Maps with Django (2): GeoDjango, PostGIS, Leaflet

A quickstart guide to create a web map with the Python-based web framework Django using its module GeoDjango, the PostgreSQL database with its spatial extension PostGIS and Leaflet, a JavaScript library for interactive maps.

© 2020 Paolo Melchiorre “Photo at the sunrise of Bivacco Fusco trail sign in Abruzzo, Italy”

Maps with Django (2 part series)

1. Maps with Django (1): GeoDjango, SpatiaLite, Leaflet
2. Maps with Django (2): GeoDjango, PostGIS, Leaflet

Contents

Abstract

Keeping in mind the Pythonic principle that “simple is better than complex” we’ll see how to create a web map with the Python based web framework Django using its GeoDjango module, storing geographic data in your PostgreSQL database on which to run geospatial queries with PostGIS.

Introduction

A *map* in a website is the best way to make geographic data easily accessible to users because it represents, in a simple way, the information relating to a specific geographical area and is in fact used by many online services.

Implementing a web *map* can be complex and many adopt the strategy of using external services, but in most cases this strategy turns out to be a major data and cost management problem.

In this guide we'll see how to create a web *map* with the **Python** based web framework **Django** using its **GeoDjango** module, storing geographic data in your local database on which to run geospatial queries.

Through this intervention you can learn how to add on your website a complex and interactive web *map* based on this 3 software:

- **GeoDjango**, the **Django** geographic module
- **PostGIS**, the **PostgreSQL** spatial extension
- **Leaflet**, a **JavaScript** library for interactive maps

Requirements

The requirements to create our map with Django are:

- a stable and supported version of Python 3 (tested with Python 3.8-3.10):

```
$ python3 --version
Python 3.10.0
```

- a Python virtual environment:

```
$ python3 -m venv ~/.mymap
$ source ~/.mymap/bin/activate
```

- the latest stable version of Django (tested with Django 3.1-4.0):

```
$ python3 -m pip install django~=4.0
```

Creating the 'mymap' project

To create the `mymap` project I switch to my `projects` directory:

```
$ cd ~/projects
```

and then use the `startproject` Django command:

```
$ python3 -m django startproject mymap
```

The basic files of our project will be created in the `mymap` directory:

```
$ tree --noreport --dirsfirst mymap/
mymap/
├── mymap
│   ├── asgi.py
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

Creating the 'markers' app

After switching to the `mymap` directory:

```
$ cd mymap
```

We can create our `markers` app with the Django startapp command:

```
$ python3 -m django startapp markers
```

Again, all the necessary files will be created for us in the `markers` directory:

```
$ tree --noreport --dirsfirst markers/
markers/
├── migrations
│   └── __init__.py
├── admin.py
├── apps.py
├── __init__.py
├── models.py
├── tests.py
└── views.py
```

Activating the 'markers' app

Now, we have to activate our `markers` application by inserting its name in the list of the `INSTALLED_APPS` in the `mymap` settings file.

`mymap/mymap/settings.py`

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "markers",
]
```

Adding an empty web map

We're going to add an empty web to the app:

Adding a template view

At this point we can proceed to insert, in the `views.py` file, a new `TemplateView` for the page of our map.

mymap/markers/views.py

```
"""Markers view."""

from django.views.generic.base import TemplateView


class MarkersMapView(TemplateView):
    """Markers map view."""

    template_name = "map.html"
```

Adding the 'map' template

We have to add a `templates/` directory in `markers/` :

```
$ mkdir templates
```

In the 'markers' templates directory we can now create a `map.html` template file for our map.

mymap/markers/templates/map.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Markers Map</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  </head>
  <body>
  </body>
</html>
```

For now we added only the usual boilerplate with a title but without a body content.

Adding ‘markers’ urls

In the `markers` URL file we must now add the path to view our map, using its template view.

mymap/markers/urls.py

