mortar-rdb Documentation

Release 3.0.0

Simplistix Ltd

Mar 07, 2019

Contents

This package ties together SQLAlchemy and the component architecture to make it easy to develop projects using SQLAlchemy through their complete lifecycle.

It also includes some common models and mixins that are useful in a variety of projects.

CHAPTER 1

Installation Instructions

The best way to install mortar_rdb is with pip:

```
pip install mortar_rdb
```

Of course, once it's installed, make sure it's in your `requirements.txt`!

Python version requirements

This package has been tested with Python 2.7, 3.4+, MySQL, Postgres and SQLite on Linux, and is also expected to work on Mac OS X and Windows with any database supported by SQLAlchemy.

CHAPTER 2

Basic Usage

This narrative documentation covers the use case of developing an application from scratch that uses *mortar_rdb* to interact with a relational database through development and testing.

2.1 Development

For this narrative, we'll assume we're developing our application in a python package called `sample` that uses the following model:

sample/model.py

```
from mortar_rdb import declarative_base
from mortar_rdb.controlled import Config, scan
from sqlalchemy import Table, Column, Integer, String

Base = declarative_base()

class User(Base):
  __tablename__ = 'user'
  id = Column(Integer, primary_key=True)
  name = Column(String(20))

source = scan('sample')
config = Config(source)
```

There's nothing particularly special about this model other than that we've used *mortar_rdb.declarative_base()* to obtain a declarative base rather than calling `sqlalchemy.ext.declarative.declarative_base()`. This means that multiple python packages can all use the same declarative base, without having to worry about which package first defines the base.

This also means that all tables and models used in our application, regardless of the package they are defined in, can refer to each other.

To allow us to take advantage of the schema controls provided by `mortar_rdb`, we have also defined a `Config` with a `Source` returned from a `scan()`. The source is defined seperately to the configuration for two reasons:

- it allows a configuration in another package to use the source defined here, which encapsulates all the tables managed by this package.
- it makes it easier to write tests for migration scripts for the tables managed by this package.

To use the above model, we have the following view code:

sample/views.py

```
from mortar_rdb import get_session
from sample.model import User

def add_user(name):
    session = get_session()
    session.add(User(name=name))

def edit_user(id,name):
    session = get_session()
    user = session.query(User).filter(User.id == id).one()
    user.name = name
```

When using `mortar_rdb`, the session is obtained by calling `mortar_rdb.get_session()`. This allows the provision of the session to be independent of its use, which makes testing and deploying to different environments easier.

It is also advised that application code does not manage committing or rollback of database transactions via the session unless absolutely necessary. These actions should be the responsibility of the framework running the application.

For the purposes of this narrative, we will use the following micro framework:

sample/run.py

```
from mortar_rdb import register_session
from sample import views
from sample.config import db_url
from sample.model import config

import sys
import transaction

def main():
    register_session(db_url)
    name = sys.argv[1]
    args = sys.argv[2:]
    with transaction.manager:
        getattr(views, name)(*args)
    print("Ran %r with %r" % (name, args))

if __name__=='__main__':
    main()
```

Although there's not much to it, the above framework shows the elements you will need to plug in to whatever framework you choose to use.

The main one of these is the call to *register_session()* which sets up the components necessary for *get_session()* to return a Session object.

The example framework is also shown to manage these sessions using the transaction package. Should your framework not use this package, you are strongly suggested to read the documentation for *register_session()* in detail to make sure you pass the correct parameters to get the behaviour required by your framework.

2.2 Testing

It's alway a good idea to write automated tests, preferably before writing the code under test. *mortar_rdb* aids this by providing the *mortar_rdb.testing* module.

