sqlservice Documentation

Release 1.1.3

Derrick Gilland

Sep 27, 2018

Contents

The missing SQLAlchemy ORM interface.

CHAPTER 1

Links

- Project: https://github.com/dgilland/sqlservice
- Documentation: http://sqlservice.readthedocs.io
- PyPI: https://pypi.python.org/pypi/sqlservice/
- TravisCI: https://travis-ci.org/dgilland/sqlservice

CHAPTER 2

Introduction

So what exactly is `sqlservice` and what does "the missing SQLAlchemy ORM interface" even mean? SQLAlchemy is a fantastic library and features a superb ORM layer. However, one thing SQLAlchemy lacks is a unified interface for easily interacting with your database through your ORM models. This is where `sqlservice` comes in. It's interface layer on top of SQLAlchemy's session manager and ORM layer that provides a single point to manage your database connection/session, create/reflect/drop your database objects, and easily persist/destroy model objects.

2.1 Features

This library is meant to enhance your usage of SQLAlchemy. SQLAlchemy is great and this library tries to build upon that by providing useful abstractions on top of it.

- Database client that helps manage an ORM scoped session.
- Base class for a declarative ORM Model that makes updating model columns and relationships easier and converting to a dictionary a breeze.
- Decorator-based event register for SQLAlchemy ORM events that can be used at the model class level. No need to register the event handler outside of the class definition.
- An application-side nestable transaction context-manager that helps implement pseudo-subtransactions for those that want implicit transaction demarcation, i.e. session autocommit, without using session subtransactions.
- And more!

2.2 Requirements

- Python >= 3.4
- SQLAlchemy >= 1.0.0
- pydash >= 3.4.3

CHAPTER 3

Quickstart

First, install using pip:

```
pip3 install sqlservice
```

Then, define some ORM models:

```
import re

from sqlalchemy import Column, ForeignKey, orm, types

from sqlservice import declarative_base, event


Model = declarative_base()

class User(Model):
    __tablename__ = 'user'

    id = Column(types.Integer(), primary_key=True)
    name = Column(types.String(100))
    email = Column(types.String(100))
    phone = Column(types.String(10))

    roles = orm.relation('UserRole')

    @event.on_set('phone', retval=True)
    def on_set_phone(self, value, oldvalue, initator):
        # Strip non-numeric characters from phone number.
        return re.sub('[^0-9]', '', value)

class UserRole(Model):
    __tablename__ = 'user_role'

    id = Column(types.Integer(), primary_key=True)
```

(continues on next page)

(continued from previous page)

```
    user_id = Column(types.Integer(), ForeignKey('user.id'), nullable=False)
    role = Column(types.String(25), nullable=False)
```

Next, configure the database client:

```
from sqlservice import SQLClient

config = {
    'SQL_DATABASE_URI': 'sqlite:///db.sql',
    'SQL_ISOLATION_LEVEL': 'SERIALIZABLE',
    'SQL_ECHO': True,
    'SQL_ECHO_POOL': False,
    'SQL_CONVERT_UNICODE': True,
    'SQL_POOL_SIZE': 5,
    'SQL_POOL_TIMEOUT': 30,
    'SQL_POOL_RECYCLE': 3600,
    'SQL_MAX_OVERFLOW': 10,
    'SQL_AUTOCOMMIT': False,
    'SQL_AUTOFLUSH': True,
    'SQL_EXPIRE_ON_COMMIT': True
}

db = SQLClient(config, model_class=Model)
```

Prepare the database by creating all tables:

```
db.create_all()
```

Finally (whew!), start interacting with the database.

Insert a new record in the database:

```
data = {'name': 'Jenny', 'email': 'jenny@example.com', 'phone': '555-867-5309'}
user = db.User.save(data)
```

Fetch records:

```
assert user is db.User.get(data.id)
assert user is db.User.find_one(id=user.id)
assert user is db.User.find(User.id == user.id)[0]
```

Serialize to a `dict`:

```
assert user.to_dict() == {'id': 1,
                          'name': 'Jenny',
                          'email': 'jenny@example.com',
                          'phone': '5558675309'}

assert dict(user) == user.to_dict()
```

Update the record and save:

```
user.phone = '222-867-5309'
db.User.save(user)
```

Upsert on primary key automatically:

```
assert user is db.User({'id': 1,
                        'name': 'Jenny',
                        'email': 'jenny@example.com',
                        'phone': '5558675309'})
```

Destroy the model record:

```
db.User.destroy(user)
# OR db.User.destroy([user])
# OR db.User.destroy(user.id)
# OR db.User.destroy([user.id])
# OR db.User.destroy(dict(user))
# OR db.User.destroy([dict(user)])
```

For more details, please see the full documentation at http://sqlservice.readthedocs.io.

CHAPTER 4

Guide

4.1 Installation

SQLService requires Python >= 3.4.

To install from PyPI:

```
pip install sqlservice
```

4.2 Model

The `Model` is the basic ORM class that represents your database schema. SQLAlchemy provides a very basic default ORM model class when one calls `sqlalchemy.ext.declarative.declarative_base`. SQLService does one better and provides the basic tools for handling your basic use cases.

The general approach to using `sqlservice.ModelBase` is to use it as the base class for your own custom `Model` class that extends/overrides `ModelBase` to fit your specific needs.

```
# in models/base.py
from sqlalchemy import MetaData
from sqlservice import ModelBase, as_declarative, declarative_base

metadata = MetaData()

@as_declarative(metadata=metadata)
class Model(ModelBase):
    pass

# Or using the declarative_base function...
# Model = declarative_base(ModelBase, metadata=metadata)
```

Note: All keyword arguments to `sqlservice.as_declarative` or `sqlservice.declarative_base` will be passed to `sqlalchemy.ext.declarative.declarative_base`.

From there you can use `Model` as the base class for your ORM model classes.

```
# in models/user.py
from sqlalchemy import Column, types

from .base import Model

class User(Model):
    __tablename__ = 'user'

    id = Column(types.Integer(), primary_key=True)
    name = Column(types.String(100))
    email = Column(types.String(100))
```

4.2.1 Instantiation and Updating

What does `ModelBase` provide for you? Out of the box, you'll be able to do things like:

Create a new instance from a `dict` or keyword arguments:

```
user = User({'name': 'Bob', 'email': 'bob@example.com})
user = User(name='Bob', email='bob@example.com')
```

Note: Under the hood `ModelBase.__init__` calls `update()` so anything `update()` does, `__init__` does too.

Update using attribute or item setters:

```
user.name = 'Bob Paulson'
user['name'] = 'Robert Paulson'
```

Update an instance using a `dict` or keyword arguments:

```
user.update(name='Bob Smith')
user.update({'email': 'bobsmith@example.com'})
```

The `update()` method is powerful enough to work with relationships and nested relationships. Consider the following:

```
# in models/user.py
from sqlalchemy import Column, ForeignKey, types, orm

from .base import Model

class User(Model):
    __tablename__ = 'user'

    id = Column(types.Integer(), primary_key=True)
    name = Column(types.String(100))
    email = Column(types.String(100))
```

(continues on next page)

(continued from previous page)

```
    about = orm.relation('UserAbout', uselist=False)
    devices = orm.relation('UserDevice')

class UserAbout(Model):
    __tablename__ = 'user_about'

    user_id = Column(types.Integer(), ForeignKey('user.id'), primary_key=True)
    nickname = Column(types.String(100))
    hometown = Column(types.String(100))

class UserDevice(Model):
    __tablename__ = 'user_device'

    id = Column(types.Integer(), primary_key=True)
    user_id = Column(types.Integer(), ForeignKey('user.id'), nullable=False)
    name = Column(types.String(100))

    keys = orm.relation('UserDeviceKey')

class UserDeviceKey(Model):
    __tablename__ = 'user_device_key'

    id = Column(types.Integer(), primary_key=True)
    device_id = Column(types.Integer(),
                       ForeignKey('user_device.id'),
                       nullable=False))
    key = Column(types.String(100))
```

You can now easily create a user, user devices, and device keys with a single data structure without having to use the relationship classes directly.

```
data = {
    'name': 'Bob Smith',
    'email': 'bobsmith@example.com',
    'about': {
        'nickname': 'Bobby',
        'hometown': 'Example City'
    },
    'devices': [
        {'name': 'device1', 'keys': [{'key': 'key1a'}, {'key': 'key1b'}]},
        {'name': 'device2', 'keys': [{'key': 'key2a'}, {'key': 'key2b'}]}
    ]
}
user = User(data)

user
# <User(id=None, name='Bob Smith', email='bobsmith@example.com')>

user.about
# <UserAbout(user_id=None, nickname='Bobby', hometown='Example City')>

user.devices
# [<UserDevice(id=None, user_id=None, name='device1')>,
   <UserDevice(id=None, user_id=None, name='device2')>]
```

(continues on next page)

(continued from previous page)

```
user.devices[0].keys
# [<UserDeviceKey(id=None, device_id=None, key='key1a')>,
   <UserDeviceKey(id=None, device_id=None, key='key1b')>]

user.devices[1].keys
# [<UserDeviceKey(id=None, device_id=None, key='key2a')>,
   <UserDeviceKey(id=None, device_id=None, key='key2b')>]
```

This is because `ModelBase.update()` works really hard to map `dict` keys to the correct relationship model class to automatically create new model instances from those `dict` objects. It works for relationships that are `1:1` or `1:M`.

