Papyrus Documentation

Release 2.2

Éric Lemoine

January 23, 2017

Contents

Papyrus provides geospatial extensions for the Pyramid web framework.

Papyrus includes an implementation of the MapFish Protocol. The MapFish Protocol defines a HTTP interface for creating, reading, updating, and deleting (CRUD) geographic objects (a.k.a. features).

Papyrus includes lower-level objects, like the GeoJSON renderer, that may come in handy for your apps even if you don't need or want MapFish views.

Papyrus GeoJSON encoder converts decimal types to float numbers, which may incur an infinitesimal loss of precision.

CHAPTER 1

Installing

Papyrus can be installed with `easy_install` or `pip`:

```
$ pip install papyrus
```

(Installing Papyrus in an isolated `virtualenv` is recommended.)

Most of the time you'll want to make Papyrus a dependency of your Pyramid application. For that add `papyrus` to the `install_requires` list defined in the Pyramid application `setup.py`. Example:

```
install_requires = [
    'pyramid',
    'pyramid_tm',
    'SQLAlchemy',
    'WebError',
    'papyrus'
    ]
```

CHAPTER 2

Documentation

2.1 Creating MapFish Views

Papyrus provides an implementation of the MapFish Protocol. This implementation relies on GeoAlchemy.

This section provides an example describing how to build a MapFish web service in a Pyramid application. (A MapFish web service is a web service that conforms to the MapFish Protocol.)

We assume here that we want to create a MapFish web service named `spots` that relies on a database table of the same name.

2.1.1 Setting up the db model

First of all we need an SQLAlchemy/GeoAlchemy mapping for that table. To comply with Papyrus' MapFish Protocol implementation the mapped class must implement the Python Geo Interface (typically through the `__geo_interface__` property), and must define `__init__` and `__update__` methods.

Implementing the Python Geo Interface is required for Papyrus to be able to serialize `Spot` objects into GeoJSON. The `__init__` and `__update__` methods are required for inserting and updating objects, respectively. Both the `__init__` and `__update__` methods receive a GeoJSON feature (`geojson.Feature`) as their first arguments.

GeoInterface

Papyrus provides a class to help create SQLAlchemy/GeoAlchemy mapped classes that implement the Python Geo Interface, and define `__init__` and `__update__` as expected by the MapFish protocol. This class is `papyrus.geo_interface.GeoInterface`.

The `GeoInterface` class can be used as the super class of the user-defined class. For example:

```
from sqlalchemy.ext.declarative import declarative_base
from papyrus.geo_interface import GeoInterface

Base = declarative_base()

class Spot(GeoInterface, Base):
    __tablename__ = 'spots'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    geom = GeometryColumn('the_geom', Point(srid=4326))

# For SQLAlchemy/GeoAlchemy to be able to create the geometry
```

```
# column when Spot.__table__.create or metadata.create_all is
# called.
GeometryDDL(Spot.__table__)
```

Or it can be used as the base class of classes generated by SQLAlchemy's declarative layer. For example:

```
from sqlalchemy.ext.declarative import declarative_base
from papyrus.geo_interface import GeoInterface

# constructor=None is required for declarative_base to not
# provide its own __init__ constructor
Base = declarative_base(cls=GeoInterface, constructor=None)

class Spot(Base):
    __tablename__ = 'spots'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    geom = GeometryColumn('the_geom', Point(srid=4326))

# For SQLAlchemy/GeoAlchemy to be able to create the geometry
# column when Spot.__table__.create or metadata.create_all is
# called.
GeometryDDL(Spot.__table__)
```

`GeoInterface` represents a convenience method. Implementing one's own `__geo_interface__`, `__init__`, and `__update__` definitions may be a better choice than relying on `GeoInterface`.

Note: When using `GeoInterface` understanding its code can be useful. It can also be a source of inspiration for those who don't use it.

One can change the behavior of `GeoInterface` by overloading its `__init__`, `__update__`, and `__read__` functions. The latter is called by the `__geo_interface__` property, and is therefore the one to overload to change the behavior of `__geo_interface__`.

