geostore-backend-crud

Release 0.3.16

Jan 27, 2020

Contents

Terralego backend app

CHAPTER 1

Installation

1.1 Requirements

1.1.1 DATABASE

Minimum configuration :

- Postgresql 10
- PostGIS 2.4
- PgRouting 2.5

Recommended configuration :

- Postgresql 11
- PostGIS 2.5
- PgRouting 2.6

Your final django project should use django.contrib.gis.backend.postgis as default DATABASE backend

USING docker image :

Prebuilt docker image builded by makinacorpus

https://cloud.docker.com/u/makinacorpus/repository/docker/makinacorpus/pgrouting/general

1.1.2 SYSTEM REQUIREMENTS

For django

libpq-dev gettext

For geodjango

gdal-bin binutils libproj-dev

1.2 With pip

From Pypi:

```
pip install xxxxxxxxxx-xxxxxxxxxxxx
```

From Github:

```
pip install -e https://github.com/Terralego/django-geostore.git@master#egg=geostore
```

1.3 With git

```
git clone https://github.com/Terralego/django-geostore.git
cd django-geostore
python setup.py install
```

CHAPTER 2

Configuration

In your project :

Add geostore to your `INSTALLED_APPS` :

```
# install required apps
INSTALLED_APPS = [
    ...
    'django.contrib.gis',  # assume contrib.gis is installed
    ...
    'rest_framework',
    'rest_framework_gis',
    'geostore',
    ...
]
```

2.1 Settings

warning:: Geostore will change the geojson serializer on app loading.

2.1.1 INTERNAL_GEOMETRY_SRID

Default: 4326

It's the installation SRID, it must be set before the first migration and never change after installation, else you must create your own migrations to change your database SRID.

2.1.2 HOSTNAME

Default: empty

Used to feed `TERRA_TILES_HOSTNAMES` setting

2.1.3 TERRA_TILES_HOSTNAMES

Default: [HOSTNAME,]

It contains the list of base URLs where are served the vector tiles. Since web browsers limit the number of connections to one domain name, a workaround is to use many domains to serve vector tiles, so browser will create more tcp connections, and the tiles loading will be faster.

2.1.4 MAX_TILE_ZOOM

Default: 15

It represent the max authorized zoom, if a tile with a zoom above this setting is requested, geostore will refuse to serve it.

2.1.5 MIN_TILE_ZOOM

Default: 10

Like for `MAX_TILE_ZOOM` setting, if a tile of a lesser zoom than this setting is requested, backend will refuse to serve it.

2.2 URLs

Add to you urls.py file this pattern:

```
urlpatterns = [
    ...
    path('', include('geostore.urls', namespace='geostore')),
    ...
]
```

You can customize default url and namespace by including geostore.views directly

2.2.1 Admin

you can disable and / or customize admin

CHAPTER 3

QUICK START

3.1 Manage layers

The simplest way to create a geographic data layer :

```
from geostore import GeometryTypes
from geostore.models import Layer

layer = Layer.objects.create(name='Mushroom spot',
                             geom_type=GeometryTypes.Point)
```

3.1.1 Geometry type validation

Layer support these geometry types :

Supported types

geostore.GeometryTypes

GeometryCollection = 7 LineString = 1 MultiLineString = 5 MultiPoint = 4 MultiPolygon = 6 Point = 0 Polygon = 3

Define a geometry type to layer to force feature geometry validation.

Without validation

```
from geostore.models import Layer, Feature
from geostore import GeometryTypes
from django.contrib.geos.geometries import GEOSGeometry

layer = Layer.objects.create(name='Mushroom spot 2')
```

(continues on next page)

(continued from previous page)

```
feature = Feature(layer=layer,
                  geom=GEOSGeometry("POINT(0 0)")
feature.clean()  # ok
# then, you can save
feature.save()
feature = Feature(layer=layer,
                  geom=GEOSGeometry("LINESTRING((0 0), (1 1))")

feature.clean()  # ok too
feature.save()
```

