distcontrib_migrate Documentation

Release 0.1.0

Richard Gomes

Sep 27, 2017

Contents

Code | Bugs | Forums | License | Contact

Python package `distcontrib-migrate` contributes utility functions to Distutils, extending its functionalities, such as database management functions and database migration functions.

The primary reason for the existence of `distcontrib-migrate` is making life a lot easier when dealing with database management and schema migration. By wiring `distcontrib-migrate` into your `setup.py` file, you make it capable of performing these tasks for you, thanks to powerful packages sqlalchemy and sqlalchemy-migrate.

See also: distcontrib

CHAPTER 1

Usage

Below you see an example of a `setup.py` file which employs distcontrib and `distcontrib-migrate`:

```
#!/usr/bin/env python

from distutils.core import setup
from Cython.Distutils import build_ext as cython_build
import distcontrib.tools as du
import distcontrib_migrate.api as dm

 ##
# This block contains settings you will eventually need to change
###

import myapp as myapp   #--- adjust to your package name

PACKAGE      = myapp.pkg_name
VERSION      = myapp.pkg_version
DESCRIPTION  = myapp.pkg_description
LICENSE      = myapp.pkg_license
URL          = myapp.pkg_url
AUTHOR       = myapp.pkg_author
AUTHOR_EMAIL = myapp.pkg_email
KEYWORDS     = myapp.pkg_keywords
REQUIREMENTS = myapp.pkg_requirements
LONG_DESCRIPTION = du.tools.read('README')
CLASSIFIERS      = [ 'License :: ' + LICENSE,
                     'Operating System :: OS Independent',
                     'Programming Language :: Python',
                     'Programming Language :: Cython',
                     'Development Status :: 3 - Alpha',
                     'Intended Audience :: Developers',
                     'Environment :: Console' ]

 ##
# From this point on, it's unlikely you will be changing anything.
```

```
###

PACKAGES      = find_packages(exclude=["*.tests", "*.tests.*", "tests.*", "tests"])
PACKAGES_DATA = du.tools.findall_package_data(PACKAGES)
EXT_MODULES   = du.tools.find_ext_modules(PACKAGES)

setup(
    name=PACKAGE,
    version=VERSION,
    description=DESCRIPTION,
    url=URL,
    author=AUTHOR,
    author_email=AUTHOR_EMAIL,
    long_description=LONG_DESCRIPTION,
    license=LICENSE,
    keywords=KEYWORDS,
    classifiers=CLASSIFIERS,
    packages=PACKAGES,
    package_data=PACKAGES_DATA,
    cmdclass={ 'build_ext' : cython_build,
               'doctest'   : du.doctests,
               'zap'       : du.zap,
               'migrate'   : dm.migrate,
               'psql'      : dm.psql, },
    ext_modules=EXT_MODULES,
    # install_requires=REQUIREMENTS
)
```

Then create under your `myapp/__init__.py` file something like this:

```
#!/usr/bin/env python

pkg_name         = __name__ if __package__ is None else __package__
pkg_description  = 'This application does everything you can imagine'
pkg_version      = '0.1.0'
pkg_license      = 'OSI Approved :: BSD License'
pkg_url          = 'http://' + pkg_name + '.readthedocs.org/'
pkg_author       = 'Richard Gomes http://rgomes-info.blogspot.com'
pkg_email        = 'rgomes.info@gmail.com'
pkg_keywords     = [ 'artificial','intelligence','magic','sorcery','voodoo' ]
pkg_requirements = [ 'lxml', 'sqlalchemy' ]
```

Requirements

Database management functions require `sudo` rights.

Quick guide for the impatient

In *all examples below*, please keep in mind that you can always specify the `--url` parameter. If not specified, the value assumed by default is:

```
postgresql://${USER}@localhost:5432/sample
```

You can create users easily from command line:

```
$ python setup.py psql --createuser         # creates an user in the database
$ python setup.py psql --dropdb             # drops user and all owned objects
$ python setup.py psql --dropdb --createdb  # drops and creates an user in one go
```

Then you can create a Postgres database easily from command line:

```
$ python setup.py psql --createdb           # creates a database
$ python setup.py psql --dropdb             # drops a database
$ python setup.py psql --dropdb --createdb  # drops and creates a database in one go
```

You can query what the model version and the database version are:

```
$ python setup.py migrate --status --url postgresql://${USER}@localhost:5432/sample
↪# show model and database versions
$ python setup.py migrate --status
↪# same as above
```

You can perform upgrade and downgrade the database easily from command line:

```
$ python setup.py migrate --upgrade --url postgresql://${USER}@localhost:5432/sample␣
↪ # upgrade database
$ python setup.py migrate --upgrade                                                 ␣
↪ # same as above
$ python setup.py migrate --upgrade --changeset=17                                  ␣
↪ # upgrade to database version 17
$ python setup.py migrate --downgrade --url postgresql://${USER}@localhost:5432/
↪sample   # downgrade database
$ python setup.py migrate --downgrade                                               ␣
↪   # same as above
$ python setup.py migrate --downgrade --changeset=15                                ␣
↪   # downgrade to database version 15
```

In order to test upgrade/downgrade scripts, you can do this:

```
$ python setup.py migrate --test
```

Note: `--test` is equirevalent to `--upgrade` followed by `--downgrade`

If you have an existing database which you would like to reverse engineer its model, you can try this:

```
$ STAGING_URL=postgresql://admin@staging.example.com:5432/crm_staging
$ python setup.py migrate --status       --url=${STAGING_URL}
Model version:    0
Database version: 17
$ python setup.py migrate --create-model --url=${STAGING_URL} # reverse engineering␣
↪to repository "admin" (the default)
$ python setup.py migrate --status       --url=${STAGING_URL}
Model version:   1
Database version: 17
```