In addition, when you update the model with relationship data, it will nest calls to the relationship class' `update()` methods.

```
user.update({'about': {'nickname': 'Bo'}})
user.about
# <UserAbout(user_id=None, nickname='Bo', hometown='Example City')>
```

Warning: Depending on whether you've set up relationship cascades, calling `update()` on relationships can result in integrity errors since SQLAlchemy will nullify orphaned relationship models when they are replaced.

```
user.update({'devices': [{'name': 'device3'}]})
db.save(user)

# sqlalchemy.exc.IntegrityError: (raised as a result of Query-invoked autoflush;
# consider using a session.no_autoflush block if this flush is occurring
# prematurely) (sqlite3.IntegrityError) NOT NULL constraint failed:
# user_device.user_id [SQL: 'UPDATE user_device SET user_id=? WHERE
# user_device.id = ?'] [parameters: ((None, 1), (None, 2))]
```

4.2.2 Dictionary Serialization

Want to serialize your models to `dict` objects?

```
user.to_dict()
dict(user)
# {'id': 1,
   'name': 'Bob Smith',
   'email': 'bobsmith@example.com',
   'about': {'nickname': 'Bo', 'hometown': 'Example City'},
   'devices': [{'id': 1, 'name': 'device1', 'user_id': 1}, {'id': 2, 'name': 'device2
↪', 'user_id': 1}]}
```

As you can see, relationships are serialized too.

But how does this handle lazy loaded models? When serializing the only data that is serialized is what is already loaded. This is done to avoid triggerring a large number of individual queries on lazily loaded attributes. Essentially, `Model.to_dict()` only looks at what's already present in `user.__dict__` and never touches any attributes directly (which could lead to additional queries). So it's up to you to ensure that your model is loaded with the data you want to be serialized before calling `to_dict()`.

Need to serialize certain types differently? Add some adapters using `__dict_args__` class attribute:

```
class User(Model):
    ...
    __dict_args__ = {
        'adapters': {
            UserAbout: lambda model, *_: {'nickname': model.nickname},
            # identical to above but using string name for Model...
            # 'UserAbout': lambda model, *_: {'nickname': model.nickname},
            'devices': lambda devices, *_: [(device.name, device.keys) for device in
↪devices],
            list: lambda items, col, *_: [item.name for item in items],
            (int, str): lambda value, col: col + ':' + str(value)
        }
    }

dict(user)
# {'id': 'id:1',
   'name': 'name:Bob Smith',
   'email': 'email:bobsmith@example.com',
   'about': {'nickname': 'Bo'},
   'devices': [('device1', ['key1a', 'key1b']),
               ('device2', ['key2a', 'key2b'])]}
```

The `adapaters` argument is expected to be a mapping to serializers where each key can be one of:

- model class object (e.g. `UserAbout`)
- string name of model class (e.g. `'UserAbout'`)
- string name of model attribute (e.g. `'about'` which corresponds to `User.about`)
- other type (e.g. `list`, `int`, `str`, etc.)
- tuple of types (e.g. `(int, float)`)
- `None` to exclude the key from serialization.

The serializer should be a callable that accepts three arguments: `(value, column, model_instance)` (the arguments passed in are based on the function definition and are automatically detected). The adapter serializer used when it's key matches the matches the value's type, descriptor name, or model class name. For relationships defined as a `list` or other list-like structure, the relationship class' `__dict_args__` will be used during nested serialization. If you need to reference classes that aren't defined yet (e.g. other model classes), you can make `__dict_args__` a `@property` or use the string class name if it's another model class.

4.2.3 Object Identity

You can get the primary key identity of any model object:

```
user.identity()
# 1
```

Note: If the model has multiple primary keys, a tuple is returned

4.2.4 Class Methods and Properties

The `Model` class includes other useful methods as well:

```
User.class_mapper()
# <Mapper at 0x7fd9e7443b70; User>

User.columns()
# (Column('id', Integer(), table=<user>, primary_key=True, nullable=False),
   Column('name', String(length=100), table=<user>),
   Column('email', String(length=100), table=<user>))

User.pk_columns()
# (Column('id', Integer(), table=<user>, primary_key=True, nullable=False),)

User.relationships()
# (<RelationshipProperty at 0x7fd9ead007b8; about>,
   <RelationshipProperty at 0x7fd9e7421f28; devices>)

for descriptor in User.descriptors():
    (str(descriptor), repr(descriptor))
# User.about, <sqlalchemy.orm.attributes.InstrumentedAttribute object at␣
↪0x7fd9e743f728>
# User.devices, <sqlalchemy.orm.attributes.InstrumentedAttribute object at␣
↪0x7fd9e743f780>
# User.name, <sqlalchemy.orm.attributes.InstrumentedAttribute object at␣
↪0x7fd9e743f938>
# User.email, <sqlalchemy.orm.attributes.InstrumentedAttribute object at␣
↪0x7fd9e743f9e8>
# User.id, <sqlalchemy.orm.attributes.InstrumentedAttribute object at 0x7fd9e743f888>
```

4.3 Event

SQLAlchemy features an ORM event API but one thing that is lacking is a way to register event handlers in a declarative way inside the Model's class definition. To bridge this gap, this module contains a collection of decorators that enable this kind of functionality.

Instead of having to write event registration like this:

```
from sqlalchemy import event

from myproject import Model

class User(Model):
    _id = Column(types.Integer(), primary_key=True)
    email = Column(types.String())

def set_email_listener(target, value, oldvalue, initiator):
    print 'received "set" event for target: {0}'.format(target)
    return value

def before_insert_listener(mapper, connection, target):
    print 'received "before_insert" event for target: {0}'.format(target)

event.listen(User.email, 'set', set_email_listener, retval=True)
event.listen(User, 'before_insert', before_insert_listener)
```

Model Events allows one to write event registration more succinctly as:

```
from sqlservice import event

from myproject import Model

class User(Model):
    _id = Column(types.Integer(), primary_key=True)
    email = Column(types.String())

    @event.on_set('email', retval=True)
    def on_set_email(target, value, oldvalue, initiator):
        print 'received set event for target: {0}'.format(target)
        return value

    @event.before_insert()
    def before_insert(mapper, connection, target):
        print ('received "before_insert" event for target: {0}'
               .format(target))
```

For details on each event type's expected function signature, see SQLAlchemy's ORM Events.

For a full listing of sqlservice event decorators, see the *sqlservice.event*.

4.4 Client

The heart and soul of sqlservice is the *sqlservice.client* module. It is a gateway to your databases that seamlessly integrates with your declarative Model's metadata to provide an abstraction layer on top of it.

Before we get to the good stuff, let's first start by creating our client database object:

```
from sqlservice import SQLClient

# Let's assume you've define the User, UserAbout, UserDevice, and UserDeviceKey
# as illustated in the "Model" section of the docs in a module called models.py
# with a declarative base named Model.
from models import Model

config = {'SQL_DATABASE_URI': 'sqlite://'}
db = SQLClient(config, model_class=Model)
```

Note: For details on the available configuration values, see *sqlservice.client.SQLClient*.

Behind the scenes when `SQLClient` is instantiated, several important steps take place:

- A SQLAlchemy engine object is created at `db.engine`.
- A thread-local, scoped ORM session is created at `db.session`.
- Instances of `sqlservice.service.SQLService` are created for all declarative model classes associated with `Model` that are accessible at `db.<model_class>` (more on this below).

4.4.1 Engine

Nothing fancy here. The `db.engine` is created using `sqlalchemy.create_engine`.

4.4.2 Session

The ORM session at `db.session` is actually a proxy to the object returned by `sqlalchemy.orm.scoped_session(session_factory)()` which is the thread-local session object. This proxy attribute is used so that `db.session` always returns an instance of the `Session` class used by `orm.sessionmaker`.

You can directly access the session at `db.session`, but there are several convenience proxy attributes available at the top-level of each `SQLClient` instance:

SQL-Client	Session Object	Description
db.add	session.add	Add an ORM object to the session.
db.add_all	session.add_all	Add a list of ORM objects to the session.
db.delete	session.delete	Mark ORM object for deletion.
db.merge	session.merge	Merge an existing ORM instance with one loaded from the database.
db.execute	session.execute	Execute SQL statement.
db.close	session.close	Close the session. This will rollback any open transactions.
db.flush	session.flush	Flush the session.
db.refresh	session.refresh	Refresh an ORM object by querying the database.
db.commit	session.commit	Commit the session. **NOTE:** The commit call is wrapped in a try/except block that calls `db.session.rollback()` on exception before re-raising.
db.rollback	session.rollback	Rollback the session.
db.query	session.query	Perform an ORM query.

In addition, there are also proxies to the scoped session object:

SQLClient	Scoped Session Object	Description
db.remove	Session.remove	Dispose of the current scoped session.
db.close_all	Session.close_all	Close all thread-local session objects.

And if you want to completely close all sessions and terminate the engine connection, use `disconnect()`, which will close all thread-local sessions, dispose of the scoped session, and dispose of the engine connection:

```
db.disconnect()
```

4.4.3 Session Query

The default query class for `db.query`/`db.session.query` uses *sqlservice.query.SQLQuery* which provides additional methods beyond SQLAlchemy's base query class.