With validation

```
from geostore.models import Layer, Feature
from geostore import GeometryTypes
from django.contrib.geos.geometries import GEOSGeometry

layer = Layer.objects.create(name='Mushroom spot 3',
                             geom_type=GeometryTypes.Point)
feature = Feature(layer=layer,
                  geom=GEOSGeometry("POINT(0 0)")

feature.clean()  # ok
feature.save()
feature = Feature(layer=layer,
                  geom=GEOSGeometry("LINESTRING((0 0), (1 1))")
feature.clean()  # validation error !
```

3.1.2 JSON schema definition / validation

You can use json schema definition to describe your data content, and improve feature properties validation.

https://json-schema.org/ https://rjsf-team.github.io/react-jsonschema-form/

```
from geostore.models import Layer, Feature
from geostore import GeometryTypes
from django.contrib.geos.geometries import GEOSGeometry

layer = Layer.objects.create(name='Mushroom spot 4',
                             geom_type=GeometryTypes.Point,
                             schema={
                               "required": ["name", "age"],
                               "properties": {
                                 "name": {
                                   "type": "string",
                                   "title": "Name"
                                 },
                                 "age": {
                                   "type": "integer",
                                   "title": "Age"
                                 }
                               }
                             })
```

(continues on next page)

(continued from previous page)

```
feature = Feature(layer=layer,
                  geom=GEOSGeometry("POINT(0 0)")
feature.clean()  # Validation Error ! name and age are required

feature = Feature(layer=layer,
                  geom=GEOSGeometry("POINT(0 0)",
                  properties={
                      "name": "Arthur",
                  })
feature.clean()  # Validation Error ! age is required

feature = Feature(layer=layer,
                  geom=GEOSGeometry("POINT(0 0)",
                  properties={
                    "name": "Arthur",
                    "age": "ten",
                  })
feature.clean()  # Validation Error ! age should be integer

feature = Feature(layer=layer,
                  geom=GEOSGeometry("POINT(0 0)",
                  properties={
                    "name": "Arthur",
                    "age": 10
                  })
feature.clean()  # ok !
feature.save()
```

3.1.3 Vector tiles

geostore provide endpoint to generate and cache MVT based on your data.

You can access these tiles through Layer and LayerGroup features.

On layers

On group of layers

3.1.4 Relations

- You can define relations between layers (and linked features)

Warning: Compute relations need celery project and worker configured in your project. Run at least 1 worker. You need to fix settings explicitly to enable asynchronous tasks. GEOSTORE_RELATION_CELERY_ASYNC = True

Manual relation

No automatic links between features. You need to create yourself FeatureRelation between Features.

Automatic relations

If any celery project worker is available, and GEOSTORE_RELATION_CELERY_ASYNC settings set to True, each layer relation creation or feature edition will launch async task to update relation between linked features.

Intersects

By selecting intersects, each feature in origin layer intersecting geometry features in destination layer, will be linked to them.

Distance

By selecting distance, each feature in origin layer with distance max geometry features in destination layer, will be linked to them.

> **Warning:** You need to define distance in settings: {"distance": 10000} # for 10km

3.1.5 Data import

ShapeFile

GeoJSON

3.1.6 Data export

3.1.7 API endpoints

CHAPTER 4

Vector Tiles

Vector tiles are served following the Mapbox Vector Tiles standard, and using the ST_AsMVT Postgis method.

Most of the work is done in the `geostore.tiles.helpers` module.

4.1 Settings

Vector tiles can be served in many ways, and it generation can be configured. This allow you to manage which data is returned, but also some tunning settings.

The `Layer` models has a `settings` attribute which is a `JSONField`.

Here we describe available json keys and its content, then we provide your an example.

4.2 metadata

Contains all data metadatas that can be added to tile content, it allows you to store it in a convenient way.

4.2.1 attribution

Default: None

Attribution of the layer's data. Must be a dict like this:

```
{'name': 'OSM contributors', href='http://openstreetmap.org'}
```

4.2.2 licence

Default: None

String containing the layer's data licence. i.e.: ODbL, CC-BY, Public Domain, ...

4.2.3 description

Default: None

Text that describe the data.

