GeoAlchemy Documentation

Release 0.4.1

Sanjiv Singh

February 01, 2014

Contents

GIS Support for SQLAlchemy.

CHAPTER 1

Contents:

1.1 Introduction

GeoAlchemy is an extension of SQLAlchemy. It provides support for Geospatial data types at the ORM layer using SQLAlchemy. It aims to support spatial operations and relations specified by the Open Geospatial Consortium (OGC). The project started under Google Summer of Code Program under the mentorship of Mark Ramm-Christensen.

1.1.1 Requirements

Requires SQLAlchemy > 0.6. Supported on Python 2.5 and Python 2.6. Should also work with Python 2.4 but has not been tested. It also requires a supported spatial database.

1.1.2 Supported Spatial Databases

At present PostGIS, Spatialite, MySQL, Oracle, and MS SQL Server 2008 are supported.

1.1.3 Support

GeoAlchemy is at an early stage of development. Its mailing list is available on Google Groups. The source code can be found on BitBucket. Also, feel free to email the author directly to send bugreports, feature requests, patches, etc.

1.1.4 Installation

To install type as usual:

```
$ easy_install GeoAlchemy
```

Or, download the package, change into geoalchemy dir and type:

```
$ python setup.py install
```

1.1.5 Documentation

Documentation is available online at http://geoalchemy.org. You can also generate full documentation using sphinx by doing *make html* in the *doc* dir and pointing the browser to *doc/_build/index.html*.

1.1.6 Package Contents

geoalchemy/ Source code of the project.

geoalchemy/tests/ Unittests for GeoAlchemy.

doc/ Documentation source.

examples/ A few examples demonstrating usage.

1.1.7 License

GeoAlchemy is released under the MIT License.

1.1.8 Author

Sanjiv Singh

Mentored by : Mark Ramm-Christensen

The project was started based on example code written by Michael Bayer, the author of SQLAlchemy project.

1.2 GeoAlchemy Installation

The prerequisites for GeoAlchemy installation are:

- Python
- A Supported Spatial Database (either of PostGIS, MySQL, Spatialite)
- Python DB-API for the database
- Setuptools

GeoAlchemy can then be installed from cheese shop using the *easy_install* command:

```
$ easy_install GeoAlchemy
```

This will download and install the following packages from the Python Package Index aka *Cheeseshop*:

- SQLAlchemy
- GeoAlchemy

You are now ready to use GeoAlchemy.

Alternatively GeoAlchemy can be installed from source.

1.3 GeoAlchemy Tutorial

This is a short tutorial to illustrate the use of GeoAlchemy and to get you started quickly. In this tutorial we will create a simple model using three vector layers in PostGIS. The same tutorial will be applicable for MySQL. Minor changes are required for spatialite as the spatialite extension needs to be loaded. Spatialite specific details are given here.

1.3.1 Setting up PostGIS

This tutorial requires a working PostGIS installation. The PostGIS Documentation has extensive installation instructions. Create a spatially enabled database called *gis* as per instructions given in the PostGIS documentation. Also create a new user and grant it permissions on this database.

On Ubuntu the following steps have to executed to create the database.

```
sudo su postgres
createdb -E UNICODE gis
createlang plpgsql gis
psql -d gis -f /usr/share/postgresql-8.3-postgis/lwpostgis.sql
psql -d gis -f /usr/share/postgresql-8.3-postgis/spatial_ref_sys.sql

#Create a new user (if a user named 'gis' does not exist already)
createuser -P gis

#Grant permissions to user 'gis' on the new database
psql gis
grant all on database gis to "gis";
grant all on spatial_ref_sys to "gis";
grant all on geometry_columns to "gis";
\q
```

1.3.2 Initializing SQLAlchemy

First of all we need some code to initialize sqlalchemy and to create an engine with a database specific dialect. We use the standard sqlalchemy dburi format to create the engine. For more details refer to the sqlalchemy documentation. We use this engine to create a session. A session could be either bound or unbound. In this example we will create a bound session.

```
from sqlalchemy import *
from sqlalchemy.orm import *

engine = create_engine('postgresql://gis:password@localhost/gis', echo=True)
Session = sessionmaker(bind=engine)
session = Session()
```

1.3.3 Model Definition

Let us assume we have to model the following GIS layers in the database:

- spots - represented as point geometry
- roads - represented as line geometry, and
- lakes - represented as polygon geometry