```
"""Markers urls."""
from django.urls import path
from markers.views import MarkersMapView
app_name = "markers"
urlpatterns = [
    path("map/", MarkersMapView.as_view()),
]
```

Updating ‘mymap’ urls

As a last step we include in turn the URL file of the `marker` app in that of the project.

mymap/mymap/urls.py

```
"""mymap URL Configuration."""

from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("admin/", admin.site.urls),
    path("markers/", include("markers.urls")),
]
```

We just made a first view in Django, but it will show a blank page.

Testing the blank map page

You can test the blank map page running this command:

```
$ python3 manage.py runserver
```

Now that the server’s running, visit http://127.0.0.1:8000/markers/map/ with your Web browser. You’ll see a working blank map page.

We can now let’s move on to something more challenging.

Leaflet

Leaflet

- Leaflet is one of the most used JavaScript libraries for web maps
- It's a Free Software
- It's desktop and mobile friendly
- Leaflet is very light *(~39 KB of gzipped JS)*
- It has a very good documentation

Updating the 'map' template

To use Leaflet, we need to link its JavaScript and CSS modules in our template. We also need a DIV tag with 'map' as ID.

In addition, using the "static" template tag, we'll also link our custom JavaScript and CSS files, which we'll now create.

```
mymap/markers/templates/map.html
```

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Markers Map</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" type="text/css" href="{% static 'map.css' %}" />
    <link rel="stylesheet" type="text/css" href="https:///unpkg.com/leaflet/dist/leaflet.css" />
    <script src="https:///unpkg.com/leaflet/dist/leaflet.js"></script>
  </head>
  <body>
    <div id="map"></div>
    <script src="{% static 'map.js' %}"></script>
  </body>
</html>
```

Creating the static directory

We have to add a `static/` directory in `markers/` :

```
$ mkdir static
```

Adding the ‘map’ CSS

We add our `map.css` file in the `static` directory and, inside it, we add only the basic rules to show a full-screen map.

mymap/markers/static/map.css

```
html,
body {
    height: 100%;
    margin: 0;
}
#map {
    height: 100%;
    width: 100%;
}
```

Adding the ‘map’ JavaScript

In our `map.js` file we add the code to view our map.

mymap/markers/static/map.js

```
const copy = "© <a href='https://www.openstreetmap.org/copyright'>OpenStreetMap</a> contributors";
const url = "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png";
const osm = L.tileLayer(url, { attribution: copy });
const map = L.map("map", { layers: [osm] });
map.fitWorld();
```

Using the defined variables, we initialize an OpenStreetMap layer and we hook it to our ‘map’.

The last statement, sets a map view, that mostly contains the whole world, with the maximum zoom level possible.

Show the empty web map

We can now start our django project with the 'runserver' command

```
$ python3 manage.py runserver
```

Now that the server's running, visit http://127.0.0.1:8000/markers/map/ with your Web browser. You'll see a "Markers map" page, with a full page map. It worked!

We just created an empty map with Django and the result is pretty much what you see now.

A map, without markers, showing the whole world.

GeoDjango

It's time to get to know and activate GeoDjango, the Django geographic module.

Django added geographic functionality a few years ago *(v1.0 ~2008)*, in the `django.contrib.gis`, with specific fields, multiple database backends, spatial queries, and also admin integration.

Since then, many new useful features have been added every year, until the latest version:

- Spatialite backend *(v1.1 ~2009)*
- Multiple backends *(v1.2 ~2010)*
- OpenLayers-based widgets *(v1.6 ~2013)*
- GeoJSON serializer *(v1.8 ~2015)*
- GeoIP2 Geolocation *(v1.9 ~2016)*

Before activating it we need to install some requirements.

GDAL

A mandatory GeoDjango requirement is GDAL.

GDAL logo

- It's an OS/Geo library for reading and writing raster and vector geospatial data formats,
- It's released with a free software license
- It has a variety of useful command lines for data translation and processing.

