alchy Documentation

Release 2.2.2

Derrick Gilland

January 04, 2017

Contents

A SQLAlchemy extension for its declarative ORM that provides enhancements for model classes, queries, and sessions.

CHAPTER 1

MAINTENANCE MODE

PROJECT IS IN MAINTENANCE MODE: NO NEW FEATURES, BUG FIXES ONLY

Use sqlservice instead.

CHAPTER 2

Links

- Project: https://github.com/dgilland/alchy
- Documentation: https://alchy.readthedocs.io
- PyPi: https://pypi.python.org/pypi/alchy/
- TravisCI: https://travis-ci.org/dgilland/alchy

CHAPTER 3

Quickstart

Let's see `alchy` in action. We'll start with some model definitions.

```
from alchy import ModelBase, make_declarative_base
from sqlalchemy import orm, Column, types, ForeignKey

class Base(ModelBase):
    # extend/override ModelBase if necessary
    pass

Model = make_declarative_base(Base=Base)

class User(Model):
    __tablename__ = 'user'

    _id = Column(types.Integer(), primary_key=True)
    name = Column(types.String())
    email = Column(types.String())
    level = Column(types.Integer())

    items = orm.relationship('UserItem')

class UserItem(Model):
    # when no __tablename__ defined,
    # one is autogenerated using class name
    # like this:
    #__tablename__ = 'user_item'

    _id = Column(types.Integer(), primary_key=True)
    user_id = Column(types.Integer(), ForeignKey('user._id'))
    name = Column(types.String())

    user = orm.relationship('User')
```

Next, we need to interact with our database. For that we will use a *alchy.manager.Manager*.

```
from alchy import Manager

# Config can be either (1) dict, (2) class, or (3) module.
config = {
    'SQLALCHEMY_DATABASE_URI': 'sqlite://'
}

# Be sure to pass in our declarative base defined previously.
```

```
# This is needed so that Model.metadata operations like
# create_all(), drop_all(), and reflect() work.
db = Manager(config=config, Model=Model)
```

Create our database tables.

```
db.create_all()
```

Now, create some records.

```
# initialize using keyword args
user1 = User(name='Fred', email='fred@example.com')
# print('user1:', user1)

# ...or initialize using a dict
user2 = User({'name': 'Barney'})
# print('user2:', user2)

# update using either method as well
user2.update(email='barney@example.org')
user2.update({'email': 'barney@example.com'})
# print('user2 updated:', user2)
```

Add them to the database.

```
# there are several options for adding records

# add and commit in one step using positional args
db.add_commit(user1, user2)

# ...or add/commit using a list
users = [user1, user2]
db.add_commit(users)

# ...or separate add and commit calls
db.add(user1, user2)
db.commit()

# ...or with a list
db.add(users)
db.commit()

# ...or separate adds and commit
db.add(user1)
db.add(user2)
db.commit()
```

Fetch model and operate.

```
# create user
db.add_commit(User(name='Wilma', email='wilma@example.com'))

# fetch from database
user = User.get(user1._id)
# print('user:', user)

# convert to dict
user_dict = user.to_dict()
# print('user dict:', user_dict)
```

```
# ...or just pass object directly to dict()
user_dict = dict(user)

# make some changes
user.update(level=5)

# and refresh
user.refresh()

# or flush
user.flush()

# access the session that loaded the model instance
assert user.session() == db.object_session(user)

# delete user
user.delete()
db.commit()

# ...or via db
db.delete(user)
db.commit()

# ...or all-in-one step
db.delete_commit(user)
```

Query records from the database.

```
# add some more users
db.add_commit(
    User(items=[UserItem()]),
    User(items=[UserItem()]),
    User(items=[UserItem()]),
    User(items=[UserItem()]),
    User(items=[UserItem()])
)

# there are several syntax options for querying records

# using db.session directly
# print('all users:', db.session.query(User).all())

# ...or using db directly (i.e. db.session proxy)
assert db.query(User).all() == db.session.query(User).all()

# ...or via query property on model class
assert User.query.all() == db.session.query(User).all()
```

Use features from the enhanced query class.

```
q = User.query.join(UserItem)

# entities
assert q.entities == [User]
assert q.join_entities == [UserItem]
assert q.all_entities == [User, UserItem]

# paging
assert str(q.page(2, per_page=2)) == str(q.limit(2).offset((2-1) * 2))
```

```
# pagination
page2 = q.paginate(2, per_page=2)
assert str(page2.query) == str(q)
assert page2.page == 2
assert page2.per_page == 2
assert page2.total == q.count()
assert page2.items == q.limit(2).offset((2-1) * 2).all()
assert page2.prev_num == 1
assert page2.has_prev == True
assert page2.next_num == 3
assert page2.has_next == True
page_1 = page2.prev()
page_3 = page2.next()

# searching

# ...extend class definitions to support advanced and simple searching
User.__advanced_search__ = User.__simple_search__ = {
    'user_email': lambda value: User.email.like('%{0}%'.format(value)),
    'user_name': lambda value: User.name.like('%{0}%'.format(value))
}

UserItem.__advanced_search__ = {
    'item_name': lambda value: UserItem.name.like('%{0}%'.format(value))
}

search = User.query.search('example.com', {'user_name': 'wilma'})
# print('search:', str(search))
assert search.count() > 0

# entity loading
User.query.join_eager(User.items)
User.query.joinedload(User.items)
User.query.lazyload(User.items)
User.query.immediateload(User.items)
User.query.noload(User.items)
User.query.subqueryload(User.items)

# column loading
User.query.load_only('_id', 'name')
User.query.defer('email')
User.query.undefer('email') # if User.email undeferred in class definition
User.query.undefer_group('group1', 'group2') # if under groups defined in class

# utilities
User.query.map(lambda user: user.level)
User.query.pluck('level')
User.query.index_by('email')
User.query.chain().value()
User.query.reduce(
    lambda result, user: result + 1 if user.level > 5 else result,
    initial=0
)
```

For more details regarding the chaining API (i.e. *Query.chain()*), see the pydash documentation.

Utilize ORM events.

```
from alchy import events

class User(Model):
    __table_args__ = {
        # this is needed since we're replacing the ``User`` class defined above
        'extend_existing': True
    }

    _id = Column(types.Integer(), primary_key=True)
    name = Column(types.String())
    email = Column(types.String())
    level = Column(types.Integer())

    @events.before_insert_update()
    def validate(self, *args, **kargs):
        '''Validate model instance'''
        # do validation
        return

    @events.on_set('email')
    def on_set_email(self, value, oldvalue, initator):
        if self.query.filter(User.email==value, User._id!=self._id).count() > 0:
            raise ValueError('Email already exists in database')

user = User(email='one@example.com')
db.add_commit(user)

try:
    User(email=user.email)
except ValueError as ex:
    pass
```

Finally, clean up after ourselves.

```
db.drop_all()
```

See also:

For further details consult *API Reference*.