In SQLAlchemy, models can be defined either by declaring the model classes and database tables separately and then mapping the classes to the tables using mapper configuration, or they can be defined declaratively using the new declarative extension. In this example we use the SQLAlchemy declarative extension to define the model. Notes on how to use GeoAlchemy with a non-declarative mapping are stated in the Usage Notes. We also create a metadata object that holds the schema information of all database objects and will thus be useful for creating the objects in the database.

```
from sqlalchemy.ext.declarative import declarative_base
from datetime import datetime
from geoalchemy import *

metadata = MetaData(engine)
Base = declarative_base(metadata=metadata)

class Spot(Base):
    __tablename__ = 'spots'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    height = Column(Integer)
    created = Column(DateTime, default=datetime.now())
    geom = GeometryColumn(Point(2))

class Road(Base):
    __tablename__ = 'roads'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    width = Column(Integer)
    created = Column(DateTime, default=datetime.now())
    geom = GeometryColumn(LineString(2))

class Lake(Base):
    __tablename__ = 'lakes'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    depth = Column(Integer)
    created = Column(DateTime, default=datetime.now())
    geom = GeometryColumn(Polygon(2))

GeometryDDL(Spot.__table__)
GeometryDDL(Road.__table__)
GeometryDDL(Lake.__table__)
```

In the above model definition we have defined an *id* field for each class which is also the primary key in the database. We have defined a set of standard attributes using datatypes available under *sqlalchemy.types*. We have also created a *geometry attribute* for each class using *GeometryColumn* and Point, LineString and Polygon datatypes of GeoAlchemy. Here we pass the dimension parameter to *GeometryColumn*. We leave out the *srid* parameter which defaults to *4326*. This means that our geometry values will be in Geographic Latitude and Longitude coordinate system.

Finally we have used *GeometryDDL*, a DDL Extension for geometry data types that support special DDLs required for creation of geometry fields in the database.

The above declaration is completely database independent, it could also be used for MySQL or Spatialite. Queries written with GeoAlchemy are generic too. GeoAlchemy translates these generic expressions into the function names that are known by the database, that is currently in use. If you want to use a database specific function on a geometry column, like *AsKML* in PostGIS, you will have to set a comparator when defining your mapping. For the above example the mapping for *Spot* then would look like this:

```
from geoalchemy.postgis import PGComparator

class Spot(Base):
    __tablename__ = 'spots'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    height = Column(Integer)
    created = Column(DateTime, default=datetime.now())
    geom = GeometryColumn(Point(2), comparator=PGComparator)

    # [..]
```

Now you can also use PostGIS specific functions on geometry columns.

```
>>> s = session.query(Spot).filter(Spot.geom.kml == '<Point><coordinates>-81.4,38.08</coordinates></P
>>> session.scalar(s.geom.wkt)
'POINT(-81.4 38.08)'
```

Note that you do not have to set a comparator, when you want to execute a database specific function on a geometry attribute of an object (*s.geom.kml*) or when you are directly using a function (*pg_functions.kml('POINT(..)')*). You only have to set a comparator, when you are using a function on a geometry column (*Spot.geom.kml*).

The following comparators and database specific function declarations are available:

- PostGIS: *geoalchemy.postgis.PGComparator* and *geoalchemy.postgis.pg_functions*
- MySQL: *geoalchemy.mysql.MySQLComparator* and *geoalchemy.mysql.mysql_functions*
- Spatialite: *geoalchemy.spatialite.SQLiteComparator* and *geoalchemy.spatialite.sqlite_functions*
- Oracle: *geoalchemy.oracle.OracleComparator* and *geoalchemy.oracle.oracle_functions*

1.3.4 Creating Database Tables

Now we use the metadata object to create our tables. On subsequent use we will also first drop the tables so that the database is emptied before creating tables.

```
metadata.drop_all()   # comment this on first occassion
metadata.create_all()
```

1.3.5 Adding GIS Features

Adding GIS features is now as simple as instantiating the model classes and adding them to the SQLAlchemy session object that we created earlier. GeoAlchemy enables creation of spatial attributes specified using the Well Known Text (WKT) format using GeoAlchemy *WKTSpatialElement* class.

```
    wkt_spot1 = "POINT(-81.40 38.08)"
    spot1 = Spot(name="Gas Station", height=240.8, geom=WKTSpatialElement(wkt_spot1))
    wkt_spot2 = "POINT(-81.42 37.65)"
    spot2 = Spot(name="Restaurant", height=233.6, geom=WKTSpatialElement(wkt_spot2))

    wkt_road1 = "LINESTRING(-80.3 38.2, -81.03 38.04, -81.2 37.89)"
    road1 = Road(name="Peter St", width=6.0, geom=WKTSpatialElement(wkt_road1))
    wkt_road2 = "LINESTRING(-79.8 38.5, -80.03 38.2, -80.2 37.89)"
    road2 = Road(name="George Ave", width=8.0, geom=WKTSpatialElement(wkt_road2))

    wkt_lake1 = "POLYGON((-81.3 37.2, -80.63 38.04, -80.02 37.49, -81.3 37.2))"
    lake1 = Lake(name="Lake Juliet", depth=36.0, geom=WKTSpatialElement(wkt_lake1))
    wkt_lake2 = "POLYGON((-79.8 38.5, -80.03 38.2, -80.02 37.89, -79.92 37.75, -79.8 38.5))"
    lake2 = Lake(name="Lake Blue", depth=58.0, geom=WKTSpatialElement(wkt_lake2))

session.add_all([spot1, spot2, road1, road2, lake1, lake2])
session.commit()
```