By default `__read__` reads from column properties only. Likewise, `__update__` writes to column properties only. Other property types are ignored. To make `__read__` and `__update__` consider other properties the `__add_properties__` class-level property can be used. This property should reference a collection of property names. For example:

```
from papyrus.geo_interface import GeoInterface

class Type(Base):
    __tablename__ = 'type'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)

class Spot(GeoInterface, Base):
    __tablename__ = 'spots'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    geom = GeometryColumn('the_geom', Point(srid=4326))
    type_id = Column(Integer, ForeignKey('type.id'))
    type_ = relationship(Type)

    type = association_proxy('type_', 'name')
    __add_properties__ = ('type',)
```

With the above code features returned by the `__geo_interface__` will include `type` properties. And `__update__` will set `type` in `Spot` object being updated.

Without GeoInterface

Without using `GeoInterface` our `Spot` class could look like this:

```
class Spot(Base):
    __tablename__ = 'spots'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    geom = GeometryColumn('the_geom', Point(srid=4326))

    def __init__(self, feature):
        self.id = feature.id
        self.__update__(feature)

    def __update__(self, feature):
        geometry = feature.geometry
        if geometry is not None and \
           not isinstance(geometry, geojson.geometry.Default):
            shape = asShape(geometry)
            self.geom = WKBSpatialElement(buffer(shape.wkb), srid=4326)
            self._shape = shape
        self.name = feature.properties.get('name', None)

    @property
    def __geo_interface__(self):
        id = self.id
        if hasattr(self, '_shape') and self._shape is not None:
            geometry = self_shape
        else:
            geometry = loads(str(self.geom.geom_wkb))
        properties = dict(name=self.name)
        return geojson.Feature(id=id, geometry=geometry, properties=properties)

# For SQLAlchemy/GeoAlchemy to be able to create the geometry
# column when Spot.__table__.create or metadata.create_all is
# called.
GeometryDDL(Spot.__table__)
```

Notes:

- the `pyramid_routesalchemy` template, provided by Pyramid, places SQLAlchemy models in a `models.py` file located at the root of the application's main module (`myapp.models`).
- the `akhet` template, provided by the Akhet package, places SQLAlchemy models in the `__init__.py` file of the `models` module.

2.1.2 Setting up the web service

Now that database model is defined we can now create the core of our MapFish web service.

The web service can be defined through *view callables*, or through an *handler* class. View callables are a concept of Pyramid itself. Handler classes are a concept of the `pyramid_handlers` package, which is an official Pyramid add-on.

With view callables

Using view functions here's how our web service implementation would look like:

```
from myproject.models import Session, Spot
from papyrus.protocol import Protocol

# 'geom' is the name of the mapped class' geometry property
proto = Protocol(Session, Spot, 'geom')

@view_config(route_name='spots_read_many', renderer='geojson')
def read_many(request):
    return proto.read(request)

@view_config(route_name='spots_read_one', renderer='geojson')
def read_one(request):
    id = request.matchdict.get('id', None)
    return proto.read(request, id=id)

@view_config(route_name='spots_count', renderer='string')
def count(request):
    return proto.count(request)

@view_config(route_name='spots_create', renderer='geojson')
def create(request):
    return proto.create(request)

@view_config(route_name='spots_update', renderer='geojson')
def update(request):
    id = request.matchdict['id']
    return proto.update(request, id)

@view_config(route_name='spots_delete')
def delete(request):
    id = request.matchdict['id']
    return proto.delete(request, id)

@view_config(route_name='spots_md', renderer='xsd')
def md(request):
    return Spot.__table__
```

View functions are typically defined in a file named `views.py`. The first six views define the MapFish web service. The seventh view (`md`) provides a metadata view of the `Spot` model/table.