You can paginate results with `paginate()`:

```
# Return the first 25 results
db.query(User).paginate(25)
db.query(User).paginate((25, 1))

# Return the second 25 results
db.query(User).paginate((25, 2))
```

You can filter, paginate, and order results in a single method call with `search()`:

```
# Criteria is passed in by position and can be a dict-mapping to query.filter_by()
# or a query expression.
db.query(User).search({'name': 'Bob'}, User.email.like('%@gmail.com')).all()

# Pagination and ordering is by keyword argument.
db.query(User).search(per_page=25, page=2, order_by=User.name).all()
```

For more details, see the *sqlservice.query* module.

4.4.4 Application-Level Nested Transactions

Some times you may find yourself with several methods that are all self-contained within a transaction:

```
def insert_company(db, data):
    with db.transaction():
        db.save(Company(data))

def insert_company_ledger(db, data):
    with db.transaction():
        db.save(CompanyLedger(data))

def insert_initial_order(db, data):
    with db.transaction():
        db.save(Order(data))
```

In all cases, you want to ensure that any of these methods called in isolation will take place within a database transaction. But in addition, you want any combination of these function calls to also be within a single transaction and not in three separate transactions. Essentiall you want behavior like the following:

```
def create_company(db, data):
    with db.transaction():
        insert_company(db, data['company'])

def create_company_and_ledger(db, data):
    with db.transaction():
        insert_company(db, data['company'])
        insert_company_ledger(db, data['ledger'])

def create_company_and_ledger_and_order(db, data):
    with db.transaction():
        insert_company(db, data['company'])
        insert_company_ledger(db, data['ledger'])
        insert_initial_order(db, data['ledger'])
```

But you don't want each transaction context to commit if it's a nested transaction.

Not to worry because that's exactly how `db.transaction` works. It maintains a session-local transaction count based on the number of times `db.transaction` is called so that there will only be a single commit in the top-most

transaction context. This means you can define small, transactionally safe functions that can be used on their own or combined with others into larger transactions without having to worry about any of the nested transactions from committing.

4.4.5 ORM Models

Whenever the declarative base Model is passed into `SQLCLient`, its metadata is available at `db.metadata`. Several `metadata` based methods are then accessible.

Create Model Tables

Create all ORM model tables with:

```
db.create_all()
```

This will issue the appropriate SQL DDL statments that can get your database up and running quickly. For full migration integration, see alembic.

Drop Model Tables

Drop all ORM model tables with:

```
db.drop_all()
```

Reflect Models

Reflect existing database schema without predefining ORM models or Table objects:

```
db.reflect()
print(db.tables)
```

4.4.6 ORM Model Queries

ORM model queries are accessible via attribute access which provides a shorthand for `db.query(<ModelClass>)`:

```
db.User.<ModelClass>
```

So now you can easily query models:

```
users = db.User.filter(User.name.like('Mc%')).all()
```

You can save a model:

```
# Using a dict.
user = db.User.save({'name': 'Elliot', 'email': 'mr@example.com'})

# Using a model.
user['name'] += ' Alderson'
db.User.save(user)
```

(continues on next page)

(continued from previous page)

```
# Using multiple dicts and models.
users = db.User.save([{...}, {...}, User(...), User(...)])
```

You can destroy a model:

```
# Using a primary key value.
db.User.destroy(134)

# Using a dict with the primary key.
db.User.destroy({'id': 134})

# Using a model.
db.User.destroy(user)

# Using multiple values.
db.User.destroy([134, {'id': 135}, user])
```

For more details, see the *sqlservice.query* module.

4.4.7 Generic ORM Model Methods

While working with model services is the recommended way to interact with ORM models, you can save and destroy any ORM model using the `db.save()` and `db.destroy()` methods directly.

save()

You can save any ORM model instance with `db.save()`:

```
# Save a single user
db.save(user1)

# Define before/after functions around saving a user.
def before_save_user(model, is_new):
    pass

def after_save_user(model, is_new):
    pass

# Save a single user while calling before_save_user() before user is saved
# and after_save_user() after user is saved.
db.save(user1, before=before_save_user, after=after_save_user)

# Save multiple models.
# NOTE: If before/after supplied, it will be called for each individual model
# saved.
db.save([user1, user2, company1, company2])
```

When saving the SQL client will perform an upsert using the primary key values (if set) of the model(s) being saved. As a result of this, a database query will be issued to select any existing records that may match the models being saved based on their primary key values. This allows you to save model objects that are not yet associated with the SQLAlchemy session's identity map without having to first fetch the object.

This behavior can be overridden by supplying a custom "identity" function that will be applied to the model(s) being saved. The "identity" function must accept a single argument, a model, and return an identity mapping tuple where

each tuple item is a 2-element tuple containing a model column object and its value.

For example, if we wanted to upsert using a user's email address, then the identity function would be:

```
def user_identity_by_email(model):
    return ((User.email, model.email),)
```

If you wanted to upsert using a combination of the user's email address and their name, then the identity function function would be:

```
def user_identity_by_email_name(model):
    return ((User.email, model.email),
            (User.name, model.name))
```

You would then pass one of these functions to `save()`:

```
db.save(user, identity=user_identity_by_email)
```

This effectively allows you to easily create your own upsert methods independent of the database-backend.

destroy()

```
# Destroy a single user.
db.destroy(user1)

# Destroy multiple models.
db.destroy([user1, user2, company1, company2])

# Destroy using primary key only.
db.destroy(3618, model_class=User)
db.destroy(3618, model_class=User, synchronize_session=True)
```

4.4.8 Bulk Inserts and Updates

Several bulk methods are available on `SQLClient`:

- *sqlservice.client.SQLClient.bulk_insert()*: Similar to `session.bulk_insert_mappings` except it supports SQLAlchemy insert-statement objects for the mapper value.
- *sqlservice.client.SQLClient.bulk_insert_many()*: Like *sqlservice.client.SQLClient.bulk_insert()* expect that it will use a multi-row VALUES clause for a single INSERT statement instead of the DBAPI `executemany()` method.
- *sqlservice.client.SQLClient.bulk_common_update()*: Group common updates by a set of columns that define the row identity (typically primary keys) into the fewest number of update statements.
- *sqlservice.client.SQLClient.bulk_diff_update()*: Given a list of previous mappings with old values and a list of current mappings with new values, only update or insert rows that have changed.

Each of these methods expects at least a mapper (e.g. ORM object class) and a list of mappings (list of dictionaries).

```
# Bulk insert
db.bulk_insert(User, [{'name': 'aaa'}, {'name': 'bbb'}, {'name': 'ccc'}])

# Bulk insert many
```

(continues on next page)

(continued from previous page)

```
db.bulk_insert_many(User, [{'name': 'aaa'}, {'name': 'bbb'}, {'name': 'ccc'}])

# Bulk common update
# would result in 2 UPDATE statements where ids 1+2 and 3+4 are grouped together.
db.bulk_common_update(User,
                      User.id,
                      [{'id': 1, 'phone': '1234567890'},
                       {'id': 2, 'phone': '1234567890'},
                       {'id': 3, 'phone': '0987654321'},
                       {'id': 4, 'phone': '0987654321'}])

# Bulk diff update
# would result in 2 UPDATE statements [(id=1, name='AA'), (id=2, name='CC')]
# and 1 INSERT statement [(id=4, name='D')].
db.bulk_diff_update(User,
                    User.id,
                    previous_mappings=[{'id': 1, 'name': 'A'},
                                       {'id': 2, 'name': 'B'},
                                       {'id': 3, 'name': 'C'}],
                    mappings=[{'id': 1, 'name': 'AA'},
                              {'id': 2, 'name': 'B'},
                              {'id': 3, 'name': 'CC'},
                              {'id': 4, 'name': 'D'])
```

4.5 API Reference

The sqlservice package imports commonly used objects into it's top-level namespace:

```
from sqlservice import (
    ModelBase,
    SQLClient,
    SQLQuery,
    as_declarative,
    declarative_base,
    destroy,
    event,
    make_identity,
    save,
    transaction)
```

4.5.1 Client

The database client module.

class sqlservice.client.**SQLClient**(*config=None*, *model_class=None*, *query_class=<class 'sqlservice.query.SQLQuery'>*, *session_class=<class 'sqlalchemy.orm.session.Session'>*, *session_options=None*, *engine_options=None*)

Database client for interacting with a database.

The following configuration values can be passed into a new *SQLClient* instance as a `dict`. Alternatively, this class can be subclassed and `DEFAULT_CONFIG` overridden with a custom defaults.

