SQLAHelper Documentation

Release 1.0

Mike Orr

January 14, 2016

Contents

Version 1.0, released 2011-12-25

PyPI http://pypi.python.org/pypi/SQLAHelper

Docs http://sluggo.scrapping.cc/python/SQLAHelper/

Source http://bitbucket.org/sluggo/sqlahelper (Mercurial)

SQLAHeler is a small library for SQLAlchemy web applications. It acts as a container for the application's contextual session, engines, and declarative base. This avoids circular dependencies between the application's model modules, and allows cooperating third-party libraries to use the application's session, base, and transaction. SQLAHelper does not try to hide or disguise the underlying SQLAlchemy objects; it merely provides a way to organize them.

The contextual session is initialized with the popular ZopeTransactionExtension, which allows it to work with transaction managers like pyramid_tm and repoze.tm2. A transaction manager provides automatic commit at the end of request processing, or rollback if an exception is raised or HTTP error status occurs. Some transaction managers can commit both SQL and non-SQL actions in one step. SQLAHelper does not include a transaction manager, but it works with the most common ones.

It's currently tested on Python 2.7/Linux but should work on other platforms. A set of unit tests is included. Python 3 compatibility is unknown but will be addressed soon.

CHAPTER 1

Documentation

1.1 Usage and API

1.1.1 Installation

Install SQLAHelper like any Python package, using either "pip install SQLAHelper" or "easy_install SQLAHelper". To check out the development repository: "hg clone http://bitbucket.org/sluggo/sqlahelper SQLAHelper".

1.1.2 SQLAlchemy vocabulary

These are a few SQLAlchemy terms which are critical for understanding SQLAHelper.

An **engine** is a SQLAlchemy object that knows how to connect to a certain database. All SQLAlchemy applications have at least one engine.

A **session** is a SQLAlchemy object that does housekeeping for the object-relational mapper (ORM). These sessions have nothing to do with HTTP sessions despite the identical name. A session is required when using the ORM, but is not needed for lower-level SQL access.

A **contextual session** (often called a **Session** with a capital S, or a **DBSession**) is a threadlocal session proxy. It acts like a session and has the same API, but internally it maintains a separate session for each thread. This allows it to be a golabl variable in multithreaded web applications. (SQLAlchemy manual: contextual session.)

A **declarative base** (often called a **Base**) is a common superclass for all your ORM classes. An ORM class represents one database table, and is associated with a separate table object. An instance of the class represents one record in the table.

Most SQLAlchemy applications nowadays use all of these.

1.1.3 Usage

1. When your application starts up, call `add_engine` once for each database engine you will use. You will first have to create the engine using `sqlalchemy.create_engine()` or `sqlalchemy.engine_from_config()`. See Engine Configuration in the SQLAlchemy manual.
2. In models or views or wherever you need them, access the contextual session, engines, and declarative base this way:

```
import sqlahelper

Session = sqlahelper.get_session()
engine = sqlahelper.get_dbengine()
Base = sqlahelper.get_base()
```

It gets slightly more complex with multiple engines as you'll see below.

1.1.4 API

`sqlahelper.`**`add_engine`**(*engine*, *name='default'*)
Add a SQLAlchemy engine to the engine repository.

The engine will be stored in the repository under the specified name, and can be retrieved later by calling `get_engine(name)`.

If the name is "default" or omitted, this will be the application's default engine. The contextual session will be bound to it, the declarative base's metadata will be bound to it, and calling `get_engine()` without an argument will return it.

`sqlahelper.`**`get_session`**()
Return the central SQLAlchemy contextual session.

To customize the kinds of sessions this contextual session creates, call its `configure` method:

```
sqlahelper.get_session().configure(...)
```

But if you do this, be careful about the 'ext' arg. If you pass it, the ZopeTransactionExtension will be disabled and you won't be able to use this contextual session with transaction managers. To keep the extension active you'll have to re-add it as an argument. The extension is accessible under the semi-private variable `_zte`. Here's an example of adding your own extensions without disabling the ZTE:

```
sqlahelper.get_session().configure(ext=[sqlahelper._zte, ...])
```

`sqlahelper.`**`get_engine`**(*name='default'*)
Look up an engine by name in the engine repository and return it.

If no argument, look for an engine named "default".

Raise `RuntimeError` if no engine under that name has been configured.

`sqlahelper.`**`get_base`**()
Return the central SQLAlchemy declarative base.

`sqlahelper.`**`set_base`**(*base*)
Set the central SQLAlchemy declarative base.

Subsequent calls to `get_base()` will return this base instead of the default one. This is useful if you need to override the default base, for instance to make it inherit from your own superclass.

You'll have to make sure that no part of your application's code or any third-party library calls `get_base()` before you call `set_base()`, otherwise they'll get the old base. You can ensure this by calling `set_base()` early in the application's execution, before importing the third-party libraries.

`sqlahelper.`**`reset`**()
Restore the initial module state, deleting all modifications.

This function is mainly for unit tests and debugging. It undoes all customizations and reverts to the initial module state.