Now apply the model you obtained to a brand new database:

```
$ DEVEL_URL=postgresql://localhost:5432/sandbox
$ python setup.py psql --createuser --url=${DEVEL_URL}
$ python setup.py psql --createdb   --url=${DEVEL_URL}
```

```
$ python setup.py migrate --status  --url=${DEVEL_URL}
Model version:    1
Database version: 0
$ python setup.py migrate --upgrade --url=${DEVEL_URL}
$ python setup.py migrate --status  --url=${DEVEL_URL}
Model version:    1
Database version: 1
```

CHAPTER 2

Concepts

All concepts presented here must be understood from the point of view of the `migration` command, which means that they have a much narrow scope than usual.

Migration script

A migration script is a Python program which has basically only two functions: `upgrade` and `downgrade`. Below you see an example of a migration script:

```
from sqlalchemy import *
meta = MetaData()

exchange_codes = Table('exchange_codes', meta,
    Column('mic', String, primary_key=True, nullable=False),
    Column('country', String, nullable=False),
    Column('iso3166', String, nullable=False),
    Column('omic', String, nullable=False),
    Column('os', String, nullable=False),
    Column('name', String, nullable=False),
    Column('acronym', String, nullable=False),
    Column('city', String, nullable=False),
    Column('url', String, nullable=False),
    Column('status', String, nullable=False),
    Column('sdata', String, nullable=False),
    Column('cdate', String, nullable=False),
    Column('comments', String, nullable=False),
)

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

def downgrade(migrate_engine):
```

```
    meta.bind = migrate_engine
    exchange_codes.drop(meta.bind)
```

For more information, please consult sqlalchemy-migrate

Tip: `distcontrib-migrate` eases your life in regards to `sqlalchemy-migrate`: the only task left to you is the maintenance of migrations scripts. You don't need to create the model repository, define the project name, for example.

Model version

A model consists a set of migration scripts.

Supose that you are creating CRM application. You start by representing people in the database. You think that *name* and *SSN* is all you need for a person, initially. As time passes, you also need to represent deparments in the database. Then you realize that you need to link people and departments. Your model would have three iterations:

- 001_creating_table_person
- 002_creating_table_department
- 003_linking_people_and_departments

In other works, your model consists of three migration scripts. When you run the command `migrate --status`, it basically tells you how many migration scripts you have defined.

Database version

Suppose you have defined a model consisting of three migration scripts, like explained above. Also suppose that you created an empty database, which means that no migration scripts have been applied yet. In this case, the `migrate --status` command will show:

```
$ python setup.py migrate --status
Model version:    3
Database version: 0
```

Now suppose you have applied only two migration scripts because you are still working on the code which links people with departments. the `migrate --status` command will show:

```
$ python setup.py migrate --upgrade --changeset=2
$ python setup.py migrate --status
Model version:    3
Database version: 2
```

Repository

A repository is the location in the file system where your model is stored. The repository consists on control files which are *automagically* created for you and migration scripts. You don't need to bother about creating a repository, since it is created by you the first time you run the `migrate --status` command.

Creating an empty model

Supposing you are starting from scratch:

```
$ python setup.py migrate --status
Model version:    0
Database version: 0
```

In the example above, a new repository is created under the directory `admin`. You can choose another directory name if you pass argument `--scripts`, like this:

```
$ python setup.py migrate --status --scripts=woodpecker
Model version:    0
Database version: 0
```

Creating a model using reverse engineering

The command `migrate --create-model` retrieves the database schema from an existing database accessible thru a connection URL and creates a new migration script onto a giving repository. For example:

```
$ STAGING_URL=postgresql://admin@staging.example.com:5432/crm_staging
$ python setup.py migrate --status       --url=${STAGING_URL}
Model version:    0
Database version: 17

$ python setup.py migrate --create-model --url=${STAGING_URL} # reverse engineering␣
↪to repository "admin" (the default)
$ python setup.py migrate --status       --url=${STAGING_URL}
Model version:    1
Database version: 17
```

This functionality depends on sqlalchemy-migrate and it is considered experimental. If you find issues, pleare report to sqlalchemy-migrate user's mailing list

Authentication

Commands `migrate` and `psql` also honour the convention of storing passwords for Postgres databases onto file `~/.pgpass`. The first time you try to access a database which does not have its password stored in `~/.pgpass`, you will be prompted to enter the password and it will be stored in the file.

CHAPTER 3

Known issues and limitations

- (issue 1208083): Installation with `pip` onto a fresh environment may fail. Workaround: attempting to install a second time is expected to work:

```
$ pip install sqlalchemy-migrate   # eventually may fail
$ pip install sqlalchemy-migrate   # expected to suceed
```

- Database management functions (createuser, dropuser, createdb, dropdb) support only Postgres databases at the moment. However, database migrations are supported on all databases supported by sqlalchemy.

CHAPTER 4

Support

- Bugs: https://bugs.launchpad.net/distcontrib-migrate
- Forums : https://answers.launchpad.net/distcontrib-migrate
- Sources: https://code.launchpad.net/distcontrib-migrate

Note: Issues related to command `--create-model` are, in most situations, originated in sqlalchemy-migrate, since the reverse engineering is considered experimental. Please report these issues at sqlalchemy-migrate user's mailing list.