We now need to provide *routes* to these actions. This is done by calling `add_papyrus_routes()` on the `Configurator` (in `__init__.py`):

```
import papyrus
from papyrus.renderers import GeoJSON, XSD
config.include(papyrus.includeme)
config.add_renderer('geojson', GeoJSON())
config.add_renderer('xsd', XSD())
config.add_papyrus_routes('spots', '/spots')
config.add_route('spots_md', '/spots/md.xsd', request_method='GET')
config.scan()
```

`add_papyrus_routes` is a convenience method, here's what it basically does:

```
config.add_route('spots_read_many', '/spots', request_method='GET')
config.add_route('spots_read_one', '/spots/{id}', request_method='GET')
config.add_route('spots_count', '/spots/count', request_method='GET')
config.add_route('spots_create', '/spots', request_method='POST')
config.add_route('spots_update', '/spots/{id}', request_method='PUT')
config.add_route('spots_delete', '/spots/{id}', request_method='DELETE')
```

With a handler

Using a handler here's what our web service implementation would look like:

```
from pyramid_handlers import action

from myproject.models import Session, Spot
from papyrus.protocol import Protocol

# create the protocol object. 'geom' is the name
# of the geometry attribute in the Spot model class
proto = Protocol(Session, Spot, 'geom')

class SpotHandler(object):
    def __init__(self, request):
        self.request = request

    @action(renderer='geojson')
    def read_many(self):
        return proto.read(self.request)

    @action(renderer='geojson')
    def read_one(self):
        id = self.request.matchdict.get('id', None)
        return proto.read(self.request, id=id)

    @action(renderer='string')
    def count(self):
        return proto.count(self.request)

    @action(renderer='geojson')
    def create(self):
        return proto.create(self.request)

    @action(renderer='geojson')
    def update(self):
        id = self.request.matchdict['id']
        return proto.update(self.request, id)

    @action()
    def delete(self):
        id = self.request.matchdict['id']
        return proto.delete(self.request, id)

    @action(renderer='xsd')
    def md(self):
        return Spot.__table__
```

The six actions of the `SpotHandler` class entirely define our MapFish web service.

We now need to provide *routes* to these actions. This is done by calling `add_papyrus_handler()` on the

`Configurator`:

```
import papyrus
from papyrus.renderers import GeoJSON
config.include(papyrus)
config.add_renderer('geojson', GeoJSON())
config.add_papyrus_handler('spots', '/spots',
                           'myproject.handlers:SpotHandler')
config.add_handler('spots_md', '/spots/md.xsd',
                   'myproject.handlers:SpotHandler', action='md',
                   request_method='GET')
```

Likewise `add_papyrus_routes` `add_papyrus_handler` is a convenience method. Here's what it basically does:

```
config.add_handler('spots_read_many', '/spots',
                   'myproject.handlers:SpotHandler',
                   action='read_many', request_method='GET')
config.add_handler('spots_read_one', '/spots/{id}',
                   'myproject.handlers:SpotHandler',
                   action='read_one', request_method='GET')
config.add_handler('spots_count', '/spots/count',
                   'myproject.handlers:SpotHandler',
                   action='count', request_method='GET')
config.add_handler('spots_create', '/spots',
                   'myproject.handlers:SpotHandler',
                   action='create', request_method='POST')
config.add_handler('spots_update', '/spots/{id}',
                   'myproject.handlers:SpotHandler',
                   action='update', request_method='PUT')
config.add_handler('spots_delete', '/spots/{id}',
                   'myproject.handlers:SpotHandler',
                   action='delete', request_method='DELETE')
```

Note: when using handlers the `pyramid_handlers` package must be set as an application's dependency.

2.1.3 API Reference

2.2 GeoJSON Renderer

Papyrus provides a GeoJSON renderer, based on Sean Gillies' geojson package.

To use it the GeoJSON renderer factory must be added to the application configuration.

For that you can either pass the factory to the `Configurator` constructor:

```
from pyramid.mako_templating import renderer_factory as mako_renderer_factory
from papyrus.renderers import GeoJSON
config = Configurator(
    renderers=(('.mako', mako_renderer_factory),
               ('geojson', GeoJSON())))
    )
```

Or you can apply the `add_renderer` method to the `Configurator` instance:

```
from papyrus.renderers import GeoJSON
config.add_renderer('geojson', GeoJSON())
```

Make sure that `add_renderer` is called before any `add_view` call that names `geojson` as an argument.