CHAPTER 4

Guide

4.1 Installation

To install from PyPi:

```
pip install alchy
```

4.1.1 Compatibility

- Python 2.6
- Python 2.7
- Python 3.2
- Python 3.3
- Python 3.4

4.1.2 Dependencies

- SQLAlchemy >= 0.9.0

4.2 Quickstart

Let's see `alchy` in action. We'll start with some model definitions.

```
from alchy import ModelBase, make_declarative_base
from sqlalchemy import orm, Column, types, ForeignKey

class Base(ModelBase):
    # extend/override ModelBase if necessary
    pass

Model = make_declarative_base(Base=Base)

class User(Model):
    __tablename__ = 'user'
```

```
    _id = Column(types.Integer(), primary_key=True)
    name = Column(types.String())
    email = Column(types.String())
    level = Column(types.Integer())

    items = orm.relationship('UserItem')

class UserItem(Model):
    # when no __tablename__ defined,
    # one is autogenerated using class name
    # like this:
    #__tablename__ = 'user_item'

    _id = Column(types.Integer(), primary_key=True)
    user_id = Column(types.Integer(), ForeignKey('user._id'))
    name = Column(types.String())

    user = orm.relationship('User')
```

Next, we need to interact with our database. For that we will use a *alchy.manager.Manager*.

```
from alchy import Manager

# Config can be either (1) dict, (2) class, or (3) module.
config = {
    'SQLALCHEMY_DATABASE_URI': 'sqlite://'
}

# Be sure to pass in our declarative base defined previously.
# This is needed so that Model.metadata operations like
# create_all(), drop_all(), and reflect() work.
db = Manager(config=config, Model=Model)
```

Create our database tables.

```
db.create_all()
```

Now, create some records.

```
# initialize using keyword args
user1 = User(name='Fred', email='fred@example.com')
# print('user1:', user1)

# ...or initialize using a dict
user2 = User({'name': 'Barney'})
# print('user2:', user2)

# update using either method as well
user2.update(email='barney@example.org')
user2.update({'email': 'barney@example.com'})
# print('user2 updated:', user2)
```

Add them to the database.

```
# there are several options for adding records

# add and commit in one step using positional args
db.add_commit(user1, user2)
```

```
# ...or add/commit using a list
users = [user1, user2]
db.add_commit(users)

# ...or separate add and commit calls
db.add(user1, user2)
db.commit()

# ...or with a list
db.add(users)
db.commit()

# ...or separate adds and commit
db.add(user1)
db.add(user2)
db.commit()
```

Fetch model and operate.

```
# create user
db.add_commit(User(name='Wilma', email='wilma@example.com'))

# fetch from database
user = User.get(user1._id)
# print('user:', user)

# convert to dict
user_dict = user.to_dict()
# print('user dict:', user_dict)

# ...or just pass object directly to dict()
user_dict = dict(user)

# make some changes
user.update(level=5)

# and refresh
user.refresh()

# or flush
user.flush()

# access the session that loaded the model instance
assert user.session() == db.object_session(user)

# delete user
user.delete()
db.commit()

# ...or via db
db.delete(user)
db.commit()

# ...or all-in-one step
db.delete_commit(user)
```

Query records from the database.

```
# add some more users
db.add_commit(
    User(items=[UserItem()]),
    User(items=[UserItem()]),
    User(items=[UserItem()]),
    User(items=[UserItem()]),
    User(items=[UserItem()])
)

# there are several syntax options for querying records

# using db.session directly
# print('all users:', db.session.query(User).all())

# ...or using db directly (i.e. db.session proxy)
assert db.query(User).all() == db.session.query(User).all()

# ...or via query property on model class
assert User.query.all() == db.session.query(User).all()
```

Use features from the enhanced query class.

```
q = User.query.join(UserItem)

# entities
assert q.entities == [User]
assert q.join_entities == [UserItem]
assert q.all_entities == [User, UserItem]

# paging
assert str(q.page(2, per_page=2)) == str(q.limit(2).offset((2-1) * 2))

# pagination
page2 = q.paginate(2, per_page=2)
assert str(page2.query) == str(q)
assert page2.page == 2
assert page2.per_page == 2
assert page2.total == q.count()
assert page2.items == q.limit(2).offset((2-1) * 2).all()
assert page2.prev_num == 1
assert page2.has_prev == True
assert page2.next_num == 3
assert page2.has_next == True
page_1 = page2.prev()
page_3 = page2.next()

# searching

# ...extend class definitions to support advanced and simple searching
User.__advanced_search__ = User.__simple_search__ = {
    'user_email': lambda value: User.email.like('%{0}%'.format(value)),
    'user_name': lambda value: User.name.like('%{0}%'.format(value))
}

UserItem.__advanced_search__ = {
    'item_name': lambda value: UserItem.name.like('%{0}%'.format(value))
}
```

```
search = User.query.search('example.com', {'user_name': 'wilma'})
# print('search:', str(search))
assert search.count() > 0

# entity loading
User.query.join_eager(User.items)
User.query.joinedload(User.items)
User.query.lazyload(User.items)
User.query.immediateload(User.items)
User.query.noload(User.items)
User.query.subqueryload(User.items)

# column loading
User.query.load_only('_id', 'name')
User.query.defer('email')
User.query.undefer('email') # if User.email undeferred in class definition
User.query.undefer_group('group1', 'group2') # if under groups defined in class

# utilities
User.query.map(lambda user: user.level)
User.query.pluck('level')
User.query.index_by('email')
User.query.chain().value()
User.query.reduce(
    lambda result, user: result + 1 if user.level > 5 else result,
    initial=0
)
```

For more details regarding the chaining API (i.e. *`Query.chain()`*), see the pydash documentation.

Utilize ORM events.

```
from alchy import events

class User(Model):
    __table_args__ = {
        # this is needed since we're replacing the ``User`` class defined above
        'extend_existing': True
    }

    _id = Column(types.Integer(), primary_key=True)
    name = Column(types.String())
    email = Column(types.String())
    level = Column(types.Integer())

    @events.before_insert_update()
    def validate(self, *args, **kargs):
        '''Validate model instance'''
        # do validation
        return

    @events.on_set('email')
    def on_set_email(self, value, oldvalue, initator):
        if self.query.filter(User.email==value, User._id!=self._id).count() > 0:
            raise ValueError('Email already exists in database')

user = User(email='one@example.com')
db.add_commit(user)
```

```
try:
    User(email=user.email)
except ValueError as ex:
    pass
```

Finally, clean up after ourselves.

```
db.drop_all()
```

See also:

For further details consult *API Reference*.