If you want to insert a geometry that has a different spatial reference system than your geometry column, a transformation is added automatically.

```
geom_spot3 = WKTSpatialElement('POINT(30250865 -610981)', 2249)
spot3 = Spot(name="Park", height=53.2, geom=geom_spot3)
session.add(spot3)
session.commit()
```

Scripts for creating sample gis objects as shown above are available in the examples directory. You could run those scripts to create the database tables and the gis objects. Running them with -i option to the interpreter will drop you at the interactive interpreter prompt. You can then follow the rest of the tutorial on the interpreter.

```
$ python -i examples/tutorial.py
>>>
```

1.3.6 Performing Spatial Queries

The GeoAlchemy project intends to cover most of the spatial operations and spatial relations supported by the underlying spatial database. Some of these are shown below and the rest are documented in the reference docs.

Functions to obtain geometry value in different formats

```
>>> s = session.query(Spot).get(1)
>>> session.scalar(s.geom.wkt)
'POINT(-81.42 37.65)'
>>> session.scalar(s.geom.gml)
'<gml:Point srsName="EPSG:4326"><gml:coordinates>-81.42,37.65</gml:coordinates></gml:Point>'
>>> session.scalar(s.geom.kml)
'<Point><coordinates>-81.42,37.65</coordinates></Point>'
>>> import binascii
>>> binascii.hexlify(session.scalar(s.geom.wkb))
'01010000007b14ae47e15a54c03333333333d34240'
```

Note that for all commands above a new query had to be made to the database. Internally GeoAlchemy uses Well-Known-Binary (WKB) to fetch the geometry, that belongs to an object of a mapped class. All the time an object is queried, the geometry for this object is loaded in WKB.

You can also access this internal WKB geometry directly and use it for example to create a Shapely geometry. In this case, no new query has to be made to the database.

```
>>> binascii.hexlify(s.geom.geom_wkb)
    '01010000007b14ae47e15a54c03333333333d34240'
```

Functions to obtain the geometry type, coordinates, etc

```
>>> s = session.query(Spot).filter(Spot.height > 240).first()
>>>
>>> session.scalar(s.geom.geometry_type)
'ST_Point'
>>> session.scalar(s.geom.x)
-81.420000000000002
>>> session.scalar(s.geom.y)
37.649999999999999
>>> s.geom.coords(session)
[-81.420000000000002, 37.649999999999999]
```

Spatial operations that return new geometries

```
>>> r = session.query(Road).first()
>>> l = session.query(Lake).first()
>>>
>>> buffer_geom = DBSpatialElement(session.scalar(r.geom.buffer(10.0)))
>>> session.scalar(buffer_geom.wkt)
'POLYGON((-77.4495270615657 28.6622373442108,-77.9569183543725 28.4304851371862,-79.8646930595254 27
>>> envelope_geom = DBSpatialElement(session.scalar(r.geom.envelope))
>>> session.scalar(envelope_geom.wkt)
'POLYGON((-81.2000045776367 37.8899993896484,-81.2000045776367 38.2000007629395,-80.2999954223633 38
>>> cv_geom = DBSpatialElement(session.scalar(r.geom.convex_hull))
>>> session.scalar(cv_geom.wkt)
'POLYGON((-81.2 37.89,-81.03 38.04,-80.3 38.2,-81.2 37.89))'
```

Spatial relations for filtering features

```
>>> r = session.query(Road).first()
>>> l = session.query(Lake).first()

>>> session.query(Road).filter(Road.geom.intersects(r.geom)).count()
1L
>>> session.query(Lake).filter(Lake.geom.touches(r.geom)).count()
0L
>>> session.query(Spot).filter(Spot.geom.covered_by(l.geom)).count()
0L
>>> session.scalar(r.geom.touches(l.geom))
False
>>> box = 'POLYGON((-82 38, -80 38, -80 39, -82 39, -82 38))'
>>> session.query(Spot).filter(Spot.geom.within(box)).count()
1L
```