The following example shows how to provides minimal coverage using *mortar_rdb.testing.register_session()* and illustrates how the abstraction of configuring a session from obtaining a session in *mortar_rdb* makes testing easier:

sample/tests.py

```
from mortar_rdb import get_session
from mortar_rdb.testing import register_session
from sample.model import User, config
from sample.views import add_user, edit_user
from unittest import TestCase

class Tests(TestCase):

    def setUp(self):
        self.session = register_session(config=config)

    def tearDown(self):
        self.session.rollback()

    def test_add_user(self):
        # code under test
        add_user('Mr Test')
        # checks
        user = self.session.query(User).one()
        self.assertEqual('Mr Test', user.name)

    def test_edit_user(self):
        # setup
        self.session.add(User(id=1, name='Mr Chips'))
        # code under test
        edit_user('1', 'Mr Fish')
        # checks
        user = self.session.query(User).one()
        self.assertEqual('Mr Fish', user.name)
```

If you wish to run these tests against a particular database, rather than using the default in-memory SQLite database, then set the DB_URL enviroment variable to the SQLAlchemy url of the database you'd like to use. For example, if you run your tests with pytest and are developing in a unix-like environment against a MySQL database, you could do:

```
$ DB_URL=mysql://scott:tiger@localhost/test pytest
```

2.3 Release

With the application developed and tested, it is now time to release and deploy it. Users of `mortar_rdb` are encouraged to create a small database management script making use of `mortar_rdb.controlled.Scripts`.

Here's is an example for the above model:

sample/db.py

```python
from mortar_rdb.controlled import Scripts
from sample.config import db_url, is_production
from sample.model import config

scripts = Scripts(db_url, config, not is_production)

if __name__=='__main__':
    scripts()
```

This script can be used to create all tables required by the applications `Config` as follows:

```
$ bin/db create
For database at sqlite:////test.db:
Creating the following tables:
user
```

Other commands are are provided by `Scripts` and both the command line help, obtained with the `--help` option to either the script or any of its commands, and documentation are well worth a read.

So, the view code, database model, tests and framework are all now ready and the database has been created. The framework is now ready to use:

```
$ bin/run add_user test
Ran 'add_user' with ['test']
```

CHAPTER 3

Sequences

Many applications require non-repeating sequences of integer numbers. While some database engines provide structures for this purpose, others, such as MySQL and SQLite do not.

Alternatively, you may wish to have a more global source of unique integer identifiers that you share across several database servers.

For either of these cases, you can use the sequence support provided by *mortar_rdb*.

Usage is very simple. During application configuration, register sequences at the same time as you register the session to use for a database:

```
from mortar_rdb import register_session, get_session
from mortar_rdb.sequence import register_sequence

import transaction

register_session(db_url)
with transaction.manager:
    session = get_session()
    register_sequence('seq1', session)
    register_sequence('seq2', session)
```

You'll notice that you need to manage a database transaction when you call *register_sequence()*, either using the `transaction` package or by manually calling `commit()` after the call. This is because registering a sequence, certainly in the default *SequenceImplementation*, may require creating a row in a table.

Once registered, whenever you need some unique integers in application code, get hold of the sequence and call its `next()` method:

```
>>> from mortar_rdb import get_session
>>> from mortar_rdb.sequence import get_sequence
>>> seq1 = get_sequence('seq1')
>>> seq2 = get_sequence('seq2')
>>> session = get_session()
>>> seq1.next(session)
```

(continues on next page)

(continued from previous page)

```
1
>>> seq1.next(session)
2
>>> seq1.next(session)
3
>>> seq2.next(session)
1
```

Note: The default implementation used by *register_sequence()* is *mortar_rdb.sequence.generic.SequenceImplementation*.

This uses a table called `sequences` in the database which needs to be created before you first call *register_sequence()*. If you are using *mortar_rdb.controlled* then you can use the source from `mortar_rdb.sequence.generic.source` as part of your *Config* to take care of table creation.

Also, please note that this implementation may cause contention on the table in question. If your database engine provides native sequences, an implementation that used those would be gratefully received!

CHAPTER 4

API Reference

4.1 mortar_rdb

mortar_rdb.**declarative_base**(***kw*)

Return a `Base` as would be returned by `declarative_base()`.

Only one `Base` will exist for each combination of parameters that this function is called with. If it is called with the same combination of parameters more than once, subsequent calls will return the existing `Base`.