4.3 Versioning

This project follows Semantic Versioning.

4.4 Upgrading

4.4.1 To v2.0.0

The logic that sets a Model class' `__table_args__` and `__mapper_args__` (unless overridden in subclass) has been modified. A model's `__local_table_args__` and `__local_mapper_args__` are now merged with `__global_table_args__` and `__global_mapper_args__` from all classes in the class's `mro()`. A `__{global,local}_{table,mapper}_args__` may be a callable or classmethod, in which case it is evaluated on the class whose `__{table,mapper}_args__` is being set.

4.4.2 To v1.0.0

The `@classproperty` decorator has been eliminated and replaced with `@classmethod` in `v1.0.0`. This means that the previous *alchy.model.ModelBase* class properties must now be accessed via method calls:

- *alchy.model.ModelBase.session()*
- *alchy.model.ModelBase.primary_key()*
- *alchy.model.ModelBase.primary_keys()*
- *alchy.model.ModelBase.primary_attrs()*
- *alchy.model.ModelBase.attrs()*
- *alchy.model.ModelBase.descriptors()*
- *alchy.model.ModelBase.relationships()*
- *alchy.model.ModelBase.column_attrs()*
- *alchy.model.ModelBase.columns()*

CHAPTER 5

API Reference

Includes links to source code.

5.1 API Reference

5.1.1 Model

Declarative base for ORM models.

class `alchy.model.`**`ModelMeta`**(*name*, *bases*, *dct*)
ModelBase's metaclass which provides:

- Tablename autogeneration when `__tablename__` not defined.
- Support for multiple database bindings via *`ModelBase.__bind_key__`*.
- Support for declarative ORM events via *`alchy.events`* decorators or *`ModelBase.__events__`*.

class `alchy.model.`**`ModelBase`**(**args*, ***kargs*)
Base class for creating a declarative base for models.

To create a declarative base:

```
# in project/core.py
from alchy import ModelBase, make_declarative_base

class Base(ModelBase):
    # augument the ModelBase with super powers
    pass

Model = make_declarative_base(Base=Base)


# in project/models/user.py
from project.core import Model

class User(Model):
    # define declarative User model
    pass
```

`__table_args__`
Default table args.

__mapper_args__
Define a default order by when not specified by query operation, e.g.: `{ 'order_by': [column1, column2] }`

__bind_key__
Bind a model to a particular database URI using keys from `Manager.config['SQLALCHEMY_BINDS']`. By default a model will be bound to `Manager.config['SQLALCHEMY_DATABASE_URI']`.

__events__
Register orm event listeners. See *alchy.events* for more details.

query_class
Query class to use for `self.query`.

query
An instance of *query_class*. Can be used to query the database for instances of this model. NOTE: Requires setting `MyClass.query = QueryProperty(session)` when session available. See *make_declarative_base()* for automatic implementation.

__getitem__(*item*)
Proxy getitem to getattr.

__init__(**args*, ***kargs*)
Initialize model instance by calling *update()*.

__iter__()
Implement `__iter__` so model can be converted to dict via `dict()`.

__setitem__(*item*, *value*)
Proxy setitem to setattr.

__to_dict__
Configuration for *to_dict()*. Do any necessary preprocessing and return a set of string attributes which represent the fields which should be returned when calling *to_dict()*.

By default this model is refreshed if it's `__dict__` state is empty and only the ORM descriptor fields are returned.

This is the property to override if you want to return more/less than the default ORM descriptor fields.

Generally, we can usually rely on `__dict__` as a representation of model when it's just been loaded from the database. In this case, whatever values are present in `__dict__` are the loaded values from the database which include/exclude lazy attributes (columns and relationships).

One issue to be aware of is that after a model has been committed (or expired), `__dict__` will be empty. This can be worked around by calling *refresh()* which will reload the data from the database using the default loader strategies.

These are the two main cases this default implementation will try to cover. For anything more complex it would be best to override this property or the *to_dict()* method itself.

classmethod attrs()
Return ORM attributes

classmethod column_attrs()
Return table columns as list of class attributes at the class level.

classmethod columns()
Return table columns.

delete(**args*, ***kargs*)
Call `session.delete()` on `self`

descriptor_dict
Return `__dict__` key-filtered by *descriptors*.

classmethod descriptors()
Return all ORM descriptors

expire(**args*, ***kargs*)
Call `session.expire()` on `self`

expunge(**args*, ***kargs*)
Call `session.expunge()` on `self`

classmethod filter(**args*, ***kargs*)
Proxy to `cls.query.filter()`

classmethod filter_by(**args*, ***kargs*)
Proxy to `cls.query.filter_by()`

flush(**args*, ***kargs*)
Call `session.flush()` on `self`

classmethod get(**args*, ***kargs*)
Proxy to `cls.query.get()`

classmethod get_by(*data_dict=None*, ***kargs*)
Return first instance filtered by values using `cls.query.filter_by()`.

is_modified(**args*, ***kargs*)
Call `session.is_modified()` on `self`

object_session
Return session belonging to `self`

classmethod primary_attrs()
Return class attributes from primary keys.

classmethod primary_key()
Return primary key as either single column (one primary key) or tuple otherwise.

classmethod primary_keys()
Return primary keys as tuple.

query_class
alias of `QueryModel`

refresh(**args*, ***kargs*)
Call `session.refresh()` on `self`

classmethod relationships()
Return ORM relationships

save(**args*, ***kargs*)
Call `session.add()` on `self`

classmethod session()
Return session from query property

to_dict()
Return dict representation of model by filtering fields using *__to_dict__*.

update(*data_dict=None*, ***kargs*)
Update model with arbitrary set of data.

alchy.model.**make_declarative_base**(*session=None*, *Model=None*, *Base=None*, *Meta=None*, *metadata=None*)
Factory function for either creating a new declarative base class or extending a previously defined one.

alchy.model.**extend_declarative_base**(*Model*, *session=None*)
Extend a declarative base class with additional properties.

•Extend *Model* with query property accessor

5.1.2 Query

Query subclass used by Manager as default session query class.

class alchy.query.**Query**(*entities*, *session=None*)
Extension of default Query class used in SQLAlchemy session queries.