To use the GeoJSON renderer in a view set `renderer` to `geojson` in the view config. Here is a simple example:

```
@view_config(renderer='geojson')
def hello_world(request):
    return {
        'type': 'Feature',
        'id': 1,
        'geometry': {'type': 'Point', 'coordinates': [53, -4]},
        'properties': {'title': 'Dict 1'},
        }
```

Views configured with the `geojson` renderer must return objects that implement the Python Geo Interface.

Here's another example where the returned object is an SQLAlchemy (or GeoAlchemy) mapped object:

```
@view_config(renderer='geojson')
def features(request):
    return Session().query(Spot).all()
```

In the above example the `Spot` objects returned by the `query` call must implement the Python Geo Interface.

Notes:

- The GeoJSON renderer supports JSONP. The renderer indeed checks if there's a `callback` parameter in the query string, and if there's one it wraps the response in a JavaScript call and sets the response content type to `text/javascript`.
- The application developer can also specify the name of the JSONP callback parameter, using this:

```
    from papyrus.renderers import GeoJSON
    config.add_renderer('geojson', GeoJSON(jsonp_param_name='cb'))
```

  With this, if there's a parameter named `cb` in the query string, the renderer will return a JSONP response.

- By default, lists and tuples passed to the renderer will be rendered as FeatureCollection. You can change this using the `collection_type` argument:

```
    from papyrus.renderers import GeoJSON
    config.add_renderer('geojson', GeoJSON(collection_type='GeometryCollection'))
```

2.2.1 API Reference

class `papyrus.renderers.`**`GeoJSON`**(*jsonp_param_name='callback'*, *collection_type=<class 'geojson.feature.FeatureCollection'>*)

GeoJSON renderer.

This class is actually a renderer factory helper, implemented in the same way as Pyramid's JSONP renderer.

Configure a GeoJSON renderer using the `add_renderer` method on the Configurator object:

```
    from papyrus.renderers import GeoJSON

    config.add_renderer('geojson', GeoJSON())
```

Once this renderer has been registered as above , you can use `geojson` as the `renderer` parameter to `@view_config` or to the `add_view` method on the Configurator object:

```
    @view_config(renderer='geojson')
    def myview(request):
        return Feature(id=1, geometry=Point(1, 2),
                       properties=dict(foo='bar'))
```

The GeoJSON renderer supports JSONP:

- If there is a parameter in the request's HTTP query string that matches the `jsonp_param_name` of the registered JSONP renderer (by default, `callback`), the renderer will return a JSONP response.
- If there is no callback parameter in the request's query string, the renderer will return a 'plain' JSON response.

By default the renderer treats lists and tuples as feature collections. If you want lists and tuples to be treated as geometry collections, set `collection_type` to `'GeometryCollection'`:

```
    config.add_renderer(
        'geojson', GeoJSON(collection_type='GeometryCollection')
```

2.3 XSD Renderer

Papyrus provides an XSD renderer, capable of serializing SQLAlchemy mapped classes (including GeoAlchemy geometry columns) into XML Schema Documents.

XSDs generated by the XSD Renderer can, for example, be parsed using OpenLayers's DescribeFeatureType format.

To use the XSD renderer the XSD renderer factory should be added to the application configuration.

This is done by either passing the factory to the `Configurator` constructor:

```
from pyramid.mako_templating import renderer_factory as mako_renderer_factory
from papyrus.renderers import XSD
config = Configurator(
    renderers=(('.mako', mako_renderer_factory),
               ('xsd', XSD())))
    )
```

Or by applying the `add_renderer` method to the `Configurator` instance:

```
from papyrus.renderers import XSD
config.add_renderer('xsd', XSD())
```

Make sure that `add_renderer` is called before any `add_view` call that names `xsd` as an argument.

To use the XSD renderer in a view set `renderer` to `xsd` in the view config. Here is a simple example:

```
from sqlalchemy import Column, types
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

@view_config(renderer='xsd')
def hello_world(request):
    class C(Base):
        __tablename__ = 'table'
        id = Column(types.Integer, primary_key=True)
    return C
```

Views configured with the `xsd` renderer should return SQLAlchemy mapped classes.