Installing GDAL

We need to install the `GDAL` *(Geospatial Data Abstraction Library)* ^:

- on Debian-based GNU/Linux distributions (e.g. Debian 10, Ubuntu 20.04, ...):

```
$ sudo apt install gdal-bin
```

^ *for other platform-specific instructions read the Django documentation.*

Activating GeoDjango

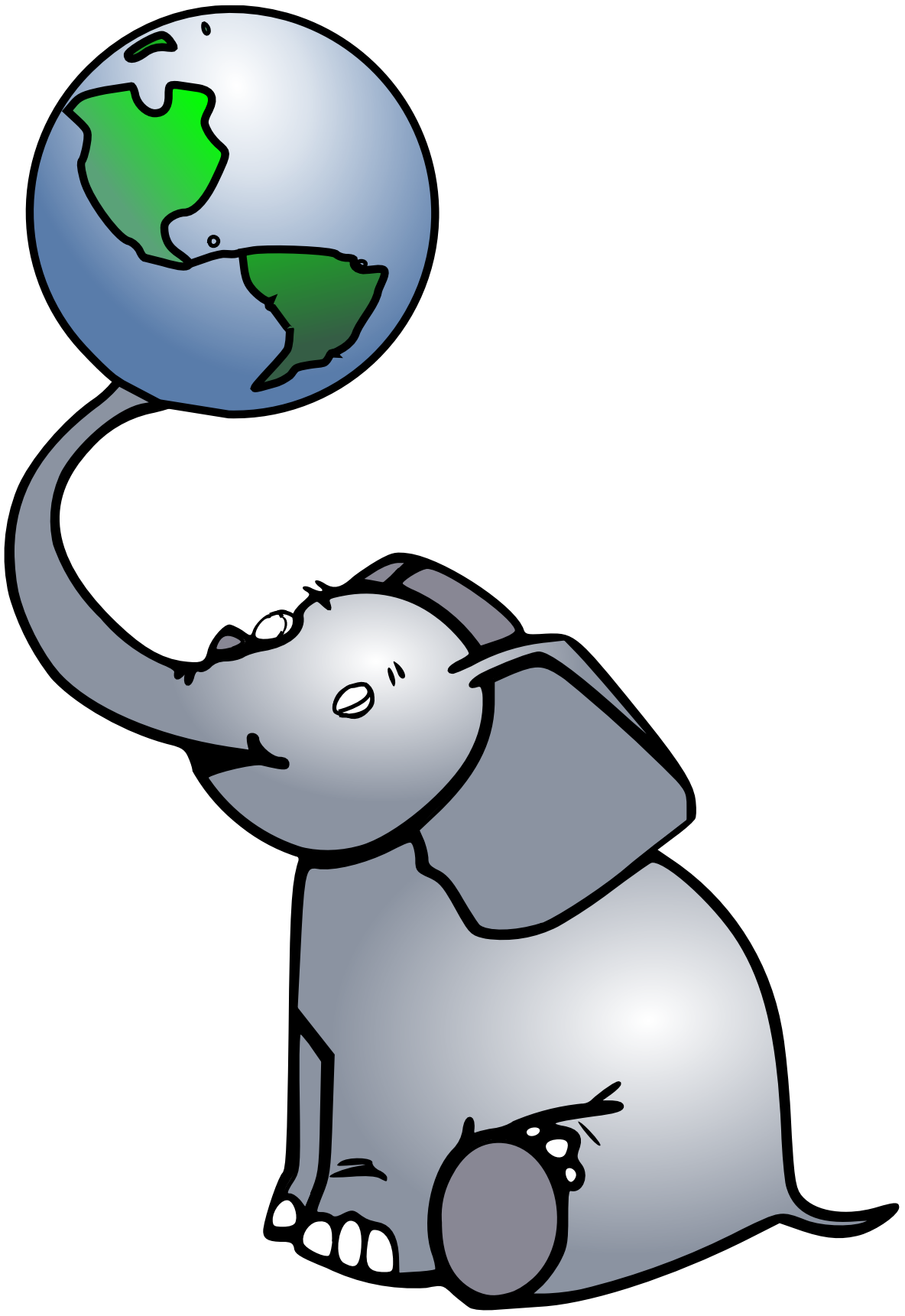
We can now activate GeoDjango by adding the `django.contrib.gis` module to the `INSTALLED_APPS`, in our project settings.