DEFAULT_PER_PAGE = 50
Default per_page argument for pagination when per_page not specified.

all_entities
Return list of entities + join_entities present in query.

chain()
Return pydash chaining instance with items returned by `all()`.

See also:

pydash's documentation on chaining

defer(**columns*)
Apply `defer()` to query.

entities
Return list of entity classes present in query.

immediateload(**keys*, ***kargs*)
Apply `immediateload()` to *keys*.

Parameters **keys** (*mixed*) – Either string or column references to join path(s).

Keyword Arguments **options** (*list*) – A list of *LoadOption* to apply to the overall load strategy, i.e., each *LoadOption* will be chained at the end of the load.

Note: Additional keyword args will be passed to initial load creation.

index_by(*callback=None*)
Index items returned by `all()` using *callback*.

join_eager(**keys*, ***kargs*)
Apply `join` + `self.options(contains_eager())`.

Parameters **keys** (*mixed*) – Either string or column references to join path(s).

Keyword Arguments

- **alias** – Join alias or `dict` mapping key names to aliases.
- **options** (*list*) – A list of *LoadOption* to apply to the overall load strategy, i.e., each *LoadOption* will be chained at the end of the load.

join_entities
Return list of the joined entity classes present in query.

joinedload(**keys*, ***kargs*)
Apply `joinedload()` to *keys*.

Parameters **keys** (*mixed*) – Either string or column references to join path(s).

Keyword Arguments **options** (*list*) – A list of *LoadOption* to apply to the overall load strategy, i.e., each *LoadOption* will be chained at the end of the load.

Note: Additional keyword args will be passed to initial load creation.

lazyload(**keys*, ***kargs*)
Apply `lazyload()` to *keys*.

Parameters **keys** (*mixed*) – Either string or column references to join path(s).

Keyword Arguments **options** (*list*) – A list of *LoadOption* to apply to the overall load strategy, i.e., each *LoadOption* will be chained at the end of the load.

Note: Additional keyword args will be passed to initial load creation.

load_only(**columns*)
Apply `load_only()` to query.

map(*callback=None*)
Map *callback* to each item returned by `all()`.

noload(**keys*, ***kargs*)
Apply `noload()` to *keys*.

Parameters **keys** (*mixed*) – Either string or column references to join path(s).

Keyword Arguments **options** (*list*) – A list of *LoadOption* to apply to the overall load strategy, i.e., each *LoadOption* will be chained at the end of the load.

Note: Additional keyword args will be passed to initial load creation.

outerjoin_eager(**keys*, ***kargs*)
Apply `outerjoin` + `self.options(contains_eager())`.

Parameters **keys** (*mixed*) – Either string keys or column references to join path(s).

Keyword Arguments

- **alias** – Join alias or `dict` mapping key names to aliases.
- **options** (*list*) – A list of *LoadOption* to apply to the overall load strategy, i.e., each *LoadOption* will be chained at the end of the load.

page(*page=1*, *per_page=None*)
Return query with limit and offset applied for page.

paginate(*page=1*, *per_page=None*, *error_out=True*)
Return *Pagination* instance using already defined query parameters.

pluck(*column*)
Pluck *column* attribute values from `all()` results and return as list.

reduce(*callback=None*, *initial=None*)
Reduce `all()` using *callback*.

reduce_right(*callback=None*, *initial=None*)
Reduce reversed `all()` using *callback*.

subqueryload(**keys*, ***kargs*)
Apply `subqueryload()` to *keys*.

Parameters **keys** (*mixed*) – Either string or column references to join path(s).

Keyword Arguments **options** (*list*) – A list of *LoadOption* to apply to the overall load strategy, i.e., each *LoadOption* will be chained at the end of the load.

Note: Additional keyword args will be passed to initial load creation.

undefer(**columns*)
Apply `undefer()` to query.

undefer_group(**names*)
Apply `undefer_group()` to query.

class `alchy.query.`**QueryModel**(*entities*, *session=None*)
Class used for default query property class for `mymanager.query`, `mymanager.session.query`, and `MyModel.query`. Can be used in other libraries/implementations when creating a session:

```
from sqlalchemy import orm

from alchy import QueryModel
# or if not using as query property
# from alchy import Query


session = orm.scoped_session(orm.sessionmaker())
session.configure(query_cls=QueryModel)
```

NOTE: If you don't plan to use the query class as a query property, then you can use the *Query* class instead since it won't include features that only work within a query property context.

__search_filters__
All available search filter functions indexed by a canonical name which will be referenced in advanced/simple search. All filter functions should take a single value and return an SQLAlchemy filter expression, i.e., `{key: lambda value: Model.column_name.contains(value)}`

__advanced_search__
Advanced search models search by named parameters. Generally found on advanced search forms where each field maps to a specific database field that will be queried against. If defined as a list, each item should be a key from *__search_filters__*. The matching *__search_filters__* function will be used in the query. If defined as a dict, it should have the same format as *__search_filters__*.

__simple_search__
Simple search models search by phrase (like Google search). Defined like *__advanced_search__*.

__order_by__
Default order-by to use when *alchy.model.ModelBase.query* used.

Model
Return primary entity model class.

advanced_filter(*search_dict=None*)
Return the compiled advanced search filter mapped to *search_dict*.

get_search_filters(*keys*)
Return *__search_filters__* filtered by keys.

search(*search_string=None*, *search_dict=None*, ***search_options*)
Perform combination of simple/advanced searching with optional limit/offset support.

simple_filter(*search_terms=None*)
Return the compiled simple search filter mapped to *search_terms*.

class alchy.query.**QueryProperty**(*session*)
Query property accessor which gives a model access to query capabilities via *alchy.model.ModelBase.query* which is equivalent to `session.query(Model)`.

class alchy.query.**Pagination**(*query*, *page*, *per_page*, *total*, *items*)
Internal helper class returned by *Query.paginate()*. You can also construct it from any other SQLAlchemy query object if you are working with other libraries. Additionally it is possible to pass `None` as query object in which case the *prev* and *next* will no longer work.

has_next = None
True if a next page exists.

has_prev = None
True if a previous page exists.

items = None
The items for the current page.

next(*error_out=False*)
Returns a *Pagination* object for the next page.

next_num = None
Number of the next page.

page = None
The current page number (1 indexed).

pages = None
The total number of pages.

per_page = None
The number of items to be displayed on a page.

prev(*error_out=False*)
Returns a *Pagination* object for the previous page.

prev_num = None
Number of the previous page.

query = None
The query object that was used to create this pagination object.

total = None
The total number of items matching the query.

class alchy.query.**LoadOption**(*strategy*, **args*, ***kargs*)
Chained load option to apply to a load strategy when calling *Query* load methods.