Using the generic functions from *geoalchemy.functions* or the database specific functions from *geoalchemy.postgis.pg_functions*, *geoalchemy.mysql.mysql_functions* and *geoalchemy.spatialite.sqlite_functions*, more complex queries can be made.

```
>>> from geoalchemy.functions import functions
>>> session.query(Spot).filter(Spot.geom.within(functions.buffer(functions.centroid(box), 10, 2))).co
2L
>>> from geoalchemy.postgis import pg_functions
>>> point = 'POINT(-82 38)'
>>> session.scalar(pg_functions.gml(functions.transform(point, 2249)))
'<gml:Point srsName="EPSG:2249"><gml:coordinates>-2369733.76351267,1553066.7062767</gml:coordinates><
```

1.4 Usage notes

1.4.1 Notes for Spatialite

Although Python2.5 and its higher versions include sqlite support, while using spatialite in python we have to use the db-api module provided by pysqlite2. So we have to install pysqlite2 separately. Also, by default the pysqlite2 disables extension loading. In order to enable extension loading, we have to build it ourselves. Download the pysqlite tarball, open the file setup.cfg and comment out the line that reads:

```
define=SQLITE_OMIT_LOAD_EXTENSION
```

Now save the file and then build and install pysqlite2:

```
$ python setup.py install
```

Now, we are ready to use spatialite in our code. While importing pysqlite in our code we must ensure that we are importing from the newly installed pysqlite2 and not from the pysqlite library included in python. Also pass the imported module as a parameter to sqlalchemy create_engine function so that sqlalchemy uses this module instead of the default module: to be used:

```
from pysqlite2 import dbapi2 as sqlite

engine = create_engine('sqlite:////tmp/devdata.db', module=sqlite, echo=True)
```

Enable sqlite extension loading and load the spatialite extension:

```
connection = engine.raw_connection().connection
connection.enable_load_extension(True)
metadata = MetaData(engine)
session = sessionmaker(bind=engine)()
session.execute("select load_extension('/usr/local/lib/libspatialite.so')")
```

When using for the database for the first time we have to initialize the database. Details are given in spatialite documentation.

```
sqlite3> SELECT InitSpatialMetaData();
sqlite3> INSERT INTO spatial_ref_sys (srid, auth_name, auth_srid, ref_sys_name, proj4text) VALUES (43
```

1.4.2 Notes for Oracle

Oracle support was tested using Oracle Standard Edition 11g Release 2 with Oracle Locator. GeoAlchemy does not work with Oracle 10g Express Edition, because it does not contain an internal JVM which is required for various geometry functions.

cx_Oracle

GeoAlchemy was tested using the Python driver cx_Oracle. cx_Oracle requires the environment variables ORACLE_HOME and LD_LIBRARY_PATH. You can set them in Python before importing the module.

```
import os
os.environ['ORACLE_HOME'] = '/usr/lib/oracle/xe/app/oracle/product/10.2.0/server'
os.environ['LD_LIBRARY'] = '/usr/lib/oracle/xe/app/oracle/product/10.2.0/server/lib'

import cx_Oracle
```

Dimension Information Array (DIMINFO)

When using a spatial index for a geometry column, Oracle requires a dimension information array (Geometry Metadata Views: DIMINFO) for this column. If you are using the DDL extension of GeoAlchemy to create your tables, you will have to set a DIMINFO parameter in your table mapping.

```
diminfo = "MDSYS.SDO_DIM_ARRAY("\
            "MDSYS.SDO_DIM_ELEMENT('LONGITUDE', -180, 180, 0.000000005),"\
            "MDSYS.SDO_DIM_ELEMENT('LATITUDE', -90, 90, 0.000000005)"\
            ")"

class Lake(Base):
    __tablename__ = ’lakes’

    lake_id = Column(Integer, Sequence(’lakes_id_seq’), primary_key=True)
    lake_name = Column(String(40))
    lake_geom = GeometryColumn(Polygon(2, diminfo=diminfo), comparator=OracleComparator)
```

Some geometry functions also expect a DIMINFO array as parameter for every geometry that is passed in as parameter. For parameters that are geometry columns or that were queried from the database, GeoAlchemy automatically will insert a subquery that selects the DIMINFO array from the metadata view `ALL_SDO_GEOM_METADATA` connected to the geometry.

