django-vectortiles

Release 1.0.0-beta3

Jean-Etienne Castagnede

Feb 16, 2024

CONTENTS

CHAPTER

ONE

INSTALLATION

1.1 Requirements

- You need to install geodjango required libraries (See here)

1.2 PostGIS 2.4+ backend usage

- You need a PostgreSQL database with PostGIS 2.4+ extension enabled. (See https://docs.djangoproject.com/en/stable/ref/contrib/gis/install/postgis/)
- You need to enable and use **django.contrib.gis.db.backends.postgis** database backend

```
pip install psycopg2
pip install django-vectortiles
```

1.3 Other database backend usages

```
pip install django-vectortiles[python]
```

This will include sub-dependencies to generate vector tiles from mapbox_vector_tiles python library.

CHAPTER

TWO

USAGE

A vector tile is composed by vector layers which represent different kind of data. Each layer is composed by features.

To start using django-vectortiles, you need GeoDjango models with geometries.

Then, you need to describe how your data will be embed in tiles.

Start by creating vector layers for your data…

```
# in your app models.py
from django.contrib.gis.db import models


class City(models.Model):
    name = models.CharField(max_length=250)
    city_code = models.CharField(max_length=10, unique=True)
    population = models.IntegerField(default=0)
    geom = models.MultiPolygonField(srid=4326)


# in a vector_layers.py file (for example)
from vectortiles import VectorLayer
from your_app.models import City


class CityVectorLayer(VectorLayer):
    model = City  # your model, as django conventions you can use queryset or get_
↪queryset method instead)
    id = "cities"  # layer id in you vector layer. each class attribute can be defined␣
↪by get_{attribute} method
    tile_fields = ('city_code', "name") # fields to include in tile
    min_zoom = 10 # minimum zoom level to include layer. Take care of this, as it could␣
↪be a performance issue. Try to not embed data that will no be shown in your style␣
↪definition.
    # all attributes available in vector layer definition can be defined
```

Well. your vector layer is ready. next step is to create a tile class and a view to serve it.

2.1 Simple layer tile view

```
# in your view file
from your_app.vector_layers import CityVectorLayer
from your_app.views import MVTView


class CityTileView(MVTView):
    layer_classes = [CityVectorLayer, CityCentroidVectorLayer]  # you can use get_layer_
↪classes method, or directly get_layers instead

# in your urls file
from django.urls import path
from yourapp import views

urlpatterns = [
    ...
    views.CityTileView.get_url(),  # serve tiles at default /tiles/<int:z>/<int:x>/
↪<int:y>
    ...
]
```

2.2 Multiple layer tile view

As vector tile layer permit it, you can embed multiple layers in your tile.

Let's create a second layer.

```
# in your app models.py
class State(models.Model):
    name = models.CharField(max_length=250)
    state_code = models.CharField(max_length=10, unique=True)
    geom = models.MultiPolygonField(srid=4326)

# in vector_layers.py file
class StateVectorLayer(VectorLayer):
    model = State
    id = "states"
    tile_fields = ('state_code', "name")
    min_zoom = 3

# in your view file
class CityAndStateTileView(MVTView):
    layer_classes = [CityVectorLayer, StateVectorLayer]

# in your urls file
urlpatterns = [
    ...
    views.CityAndStateTileView.get_url(),  # serve tiles at default /tiles/<int:z>/
↪<int:x>/<int:y>
```

(continues on next page)

(continued from previous page)

```
    ...
]
```

2.3 Using TileJSON

It's a good practice to use tilejson to tell to your map library how to gt your tiles and their defintion. django-vectortiles permit that.