Example usage:

```
qry = (db.session.query(Product)
       .join_eager('category',
                   options=[LoadOption('noload', 'images')]))
```

This would result in the `noload` option being chained to the eager option for `Product.category` and is equilvalent to:

```
    qry = (db.session.query(Product)
           .join('category')
           .options(contains_eager('category').noload('images')))
```

5.1.3 Events

Declarative ORM event decorators and event registration.

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 project.core 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 alchy import events

from project.core import Model

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

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

    @events.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.

class `alchy.events.`**`on_set`**(*attribute*, ***event_kargs*)

Event decorator for the `set` event.

class alchy.events.**on_append**(*attribute*, ***event_kargs*)
Event decorator for the append event.

class alchy.events.**on_remove**(*attribute*, ***event_kargs*)
Event decorator for the remove event.

class alchy.events.**before_delete**(***event_kargs*)
Event decorator for the before_delete event.

class alchy.events.**before_insert**(***event_kargs*)
Event decorator for the before_insert event.

class alchy.events.**before_update**(***event_kargs*)
Event decorator for the before_update event.

class alchy.events.**before_insert_update**(***event_kargs*)
Event decorator for the before_insert and before_update events.

class alchy.events.**after_delete**(***event_kargs*)
Event decorator for the after_delete event.

class alchy.events.**after_insert**(***event_kargs*)
Event decorator for the after_insert event.

class alchy.events.**after_update**(***event_kargs*)
Event decorator for the after_update event.

class alchy.events.**after_insert_update**(***event_kargs*)
Event decorator for the after_insert and after_update events.

class alchy.events.**on_append_result**(***event_kargs*)
Event decorator for the append_result event.

class alchy.events.**on_create_instance**(***event_kargs*)
Event decorator for the create_instance event.

class alchy.events.**on_instrument_class**(***event_kargs*)
Event decorator for the instrument_class event.

class alchy.events.**before_configured**(***event_kargs*)
Event decorator for the before_configured event.

class alchy.events.**after_configured**(***event_kargs*)
Event decorator for the after_configured event.

class alchy.events.**on_mapper_configured**(***event_kargs*)
Event decorator for the mapper_configured event.

class alchy.events.**on_populate_instance**(***event_kargs*)
Event decorator for the populate_instance event.

class alchy.events.**on_translate_row**(***event_kargs*)
Event decorator for the translate_row event.

class alchy.events.**on_expire**(***event_kargs*)
Event decorator for the expire event.

class alchy.events.**on_load**(***event_kargs*)
Event decorator for the load event.

class alchy.events.**on_refresh**(***event_kargs*)
Event decorator for the refresh event.

5.1.4 Search

SQLAlchemy query filter factories usable in *alchy.query.QueryModel.__search_filters__*.

These are factory functions that return common filter operations as functions which are then assigned to the model class' search config attributes. These functions are syntatic sugar to make it easier to define compatible search functions. However, due to the fact that a model's query class has to be defined before the model and given that the model column attributes need to be defined before using the search factories, there are two ways to use the search factories on the query class:

1. Define *alchy.query.QueryModel.__search_filters__* as a property that returns the filter dict.
2. Pass in a callable that returns the column.

For example, *without* *alchy.search* one would define a *alchy.query.QueryModel.__search_filters__* similar to:

```
class UserQuery(QueryModel):
    __search_filters = {
        'email': lambda value: User.email.like(value)
    }

class User(Model):
    query_class = UserQuery
    email = Column(types.String(100))
```

Using *alchy.search* the above then becomes:

```
class UserQuery(QueryModel):
    @property
    def __search_filters__(self):
        return {
            'email': like(User.email)
        }

class User(Model):
    query_class = UserQuery
    email = Column(types.String(100))
```

Or if a callable is passed in:

```
from alchy import search

class UserQuery(QueryModel):
    __search_filters__ = {
        'email': like(lambda: User.email)
    }

class User(Model):
    query_class = UserQuery
    email = Column(types.String(100))
```

The general naming convention for each comparator is:

- positive comparator: `<base>` (e.g. *like()*)
- negative comparator: `not<base>` (e.g. *notlike()*)

The basic call signature for the `search` functions is:

```
# search.<function>(column)
# e.g.
search.contains(email)
```

class alchy.search.**like**(*column*)
Return like filter function using ORM column field.

class alchy.search.**notlike**(*column*)
Return not(like) filter function using ORM column field.

class alchy.search.**ilike**(*column*)
Return ilike filter function using ORM column field.

class alchy.search.**notilike**(*column*)
Return not(ilike) filter function using ORM column field.

class alchy.search.**startswith**(*column*)
Return startswith filter function using ORM column field.

class alchy.search.**notstartswith**(*column*)
Return not(startswith) filter function using ORM column field.

class alchy.search.**endswith**(*column*)
Return endswith filter function using ORM column field.

class alchy.search.**notendswith**(*column*)
Return not(endswith) filter function using ORM column field.

class alchy.search.**contains**(*column*)
Return contains filter function using ORM column field.

class alchy.search.**notcontains**(*column*)
Return not(contains) filter function using ORM column field.

class alchy.search.**icontains**(*column*)
Return icontains filter function using ORM column field.

class alchy.search.**noticontains**(*column*)
Return not(icontains) filter function using ORM column field.

class alchy.search.**in_**(*column*)
Return in_ filter function using ORM column field.

class alchy.search.**notin_**(*column*)
Return not(in_) filter function using ORM column field.

class alchy.search.**eq**(*column*)
Return == filter function using ORM column field.

class alchy.search.**noteq**(*column*)
Return not(==) filter function using ORM column field.

class alchy.search.**gt**(*column*)
Return > filter function using ORM column field.

class alchy.search.**notgt**(*column*)
Return not(>) filter function using ORM column field.

class alchy.search.**ge**(*column*)
Return >= filter function using ORM column field.

class alchy.search.**notge**(*column*)
Return not(>=) filter function using ORM column field.

class alchy.search.**lt**(*column*)
Return < filter function using ORM column field.

class alchy.search.**notlt**(*column*)
Return not(<) filter function using ORM column field.

class alchy.search.**le**(*column*)
Return <= filter function using ORM column field.

class alchy.search.**notle**(*column*)
Return not(<=) filter function using ORM column field.

class alchy.search.**any_**(*column*, *column_operator*)
Return any filter function using ORM relationship field.

class alchy.search.**notany_**(*column*, *column_operator*)
Return not(any) filter function using ORM relationship field.

class alchy.search.**has**(*column*, *column_operator*)
Return has filter function using ORM relationship field.

class alchy.search.**nothas**(*column*, *column_operator*)
Return not(has) filter function using ORM relationship field.

class alchy.search.**eqenum**(*column*, *enum_class*)
Return == filter function using ORM DeclarativeEnum field.

class alchy.search.**noteqenum**(*column*, *enum_class*)
Return not(==) filter function using ORM DeclarativeEnum field.