SQL_DATABASE_URI	URI used to connect to the database. Defaults to `sqlite://`.
SQL_ECHO	When `True` have SQLAlchemy log all SQL statements. Defaults to `False`.
SQL_ECHO_POOL	When `True` have SQLAlchemy log all log all checkouts/checkins of the connection pool. Defaults to `False`.
SQL_ENCODING	The string encoding used by SQLAlchemy for string encode/decode operations which occur within SQLAlchemy, outside of the DBAPI. Defaults to *utf-8*.
SQL_CONVERT_UNICODE	When `True` sets the default behavior of `convert_unicode` on the `String` type to `True`, regardless of a setting of `False` on an individual `String` type, thus causing all `String` -based columns to accommodate Python unicode objects.
SQL_ISOLATION_LEVEL	String parameter interpreted by various dialects in order to affect the transaction isolation level of the database connection. The parameter essentially accepts some subset of these string arguments: `"SERIALIZABLE"`, `"REPEATABLE_READ"`, `"READ_COMMITTED"`, `"READ_UNCOMMITTED"` and `"AUTOCOMMIT"`. Behavior here varies per backend, and individual dialects should be consulted directly. Defaults to `None`.
SQL_POOL_SIZE	The size of the database pool. Defaults to the engine's default (usually `5`).
SQL_POOL_TIMEOUT	Specifies the connection timeout for the pool. Defaults to `10`.
SQL_POOL_RECYCLE	Number of seconds after which a connection is automatically recycled.
SQL_MAX_OVERFLOW	Controls the number of connections that can be created after the pool reached its maximum size. When those additional connections are returned to the pool, they are disconnected and discarded.
SQL_AUTOCOMMIT	When `True` the `Session` does not keep a persistent transaction running, and will acquire connections from the engine on an as-needed basis, returning them immediately after their use. Defaults to `False`.
SQL_AUTOFLUSH	When `True`, all query operations will issue a `flush()` call to the `Session` before proceeding. This is a convenience feature so that `flush()` need not be called repeatedly in order for database queries to retrieve results.
SQL_EXPIRE_ON_COMMIT	When `True` all instances will be fully expired after each `commit()`, so that all attribute/object access subsequent to a completed transaction will load from the most recent database state. Defaults to `True`.
SQL_POOL_PRE_PING	When `True` will enable SQLAlchemy's connection pool "pre-ping" feature that tests connections for liveness upon each checkout. Defaults to `False`. Requires SQLAlchemy >= 1.2.

Parameters

- **config** (*dict|str*) – Database engine configuration options or database URI string. Defaults to `None` which uses an in-memory SQLite database.
- **model_class** (*object*) – A SQLAlchemy ORM declarative base model.
- **query_class** (*Query, optional*) – SQLAlchemy Query derived class to use as the default class when creating a new query object.
- **session_class** (*Session, optional*) – SQLAlchemy Session derived class to use by the session maker.
- **session_options** (*dict, optional*) – Additional session options use when creating the database session.

add
Proxy property to `session.add()`.

add_all
Proxy property to `session.add_all()`.

bulk_common_update(*mapper*, *key_columns*, *mappings*)
Perform a bulk UPDATE on common shared values among *mappings*. What this means is that if multiple

records are being updated to the same values, then issue only a single update for that value-set using the identity of the records in the WHERE clause.

Parameters

- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **key_columns** (*tuple*) – A tuple of SQLAlchemy columns that represent the identity of each row (typically this would be a table's primary key values but they can be any set of columns).
- **mappings** (*list*) – List of `dict` objects to update.

Returns list[ResultProxy]

bulk_diff_update(*mapper*, *key_columns*, *previous_mappings*, *mappings*)

Perform a bulk INSERT/UPDATE on the difference between *mappings* and *previous_mappings* such that only the values that have changed are included in the update. If a mapping in *mappings* doesn't exist in *previous_mappings*, then it will be inclued in the bulk INSERT. The bulk INSERT will be deferred to *bulk_insert()*. The bulk UPDATE will be deferred to *bulk_common_update()*.

Parameters

- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **mappings** (*list*) – List of `dict` objects to update.
- **previous_mappings** (*list*) – List of `dict` objects that represent the previous values of all mappings found for this update set (i.e. these are the current database records).
- **key_columns** (*tuple*) – A tuple of SQLAlchemy columns that represent the identity of each row (typically this would be a table's primary key values but they can be any set of columns).

Returns list[ResultProxy]

bulk_insert(*mapper*, *mappings*)

Perform a bulk insert into table/statement represented by *mapper* while utilizing a special syntax that replaces the tradtional `executemany()` DBAPI call with a multi-row VALUES clause for a single INSERT statement.

See *bulk_insert_many()* for bulk inserts using `executemany()`.

Parameters

- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **mappings** (*list*) – List of `dict` objects to insert.

Returns ResultProxy

bulk_insert_many(*mapper*, *mappings*)

Perform a bulk insert into table/statement represented by *mapper* while utilizing the `executemany()` DBAPI call.

See *bulk_insert()* for bulk inserts using a multi-row VALUES clause for a single INSERT statement.

Parameters

- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **mappings** (*list*) – List of `dict` objects to insert.

Returns ResultProxy

bulk_insert_mappings
Proxy property to `session.bulk_insert_mappings()`.

bulk_save_objects
Proxy property to `session.bulk_save_objects()`.

bulk_update_mappings
Proxy property to `session.bulk_update_mappings()`.

close
Proxy property to `session.close()`.

close_all
Proxy property to `_Session.close_all()`.

commit()
Commit a session transaction but rollback if an error occurs. This helps ensure that the session is not left in an unsable state.

create_all()
Create all metadata (tables, etc) contained within *metadata*.

create_engine(*uri*, *options=None*)
Factory function to create a database engine using *config* options.

Parameters **config** (*dict*) – Database client configuration.

Returns SQLAlchemy engine instance.

Return type Engine

create_models_registry(*model_class*)
Return model registry `dict` with model names as keys and corresponding model classes as values.

create_session(*bind*, *options=None*, *session_class=<class 'sqlalchemy.orm.session.Session'>*, *query_class=<class 'sqlservice.query.SQLQuery'>*)
Factory function to create a scoped session using *bind*.

Parameters

- **bind** (*Engine|Connection*) – Database engine or connection instance.
- **options** (*dict, optional*) – Session configuration options.
- **session_class** (*obj, optional*) – Session class to use when creating new session instances. Defaults to `Session`.
- **query_class** (*obj, optional*) – Query class used for `session.query` instances. Defaults to *SQLQuery*.

Returns SQLAlchemy session instance bound to *bind*.

Return type Session

database
Proxy property to database engine's database name.

delete
Proxy property to `session.delete()`.

destroy(*data*, *model_class=None*, *synchronize_session=False*)
Delete bulk records from *data*.

The *data* argument can be any of the following:

- `dict`

- `model_class` instance
- `list`/`tuple` of `dict` objects
- `list`/`tuple` of `model_class` instances

If a `dict` or `list` of `dict` is passed in, then *model_class* must be provided

Parameters

- **data** (*mixed*) – Data to delete from database.
- **synchronize_session** (*bool | str*) – Argument passed to `Query.delete`.

Returns Number of deleted records.

Return type int

disconnect()
Disconnect all database sessions and connections.

drop_all()
Drop all metadata (tables, etc) contained within *metadata*.

execute
Proxy property to `session.execute()`.

expire
Proxy property to `session.expire()`.

expire_all
Proxy property to `session.expire()`.

expunge(**instances*)
Remove all *instances* from *session*.

expunge_all
Proxy property to `session.expunge()`.

flush
Proxy property to `session.flush()`.

get_metadata()
Return *MetaData* from `model` or raise an exception if `model` was never given.

invalidate
Proxy property to `session.invalidate()`.

is_active
Proxy property to `session.is_active`.

is_modified
Proxy property to `session.is_modified()`.

make_engine_options(*extra_options=None*)
Return engine options from `config` for use in `sqlalchemy.create_engine`.

make_session_options(*extra_options=None*)
Return session options from `config` for use in `sqlalchemy.orm.sessionmaker`.

merge
Proxy property to `session.merge()`.

metadata
Return *MetaData* from `model` or `None`.

no_autoflush
Proxy property to `session.no_autoflush()`.

ping()
Ping the database to check whether the connection is valid.

Returns `True` when connection check passes.

Return type bool

Raises `sqlalche.exc.SQLAlchemyError` – When the connection check fails.

prepare
Proxy property to `session.prepare()`.

prune
Proxy property to `session.prune()`.

query
Proxy property to `session.query()`.

reflect()
Reflect tables from database into *metadata*.

refresh
Proxy property to `session.refresh()`.

remove
Proxy propery to `_Session.remove()`.

rollback
Proxy property to `session.rollback()`.

save(*models*, *before=None*, *after=None*, *identity=None*)
Save *models* into the database using insert, update, or upsert-on-primary-key.

The *models* argument can be any of the following:

- Model instance
- `list`/`tuple` of Model instances

Parameters

- **models** (*mixed*) – Models to save to database.
- **before** (*function, optional*) – Function to call before each model is saved via `session.add`. Function should have signature `before(model, is_new)`.
- **after** (*function, optional*) – Function to call after each model is saved via `session.add`. Function should have signature `after(model, is_new)`.
- **identity** (*function, optional*) – Function used to return an idenity map for a given model. Function should have the signature `identity(model)`. By default *core.primary_identity_map()* is used.

Returns If a single item passed in. list: A `list` of Model instaces if multiple items passed in.

Return type Model

scalar
Proxy property to `session.scalar()`.

session
Proxy to threadlocal session object returned by scoped session object.

Note: Generally, the scoped session is sufficient to work with directly. However, the scoped session doesn't provide access to the custom session class used by the session factory. This property returns an instance of our custom session class. Multiple calls to the scoped session always returns the same active theadlocal session (i.e. `self._Session() is self._Session()`).