`mymap/mymap/settings.py`

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "django.contrib.gis",
    "markers",
]
```

PostGIS

And now let's start using PostGIS as a new backend engine.



PostGIS logo

- PostGIS is a PostgreSQL extension, and it's also the best database backend for GeoDjango.
- It internally integrates spatial data, and has spatial data types, indexes and functions.

Installing PostgreSQL C client library

In order to use `PostGIS` as a database backend, we need to install the `PostgreSQL C client library` ^.

- on Debian-based GNU/Linux distributions (e.g. Debian 11, Ubuntu 22.04, ...):

```
$ sudo apt install libpq5
```

^ *for other platform-specific instructions read the Django documentation.*

Requirements file

The python requirements of our project are increasing and therefore a good practice is to create a

requirements file, with the package list.

We'll use the Python PostgreSQL database adapter in addition to the already installed Django package.

mymap/requirements.txt

```
django~=4.0.0
psycopg2-binary~=2.9.0
```

> **Note**: The psycopg2-binary package is meant for beginners to start playing with Python and PostgreSQL without the need to meet the build requirements. For production use you are advised to use the source distribution. 🔗

Installing requirements

We install all the Python requirements, using the python package installer module.

```
$ python3 -m pip install -r requirements.txt
```

Activating PostGIS

We modify the project database settings, adding the PostGIS engine and the connection parameters of our PostgreSQL database, which you may have locally or remotely.

mymap/mymap/settings.py

```
DATABASES = {
    "default": {
        "ENGINE": "django.contrib.gis.db.backends.postgis",
        "HOST": "database",
        "NAME": "mymap",
        "PASSWORD": "password",
        "PORT": 5432,
        "USER": "postgres",
    }
}
```

> **Note**: You need to create/activate a PostgreSQL database instance (e.g. with system packages, with Docker or as a remote service) and then replace your credentials in the DATABASE settings.

Adding some markers

Now we can add some markers to the map.

Adding the Marker model

We can now define our `Marker` model to store a location and a name.

mymap/markers/models.py

```
"""Markers models."""

from django.contrib.gis.db import models


class Marker(models.Model):
    """A marker with name and location."""

    name = models.CharField(max_length=255)
    location = models.PointField()

    def __str__(self):
        """Return string representation."""
        return self.name
```

Our two fields are both mandatory, the location is a simple PointField, and we'll use the name to represent the model.

Activating the Marker admin

To easily insert new markers in the map we use the Django admin interface.

mymap/markers/admin.py

```
"""Markers admin."""

from django.contrib.gis import admin
from markers.models import Marker


@admin.register(Marker)
class MarkerAdmin(admin.OSMGeoAdmin):
    """Marker admin."""

    list_display = ("name", "location")
```

We define a Marker admin class, by inheriting the GeoDjango admin class, which uses the OpenStreetMap layer in its widget.

Updating the database

We can now generate a new database migration and then apply it to our database.