Following functions expect a DIMINFO array:

- functions.length
- functions.area
- functions.centroid
- functions.buffer
- functions.convex_hull
- functions.distance
- functions.within_distance
- functions.intersection
- all functions in oracle_functions.sdo_geom_sdo_*

Example:

```
session.query(Lake).filter(Lake.lake_geom.area > 0).first()

l = session.query(Lake).filter(Lake.lake_name==’Lake White’).one()
session.scalar(l.lake_geom.area)
```

For geometries that do not have an entry in `ALL_SDO_GEOM_METADATA`, you manually have to set a DIMINFO array.

```
from sqlalchemy.sql.expression import text
diminfo = text("MDSYS.SDO_DIM_ARRAY("\
        "MDSYS.SDO_DIM_ELEMENT(’LONGITUDE’, -180, 180, 0.000000005),"\
        "MDSYS.SDO_DIM_ELEMENT(’LATITUDE’, -90, 90, 0.000000005)"\
        ")")

session.scalar(functions.area(WKTSpatialElement(’POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))’, 4326), diminfo
```

If you want to use a different DIMINFO array or a tolerance value as parameter, you have to set the flag `auto_diminfo`, so that GeoAlchemy is not trying to insert the DIMINFO array for you.

```
session.scalar(l.lake_geom.area(0.000000005, auto_diminfo=False))
```

Using Oracle functions

ST_GEOMETRY functions

Oracle supports the SQL/MM geometry type `ST_GEOMETRY` (see ST_GEOMETRY and SDO_GEOMETRY Interoperability). The functions inside the package `MDSYS.OGC_*` implement the specification *OGC Simple Feature Access* and the functions inside the package `MDSYS.ST_GEOMETRY.ST_*` the specification *SQL/MM*.

GeoAlchemy uses the functions from these two packages for all functions in `functions.*` except for the following:

- functions.wkt
- functions.wkb
- functions.length
- functions.area
- functions.centroid
- functions.buffer
- functions.convex_hull
- functions.distance
- functions.within_distance
- functions.intersection
- functions.transform

The functions in `MDSYS.OGC_*` and `MDSYS.ST_GEOMETRY.ST_*` expect the geometry type `ST_GEOMETRY`. GeoAlchemy automatically adds a cast from `SDO_GEOMETRY`.

Some functions, like `OGC_X` or `OGC_IsClosed`, also only work for subtypes of `ST_GEOMETRY`, so that a cast to the subtype has to be added. If you are executing a function on a geometry column or on a geometry attribute of a mapped instance, GeoAlchemy will insert the cast to the subtype. Otherwise you will have to specify the geometry type:

```
session.scalar(functions.is_ring(WKTSpatialElement('LINESTRING(0 0, 0 1, 1 0, 1 1, 0 0)', geometry_ty
```

Functions that return a Boolean value

Functions in Oracle return 'TRUE' or 1/0 instead of a Boolean value. When those functions are used inside a where clause, GeoAlchemy adds a comparision, for example:

```
session.query(Lake).filter(
            Lake.lake_geom.sdo_contains('POINT(-88.9055734203822 43.0048567324841)'))

# is compiled to:
SELECT ..
FROM lakes
WHERE SDO_CONTAINS(lakes.lake_geom, MDSYS.SDO_GEOMETRY(:SDO_GEOMETRY_1, :SDO_GEOMETRY_2)) = 'TRUE'
```

Measurement functions

Measurement functions like `area`, `length`, `distance` and `within_distance` by default use meter as unit for geodetic data (like WGS 84) and otherwise the unit 'associated with the data'. If you want to use a different unit, you can set it as parameter.

```
session.scalar(l.lake_geom.area('unit=SQ_KM'))
```

Member functions

For member functions of `SDO_Geometry` (currently only `oracle_functions.gtype` and `oracle_functions.dims`) a table alias has to be used, when the function is called on a geometry column.

```
from sqlalchemy.orm import aliased
spot_alias = aliased(Spot)

session.query(spot_alias).filter(spot_alias.spot_location.dims == 2).first()
```

Inserting NULL in geometry columns

When synchronizing mapped instances with the database, SQLAlchemy uses bind parameters for insert/update statements. Unfortunately cx_Oracle currently does not support the insertion of `None` into geometry columns when using bind parameters.

If you want to insert `NULL` you have to use the constant `oracle.ORACLE_NULL_GEOMETRY`:

```
from geoalchemy.oracle import ORACLE_NULL_GEOMETRY

spot_null = Spot(spot_height=None, spot_location=ORACLE_NULL_GEOMETRY)
session.add(spot_null)
session.commit();
```

Notes on DBSpatialElement

For the other databases the result of a function call, that returned a new geometry, could be wrapped into a DBSpatialElement, so that new queries could be executed on that instance.

For Oracle the returned geometry is an object of `SDO_Geometry`. cx_Oracle currently does not support Oracle objects as argument in queries, so `DBSpatialElement` can not be used for Oracle.