See also:

http://docs.sqlalchemy.org/en/latest/orm/contextual.html

tables
Return `dict` of table instances found in *`metadata`* with table names as keys and corresponding table objects as values.

transaction(*commit=True*, *rollback=False*, *autoflush=None*)
Nestable session transaction context manager where only a single commit will be issued once all contexts have been exited. If an exception occurs either at commit time or before, the transaction will be rolled back and the original exception re-raised.

Parameters

- **commit** (*bool, optional*) – Whether to commit the transaction or leave it open. Defaults to `True`.
- **rollback** (*bool, optional*) – Whether to rollback the transaction. Defaults to `False`. WARNING: This overrides *commit*.
- **autoflush** (*bool, optional*) – Whether to override `session.autoflush`. Original `session.autoflush` will be restored after transaction. Defaults to `None` which doesn't modify `session.autoflush`.

Yields *`session`*

update_models_registry()
Update `models` registry as computed from `model_class`.

url
Proxy property to database engine's database URL.

4.5.2 Query

The query module.

class sqlservice.query.**SQLQuery**(*entities*, *session=None*)
Extended SQLAlchemy query class.

all_entities
Return list of all entities for query.

all_model_classes
Return list of all model classes for query.

destroy(*data*, *synchronize_session=False*)
Delete bulk records identified by *data*.

> **Warning:** This requires that the `Query` has been generated using `Query(<ModelClass>)`; otherwise, and exception will be raised.

The *data* argument can be any of the following:

- `dict`
- *model_class* instance
- `list`/`tuple` of `dict` objects
- `list`/`tuple` of *model_class* instances

Parameters

- **data** (*mixed*) – Data to delete from database.
- **synchronize_session** (*bool|str*) – Argument passed to `Query.delete`.

Returns Number of deleted records.

Return type int

Raises `InvalidRequestError` – When *model_class* is `None`.

entities
Return list of entity classes for query.

find(**criterion*, ***kargs*)
Return list of models matching *criterion*.

Parameters ***criterion** (*sqlaexpr, optional*) – SQLA expression to filter against.

Keyword Arguments

- **per_page** (*int, optional*) – Number of results to return per page. Defaults to `None` (i.e. no limit).
- **page** (*int, optional*) – Which page offset of results to return. Defaults to `1`.
- **order_by** (*sqlaexpr, optional*) – Order by expression. Defaults to `None`.

Returns List of *model_class*

Return type list

find_one(**criterion*, ***criterion_kargs*)
Return a single model or `None` given *criterion* `dict` or keyword arguments.

Parameters

- **criterion** (*dict, optional*) – Filter-by dict.
- ****criterion_kargs** (*optional*) – Mapping of filter-by arguments.

Returns When filtered record exists. None: When filtered record does not exist.

Return type *model_class*

join_entities
Return list of the joined entity classes for query.

join_model_classes
Return model classes contained in joins for query.

mapper_entities
Return mapper entities for query.

model_class
Return primary model class if query generated using `session.query(model_class)` or `None` otherwise.

model_classes
Return model classes used as selectable for query.

paginate(*pagination*)
Return paginated query.

Parameters **pagination** (*tuple|int*) – A `tuple` containing `(per_page, page)` or an `int` value for `per_page`.

Returns

New `Query` instance with `limit` and `offset` parameters applied.

Return type Query

save(*data*, *before=None*, *after=None*, *identity=None*)
Save *data* into the database using insert, update, or upsert-on-primary-key.

> **Warning:** This requires that the `Query` has been generated using `Query(<ModelClass>)`; otherwise, and exception will be raised.

The *data* argument can be any of the following:

- `dict`
- *model_class* instance
- `list`/`tuple` of `dict` objects
- `list`/`tuple` of *model_class* instances

This method will attempt to do the "right" thing by mapping any items in *data* that have their primary key set with the corresponding record in the database if it exists.

Parameters

- **data** (*mixed*) – Data to save to database.
- **before** (*function, optional*) – Function to call before each model is saved via `session.add`. Function should have signature `before(model, is_new)`.
- **after** (*function, optional*) – Function to call after each model is saved via `session.add`. Function should have signature `after(model, is_new)`.
- **identity** (*function, optional*) – Function used to return an idenity map for a given model. Function should have the signature `identity(model)`. By default *core.primary_identity_map()* is used.

Returns If a single item passed in. list: A `list` of *model_class* when multiple items passed.

Return type *model_class*

Raises `InvalidRequestError` – When *model_class* is `None`.

search(**criterion*, ***kargs*)
Return search query object.

Parameters ***criterion** (*sqlaexpr, optional*) – SQLA expression to filter against.

Keyword Arguments

- **per_page** (*int, optional*) – Number of results to return per page. Defaults to None (i.e. no limit).
- **page** (*int, optional*) – Which page offset of results to return. Defaults to 1.
- **order_by** (*sqlaexpr, optional*) – Order by expression. Defaults to None.

Returns

New Query instance with criteria and parameters applied.

Return type Query

4.5.3 Model

The declarative base model class for SQLAlchemy ORM.

class sqlservice.model.**ModelBase**(*data=None*, ***kargs*)

Declarative base for all ORM model classes.

classmethod class_mapper()

Return class mapper instance of model.

classmethod class_registry()

Returns declarative class registry containing declarative model class names mapped to class objects.

classmethod columns()

Return model columns as dict like OrderProperties object.

classmethod descriptors()

Return all ORM descriptors

descriptors_to_dict()

Return a dict that maps data loaded in __dict__ to this model's descriptors. The data contained in __dict__ represents the model's state that has been loaded from the database. Accessing values in __dict__ will prevent SQLAlchemy from issuing database queries for any ORM data that hasn't been loaded from the database already.

Note: The dict returned will contain model instances for any relationship data that is loaded. To get a dict containing all non-ORM objects, use *to_dict()*.

Returns dict

identity()

Return primary key identity of model instance. If there is only a single primary key defined, this method returns the primary key value. If there are multiple primary keys, a tuple containing the primary key values is returned.

identity_map()

Return primary key identity map of model instance as an ordered dict mapping each primary key column to the corresponding primary key value.

classmethod pk_columns()

Return tuple of primary key(s) for model.

classmethod relationships()
Return ORM relationships

to_dict()
Return a `dict` of the current model's state (i.e. data returned is limited to data already fetched from the database) if some state is loaded. If no state is loaded, perform a session refresh on the model which will result in a database query. For any relationship data that is loaded, `to_dict` be called recursively for those objects.

Returns dict

update(*data=None*, ***kargs*)
Update model by positional `dict` or keyword arguments.

Note: If both *data* and keyword arguments are passed in, the keyword arguments take precedence.

Parameters

- **data** (*dict, optional*) – Data to update model with.
- ****kargs** (*optional*) – Mapping of attributes to values to update model with.

Raises - *TypeError* – If *data* is not `None` or not a `dict`.

class sqlservice.model.**ModelMeta**(*name*, *bases*, *dct*)
Model metaclass that prepares model classes for event registration hooks.

sqlservice.model.**as_declarative**(***kargs*)
Decorator version of *declarative_base()*.

sqlservice.model.**declarative_base**(*cls=<class 'sqlservice.model.ModelBase'>*, *metadata=None*, *metaclass=<class 'sqlservice.model.ModelMeta'>*, ***kargs*)
Function that converts a normal class into a SQLAlchemy declarative base class.

Parameters

- **cls** (*type*) – A type to use as the base for the generated declarative base class. May be a class or tuple of classes. Defaults to *ModelBase*.
- **metadata** (*MetaData, optional*) – An optional MetaData instance. All Table objects implicitly declared by subclasses of the base will share this MetaData. A MetaData instance will be created if none is provided. If not passed in, *cls.metadata* will be used if set. Defaults to `None`.
- **metaclass** (*DeclarativeMeta, optional*) – A metaclass or `__metaclass__` compatible callable to use as the meta type of the generated declarative base class. If not passed in, *cls.metaclass* will be used if set. Defaults to *ModelMeta*.

Keyword Arguments

- **other keyword arguments are passed to**(*All*) –
- **sqlalchemy.ext.declarative.declarative_base.** –

Returns Declarative base class

Return type class

sqlservice.model.**default_dict_adapter**(*value*, *key*, *model*)
Default *ModelBase.to_dict()* adapter that handles nested serialization of model objects.