5.1.5 Types

Collection of custom column types.

class alchy.types.**DeclarativeEnumType**(*enum*, *name=None*)
Column type usable in table column definitions.

class alchy.types.**DeclarativeEnum**
Declarative enumeration.

For example:

```
class OrderStatus(DeclarativeEnum):
    pending = ('p', 'Pending')
    submitted = ('s', 'Submitted')
    complete = ('c', 'Complete')

class Order(Model):
    status = Column(OrderStatus.db_type(), default=OrderStatus.pending)
```

classmethod db_type(*name=None*)
Return database column type for use in table column definitions.

classmethod from_string(*string*)
Return enum symbol given string value.

Raises ValueError – If *string* doesn't correspond to an enum value.

classmethod values()
Return list of possible enum values. Each value is a valid argument to *from_string()*.

5.1.6 Manager

Manager class and mixin.

The *Manager* class helps manage a SQLAlchemy database session as well as provide convenience functions for commons operations.

Configuration

The following configuration values can be passed into a new *Manager* instance as a `dict`, `class`, or `module`.

`SQLALCHEMY_DATABASE_URI`	URI used to connect to the database. Defaults to `sqlite://`.
`SQLALCHEMY_BINDS`	A `dict` that maps bind keys to database URIs. Optionally, in place of a database URI, a configuration `dict` can be used to overrided connection options.
`SQLALCHEMY_ECHO`	When `True` have SQLAlchemy echo all SQL statements. Defaults to `False`.
`SQLALCHEMY_POOL_SIZE`	The size of the database pool. Defaults to the engine's default (usually `5`).
`SQLALCHEMY_POOL_TIMEOUT`	Specifies the connection timeout for the pool. Defaults to `10`.
`SQLALCHEMY_POOL_RECYCLE`	Number of seconds after which a connection is automatically recycled.
`SQLALCHEMY_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.

class `alchy.manager.`**`ManagerMixin`**

Extensions for *Manager.session*.

`add`(**instances*)

Override `session.add()` so it can function like `session.add_all()`.

> **Note:** Supports chaining.

`add_commit`(**instances*)

Add instances to session and commit in one call.

`delete`(**instances*)

Override `session.delete()` so it can function like `session.add_all()`.

> **Note:** Supports chaining.

`delete_commit`(**instances*)

Delete instances to session and commit in one call.

class `alchy.manager.`**`Manager`**(*config=None*, *session_options=None*, *Model=None*, *session_class=None*)

Manager class for database session.

Initialization of *Manager* accepts a config object, session options, and an optional declarative base. If `Model` isn't provided, then a default one is generated using *alchy.model.make_declarative_base()*. The declarative base model is accessible at *Model*.

By default the `session_options` are:

```
{
    'query_cls': alchy.Query,
    'autocommit': False,
    'autoflush': True
}
```

The default *session_class* is `alchy.Session`. If you want to provide your own session class, then it's suggested that you subclass `alchy.Session` and pass it in via *session_class*. This way your subclass will inherit the functionality of `alchy.Session`.

Model = None
Declarative base model class.

__getattr__(*attr*)
Delegate all other attributes to *session*.

binds
Returns config options for all binds.

binds_map
Returns a dictionary with a table->engine mapping. This is suitable for use in `sessionmaker(binds=binds_map)`.

config = None
Database engine configuration options.

create_all(*bind='__all__'*)
Create database schema from models.

create_engine(*uri_or_config*)
Create engine using either a URI or a config dict. If URI supplied, then the default *config* will be used. If config supplied, then URI in config will be used.

create_scoped_session(*options=None*)
Create scoped session which internally calls *create_session()*.

create_session(*options*)
Create session instance using custom Session class that supports multiple bindings.

drop_all(*bind='__all__'*)
Drop tables defined by models.

engine
Return default database engine.

get_engine(*bind=None*)
Return engine associated with bind. Create engine if it doesn't already exist.

get_tables_for_bind(*bind=None*)
Returns a list of all tables relevant for a bind.

metadata
Return *Model* metadata object.

reflect(*bind='__all__'*)
Reflect tables from database.

session = None
Scoped session object.

session_class = None
Class to used for session object.

class `alchy.manager.`**Config**(*defaults=None*)
Configuration loader which acts like a dict but supports loading values from an object limited to `ALL_CAPS_ATTRIBUTES`.

from_object(*obj*)
Pull `dir(obj)` keys from *obj* and set onto `self`.

5.1.7 Session

Session class that supports multiple database binds.

class alchy.session.**Session**(*manager*, ***options*)

The default session used by *alchy.manager.Manager*. It extends the default session system with bind selection.

Parameters

- **manager** (alchy.manager.Manager) – Alchy manager instance
- **options** (*dict*) – pass-through to SessionBase call

get_bind(*mapper=None*, *clause=None*)

Return engine bind using mapper info's bind_key if present.

5.1.8 Utils

Generic utility functions used in package.

alchy.utils.**is_sequence**(*obj*)

Test if *obj* is an iterable but not `dict` or `str`. Mainly used to determine if *obj* can be treated like a `list` for iteration purposes.

alchy.utils.**camelcase_to_underscore**(*string*)

Convert string from `CamelCase` to `under_score`.

alchy.utils.**iterflatten**(*items*)

Return iterator which flattens list/tuple of lists/tuples:

```
>>> to_flatten = [1, [2,3], [4, [5, [6]], 7], 8]
>>> assert list(iterflatten(to_flatten)) == [1,2,3,4,5,6,7,8]
```

alchy.utils.**flatten**(*items*)

Return flattened list of a list/tuple of lists/tuples:

```
>>> assert flatten([1, [2,3], [4, [5, [6]], 7], 8]) == [1,2,3,4,5,6,7,8]
```

CHAPTER 6

Project Info

6.1 License

6.2 Changelog

6.2.1 v2.2.2 (2017-01-03)

- Fix bug in handling of session options when providing explicit `binds` value via `session_options` during `Manager` in initialization. Thanks brianbruggeman!

6.2.2 v2.2.1 (2016-05-18)

- Fix bug with `events.before_delete` where decorator was defined using invalid parent class making it completely non-functional as a decorator.