1.4.3 Notes for MS Sql Server

The MS Sql Server spatial support has been tested using MS SQL Server 2008, connecting to it via pyODBC on Windows.

There is one important difference between SQL Server 2008 spatial support and PostGIS in that it is **not** possible to restrict the spatial column to a specific type of geometry. All columns will be `geoalchemy.geometry.Geometry`.

Supported functions

Most of the standard functions defined in GeoAlchemy are available and work as expected, but there are a few exceptions:

- g.centroid – Only returns results for `Polygon` and `MultiPolygon`. Returns 'NULL' for all other `Geometry`
- g.envelope – Will always return a `Polygon` regardless of the type of `Geometry` it was called on
- g.buffer – Only supports the buffer distance as a parameter
- g.transform – Not defined

- g.within_distance – Not defined
- g.covers – Not defined
- g.covers_by – Not defined
- g.intersection – Not defined

MS Sql Server specific functions

Sql Server provides a number of additional spatial functions, details of which can be found in the documentation of `geoalchemy.mssql.ms_functions`. These additional functions can be used like any other function, or via *ms_functions.function_name*.

```
session.query(Road).filter(Road.road_geom.instance_of('LINESTRING'))
```

```
from geoalchemy.mssql import ms_functions
ms_functions.buffer_with_tolerance('POINT(-88.5945861592357 42.9480095987261)', 10, 2, 0)
```

- `text_zm`
- `buffer_with_tolerance`
- `filter`
- `instance_of`
- `m`
- `make_valid`
- `reduce`
- `to_string`
- `z`

Creating a spatial index

Sql Server requires a bounding box that is equal to the extent of the data in the indexed spatial column. As the necessary information is not available when the DDL statements are executed no spatial indexes are created by default. To create a spatial index the bounding box must be specified explicitly when the `GeometryColumn` is defined:

```
class Road(Base):
    __tablename__ = 'ROADS'

    road_id = Column(Integer, primary_key=True)
    road_name = Column(String(255))
    road_geom = GeometryColumn(Geometry(2, bounding_box='(xmin=-180, ymin=-90, xmax=180, ymax=90)'), r
```

Inserting NULL geometries

Due to a bug in the underlying libraries there is currently no support for inserting NULL geometries that have a *None* geometry. The following code will not work:

```
session.add(Road(road_name=u'Destroyed road', road_geom=None))
```

To insert NULL you must use `geoalchemy.mssql.MS_SPATIAL_NULL` to explicitly specify the NULL geometry.

```
session.add(Road(road_name=u'Destroyed road', road_geom=MS_SPATIAL_NULL))
```

This is an issue with pyODBC and can be tracked via http://code.google.com/p/pyodbc/issues/detail?id=103.

1.4.4 Notes on non-declarative mapping

In some cases it may be favored to define the database tables and the model classes separately. GeoAlchemy also supports this way of non-declarative mapping. The following example demonstrates how a mapping can be set up.

```
from sqlalchemy import *
from sqlalchemy.orm import *
from geoalchemy import *
from geoalchemy.postgis import PGComparator

engine = create_engine('postgresql://gis:gis@localhost/gis', echo=True)
metadata = MetaData(engine)
session = sessionmaker(bind=engine)()

# define table
spots_table = Table('spots', metadata,
                    Column('spot_id', Integer, primary_key=True),
                    Column('spot_height', Numeric),
                    GeometryExtensionColumn('spot_location', Geometry(2)))

# define class
class Spot(object):
    def __init__(self, spot_id=None, spot_height=None, spot_location=None):
        self.spot_id = spot_id
        self.spot_height = spot_height
        self.spot_location = spot_location

# set up the mapping between table and class
mapper(Spot, spots_table, properties={
            'spot_location': GeometryColumn(spots_table.c.spot_location,
                                            comparator=PGComparator)})

# enable the DDL extension
GeometryDDL(spots_table)

# create table
metadata.create_all()

# add object
session.add(Spot(spot_height=420.40, spot_location='POINT(-88.5945861592357 42.9480095987261)'))
session.commit()

s = session.query(Spot).get(1)
print session.scalar(s.spot_location.wkt)
```

1.5 Tutorial : FeatureServer Using GeoAlchemy

1.5.1 Introduction

FeatureServer is a simple Python-based geographic feature server. It allows you to store geographic vector features in a number of different backends, and interact with them – creating, updating, and deleting – via a REST-based API. It is distributed under a BSD-like open source license.

text.geo, the TurboGears2 extension for GIS makes it possible to use FeatureServer with GeoAlchemy as datasource. It provides two main components:

- GeoAlchemy Datasource - This allows geographic features to be stored in any of the spatial databases supported by GeoAlchemy.
- FeatureServer Controller - This creates a new controller that reads the config and makes use of the FeatureServer API to dispatch requests to featureserver.