4.5.4 Event

The event module with declarative ORM event decorators and event registration.

class sqlservice.event.**AttributeEventDecorator**(*attribute*, ***event_kargs*)
Base class for an attribute event decorators.

class sqlservice.event.**Event**(*name*, *attribute*, *listener*, *kargs*)
Universal event class used when registering events.

class sqlservice.event.**EventDecorator**(***event_kargs*)
Base class for event decorators that attaches metadata to function object so that *register()* can find the event definition.

class sqlservice.event.**after_delete**(***event_kargs*)
Event decorator for the `after_delete` event.

class sqlservice.event.**after_insert**(***event_kargs*)
Event decorator for the `after_insert` event.

class sqlservice.event.**after_save**(***event_kargs*)
Event decorator for the `after_insert` and `after_update` events.

class sqlservice.event.**after_update**(***event_kargs*)
Event decorator for the `after_update` event.

class sqlservice.event.**before_delete**(***event_kargs*)
Event decorator for the `before_delete` event.

class sqlservice.event.**before_insert**(***event_kargs*)
Event decorator for the `before_insert` event.

class sqlservice.event.**before_save**(***event_kargs*)
Event decorator for the `before_insert` and `before_update` events.

class sqlservice.event.**before_update**(***event_kargs*)
Event decorator for the `before_update` event.

class sqlservice.event.**on_append**(*attribute*, ***event_kargs*)
Event decorator for the `append` event.

class sqlservice.event.**on_bulk_replace**(*attribute*, ***event_kargs*)
Event decorator for the `bulk_replace` event.

class sqlservice.event.**on_dispose_collection**(*attribute*, ***event_kargs*)
Event decorator for the `dispose_collection` event.

class sqlservice.event.**on_expire**(***event_kargs*)
Event decorator for the `expire` event.

class sqlservice.event.**on_init_collection**(*attribute*, ***event_kargs*)
Event decorator for the `init_collection` event.

class sqlservice.event.**on_init_scalar**(*attribute*, ***event_kargs*)
Event decorator for the `init_scalar` event.

class sqlservice.event.**on_load**(***event_kargs*)
Event decorator for the `load` event.

class sqlservice.event.**on_modified**(*attribute*, ***event_kargs*)
Event decorator for the `modified` event.

class sqlservice.event.**on_refresh**(***event_kargs*)
Event decorator for the `refresh` event.

class sqlservice.event.**on_remove**(*attribute*, ***event_kargs*)
Event decorator for the `remove` event.

class sqlservice.event.**on_set**(*attribute*, ***event_kargs*)
Event decorator for the `set` event.

sqlservice.event.**register**(*cls*, *dct*)
Register events defined on a class during metaclass creation.

sqlservice.core.**bulk_common_update**(*session*, *mapper*, *key_columns*, *mappings*)
Perform a bulk UPDATE on common shared values among *mappings*. What this means is that if multiple records are being updated to the same values, then issue only a single update for that value-set using the identity of the records in the WHERE clause.

Parameters

- **session** (*Session*) – SQLAlchemy session object.
- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **key_columns** (*tuple*) – A tuple of SQLAlchemy columns that represent the identity of each row (typically this would be a table's primary key values but they can be any set of columns).
- **mappings** (*list*) – List of `dict` objects to update.

Returns list[ResultProxy]

sqlservice.core.**bulk_diff_update**(*session*, *mapper*, *key_columns*, *previous_mappings*, *mappings*)
Perform a bulk INSERT/UPDATE on the difference between *mappings* and *previous_mappings* such that only the values that have changed are included in the update. If a mapping in *mappings* doesn't exist in *previous_mappings*, then it will be inclued in the bulk INSERT. The bulk INSERT will be deferred to *bulk_insert()*. The bulk UPDATE will be deferred to *bulk_common_update()*.

Parameters

- **session** (*Session*) – SQLAlchemy session object.
- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **key_columns** (*tuple*) – A tuple of SQLAlchemy columns that represent the identity of each row (typically this would be a table's primary key values but they can be any set of columns).
- **previous_mappings** (*list*) – List of `dict` objects that represent the previous values of all mappings found for this update set (i.e. these are the current database records).
- **mappings** (*list*) – List of `dict` objects to update.

Returns list[ResultProxy]

sqlservice.core.**bulk_insert**(*session*, *mapper*, *mappings*)
Perform a bulk insert into table/statement represented by *mapper* while utilizing a special syntax that replaces the tradtional `executemany()` DBAPI call with a multi-row VALUES clause for a single INSERT statement.

See *bulk_insert_many()* for bulk inserts using `executemany()`.

Parameters

- **session** (*Session*) – SQLAlchemy session object.
- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **mappings** (*list*) – List of `dict` objects to insert.

Returns ResultProxy

sqlservice.core.**bulk_insert_many**(*session*, *mapper*, *mappings*)

Perform a bulk insert into table/statement represented by *mapper* while utilizing the `executemany()` DBAPI call.

See *bulk_insert()* for bulk inserts using a multi-row VALUES clause for a single INSERT statement.

Parameters

- **session** (*Session*) – SQLAlchemy session object.
- **mapper** – An ORM class or SQLAlchemy insert-statement object.
- **mappings** (*list*) – List of `dict` objects to insert.

Returns ResultProxy

sqlservice.core.**destroy**(*session*, *data*, *model_class=None*, *synchronize_session=False*)

Delete bulk *data*.

The *data* argument can be any of the following:

- Single instance of *model_class*
- List of *model_class* instances
- Primary key value (single value or `tuple` of values for composite keys)
- List of primary key values.
- Dict containing primary key(s) mapping
- List of dicts with primary key(s) mappings

If a non-*model_class* instances are passed in, then *model_class* is required to know which table to delete from.

Parameters

- **session** (*Session*) – SQLAlchemy session object.
- **data** (*mixed*) – Data to delete from database.
- **synchronize_session** (*bool | str*) – Argument passed to `Query.delete`.

Returns Number of deleted records.

Return type int

sqlservice.core.**identity_map_filter**(*models*, *identity=None*)

Return SQLAlchemy filter expression for a list of *models* based on the identity map returned by the given *identity* function.

sqlservice.core.**make_identity**(**columns*)

Factory function that returns an identity function that can be used in *save()*. The identity function returns an identity-tuple mapping from a model instance with the given *columns* and their values.

sqlservice.core.**mapper_primary_key**(*model_class*)

Return primary keys of *model_class*.

sqlservice.core.**primary_identity_map**(*model*)

Return identity-map of a model as a N-element tuple where N is the number of primary key columns. Each element of the tuple is a 2-element tuple containing the primary key column and the corresponding model value.

sqlservice.core.**primary_identity_value**(*model*)

Return primary key identity of model instance. If there is only a single primary key defined, this function returns the primary key value. If there are multiple primary keys, a tuple containing the primary key values is returned.

sqlservice.core.**primary_key_filter**(*items*, *model_class*)

Given a set of *items* that have their primary key(s) set and that may or may not exist in the database, return a filter that queries for those records.

Parameters

- **items** (*list*) – List of `dict` or *model_class* instances to query.
- **model_class** (*Model*) – ORM model class to query against.

Returns sqlalchemy.sql.elements.BinaryExpression

sqlservice.core.**save**(*session*, *models*, *before=None*, *after=None*, *identity=None*)

Save *models* into the database using insert, update, or upsert on *identity*.

The *models* argument can be any of the following:

- Model instance
- `list`/`tuple` of Model instances

The required function signature and return of *identity* is:

```
def identity(model):
    return ((column1, value1), (column2, value2), ... (colN, valN))
```

For example, a model with a single primary key, the return might look like:

```
model = Model(id=1)
identity(model) == ((Model.id, 1))
```

And for a model with a composite primary key:

```
model = Model(id1=1, id2=2)
identity(model) == ((Model.id1, 1), (Model.id2, 2))
```

This requirement is necessary so that the *save()* function can correctly generate a query filter to select existing records to determine which of those records should be updated vs inserted.

If no *identity* function is provided, then *primary_identity_map()* will be used which will result in upserting on primary key.

Parameters

- **session** (*Session*) – SQLAlchemy session object.
- **models** (*mixed*) – Models to save to database.
- **before** (*function, optional*) – Function to call before each model is saved via `session.add`. Function should have signature `before(model, is_new)`.
- **after** (*function, optional*) – Function to call after each model is saved via `session.add`. Function should have signature `after(model, is_new)`.
- **identity** (*function, optional*) – Function used to return an idenity map for a given model. Function should have the signature `identity(model)`. By default *primary_identity_map()* is used.

Returns If a single item passed in. list: A `list` of Model instaces if multiple items passed in.

Return type Model

sqlservice.core.**transaction**(*session*, *commit=True*, *rollback=False*, *autoflush=None*)

Nestable session transaction context manager where only a single commit will be issued once all contexts have

been exited. If an exception occurs either at commit time or before, the transaction will be rolled back and the original exception re-raised.

Parameters

- **session** (*Session*) – SQLAlchemy session object.
- **commit** (*bool, optional*) – Whether to commit the transaction or leave it open. Defaults to `True`.
- **rollback** (*bool, optional*) – Whether to rollback the transaction. Defaults to `False`. WARNING: This overrides *commit*.
- **autoflush** (*bool, optional*) – Whether to override `session.autoflush`. Original `session.autoflush` will be restored after transaction. Defaults to `None` which doesn't modify `session.autoflush`.

Yields `session`

CHAPTER 5

Project Info

5.1 License

5.2 Versioning

This project follows Semantic Versioning with the following caveats:

- Only the public API (i.e. the objects imported into `sqlservice.__init__`) will maintain backwards compatibility between MINOR version bumps.
- Objects within any other parts of the library are not guaranteed to not break between MINOR version bumps.

With that in mind, it is recommended to only use or import objects from the main module.

5.3 Changelog

5.3.1 v1.1.3 (2018-09-26)

- If a key in `Model.__dict_args__['adapters']` is `None`, then don't serialize that key when calling `Model.to_dict()`.

5.3.2 v1.1.2 (2018-09-23)

- Fix handling of string keys in `Model.__dict_args__['adapters']` that resulted in an unhandled `TypeError` exception in some cases.

5.3.3 v1.1.1 (2018-09-07)

- Fix mishandling of case where new mappings passed to `SQLClient.bulk_diff_update()` aren't different than previous mappings.