TileJSON and tile views share some data, as vactor layers definition. So we need to factorize some things.

```
# in your view file

class CityAndStateBaseLayer:
    # mixin for your two views
    layer_classes = [CityVectorLayer, StateVectorLayer]
    prefix_url = 'city-and-states'  # as tilejson need to known tiles URL, we need to␣
↪define a url prefix for our tiles

class CityAndStateTileView(CityAndStateBaseLayer, MVTView):
    pass


class CityAndStateTileJSON(CityAndStateBaseLayer, TileJSONView):
    pass


# in your urls file
urlpatterns = [
    ...
    views.CityAndStateTileView.get_url(),  # serve tiles at /city-and-states/<int:z>/
↪<int:x>/<int:y>
    views.CityAndStateTileJSON.get_url(),  # serve tilejson at /city-and-states/tiles.
↪json
    ...
]
```

Now you can use your tiles with a map library like MapLibre or Mapbox GL JS, directly wit hthe tileJSON provided.

Warning: By default, it's your browser URL that will be used to generate tile url in tilejson. Take care about django and SSL configuration (django settings, web server headers) if you want to generate an URL with https://

Note: If your application is hosted on server with many workers, and you want to optimized tile loading, you can add several urls in your tilejson file.

```
# add in your settings.py file
ALLOWED_HOSTS = [
    "a.tiles.xxxx",
    "b.tiles.xxxx",
```

(continues on next page)

(continued from previous page)

```
    "c.tiles.xxxx",
    ...
]

VECTOR_TILES_URLS = [
    "https://a.tiles.xxxx",
    "https://b.tiles.xxxx",
    "https://c.tiles.xxxx",
]
```

With these settings, each tilejson file will contain several urls, and your map library will be able to parallel load tiles at time.

2.4 More complex multiple layer tile view

You can customize geometry data embed in your tiles.

```
class CityCentroidVectorLayer(VectorLayer):
    queryset = City.objects.annotate(
        centroid=Centroid("geom"), # compute the city centroïd
        area=Area("geom"), # compute the city area
    )
    geom_field = "centroid"  # use the centroid field as geometry feature
    id = "city_centroids"
    tile_fields = ('name', 'city_code', 'area', 'population')  # add area and population␣
↪properties in each tile feature
    min_zoom = 7  # let's show city name at zoom 7
```

2.5 Complex related model tile view

```
# in your app models.py
from django.contrib.gis.db import models


class Layer(models.Model):
    name = models.CharField(max_length=250)


class Feature(models.Model):
    geom = models.GeometryField(srid=4326)
    name = models.CharField(max_length=250)
    layer = models.ForeignKey(Layer, on_delete=models.CASCADE, related_name='features')

# in your views.py file
from django.views.generic import DetailView
from your_app.models import Layer
```

(continues on next page)

(continued from previous page)

```
class LayerTileView(MVTView, DetailView):
    model = Layer
    tile_fields = ('name', )

    def get_id(self):
        return self.get_object().name

    def get_queryset(self):
        return self.get_object().features.all()

    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        return BaseVectorTileView.get(self,request=request, z=kwargs.get('z'), x=kwargs.
↪get('x'), y=kwargs.get('y'))


# in your urls file
from django.urls import path
from your_app import views


urlpatterns = [
    ...
    path('layer/<int:pk>/tile/<int:z>/<int:x>/<int:y>', views.LayerTileView.as_view(),␣
↪name="layer-tile"),
    ...
]
```

2.6 Django Rest Framework

```
# in your views.py file
from vectortiles.rest_framework.renderers import MVTRenderer


class FeatureAPIView(BaseVectorTile, APIView):
    queryset = Feature.objects.all()
    id = "features"
    tile_fields = ('name', )
    queryset_limit = 100
    renderer_classes = (MVTRenderer, )

    def get(self, request, *args, **kwargs):
        return Response(self.get_tile(kwargs.get('x'), kwargs.get('y'), kwargs.get('z')))

# in your urls file
urlpatterns = [
    ...
    path('features/tiles/<int:z>/<int:x>/<int:y>', FeatureAPIView.as_view(),
         name="feature-tile-drf"),
```