1.1.5 Examples

This application connects to one database. There's only one engine so we make it the default engine.

```
import sqlalchemy as sa
import sqlahelper

engine = sa.create_engine("sqlite:///db.sqlite")
sqlahelper.add_engine(engine)
```

This second application is a typical Pyramid/Pylons/TurboGears application. Its engine args are embedded in a general settings dict, which was parsed from an application-wide INI file. All the values are strings because the INI parser is unaware of the appropriate type for each value.

```
import sqlalchemy as sa
import sqlahelper

settings = {
    "debug_notfound": "false",
    "mako.directories": "myapp:templates",
    "sqlalchemy.url": "sqlite:////home/me/applications/myapp/db.sqlite",
    "sqlalchemy.logging_name": "main",
    "sqlalchemy.pool_size": "10",
    }
engine = sa.engine_from_config(settings, prefix="sqlalchemy.")
sqlahelper.add_engine(engine)
```

The `engine_from_config` method finds the keys with the matching prefix, strips the prefix, converts the values to their proper type, and calls `add_engine` with the extracted arguments. It ignores keys that don't have the prefix. The only required key is the database URL ("sqlalchemy.url" in this case). (Note: type conversion covers only a few most common arguments.)

If `engine_from_config` raises "KeyError: 'pop(): dictionary is empty"', make sure the prefix is correct. In this case it includes a trailing dot.

Multiple databases are covered in the next section.

1.1.6 Multiple databases

A default engine plus other engines

In this scenario, the default engine is used for most operations, but two other engines are also used occasionally:

```
import sqlalchemy as sa
import sqlahelper

# Initialize the default engine.
default = sa.engine_from_config(settings, prefix="sqlalchemy.")
sqlahelper.add_engine(default)

# Initialize the other engines.
engine1 = sa.engine_from_config(settings, prefix="engine1.")
engine2 = sa.engine_from_config(settings, prefix="engine2.")
sqlahelper.add_engine(engine1, "engine1")
sqlahelper.add_engine(engine2, "engine2")
```

Queries will use the default engine by default. To use a different engine you have to use the `bind=` argument on the method that executes the query; or execute low-level SQL directly on the engine (`engine.execute(sql)`).

Two engines, but no default engine

In this scenario, two engines are equally important, and neither is predominent enough to deserve being the default engine. This is useful in applications whose main job is to copy data from one database to another.

```
sqlahelper.add_engine(settings, name="engine1", prefix="engine1.")
sqlahelper.add_engine(settings, name="engine2", prefix="engine2.")
```

Because there is no default engine, queries will fail unless you specify an engine every time using the `bind=` argument or `engine.execute(sql)`.

Different tables bound to different engines

It's possible to bind different ORM classes to different engines in the same database session. Configure your application with no default engine, and then call the Session's `.configure` method with the `binds=` argument to specify which classes go to which engines. For instance:

```
import myapp.models as models

sqlahelper.add_engine(engine1, "engine1")
sqlahelper.add_engine(engine2, "engine2")
Session = sqlahelper.get_session()
binds = {models.Person: engine1, models.Score: engine2}
Session.configure(binds=binds)
```

The keys in the `binds` dict can be SQLAlchemy ORM classes, table objects, or mapper objects.

1.2 Full Changelog

1.2.1 2.0 (unreleased)

- New version 2 API.
- Old version 1 API is supported for backward compatibility.
- Convert repository to Git (from Mercurial).
- Repository is now part of the Pylons project. (https://github.com/Pylons/SQLAHelper).

1.2.2 1.0 (2011-12-25)

- Add `set_base()` function and unit test.
- Change all remaining references to pyramid_sqla to sqlahelper.
- Delete demo application, which was for an old version of Pyramid.

1.2.3 1.0b1 (2011-03-11)

- Remove engine-creation features from `add_engine()`. It was getting too complex to document all the permutations. You'll have to create the engine yourself and pass it to `add_engine`.

1.2.4 Repository SQLAHelper created

- Clone repository 'SQLAHelper' from 'pyramid_sqla'.
- Delete all non-SQLAlchemy code and docs; they've moved to the Akhet package.
- Rename `pyramid_sqla` to `sqlahelper` and change it from a package to a module.
- Move pyramid_sqla/tests/test.py to tests.py.

1.2.5 pylons_sqla-1.0rc2 (2010-02-20, never released)

- `add_static_route` is now a Pyramid config method if you call the new `includeme` function. This is used in the application template.
- Add `pyramid_sqla` as a dependency in the application template.
- Delete websetup.py. Console scripts are more flexible than "paster setup-app".
- Fix but that may have prevented create_db.py from finding the INI stanza.
- 100% test coverage contributed by Chris McDonough.
- Delete unneeded development code in static.py.
- Set Mako's 'strict_undefined' option in the application template.

1.2.6 pyramid_sqla-1.0rc1 (2010-01-26)

- 'pyramid_sqla' application template supports commit veto feature in repoze.tm2 1.0b1.
- Add production.ini to application template.
- Delete stray files in application template that were accidentally included.

1.2.7 pyramid_sqla-0.2 (2011-01-19)

- Pyramid 1.0a10 spins off view handler support to 'pyramid_handlers' package.
- 'pyramid_sqla' application template depends on Pyramid>=1.0a10.

1.2.8 pyramid_sqla-0.1 (2011-01-12)

- Initial release.
- Warning: a change in Pyramid 1.0a10 broke applications created using the this version's application template. To run existing applications under Pyramid 1.0a10 and later, add a 'pyramid_handlers' dependency to the `requires` list in setup.py and reinstall the application.

A

G

R

S