5.3.4 v1.1.0 (2018-09-05)

- Add `SQLClient.bulk_common_update()` and `core.bulk_common_update()`.
- Add `SQLClient.bulk_diff_update()` and `core.bulk_diff_update()`.
- Move logic in `SQLClient.bulk_insert()` and `bulk_insert_many()` to `core.bulk_insert()` and `core.bulk_insert_many()` respectively.

5.3.5 v1.0.2 (2018-08-20)

- Minor optimization to `SQLQuery.save()` to not create an intermediate list when saving multiple items.

5.3.6 v1.0.1 (2018-08-20)

- Add missing handling for generators in `SQLQuery.save()`.

5.3.7 v1.0.0 (2018-08-19)

- Drop support for Python 2.7. (**breaking change**)
- Don't mutate `models` argument when passed in as a list to `SQLClient.save|core.save`.
- Allow generators to be passed into `SQLClient.save|core.save` and `SQLClient.destroy|core.destroy`.
- Remove deprecated methods: (**breaking change**)
  - `SQLClient.shutdown()` (use `SQLClient.disconnect()`)
  - `SQLQuery.chain()`
  - `SQLQuery.pluck()`
  - `SQLQuery.key_by()`

  - `SQLQuery.map()`
  - `SQLQuery.reduce()`
  - `SQLQuery.reduce_right()`
  - `SQLQuery.stack_by()`

5.3.8 v0.23.0 (2018-08-06)

- Add `SQLClient.DEFAULT_CONFIG` class attribute as way to override config defaults at the class level via subclassing.
- Rename `SQLClient.shutdown()` to `disconnect()` but keep `shutdown()` as a deprecated alias.
- Deprecate `SQLClient.shutdown()`. Use `SQLClient.disconnect()` instead. Will be removed in `v1`.
- Deprecate `SQLQuery` methods below. Use `pydash` library directly or re-implement in subclass of `SQLQuery` and pass to `SQLClient()` via `query_class` argument. Methods will be removed in `v1`:
  - `SQLQuery.chain()`
  - `SQLQuery.pluck()`
  - `SQLQuery.key_by()`
  - `SQLQuery.map()`
  - `SQLQuery.reduce()`
  - `SQLQuery.reduce_right()`
  - `SQLQuery.stack_by()`

5.3.9 v0.22.1 (2018-07-15)

- Support Python 3.7.

5.3.10 v0.22.0 (2018-04-12)

- Change default behavior of `SQLClient.transaction()` to **not** override the current session's `autoflush` setting (use `SQLClient.transaction(autoflush=True)` instead. (**breaking change**)
- Add boolean `autoflush` option to `SQLClient.transaction()` to set session's `autoflush` value for the duration of the transaction.
- Add new `sqlservice.event` decorators:
  - `on_init_scalar`
  - `on_init_collection`
  - `on_modified`
  - `on_bulk_replace`
  - `on_dispose_collection`

5.3.11 v0.21.0 (2018-04-02)

- Add `SQLClient.ping()` method that performs a basic connection check.

5.3.12 v0.20.0 (2018-03-20)

- Add `ModelBase.class_registry()` that returns the declarative class registry from declarative metadata. Roughly equivalent to `ModelBase._decl_class_registry` but with `_sa_*` keys removed.
- Pass model instance as third optional argument to `ModelBase.__dict_args__['adapters']` handlers.
- Expose default `dict` adapater as `sqlservice.model.default_dict_adapter`.

5.3.13 v0.19.0 (2018-03-19)

- Support model class names as valid keys in `ModelBase.__dict_args__['adapaters']`. Works similar to string namesused in `sqlalchemy.orm.relation`.
- Support model class orm descriptors (e.g. columns, relationships) as valid keys in `ModelBase.__dict_args__['adapaters']`.

5.3.14 v0.18.0 (2018-03-12)

- Remove `readonly` argument from `SQLClient.transaction` and replace with separate `commit` and `rollback`. (**breaking change**)
  - The default is `commit=True` and `rollback=False`. This behavior mirrors the previous behavior.
  - When `rollback=True`, the `commit` argument is ignored and the top-level transaction is always rolled back. This is like `readonly=True` in version `0.17.0`.
  - When `commit=False` and `rollback=False`, the "transaction" isn't finalized and is left open. This is like `readonly=True` in versions `<=0.16.1`.

5.3.15 v0.17.0 (2018-03-12)

- Rollback instead of commit in a readonly transaction issued by `SQLClient.transaction`. (**potential breaking change**)
  - There's a potential breaking change for the case where there's nested a write transaction under a readonly transaction. Previously, the write transaction would be committed when the readonly transaction finalized since commit was being called instead of rollback. However with this change, the settings of the first transaction before any nesting will now determine whether the entire transaction is committed or rollbacked.

5.3.16 v0.16.1 (2018-02-26)

- Use `repr(self.url)` in `SQLClient.__repr__()` instead of `str()` to mask connection password if provided.

5.3.17 v0.16.0 (2018-02-21)

- Support a database URI string as the configuration value for `SQLClient`. For example, previously had to do `SQLClient({'SQL_DATABASE_URI': '<db_uri>'})` but now can do `SQLClient('<db_uri>')`.
- Add `repr()` support to `SQLClient`.

5.3.18 v0.15.0 (2018-02-13)

- Add `SQL_POOL_PRE_PING` config option to `SQLClient` that sets `pool_pre_ping` argument to engine. Requires SQLAlchemy >= 1.2. Thanks dsully!

5.3.19 v0.14.2 (2017-10-17)

- Fix `Query.search()` so that `dict` filter-by criteria will be applied to the base model class of the query if it's set (i.e. make `db.query(ModelA).join(ModelB).search({'a_only_field': 'foo'})` work so that `{'a_only_field': 'foo'}` is filtered on `ModelA.a_only_field` instead of `ModelB`). This also applies to `Query.find()` and `Query.find_one()` which use `search()` internally.

5.3.20 v0.14.1 (2017-09-09)

- Fix typo in `SQL_ENCODING` config option mapping to SQLAlchemy parameter. Thanks dsully!

5.3.21 v0.14.0 (2017-08-03)

- Make `declarative_base` pass extra keyword arguments to `sqlalchemy.ext.declarative.declarative_base`.
- Remove `ModelBase.metaclass` and `ModelBase.metadata` hooks for hoisting those values to `declarative_base()`. Instead, pass optional `metadata` and `metaclass` arguments directly to `declarative_base`. (**breaking change**)
- Replace broken `declarative_base` decorator usage with new decorator-only function, `as_declarative`. Previously, `@declarative_base` only worked as a decorator when not "called" (i.e. `@declarative_base` worked but `@declarative_base(...)` failed).

5.3.22 v0.13.0 (2017-07-11)

- Add `ModelBase.__dict_args__` attribute for providing arguments to `ModelBase.to_dict`.
- Add `adapters` option to `ModelBase.__dict_args__` for mapping model value types to custom serializatoin handlers during `ModelBase.to_dict()` call.

5.3.23 v0.12.1 (2017-04-04)

- Bump minimum requirement for pydash to `v4.0.1`.

- Revert removal of `Query.pluck` but now `pluck` works with a deep path *and* path list (e.g. `['a', 'b', 0, 'c']` to get `'value'` in `{'a': {'b': [{'c': 'value'}]}}` which is something that `Query.map` doesn't support.

5.3.24 v0.12.0 (2017-04-03)

- Bump minimum requirement for pydash to `v4.0.0`. (**breaking change**)
- Remove `Query.pluck` in favor or `Query.map` since `map` can do everything `pluck` could. (**breaking change**)
- Rename `Query.index_by` to `Query.key_by`. (**breaking change**)
- Rename `callback` argument to `iteratee` for `Query` methods:
  - `key_by`
  - `stack_by`
  - `map`
  - `reduce`
  - `reduce_right`

5.3.25 v0.11.0 (2017-03-10)

- Make `SQLClient.save()` update the declarative model registry whenever an model class isn't in it. This allows saving to work when a `SQLClient` instance was created before models have been imported yet.
- Make `SQLClient.expunge()` support multiple instances.
- Make `SQLClient.save()` and `SQLQuery.save()` handle saving empty dictionaries.

5.3.26 v0.10.0 (2017-02-13)

- Add `engine_options` argument to `SQLClient()` to provide additional engine options beyond what is supported by the `config` argument.
- Add `SQLClient.bulk_insert` for performing an INSERT with a multi-row VALUES clause.
- Add `SQLClient.bulk_insert_many` for performing an `executemany()` DBAPI call.
- Add additional `SQLClient.session` proxy properties on `SQLClient.<proxy>`:
  - `bulk_insert_mappings`
  - `bulk_save_objects`
  - `bulk_update_mappings`
  - `is_active`
  - `is_modified`
  - `no_autoflush`
  - `preapre`
- Store `SQLClient.models` as a static `dict` instead of computed property but recompute if an attribute error is detected for `SQLClient.<Model>` to handle the case of a late model class import.

- Fix handling of duplicate base class names during `SQLClient.models` creation for model classes that are defined in different submodules. Previously, duplicate model class names prevented those models from being saved via `SQLClient.save()`.

5.3.27 v0.9.1 (2017-01-12)

- Fix handling of `scopefunc` option in `SQLClient.create_session`.