(continues on next page)

(continued from previous page)

```
    ...
]

# or extending viewset

class FeatureViewSet(BaseVectorTile, viewsets.ModelViewSet):
    queryset = Feature.objects.all()
    id = "features"
    tile_fields = ('name', )
    queryset_limit = 100

    @action(detail=False, methods=['get'], renderer_classes=(MVTRenderer, ),
            url_path='tiles/(?P<z>\d+)/(?P<x>\d+)/(?P<y>\d+)', url_name='tile')
    def tile(self, request, *args, **kwargs):
        return Response(self.get_tile(x=int(kwargs.get('x')), y=int(kwargs.get('y')),␣
↪z=int(kwargs.get('z'))))

# in your urls file
router = SimpleRouter()
router.register(r'features', FeatureViewSet, basename='features')

urlpatterns += router.urls
```

then use http://your-domain/features/tiles/{z}/{x}/{y}.pbf

2.7 MapLibre Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-
↪scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>MapBox / MapLibre example</title>
    <style>
        html, body {
            margin: 0;
            padding: 0;
        }
    </style>
    <link href='https://unpkg.com/maplibre-gl@2.4.0/dist/maplibre-gl.css' rel='stylesheet
↪'/>
</head>
<body>
<div id="map" style="width: 100%; height: 100vh"></div>
<script src='https://unpkg.com/maplibre-gl@2.4.0/dist/maplibre-gl.js'></script>

<script>
```

(continues on next page)

(continued from previous page)

```
    var map = new maplibregl.Map({
        container: 'map',
        hash: true,
        style: 'https://demotiles.maplibre.org/style.json', // stylesheet location
        center: [1.77, 44.498], // starting position [lng, lat]
        zoom: 8 // starting zoom
    });
    var nav = new maplibregl.NavigationControl({visualizePitch: true});
    map.addControl(nav, 'top-right');
    var scale = new maplibregl.ScaleControl({
        maxWidth: 80,
        unit: 'metric'
    });
    map.addControl(scale);
    map.on('load', function () {
        map.addSource('layers', {
            'type': 'vector',
            'url': '{% url "city-tilejson" %}'
        });
        map.addLayer(
            {
                'id': 'background2',
                'type': 'background',
                'paint': {
                    'background-color': '#F8F4F0',
                }

            }
        );
        map.addLayer(
            {
                'id': 'cities',
                'type': 'line',
                'filter': ['==', ['geometry-type'], 'Polygon'],
                'source': 'layers',
                'source-layer': 'cities',
                'layout': {
                    'line-cap': 'round',
                    'line-join': 'round'
                },
                'paint': {
                    'line-opacity': 0.4,
                    'line-color': '#3636a8',
                    'line-width': 0.5,
                    'line-dasharray': [10, 10]
                }

            }
        );

        map.addLayer(
            {
```

(continues on next page)

(continued from previous page)

```
                "id": "city-borders",
                "type": "symbol",
                "source": "layers",
                "source-layer": "cities",
                "minzoom": 13,
                "layout": {
                    "symbol-placement": "line",
                    "symbol-spacing": 350,
                    "text-field": "{nom}",
                    "text-font": ["Noto Sans Italic"],
                    "text-letter-spacing": 0.2,
                    "text-max-width": 5,
                    "text-rotation-alignment": "map",
                    "text-size": 10
                },
                "paint": {
                    "text-color": "#3636a8",
                    "text-halo-color": "rgba(255,255,255,0.7)",
                    "text-halo-width": 1
                }
            }
        );
        map.addLayer(
            {
                "id": "cities_marker",
                "type": "symbol",
                "source": "layers",
                "source-layer": "city-centroids",
                "minzoom": 10,
                "maxzoom": 12,
                "layout": {
                    "symbol-placement": "point",
                    "symbol-spacing": 350,
                    "text-field": "{nom}",
                    "text-font": ["Noto Sans Italic"],
                    "text-letter-spacing": 0.2,
                    "text-max-width": 5,
                    "text-rotation-alignment": "map",
                    "text-size": 14
                },
                "paint": {
                    "text-color": "#3636a8",
                    "text-halo-color": "rgba(255,255,255,0.7)",
                    "text-halo-width": 1.5
                }
            }
        );