```
$ python3 manage.py makemigrations
```

```
$ python3 manage.py migrate
```

Testing the admin

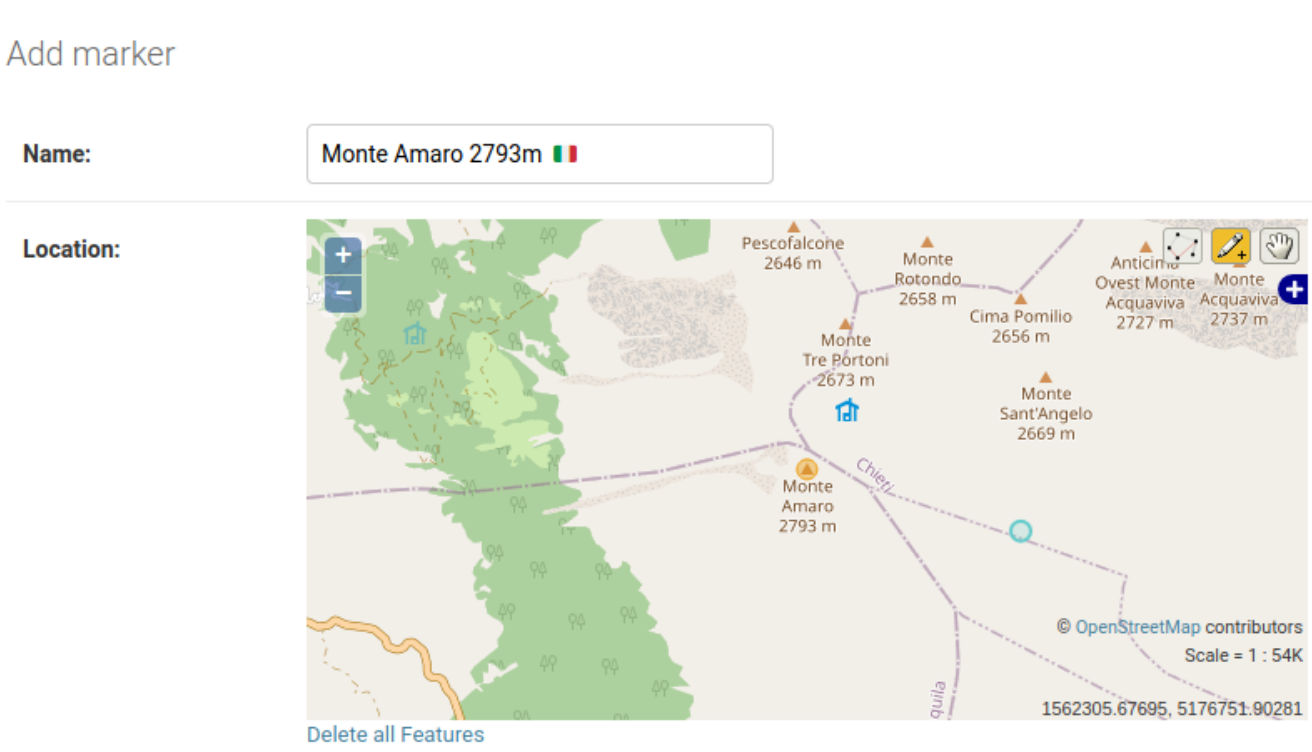
We have to create an admin user to login and test it:

```
$ python3 manage.py createsuperuser
```

Now you can test the admin running this command:

```
$ python3 manage.py runserver
```

Now that the server's running, visit http://127.0.0.1:8000/admin/markers/marker/add/ with your Web browser. You'll see a "Markers" admin page, to add new markers with a map widget. I added a marker to the latest peak I climbed: "Monte Amaro 2793m 🇮🇹"



Adding a marker in the admin page.

> **Note**: In this version of the Marker admin you have to manually navigate on the map and pin the marker to your desired location.

Showing markers in the map

Requirements file

We're going to use additional packages for our advanced map: Django filter, Django REST Framework and its geographic add-on.

mymap/requirements.txt

```
django-filter~=21.1.0
djangorestframework-gis~=0.18.0
djangorestframework~=3.13.0
django~=4.0.0
psycopg2-binary~=2.9.0
```

Installing requirements

We install all the Python requirements, using the python package installer module.