Here's another example:

```
@view_config(renderer='xsd')
def spots_md(request):
    return Spot
```

Where `Spot` is an SQLAlchemy mapped class created using SQLAlchemy's declarative layer.

Notes:

- By default the XSD renderer skips columns which are primary keys. If you wish to include primary keys then pass `include_primary_keys=True` when creating the `XSD` objects, for example:

```
    from papyrus.renderers import XSD
    config.add_renderer('xsd', XSD(include_primary_keys=True))
```

- By default the XSD renderer skips columns which are foreign keys. Use `include_foreign_keys=True` to change that behavior. For example:

```
    from papyrus.renderers import XSD
    config.add_renderer('xsd', XSD(include_foreign_keys=True))
```

- The XSD renderer adds `xsd:element` nodes for the column properties it finds in the class. The XSD renderer will ignore other property types. For example it will ignore relationship properties and association proxies. If you want to add `xsd:element` nodes for other elements in the class then use a `sequence_callback`. For example:

```
    from papyrus.renderers import XSD
    def callback(tb, cls):
        attrs = {}
        attrs['minOccurs'] = str(0)
        attrs['nillable'] = 'true'
        attrs['name'] = 'gender'
        with tag(tb, 'xsd:element', attrs) as tb:
            with tag(tb, 'xsd:simpleType') as tb:
                with tag(tb, 'xsd:restriction', {'base': 'xsd:string'}) as tb:
                    with tag(tb, 'xsd:enumeration', {'value': 'male'}):
                            pass
                    with tag(tb, 'xsd:enumeration', {'value': 'female'}):
                            pass
    config.add_renderer('xsd', XSD(sequence_callback=callback))
```

2.3.1 API Reference

class papyrus.renderers.**XSD**(*include_primary_keys=False*, *include_foreign_keys=False*, *sequence_callback=None*)

XSD renderer.

An XSD renderer generate an XML schema document from an SQLAlchemy Table object.

Configure a XSD renderer using the `add_renderer` method on the Configurator object:

```
    from papyrus.renderers import XSD

    config.add_renderer('xsd', XSD())
```

Once this renderer has been registered as above , you can use `xsd` as the `renderer` parameter to `@view_config` or to the `add_view` method on the Configurator object:

```
from myapp.models import Spot

@view_config(renderer='xsd')
def myview(request):
    return Spot
```

By default, the XSD renderer will skip columns which are primary keys or foreign keys.

If you wish to include primary keys then pass `include_primary_keys=True` when creating the XSD object, for example:

```
from papyrus.renderers import XSD

config.add_renderer('xsd', XSD(include_primary_keys=True))
```

If you wish to include foreign keys then pass `include_foreign_keys=True` when creating the XSD object, for example:

```
from papyrus.renderers import XSD

config.add_renderer('xsd', XSD(include_foreign_keys=True))
```

The XSD renderer adds `xsd:element` nodes for the column properties it finds in the class. The XSD renderer will ignore other property types. For example it will ignore relationship properties and association proxies. If you want to add `xsd:element` nodes for other elements in the class then use a `sequence_callback`. For example:

The callback receives an `xml.etree.ElementTree.TreeBuilder` object and the mapped class being serialized.

CHAPTER 3

Contributing

Papyrus is on GitHub: http://github.com/elemoine/papyrus. Fork away. Pull requests are welcome!

CHAPTER 4

Running the tests

Papyrus includes unit tests. Most of the time patches should include new tests.

To run the Papyrus tests, in addition to Papyrus and its dependencies the following packages need to be installed: `nose`, `mock`, `psycopg2`, `pyramid_handlers`, `coverage`, and `WebTest`.

For these packages to install correctly, you have to have header files for `PostgreSQL`, `Python`, and `GEOS`. On Debian-based systems install the following system packages: `libpq-dev`, `python-dev`, `libgeos-c1`.

Use `pip` and the `dev_requirements.txt` file to install these packages in the virtual environment:

```
$ pip install -r dev_requirements.txt
```

To run the tests:

```
$ nosetests
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

G

X