5.3.28 v0.9.0 (2017-01-10)

- Add `session_class` argument to `SQLClient()` to override the default session class used by the session maker.
- Add `session_options` argument to `SQLClient()` to provide additional session options beyond what is supported by the `config` argument.

5.3.29 v0.8.0 (2016-12-09)

- Rename `sqlservice.Query` to `SQLQuery`. (**breaking change**)
- Remove `sqlservice.SQLService` class in favor of utilizing `SQLQuery` for the `save` and `destroy` methods for a model class. (**breaking change**)
- Add `SQLQuery.save()`.
- Add `SQLQuery.destroy()`.
- Add `SQLQuery.model_class` property.
- Replace `service_class` argument with `query_class` in `SQLClient.__init__()`. (**breaking change**)
- Remove `SQLClient.services`. (**breaking change**)
- When a model class name is used for attribute access on a `SQLClient` instance, return an instance of `SQLQuery(ModelClass)` instead of `SQLService(ModelClass)`. (**breaking change**)

5.3.30 v0.7.2 (2016-11-29)

- Fix passing of `synchronize_session` argument in `SQLService.destroy` and `SQLClient.destroy`. Argument was mistakenly not being used when calling underlying delete method.

5.3.31 v0.7.1 (2016-11-04)

- Add additional database session proxy attributes to `SQLClient`:
  - `SQLClient.scalar -> SQLClient.session.scalar`
  - `SQLClient.invalidate -> SQLClient.session.invalidate`
  - `SQLClient.expire -> SQLClient.session.expire`
  - `SQLClient.expire_all -> SQLClient.session.expire_all`
  - `SQLClient.expunge -> SQLClient.session.expunge`
  - `SQLClient.expunge_all -> SQLClient.session.expunge_all`

  - `SQLClient.prune -> SQLClient.session.prune`

- Fix compatibility issue with pydash `v3.4.7`.

5.3.32 v0.7.0 (2016-10-28)

- Add `core.make_identity` factory function for easily creating basic identity functions from a list of model column objects that can be used with `save()`.
- Import `core.save`, `core.destroy`, `core.transaction`, and `core.make_identity` into make package namespace.

5.3.33 v0.6.3 (2016-10-17)

- Fix model instance merging in `core.save` when providing a custom identity function.

5.3.34 v0.6.2 (2016-10-17)

- Expose `identity` argument in `SQLClient.save` and `SQLService.save`.

5.3.35 v0.6.1 (2016-10-17)

- Fix bug where the `models` variable was mistakenly redefined during loop iteration in `core.save`.

5.3.36 v0.6.0 (2016-10-17)

- Add `identity` argument to `save` method to allow a custom identity function to support upserting on something other than just the primary key values.
- Make `Query` entity methods `entities`, `join_entities`, and `all_entities` return entity objects instead of model classes. (**breaking change**)
- Add `Query` methods `model_classes`, `join_model_classes`, and `all_model_classes` return the model classes belonging to a query.

5.3.37 v0.5.1 (2016-09-28)

- Fix issue where calling `<Model>.update(data)` did not correctly update a relationship field when both `<Model>.<relationship-column>` and `data[<relationship-column>]` were both instances of a model class.

5.3.38 v0.5.0 (2016-09-20)

- Allow `Service.find_one`, `Service.find`, and `Query.search` to accept a list of lists as the criterion argument.
- Rename ModelBase metaclass class attribute from `ModelBase.Meta` to `ModelBase.metaclass`. (**breaking change**)
- Add support for defining the `metadata` object on `ModelBase.metadata` and having it used when calling `declarative_base`.

- Add `metadata` and `metaclass` arguments to `declarative_base` that taken precedence over the corresponding class attributes set on the passed in declarative base type.
- Rename Model argument/attribute in `SQLClient` to `__init__` to `model_class`. (**breaking change**)
- Remove `Query.top` method. (**breaking change**)
- Proxy `SQLService.__getattr__` to `getattr(SQLService.query(), attr)` so that `SQLService` now acts as a proxy to a query instance that uses its `model_class` as the primary query entity.
- Move `SQLService.find` and `SQLService.find_one` to `Query`.
- Improve docs.

5.3.39 v0.4.3 (2016-07-11)

- Fix issue where updating nested relationship values can lead to conflicting state assertion error in SQLAlchemy's identity map.

5.3.40 v0.4.2 (2016-07-11)

- Fix missing `before` and `after` callback argument passing from `core.save` to `core._add`.

5.3.41 v0.4.1 (2016-07-11)

- Fix missing `before` and `after` callback argument passing from `SQLService.save` to `SQLClient.save`.

5.3.42 v0.4.0 (2016-07-11)

- Add support for `before` and `after` callbacks in `core.save`, `SQLClient.save`, and `SQLService.save` which are invoked before/after `session.add` is called for each model instance.

5.3.43 v0.3.0 (2016-07-06)

- Support additional engine and session configuration values for `SQLClient`.
  - New engine config options:
    * `SQL_ECHO_POOL`
    * `SQL_ENCODING`
    * `SQL_CONVERT_UNICODE`
    * `SQL_ISOLATION_LEVEL`
  - New session config options:
    * `SQL_EXPIRE_ON_COMMIT`
- Add `SQLClient.reflect` method.
- Rename `SQLClient.service_registry` and `SQLClient.model_registry` to `services` and `models`. (**breaking change**)

- Support `SQLClient.__getitem__` as proxy to `SQLClient.__getattr__` where both `db[User]` and `db['User']` both map to `db.User`.
- Add `SQLService.count` method.
- Add `Query` methods:
  - `index_by`: Converts `Query.all()` to a `dict` of models indexed by `callback` (pydash.index_by)
  - `stack_by`: Converts `Query.all()` to a `dict` of lists of models indexed by `callback` (pydash.group_by)
  - `map`: Maps `Query.all()` to a `callback` (pydash.map_)
  - `reduce`: Reduces `Query.all()` through `callback` (pydash.reduce_)
  - `reduce_right`: Reduces `Query.all()` through `callback` from right (pydash.reduce_right)
  - `pluck`: Retrieves value of of specified property from all elements of `Query.all()` (pydash.pluck)
  - `chain`: Initializes a chain object with `Query.all()` (pydash.chain)
- Rename `Query` properties: (**breaking change**)
  - `model_classes` to `entities`
  - `joined_model_classes` to `join_entities`
  - `all_model_classes` to `all_entities`

5.3.44 v0.2.0 (2016-06-15)

- Add Python 2.7 compatibility.
- Add concept of `model_registry` and `service_registry` to `SQLClient` class:
  - `SQLClient.model_registry` returns mapping of ORM model names to ORM model classes bound to `SQLClient.Model`.
  - `SQLService` instances are created with each model class bound to declarative base, `SQLClient.Model` and stored in `SQLClient.service_registry`.
  - Access to each model class `SQLService` instance is available via attribute access to `SQLClient`. The attribute name corresponds to the model class name (e.g. given a `User` ORM model, it would be accessible at `sqlclient.User`.
- Add new methods to `SQLClient` class:
  - `save`: Generic saving of model class instances similar to `SQLService.save` but works for any model class instance.
  - `destroy`: Generic deletion of model class instances or `dict` containing primary keys where model class is explicitly passed in. Similar to `SQLService.destroy`.
- Rename `SQLService.delete` to `destroy`. (**breaking change**)
- Change `SQLService` initialization signature to `SQLService(db, model_class)` and remove class attribute `model_class` in favor of instance attribute. (**breaking change**)
- Add properties to `SQLClient` class:
  - `service_registry`
  - `model_registry`
- Add properties to `Query` class:

  - `model_classes`: Returns list of model classes used to during `Query` creation.
  - `joined_model_classes`: Returns list of joined model classes of `Query`.
  - `all_model_classes`: Returns `Query.model_classes` + `Query.joined_model_classes`.
- Remove methods from `SQLService` class: (**breaking change**)
  - `query_one`
  - `query_many`
  - `default_order_by` (default order by determination moved to `Query.search`)
- Remove `sqlservice.service.transaction` decorator in favor of using transaction context manager within methods. (**breaking change**)
- Fix incorrect passing of `SQL_DATABASE_URI` value to `SQLClient.create_engine` in `SQLClient.__init__`.

5.3.45 v0.1.0 (2016-05-24)

- First release.

5.4 Authors

5.4.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

5.4.2 Contributors

- Dan Sully, dsully@github
- Cooper Benson, skycoop@github

5.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/dgilland/sqlservice.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" or "help wanted" is open to whoever wants to implement it.

Write Documentation

SQLService could always use more documentation, whether as part of the official SQLService docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dgilland/sqlservice.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.5.2 Get Started!

Ready to contribute? Here's how to set up `sqlservice` for local development.

1. Fork the `sqlservice` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/sqlservice.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenv installed, this is how you set up your fork for local development:

```
$ cd sqlservice
$ pip install -r requirements-dev.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass linting and all unit tests by testing with tox across all supported Python versions:

```
$ invoke tox
```

6. Add yourself to `AUTHORS.rst`.
7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

5.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the README.rst.
3. The pull request should work for Python 2.7, 3.4, and 3.5. Check https://travis-ci.org/dgilland/sqlservice/pull_requests and make sure that the tests pass for all supported Python versions.

CHAPTER 6

Indices and Tables

- genindex
- modindex
- search

Python Module Index

S

Index