1.5.2 About this Tutorial

In this tutorial we will create a TG2 app and use tgext.geo extension to configure and use featureserver to store, manipulate and retreive GIS features in a PostGIS database using GeoAlchemy as the ORM layer.

1.5.3 Installation

It is assumed that a fresh virtualenv has been created and TG2 installed following the TG2 Installation Guide. Install tgext.geo using easy_install:

```
(tg2env)$ easy_install -i http://www.turbogears.org/2.0/downloads/current/index/ tgext.geo
```

We assume that a PostgreSQL server is installed and ready for use. Install PostGIS and create a new PostGIS enabled database called *gis*. Refer the docs here to achieve this. Install GeoAlchemy and the python db-api for postgres:

```
(tg2env)$ easy_install GeoAlchemy
(tg2env)$ easy_install egenix-mx-base
(tg2env)$ easy_install psycopg2
```

1.5.4 Creating a New TG2 App

Create a new TG2 app with gis capability:

```
(tg2env)$ paster quickstart TGFeature --geo
(tg2env)$ cd TGFeature
```

1.5.5 Model Definition for Features

We assume that we have to model a layer of roads in our application. We open the tgfeature/model/__init__.py file in the package and add the following model definition:

```
class Road(DeclarativeBase):
    __tablename__ = 'roads'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    width = Column(Integer)
    created = Column(DateTime, default=datetime.now())
    geom = GeometryColumn(LineString(2))

GeometryDDL(Road.__table__)
```

Apart from the standard attributes, we have defined a spatial attribute called *geom* as a *GeometryColumn*. We will use this attribute to store geometry values of data type *LineString* in the database. GeoAlchemy supports other geometry types such as Point, Polygon and Mutiple Geometries. We also pass the dimension of the geometry as a parameter. The Geometry type takes another parameter for the *SRID*. In this case we leave it to its default value of *4326* which means that our geometry values will have geographic latitude and longitude coordinate system. We finally call the GeometryDDL DDL Extension that enables creation and deletion of geometry columns just after and before table create and drop statements respectively. The GeometryColumn, LineString and GeometryDDL must be imported from the geoalchemy package.

1.5.6 Creating Tables in the Database

The database tables can now be created using the setup-app paster command

```
$ (tg2env) paster setup-app development.ini
```

In case we need sample data to be inserted during application startup, we must add them into the setup script, i.e. tgfeature/websetup.py prior ro running the setup command. Let us add some sample data.

```
wkt = "LINESTRING(-80.3 38.2, -81.03 38.04, -81.2 37.89)"
road1 = model.Road(name="Peter St", width=6, geom=WKTSpatialElement(wkt))
wkt = "LINESTRING(-79.8 38.5, -80.03 38.2, -80.2 37.89)"
road2 = model.Road(name="George Ave", width=8, geom=WKTSpatialElement(wkt))
model.DBSesion.add_all([road1, road2])
```

1.5.7 FeatureServer Config

Now we need to configure our app by declaring certain parameters under the [app:main] section of the ini file. In this case we use development.ini as we are in development mode right now.

```
geo.roads.model=tgfeature.model
geo.roads.cls=Road
geo.roads.table=roads
geo.roads.fid=id
geo.roads.geometry=geom
```

The config parameters use a geo.<layer>.param=value format. This allows additional layers to be defined within the same app as follows:

```
geo.lakes.model=tgfeature.model
geo.lakes.cls=Lake
geo.lakes.table=lakes
geo.lakes.fid=id
geo.lakes.geometry=geom
```

In this tutorial, however, we will use only the roads layer.

1.5.8 Using the FeatureServerController

We can now import and mount the FeatureServer Controller inside our root controller.

```
from tgfeature.model import DBSession
from tgext.geo.featureserver import FeatureServerController

class RootController(BaseController):
    ....
    roads = FeatureServerController("roads", DBSession)
```

We pass two parameters here. The first one being the layer name. This must be the same as layer name used in development.ini. The second parameter is the sqlalchemy session. In case we were using the lakes layer too, as shown in the sample config, we would create two controller instances as:

```
class RootController(BaseController):
    ....
    roads = FeatureServerController("roads", DBSession)
    lakes = FeatureServerController("lakes", DBSession)
```

1.5.9 Testing the Server using curl

We are now ready to start and test out new geo-enabled TG2 app. Start the server in development mode by running:

```
$(tg2env) paster serve --reload development.ini
```

