
GraphAlchemy Documentation

Release 0.1.0

Jeffrey Tratner

July 31, 2012

CONTENTS

1	Base Classes	3
2	Creating Declarative Base Classes for SQLAlchemy	5
3	Creating Base Classes for Flask-SQLAlchemy	7
4	Other Methods	9
5	Notes on integrating GraphAlchemy with web frameworks	11
5.1	Flask	11
5.2	Pyramid (prev Pylons)	11
5.3	webapp2	11
5.4	Incompatible frameworks (for now)	11
6	Indices and tables	13
	Python Module Index	15

Contents:

BASE CLASSES

All Node and Edge classes are subclasses of `BaseNode` and `BaseEdge`.

CREATING DECLARATIVE BASE CLASSES FOR SQLALCHEMY

```
graphalchemy.sqlmodels.create_base_classes(NodeClass, EdgeClass, [NodeTable = None,  
                                                                    [EdgeTable = None, [declared_attr, [Column,  
                                                                    [Integer, [Unicode, [Float, [Boolean,  
                                                                    [ForeignKey, [relationship, [backref, [Base =  
                                                                    None,)
```

creates base classes (BaseEdge and BaseNode) for use as mixins for graph nodes and edges. ALL parameters must be strings convertible to unicode! Classes need to be subclassed/composited with a declarative_base class

Parameters:

param NodeTable the table for node (unicode)

param NodeClass the class for node (unicode)

param EdgeTable the table for edge (unicode)

param EdgeClass the class for edge (unicode)

param Base (optional) if a Base is passed, it will be added to the class type for you, thereby requiring no subclassing on your part.

type Base SQLAlchemy declarative base

Returns tuple of Node, Edge classes

Return type (Node, Edge)

NOTE: To overwrite the default inheritance, you can pass in any SQLAlchemy classes used in creating the functions:

```
declared_attr, Column, Unicode, Integer, Float, Boolean,  
relationship, backref, ForeignKey
```


CREATING BASE CLASSES FOR FLASK-SQLALCHEMY

```
graphalchemy.sqlmodels.create_flask_classes(db, NodeClass, EdgeClass,  
                                           NodeTable=None, EdgeTable=None)
```

Convenience method for creating *Node* and *Edge* base classes for use with Flask-SQLAlchemy. Has nearly the same signature as `create_base_classes()` But does not take in any overriding methods. Only *NodeClass* and *EdgeClass* are required.

The one required parameter is *db*, which you must create first from the sqlalchemy directions. Example usage:

```
>>> from flask import Flask
>>> from flask.ext.sqlalchemy import SQLAlchemy
>>> from graphalchemy.sqlmodelss import create_flask_classes
>>>
>>> app = Flask(__name__)
>>> app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db'
>>> db = SQLAlchemy(app)
>>>
>>> Node, Edge = create_flask_classes(db, "Node", "Edge")
```

At this point, you can subclass *Node* and *Edge* to add additional traits; however, both *Node* and *Edge* will *already* be subclasses of *db.Model*, so you don't need to mix that in.

Or you can just start up your database with:

```
>>> db.create_all()
```

Otherwise, the classes created by `create_flask_classes()` and `create_base_classes()` are pretty much the same, except that Flask-SQLAlchemy provides some additional features that can be accessed on the Models.

OTHER METHODS

```
graphalchemy.sqlmodels.sqlite_connect (dbpath, metadata[, create_engine[, sessionmaker[,  
echo=True]]])
```

return an sqllite connection to the given dbpath. Optional arguments default to sqlalchemy functions.

Parameter:

param dbpath path (relative or absolute) to database (unicode/string) NOT “sqlite://”

param metadata something that supports create_all() to create/load tables and has a bind attribute

type metadata should create tables with *create_all()*

type create_engine function (dbpath) -> engine

param sessionmaker (optional) must take *bind=engine*, return a class that can be called to create a session

type sessionmaker function (bind=engine) -> Session

param event event creator for engine (from SQLAlchemy)

param bool enforce_fk set database to enforce foreign key relationships

default enforce_fk True

Returns:

returns (engine, session)

raises ValueError if passed a path that does not exist or a non-valid path.

NOTES ON INTEGRATING GRAPHALCHEMY WITH WEB FRAMEWORKS

5.1 Flask

There are a few different options for *using Flask with SQLAlchemy*, which you can read about on [Flask's docs](#).

5.1.1 Using Flask-SQLAlchemy plugin

The only real caveat is that if you want to use Flask-SQLAlchemy, you should use the `:func~graphalchemy.sqlmodels.create_flask_classes` function, and pass it an SQLAlchemy instance (usually called `db`).

5.2 Pyramid (prev Pylons)

Pyramid has a cookbook entry on *using SQLAlchemy with Pyramid*. But it's basically just normal use of SQLAlchemy, with a few specific notes on using Pyramid's `DBSession` for sessions and some advanced topics that aren't really relevant here.

5.3 webapp2

`webapp2` should work without a problem, just import it and use it like you would with SQLAlchemy (probably just use it straight up?)

5.4 Incompatible frameworks (for now)

5.4.1 Google App Engine

Google App Engine doesn't have a (standard) relational database, but a future version of `graphalchemy` will have a version that works with App Engine (though it may or may not be a really efficient solution).

5.4.2 Django

Django uses its own ORM (which you can replace with SQLAlchemy, but it means you lose much of Django's functionality). There may be a future version of `graphalchemy` that will support Django, but for the moment, you'd have to choose to use SQLAlchemy.

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

g

graphalchemistry, 1