6.2.3 v2.2.0 (2016-03-21)

- Add `metadata``argument to ``alchy.model.make_declarative_base` to provide custom metaclass for declarative base model. Thanks fabioramponi!

6.2.4 v2.1.0 (2016-03-11)

- Add `Meta` argument to `alchy.model.make_declarative_base` to provide custom metaclass for declarative base model. Thanks elidchan!

6.2.5 v2.0.1 (2015-07-29)

- Make `Session.get_bind() mapper` argument have default value of `None`.

6.2.6 v2.0.0 (2015-04-29)

- Add `Query.index_by`.
- Add `Query.chain`.
- Add `pydash` as dependency and incorporate into existing `Query` methods: `map`, `reduce`, `reduce_right`, and `pluck`.
- Improve logic for setting `__tablename__` to work with all table inheritance styles (joined, single, and concrete), to handle `@declared_attr` columns, and not to duplicate underscore characters. Thanks sethp!
- Modify logic that sets a Model class' `__table_args__` and `__mapper_args__` (unless overridden in subclass) by merging `__global_table_args__` and `__global_mapper_args__` from all classes in the class's `mro()` with `__local_table_args__` and `__local_mapper_args__` from the class itself. A `__{global,local}_{table,mapper}_args__` may be callable or classmethod, in which case it is evaluated on the class whose `__{table,mapper}_args__` is being set. Thanks sethp! (**breaking change**)

6.2.7 v1.5.1 (2015-01-13)

- Add support for callable `__table_args__` and `__local_table_args__`. Thanks sethp!

6.2.8 v1.5.0 (2014-12-16)

- Add `Model.is_modified()`. Thanks sethp!
- Add `Model.filter()`.
- Add `Model.filter_by()`.

6.2.9 v1.4.2 (2014-11-18)

- Add `search.inenum` and `search.notinenum` for performing an `in_` and `not(in_)` comparision using `DeclarativeEnum`.

6.2.10 v1.4.1 (2014-11-17)

- Allow `Model.__bind_key__` to be set at the declarative base level so that model classes can properly inherit it.

6.2.11 v1.4.0 (2014-11-09)

- Make `ModelBase`'s `__table_args__` and `__mapper_args__` inheritable via mixins. Thanks sethp!
- Add `__enum_args__` to `DeclarativeEnum`. Thanks sethp!
- Allow enum name to be overridden when calling `DeclarativeEnum.db_type()`. Thanks sethp!

6.2.12 v1.3.1 (2014-10-14)

- During `Model.update()` when setting a non-list relationship automatically instantiate `dict` values using the relationship model class.

6.2.13 v1.3.0 (2014-10-10)

- Convert null relationships to `{}` when calling `Model.to_dict()` instead of leaving as `None`.

6.2.14 v1.2.0 (2014-10-10)

- During `Model.update()` when setting a list relationship automatically instantiate `dict` values using the relationship model class.

6.2.15 v1.1.2 (2014-09-25)

- Allow `alias` keyword argument to `Query.join_eager()` and `Query.outerjoin_eager()` to be a `dict` mapping aliases to join keys. Enables nested aliases.

6.2.16 v1.1.1 (2014-09-01)

- Fix handling of nested `Model.update()` calls to relationship attributes so that setting relationship to empty `dict` will propagate `None` to relationship attribute value correctly.

6.2.17 v1.1.0 (2014-08-30)

- Add `query.LoadOption` to support nesting load options when calling the `query.Query` load methods: `join_eager`, `outerjoin_eager`, `joinedload`, `immediateload`, `lazyload`, `noload`, and `subqueryload`.

6.2.18 v1.0.0 (2014-08-25)

- Replace usage of `@classproperty` decorators in `ModelBase` with `@classmethod`. Any previously defined class properties now require method access. Affected attributes are: `session`, `primary_key`, `primary_keys`, `primary_attrs`, `attrs`, `descriptors`, `relationships`, `column_attrs`, and `columns`. (**breaking change**)
- Proxy `getitem` and `setitem` access to `getattr` and `setattr` in `ModelBase`. Allows models to be accessed like dictionaries.
- Make `alchy.events` decorators class based.

- Require `alchy.events` decorators to be instantiated using a function call (e.g. `@events.before_update()` instead of `@events.before_update`). (**breaking change**)
- Add `alchy.search` comparators, `eqenum` and `noteqenum`, for comparing `DeclarativeEnum` types.

6.2.19 v0.13.3 (2014-07-26)

- Fix `utils.iterflatten()` by calling `iterflatten()` instead of `flatten` in recursive loop.

6.2.20 v0.13.2 (2014-06-12)

- Add `ModelBase.primary_attrs` class property that returns a list of class attributes that are primary keys.
- Use `ModelBase.primary_attrs` in `QueryModel.search()` so that it handles cases where primary keys have column names that are different than the class attribute name.

6.2.21 v0.13.1 (2014-06-11)

- Modify internals of `QueryModel.search()` to better handle searching on a query object that already has joins and filters applied.

6.2.22 v0.13.0 (2014-06-03)

- Add `search.icontains` and `search.noticontains` for case insensitive contains filter.
- Remove strict update support from `Model.update()`. Require this to be implemented in user-land. (**breaking change**)

6.2.23 v0.12.0 (2014-05-18)

- Merge originating query where clause in `Query.search` so that pagination works properly.
- Add `session_class` argument to `Manager` which can override the default session class used.

6.2.24 v0.11.3 (2014-05-05)

- In `ModelMeta` when checking whether to do tablename autogeneration, tranverse all base classes when trying to determine if a primary key is defined.
- In `ModelMeta` set `bind_key` in `__init__` method instead of `__new__`. This also fixes an issue where `__table_args__` was incorrectly assumed to always be a `dict`.

6.2.25 v0.11.2 (2014-05-05)

- Support `order_by` as list/tuple in `QueryModel.search()`.

6.2.26 v0.11.1 (2014-05-05)

- Fix bug in `QueryModel.search()` where `order_by` wasn't applied in the correct order. Needed to come before limit/offset are applied.