This method should be used so that even if more than one package used by a project defines models, they will all end up in the same `MetaData` instance and all have the same declarative registry.

mortar_rdb.**drop_tables**(*engine*)

Drop all the tables in the database attached to by the supplied engine.

As many foreign key constraints as possible will be dropped first making this quite brutal!

mortar_rdb.**get_session**(*name=''*)

Return a `Session` instance from the current registry as registered with the supplied *name*.

mortar_rdb.**register_session**(*url=None*, *name=''*, *engine=None*, *echo=None*, *transactional=True*, *scoped=True*, *twophase=True*)

Create a `Session` class and register it for later use.

Generally, you'll only need to pass in a `SQLAlchemy` connection URL. If you want to register multiple sessions for a particular application, then you should name them. If you want to provide specific engine configuration, then you can pass in an `Engine` instance. In that case, you must not pass in a URL.

Parameters

- **echo** – If *True*, then all SQL will be echoed to the python logging framework. This option cannot be specified if you pass in an engine.
- **scoped** – If *True*, then `get_session()` will return a distinct session for each thread that it is called from but, within that thread, it will always return the same session. If it is *False*, every call to `get_session()` will return a new session.

- **transactional** – If *True*, a SQLAlchemy extension will be used that that enables the transaction package to manage the lifecycle of the SQLAlchemy session (eg: begin()/commit()/rollback()). This can only be done when scoped sessions are used.

  If *False*, you will need to make sure you call begin()/commit()/rollback(), as appropriate, yourself.
- **twophase** – By default two-phase transactions are used where supported by the underlying database. Where this causes problems, single-phase transactions can be used for all engines by passing this parameter as *False*.

4.2 mortar_rdb.controlled

When a database is used, it's essential that code using the database only interacts with a database that is of the form it expects. A corollary of that is that it is important to be able to update the structure of a database from the form expected by one version of the code to that expected by another version of the code.

mortar_rdb.controlled aims to facilitate this along with providing a command line harness for creating necessary tables within a database, emptying out a non-production database and upgrading a database to a new structure where SQLAlchemy is used.

4.2.1 Packages, Models and Tables

SQLAlchemy uses Table objects that are mapped to one or more model classes. These objects are defined within python packages.

4.2.2 Configurations

A single database may contain tables that are defined in more that one package. For example, an authentication package may contain some table definitions for storing users and their permissions. That package may be used by an application which also contains a package that defines its own tables.

A *Config* is a way of expressing which tables should be expected in a database.

In general, it is recommended that a *Config* is defined once, in whatever package 'owns' a particular database. For example, an application may define a configuration for its own tables and those of any packages on which it relies, such as the hypothetical authentication package described above. If another application wants to use this application's database, it can import the configuration and check that the database structure matches that expected by the code it is currently using.

class mortar_rdb.controlled.**Config**(**sources*)

A configuration for a particular database to allow control of the schema of that database.

Parameters **sources** – The *Source* instances from which to create this configuration.

class mortar_rdb.controlled.**Scripts**(*url*, *config*, *failsafe*)

A command-line harness for performing schema control functions on a database. You should instantiate this in a small python script and call it when the script is run as a command, eg:

```
from mortar_rdb.controlled import Scripts
from sample.model import config

scripts = Scripts('sqlite://', config, True)
```

(continues on next page)

(continued from previous page)

```
if __name__=='__main__':
    script()
```

Writing the script in this style also allows `scripts` to be used as a :mod:setuptools entry point.

Parameters

- **`url`** – The `SQLAlchemy` url to connect to the database to be managed. If this isn't known at the time when this class is instantiated, then use `None`.
- **`config`** – A *`Config`* instance describing the schema of the database to be managed.
- **`failsafe`** – A boolean value that should be `True` if it's okay for the database being managed to have all its tables dropped. For obvious reasons, this should be `False` when managing your production database.

`create``()`
Create all the tables in the configuration in the database

`drop``()`
Drop all tables in the database

`class` `mortar_rdb.controlled.`**`Source`**`(`**tables*`)`
A collection of tables that should have their versioning managed together. This usually means they originate from one package.