        // Create a popup, but don't add it to the map yet.
        var popup = new maplibregl.Popup({
            closeButton: false,
            closeOnClick: false
```

(continues on next page)

(continued from previous page)

```
        });

        map.on('mouseenter', 'cities_marker', function (e) {
            // Change the cursor style as a UI indicator.
            map.getCanvas().style.cursor = 'pointer';
            var coordinates = e.features[0].geometry.coordinates.slice();
            var description = `${e.features[0].properties.name} (${e.features[0].
↪properties.population} hab.)`;

            // Ensure that if the map is zoomed out such that multiple
            // copies of the feature are visible, the popup appears
            // over the copy being pointed to.
            while (Math.abs(e.lngLat.lng - coordinates[0]) > 180) {
                coordinates[0] += e.lngLat.lng > coordinates[0] ? 360 : -360;
            }

            // Populate the popup and set its coordinates
            // based on the feature found.
            popup.setLngLat(coordinates).setHTML(description).addTo(map);
        });

        map.on('mouseleave', 'cities_marker', function () {
            map.getCanvas().style.cursor = '';
            popup.remove();
        });
    }
);
</script>
</body>
</html>
```

2.8 Cache policy

CHAPTER

THREE

DEVELOPMENT

3.1 With docker and docker-compose

Copy `.env.dist` to `.env` and fill `SECRET_KEY` and `POSTGRES_PASSWORD`

```
docker-compose build
# docker-compose up
docker-compose run /code/venv/bin/python ./manage.py test
```

3.2 Local

- Install python and django requirements (python 3.8+, django 3.2+)
- Install geodjango requirements
- Have a postgresql / postgis 2.4+ enabled database
- Use a virtualenv

```
pip install .[dev] -U
```

CHAPTER

FOUR

CHANGELOG

4.1 1.0.0-beta

- Drop python 3.6 and Django 2.2
- Add python 3.11 and Django 4.2

Breaking changes

- Refactor PostGIS and Python (old named MapBox) backends usage. Use setting to set (default postgis)
- No DetailView anymore. As Tile can have many layers, declare VectorLayer on MVTView (one or many).
- Enhancements
  - Add compatibility to use with psycopg v3
- Features
  - Native MVTRenderer for django-rest-framework
- Quality
  - Black-ified
  - iSort-ed
- Documentation
  - Add DRF and MapLibre examples

4.2 0.2.0 (2022-10-17)

- Possibly breaking change: * Base Mixin method get_tile use now class attributes for extent / buffer or clip_geom. Remove this parameters in your sub class method if needed.
- Bug fixes: * Correct usage for vector_tile_extent / vector_tile_buffer and vector_tile_clip_geom * Clipped geom is now working for mapbox
- Support Python 3.10 and django 4.1

4.3 0.1.0 (2021-02-25)

First beta release

- Add attribute to limit features in tile (unable to use a sliced queryset)

4.4 0.0.3 (2021-02-18)

- Delete useless Envelope transformation because django implicitly transform on intersects lookup (thanks to StefanBrand)
- Avoid useless queryset evaluation in some cases (thanks to StefanBrand)

4.5 0.0.2 (2021-02-12)

- Fix required 'fields' key in tilejson. Will be filled later
- Fix generated subquery to deal with DateField (thanks to StefanBrand)

4.6 0.0.1 (2020-10-22)

First Release

- **Generate Vector Tiles from django models**
  - in python
  - with PostGIS
- Generate associated TileJSON
- Default views to handle Vector tiles and tilejson

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search