4.3 tiles

4.3.1 minzoom

Default: 0

Min zoom when the layer is served in tiles. Must be higher or equal to `MIN_ZOOM` setting.

4.3.2 maxzoom

Default: 22

Max zoom when the layer is served in tiles. Must be lower or equal to `MAX_ZOOM` setting.

4.3.3 pixel_buffer

Default: 4

Buffer size around a tile, to match more features and clip features at a larger size than the tile.

Mostly, the default value is enough, but sometimes, depending of the display style (width border of lines or polygons), you will need to increase this value.

4.3.4 features_filter

Default: None

Filter the features queryset, by this value. Could be used to not return all features of your layers on the tiles.

The complete object is passed to a `filter(properties__contains)` method

4.3.5 properties_filter

Default: None

List of allowed properties in tiles. This must be a list of properties that will be the only one present in vector tiles. If set to `None`, all properties will be returned, else only properties present in the list will be returned.

4.3.6 features_limit

Default: 10000

Maximal number of features in a tile. Used to prevent tiles to have too much data, since MVT standard tells a tile must not be high than 500ko.

4.3.7 Example

```
{
      'metadata': {
          'attribution': {'name': 'OSM contributors', href='http://openstreetmap.org'}
          'licence': 'ODbL,
          'description': "Good Licence",
      },
      # Tilesets attributes
      'tiles': {
          'minzoom': 10,
          'maxzoom': 14,
          'pixel_buffer': 4,
          'features_filter': 500,
          'properties_filter': ['my_property', ],
          'features_limit': 10000,
      }
}
```

CHAPTER 5

Routing

Django-Geostore integrate a way to use your LineString layer as a routing one. It uses pgRouting as backend.

5.1 Prerequisites

- pgRouting>=2.5

5.2 Settings

pgRouting needs to update a table that contains all linestring connections, to do you need to execute the management command we made:

```
./manage.py update_topology -pk <layer_pk>
```

You must provide the pk of the layer you want to use.

5.3 Usage

The layer viewset has a route that provide a endpoint to get a routing result between two or more points.

`^layer/<pk>/route`

5.3.1 Arguments

First attribute needed, and mandatory, is `geom`, it must contrains a LineString from start to endpoint, passing through all the waypoints. Geostore will create a path passing on the intersection the closest of those point, in the order you provided it.

It can also be provided a `callbackid`, that is used to identify the request. It can be usefull in async environment. The `callbackid` is provided «as is» in the response.

Query content can provided in a POST or a GET request.

An example of response:

```
{
    'request': {
        'callbackid': 'my_callback',
        'geom': {
            "type": 'LineString',
            "coordinates": [
            [
                10.8984375,
                52.1874047455997
            ],
            [
                1.58203125,
                46.042735653846506
            ]
            ]
        }
    },
    'geom': {
        'type': 'LineString',
        'coordinates': [
        [
            1.6259765625,
            45.767522962149876
        ],
        [
            5.2294921875,
            46.558860303117164
        ],
        [
            10.986328125,
            52.10650519075632
        ]
        ]
    }
}
```

CHAPTER 6

Vector Tiles Group Access

Django-Geostore has a mecanism to authorize only some django's user Groups to access layer's on vector tiles.

This can be used to manage layer access through vector tiles.

Here we're going to describe how it works.

6.1 Where to add a group

Each layer has a ManyToMany relationship to django's Group model, that authorized only users present is thoses groups to have access to thoses layers through vector tiles.

You can add a group, with the normal django's ORM API:

```
from django.contrib.auth.models import Group
from geostore.models import Layer
g = Group.objects.first()
l = Layer.objects.first()

l.authorized_groups.add(g)
```

6.2 Then ?

Then, you can generate the autheticated URL by using a QueryString like above, where user_groups are a list of user_groups names, and layergroup is the group of the layer:

You'll have available an authenticated url, this will filter layers in tiles that are accessible to the authenticated user groups.

All authenticated informations will be provided by the authenticated tilejson, that will provide to frontend all authenticated urls.

Usually, mapbox needs only the tilejson, geostore will do all the remaining work.