in .sql scripts don't make things fail silently now (#74)
- Errors while downgrading (and probably other places) are shown on their own line
- Created mailing list and announcements list, updated documentation accordingly
- Automated tests now require py.test (#66)
- Documentation fix to .sql script commits (#72)
- Fixed a pretty major bug involving logengine, dealing with commits/tests (#64)
- Fixes to the online docs - default DB versioning table name (#68)
- Fixed the engine name in the scripts created by the command 'migrate script' (#69)
- Added Evan's email to the online docs

6.20 0.2.1

- Created this changelog
- Now requires (and is now compatible with) SA 0.3
- Commits across filesystems now allowed (shutil.move instead of os.rename) (#62)

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

m