6.2.27 v0.11.0 (2014-05-04)

- PEP8 compliance with default settings.
- Remove `query_property` argument from `make_declarative_base()` and `extend_declarative_base()`. (**breaking change**)
- Add `ModelBase.primary_keys` class property which returns a tuple always (`ModelBase.primary_key` returns a single key if only one present or a tuple if multiple).
- Move location of class `QueryProperty` from `alchy.model` to `alchy.query`. (**breaking change**)
- Create new `Query` subclass named `QueryModel` which is to be used within a query property context. Replace `Query` with `QueryModel` as default query class. (**breaking change**)
- Move `__advanced_search__` and `__simple_search__` class attributes from `ModelBase` to `QueryModel`. (**breaking change**)
- Introduce `QueryModel.__search_filters__` which can define a canonical set of search filters which can then be referenced in the list version of `__advanced_search__` and `__simple_search__`.
- Modify the logic of `QueryModel.search()` to use a subquery joined onto the originating query in order to support pagination when one-to-many and many-to-many joins are present on the originating query. (**breaking change**)
- Support passing in a callable that returns a column attribute for `alchy.search.<method>()`. Allows for `alchy.search.contains(lambda: Foo.id)` to be used at the class attribute level when `Foo.id` will be defined later.
- Add search operators `any_/notany_` and `has/nothas` which can be used for the corresponding relationship operators.

6.2.28 v0.10.0 (2014-04-02)

- Issue warning instead of failing when installed version of SQLAlchemy isn't compatible with `alchy.Query`'s loading API (i.e. missing `sqlalchemy.orm.strategy_options.Load`). This allows `alchy` to be used with earlier versions of SQLAlchemy at user's own risk.
- Add `alchy.search` module which provides compatible search functions for `ModelBase.__advanced_search__` and `ModelBase.__simple_search__`.

6.2.29 v0.9.1 (2014-03-30)

- Change `ModelBase.session` to proxy `ModelBase.query.session`.
- Add `ModelBase.object_session` proxy to `orm.object_session(ModelBase)`.

6.2.30 v0.9.0 (2014-03-26)

- Remove `engine_config_prefix` argument to `Manager()`. (**breaking change**)
- Add explicit `session_options` argument to `Manager()`. (**breaking change**)
- Change the `Manager.config` options to follow Flask-SQLAlchemy. (**breaking change**)
- Allow `Manager.config` to be either a `dict`, `class`, or `module object`.
- Add multiple database engine support using a single `Manager` instance.

- Add `__bind_key__` configuration option for `ModelBase` for binding model to specific database bind (similar to Flask-SQLAlchemy).

6.2.31 v0.8.0 (2014-03-18)

- For `ModelBase.update()` don't nest `update()` calls if field attribute is a `dict`.
- Deprecated `refresh_on_empty` argument to `ModelBase.to_dict()` and instead implement `ModelBase.__to_dict__` configuration property as place to handle processing of model before casting to `dict`. (**breaking change**)
- Add `ModelBase.__to_dict__` configuration property which handles preprocessing for model instance and returns a set of fields as strings to be used as dict keys when calling `to_dict()`.

6.2.32 v0.7.0 (2014-03-13)

- Rename `alchy.ManagerBase` to `alchy.ManagerMixin`. (**breaking change**)
- Add `pylint` support.
- Remove dependency on `six`.

6.2.33 v0.6.0 (2014-03-10)

- Prefix event decorators which did not start with `before_` or `after_` with `on_`. Specifically, `on_set`, `on_append`, `on_remove`, `on_append_result`, `on_create_instance`, `on_instrument_class`, `on_mapper_configured`, `on_populate_instance`, `on_translate_row`, `on_expire`, `on_load`, and `on_refresh`. (**breaking change**)
- Remove lazy engine/session initialization in `Manager`. Require that `Model` and `config` be passed in at init time. While this removes some functionality, it's done to simplify the `Manager` code so that it's more straightforward. If lazy initialization is needed, then a proxy class should be used. (**breaking change**)

6.2.34 v0.5.0 (2014-03-02)

- Add `ModelBase.primary_key` class property for retrieving primary key(s).
- Add `Base=None` argument to `make_declarative_base()` to support passing in a subclass of `ModelBase`. Previously had to create a declarative `Model` to pass in a subclassed `ModelBase`.
- Let any exception occurring in `ModelBase.query` attribute access bubble up (previously, `UnmappedClassError` was caught).
- Python 2.6 and 3.3 support.
- PEP8 compliance.
- New dependency: `six` (for Python 3 support)

6.2.35 v0.4.2 (2014-02-24)

- In `ModelBase.to_dict()` only include fields which are mapper descriptors.
- Support `to_dict` method hook when iterating over objects in `ModelBase.to_dict()`.
- Add `to_dict` method hook to `EnumSymbol` (propagates to `DeclarativeEnum`).

6.2.36 v0.4.1 (2014-02-23)

- Support `__iter__` method in model so that `dict(model)` is equilvalent to `model.to_dict()`.
- Add `refresh_on_empty=True` argument to `ModelBase.to_dict()` which supports calling `ModelBase.refresh()` if `__dict__` is empty.

6.2.37 v0.4.0 (2014-02-23)

- Add `ModelBase.save()` method which adds model instance loaded from session to transaction.
- Add `ModelBase.get_by()` which proxies to `ModelBase.query.filter_by().first()`.
- Add model attribute `events`.
- Add support for multiple event decoration.
- Add named events for all supported events.
- Add composite events for `before_insert_update` and `after_insert_update`.

6.2.38 v0.3.0 (2014-02-07)

- Rename `ModelBase.advanced_search_config` to `ModelBase.__advanced_search__`.
- Rename `ModelBase.simple_search_config` to `ModelBase.__simple_search__`
- Add `ModelMeta` metaclass.
- Implement `__tablename__` autogeneration from class name.
- Add mapper event support via `ModelBase.__events__` and/or `model.event` decorator.

6.2.39 v0.2.1 (2014-02-03)

- Fix reference to `model.make_declarative_base` in `Manager` class.

6.2.40 v0.2.0 (2014-02-02)

- Add default `query_class` to declarative model if none defined.
- Let `model.make_declarative_base()` accept predefined base and just extend its functionality.

6.2.41 v0.1.0 (2014-02-01)

- First release

6.3 Authors

6.3.1 Lead

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

6.3.2 Contributors

- Jeff Knupp, http://www.jeffknupp.com/, jeffknupp@github
- Seth P., seth-p@github
- Eli Chan, eli.d.chan@gmail.com, elidchan@github
- Brian Bruggeman, brian.m.bruggeman@gmail.com, brianbruggeman@github

6.4 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:

6.4.1 Types of Contributions

Report Bugs

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

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

Pydash could always use more documentation, whether as part of the official alchy 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/alchy.

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 :)

6.4.2 Get Started!

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

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

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

6.4.3 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/alchy/pull_requests and make sure that the tests pass for all supported Python versions.

6.5 Kudos

Thank you to SQLAlchemy for providing such a great library to work with.

CHAPTER 7

Indices and Tables

- genindex
- modindex
- search

a

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V