Note the *–reload* option. This tells the server to reload the app whenever there is change in any of the package files that are in its dependency chain. Now we will open up a new command window and test the server using the *curl* utility.

```
# Request the features in GeoJSON format (default)
$ curl http://localhost:8080/roads/all.json
or simply
$ curl http://localhost:8080/roads
{"crs": null, "type": "FeatureCollection", .... long GeoJSON output

# Request the features in GML format
$ curl http://localhost:8080/8080/roads/all.gml
<wfs:FeatureCollection
    xmlns:fs="http://example.com/featureserver
    ....   long GML output

# Request the features in KML format
$ curl http://localhost:8080/roads/all.kml
<?xml version="1.0" encoding="UTF-8"?>
    <kml xmlns="http://earth.google.com/kml/2.0"
    ....  long KML output
```

Now lets create a new feature using curl. Store the following json data in a new file postdata.json:

```
{"features": [{
    "geometry": {
        "type": "LineString",
        "coordinates": [[-88.913933292993605, 42.508280299363101],
                        [-88.8203027197452, 42.598566923566899],
                        [-88.738375968152894, 42.723965012738901],
                        [-88.611305904458604, 42.968073292993601],
                        [-88.365525649681501, 43.140286668789798]
        ]
    },
    "type": "Feature",
    "id": 10,
    "properties": {"name": "Broad Ave", "width": 10}
}]}
```

Create a POST request using this data and send it to the server.

```
$(tg2env) curl -d @postdata.json http://localhost:8080/roads/create.json
```

This creates a new feature and returns back the features in json format. To modify the feature edit the postdata.json file and change the properties. Lets change the name property from *Broad Ave* to *Narrow St* and the width property from *10* to *4*. The modify url should include the feature id as shows below:

```
$(tg2env)  curl -d @postdata.json http://localhost:8080/roads/3.json
```

For deleting the feature simly send a DELETE request with the feature id in the url:

```
$(tg2env) curl -X DELETE http://localhost:8080/roads/3.json
```

1.5.10 An OpenLayers Application Using FeatureServer

The server is now ready to be accessed by client applications for storing, manipulating and deleting featues. OpenLayers is an open source javascript web mapping application. It is quite matured and is under active development. To develop an OpenLayers web application using featureserver the developer is strongly recommended to have a look at the demo application available with the featureserver source code. Copy the demo app (index.html in side featureserver source code directory) to the public folder under the different name:

```
$(tg2env) cp /path/to/featureserversource/index.html tgfeature/public/demo.html
$(tg2env) cp /path/to/featureserversource/json.html tgfeature/public/
$(tg2env) cp /path/to/featureserversource/kml.html tgfeature/public/
```

Now modify these files to change the following:

```
* change all references to featureserver.cgi to '' (null)
* change all references to scribble to 'roads' (layer)
```

Point your browser to http://localhost:8080/demo.html. You should now be able to view, create and modify features using featureserver running inside your TG2 app.

1.6 Testing GeoAlchemy

1.6.1 Requirements

In order to test GeoAlchemy you must have nose tools installed. Also you should have installed SQLAlchemy, GeoAlchemy and all the DB-APIs. Refer the installation docs for details. In case you are interested in testing against only one DB-API (say Postgres), you need not have the other APIs installed. However, you will be able to run those specific tests only.

If you intend to run the tests for Spatialite, you must also have Spatialite extension installed. Also make you have enabled extension loading in your pysqlite build. Refer the Spatialite specific notes for details.

1.6.2 Preparation

You must prepare the databases to run the tests on. Spatial feature is enabled by default in MySQL. You must setup a spatial database in PostgreSQL using PostGIS. Then modify the test_mysql.py and test_postgis.py files in geoalchemy/tests to use the correct dburi (host, username and password) combination. No database preparation is required for Spatialite, as we would use an in-memory database.

1.6.3 Running the Tests

The tests can be run using nosetests as:

```
$ nosetests
```

For running the tests against a single database at a time:

```
$ nosetests geoalchemy/tests/test_postgis.py
$ nosetests geoalchemy/tests/test_mysql.py
$ nosetests geoalchemy/tests/test_spatialite.py
```

1.7 GeoAlchemy Reference

1.7.1 Core

geoalchemy.base

geoalchemy.geometry

geoalchemy.functions

geoalchemy.dialect

geoalchemy.utils

1.7.2 Dialects

geoalchemy.postgis

geoalchemy.mysql

geoalchemy.spatialite

geoalchemy.oracle

geoalchemy.mssql

CHAPTER 2

Indices and tables

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