Parameters **`tables`** – A sequence of `Table` objects that contain all the tables that will be managed by the repository in this Source.

`mortar_rdb.controlled.`**`scan`**`(`*package*, *tables=()*`)`
Scan a package or module and return a *`Source`* containing the tables from any declaratively mapped models found, any `Table` objects explicitly passed in and the `sqlalchemy-migrate` repository contained within the package.

Note: While the *package* parameter is passed as a string, this will be resolved into a module or package object. It is not a distribution name, although the two are often very similar.

Parameters

- **`package`** – A dotted path to the package to be scanned for `Table` objects.
- **`tables`** – A sequence of `Table` objects to be added to the returned *`Source`*. Any tables not created as part of declaratively mapping a class will need to be passed in using this sequence as *`scan()`* cannot sensibly scan for these objects.

4.3 mortar_rdb.testing

Helpers for unit testing when using *`mortar_rdb`*

`class` `mortar_rdb.testing.`**`TestingBase`**
This is a helper class that can either be used to make *`declarative_base()`* return a new, empty `Base` for testing purposes.

If writing a suite of unit tests, this can be done as follows:

```
from mortar_rdb.testing import TestingBase
from unittest import TestCase

class YourTestCase(TestCase):

    def setUp(self):
        self.tb = TestingBase()

    def tearDown(self):
        self.tb.restore()
```

If you need a fresh `Base` for a short section of Python code, `TestingBase` can also be used as a context manager:

```
with TestingBase():
    base = declarative_base()
    # your test code here
```

`mortar_rdb.testing.`**`register_session`**(*url=None*, *name=''*, *engine=None*, *echo=False*, *transactional=True*, *scoped=True*, *config=None*, *metadata=None*)

This will create a `Session` class for testing purposes and register it for later use.

The calling parameters mirror those of `mortar_rdb.register_session()` but if neither *url* nor *engine* is specified then:

- The environment will be consulted for a variable called `DB_URL`. If found, that will be used for the *url* parameter.
- If *url* is still *None*, an implicit *url* of `sqlite://` will be used.

If a `Config` is passed in then, once any existing content in the database has been removed, any tables controlled by that config will be created.

If a `MetaData` instance is passed in, then all tables within it will be created.

Unlike the non-testing `register_session`, this will also return an instance of the registered session.

Warning: No matter where the *url* or *engine* come from, the entire contents of the database they point at will be destroyed!

4.4 mortar_rdb.sequence

Database independent provision of non-repeating, always-incrementing sequences of integers.

`mortar_rdb.sequence.`**`get_sequence`**(*name*)

Obtain a previously registered sequence. Once obtained, the `next()` method should be called as many times as necessary.

Each call will return one system-wide unique integer that will be greater than any integers previously returned.

`mortar_rdb.sequence.`**`register_sequence`**(*name*, *session*, *impl=<class 'mortar_rdb.sequence.generic.SequenceImplementation'>*)

Register a sequence for later user.

Parameters

- **name** – A string containing the name of the sequence.
- **session** – A `Session` instance that will be used to set up anything needed in the database for the sequence to be functional. It will not be retained and may be closed and discarded once this function has returned.
- **impl** – A class whose instances implement *`ISequence`*. Defaults to *`mortar_rdb.sequence.generic.SequenceImplementation`*.

class `mortar_rdb.sequence.generic.`**`SequenceImplementation`**(*name*, *session*)

A sequence implementation that uses a table in the database with one row for each named sequence.

`next`(*session*)

Return the next integer in the sequence using the `Session` provided.

Warning: The current implementation will lock the row (or table, depending on which database you use) for the sequence in question. This could conceivably cause contention problems if more than one connection is trying to generate integers from the sequence at one time.

4.5 mortar_rdb.interfaces

Internal interface definitions. Unless you're doing something pretty special, you don't need to know about these.

interface `mortar_rdb.interfaces.`**`ISession`**

A marker interface for SQLAlchemy Sessions. This is so that we can register factories that return them.

interface `mortar_rdb.interfaces.`**`ISequence`**

An interface for sequence utility impementations. A sequence is a non-repeating, always-incrementing sequence of integers.

Implementations of this interface will be instantiated once and then have their `next()` method called often.

CHAPTER 5

Development

This package is developed using continuous integration which can be found here:

https://travis-ci.org/Mortar/mortar_rdb

The latest development version of the documentation can be found here:

http://mortar_rdb.readthedocs.org/en/latest/

If you wish to contribute to this project, then you should fork the repository found here:

https://github.com/Mortar/mortar_rdb

Once that has been done and you have a checkout, you can follow these instructions to perform various development tasks:

5.1 Setting up a virtualenv

The recommended way to set up a development environment is to turn your checkout into a virtualenv and then install the package in editable form as follows:

```
$ virtualenv .
$ bin/pip install -U -e .[test,build]
```

5.2 Running the tests

Once you've set up a virtualenv, the tests can be run as follows:

```
$ bin/pytest
```

Some of the tests can be run against a specific database to check compatibility with specific database back ends. To do this, set the `DB_URL` environment variable to the SQLAlchemy url of the database you'd like to use. For example, if you are testing in a unix-like environment and want to test against a MySQL database, you could do:

```
$ DB_URL=mysql://scott:tiger@localhost/mortar_rdb_tests bin/test
```

5.3 Building the documentation

The Sphinx documentation is built by doing the following from the directory containing setup.py:

```
$ source bin/activate
$ cd docs
$ make html
```

To check that the description that will be used on PyPI renders properly, do the following:

```
$ python setup.py --long-description | rst2html.py > desc.html
```

The resulting `desc.html` should be checked by opening in a browser.

5.4 Making a release

To make a release, just update the version in `setup.py`, update the change log, tag it and push to https://github.com/Mortar/mortar_rdb and Travis CI should take care of the rest.

Once Travis CI is done, make sure to go to https://readthedocs.org/projects/mortar_rdb/versions/ and make sure the new release is marked as an Active Version.

CHAPTER 6

Changes

6.1 3.0.0 (7 Mar 2019)

- Drop support for the ancient SQLAlchemy extension mechanism.

6.2 2.2.1 (15 Aug 2016)

- Stop passwords showing in logging from *`Scripts`*.

6.3 2.2.0 (29 Oct 2015)

- More careful password masking.
- Better support for using *`controlled.Scripts`* as part of another script framework.

6.4 2.1.1 (4 Oct 2015)

- Deploy to PyPI using Travis.

6.5 2.1.0 (4 Oct 2015)

- Drop support for Python 2.6.
- Add support for Python 3.4+.
- Move to Read The Docs for documentation.
- Move to virtualenv and nose for development.

- Move to Travis CI and Coveralls for automated continuous testing.

6.6 2.0.0 (29 Oct 2013)

- Remove use of `sqlalchemy-migrate`, `alembic` is a better bet but not yet introduced.
- Much work to better adhere to PEP 8, including renaming the major functions.

6.7 1.2.1 (30 Jun 2011)

- Add `setuptools_git` to the build chain so that `setuptools` *include_package_data* works once more.

6.8 1.2.0 (30 Jun 2011)

- Pass *None* as the default for *echo* rather than *False* on the advice of Daniel Holth.
- When using `register_session()`, allow explicit disabling of two-phase commit.
- No longer log passwords during session registration.
- Specify `sqlalchemy` 0.6 as a requirement, until `zope.sqlalchemy` is ported, `mortar_rdb` shouldn't be used with `sqlalchemy` 0.7.

6.9 1.1.0 (27 Feb 2011)

- Allow passing in `SessionExtension` instances to both `registerSession()` functions.
- Fixed a bug that resulted in an exception when passing `echo=True` to `mortar_rdb.testing.registerSession()` but not passing a `url`.

6.10 1.0.1 (19 Feb 2011)

- Fixed a missing declaration of dependency on `zope.dottedname`.

6.11 1.0.0 (18 Feb 2011)

- Initial Release.

CHAPTER 7

License

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index