```
$ python3 -m pip install -U -r requirements.txt
```

Activating Django REST Framework

The packages that we'll use directly in the code of our project are Django REST Framework and its geographic add-on which we then insert in the list of `INSTALLED_APPS` of our project settings.

mymap/mymap/settings.py

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "django.contrib.gis",
    "rest_framework",
    "rest_framework_gis",
    "markers",
]
```

Adding the Marker serializer

Let's create a serializer for our Marker class.

mymap/markers/serializers.py

```
"""Markers serializers."""

from rest_framework_gis import serializers

from markers.models import Marker


class MarkerSerializer(serializers.GeoFeatureModelSerializer):
    """Marker GeoJSON serializer."""

    class Meta:
        """Marker serializer meta class."""
```

```
        fields = ("id", "name")
        geo_field = "location"
        model = Marker
```

Inheriting from a 'rest_framework_gis' serializer, we only have to define the Marker model, the geographical field 'location' and also the optional fields, to be shown as additional properties.

Serialized GeoJSON

The `GeoFeatureModelSerializer` serializer will generate a GeoJSON like this:

```
{
    "type": "FeatureCollection",
    "features": [
    {
        "id": 1,
        "type": "Feature",
        "geometry": {
        "type": "Point",
        "coordinates": [14.085910318319995, 42.086280141658]
        },
        "properties": {
        "name": "Monte Amaro 2793m 🇮🇹"
        }
    }
    ]
}
```

Adding the Marker viewset

Our intention is to expose our markers via a RESTful API and to do so we define a read-only viewset.

We set the location as a field to filter our markers, and then a filter based on the bound box.

We also return all our Marker instances, without limitations or filters.

mymap/markers/viewsets.py

```
"""Markers API views."""
from rest_framework import viewsets
from rest_framework_gis import filters

from markers.models import Marker
from markers.serializers import MarkerSerializer


class MarkerViewSet(viewsets.ReadOnlyModelViewSet):
    """Marker view set."""

    bbox_filter_field = "location"
    filter_backends = (filters.InBBoxFilter,)
    queryset = Marker.objects.all()
    serializer_class = MarkerSerializer
```

Adding API 'markers' urls

In the `markers` application, we define the URL of our new endpoint using the Django REST Framework default router, to create our path.

mymap/markers/api.py

```python
"""Markers API URL Configuration."""

from rest_framework import routers

from markers.viewsets import MarkerViewSet

router = routers.DefaultRouter()
router.register(r"markers", MarkerViewSet)

urlpatterns = router.urls
```

Updating 'mymap' urls

Finally, we add to the definition of the URL of our project, a new path for the API that includes the path just specified for our 'marker' app.

mymap/mymap/urls.py

```python
"""mymap URL Configuration."""

from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/", include("markers.api")),
    path("markers/", include("markers.urls")),
]
```

Trying to locate the user

After finishing our RESTful API we move-on to updating our javascript file.

We try to locate the user: in the positive case we'll use it's location to center the map, in the negative case we'll locate him on an arbitrary point in the map, with a low zoom level.

mymap/markers/static/map.js

```javascript
const copy = "© <a href='https://www.openstreetmap.org/copyright'>OpenStreetMap</a> contributors";
const url = "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png";
const osm = L.tileLayer(url, { attribution: copy });
const map = L.map("map", { layers: [osm], minZoom: 5 });
map.
    locate()
    .on("locationfound", (e) => map.setView(e.latlng, 8))
    .on("locationerror", () => map.setView([0, 0], 5));
// …
```

Rendering markers incrementally

We ask our endpoint to return only the markers of the specific displayed area, passed as a boundbox string.

To build the marker layer, we ask our endpoint for data asynchronously and extract the properties we want to show in the pop-ups.

We invoke this flow, every time the user stops moving on the map.

mymap/markers/static/map.js

```
// …

async function load_markers() {
    const markers_url = `/api/markers/?in_bbox=${map.getBounds().toBBoxString()}`
    const response = await fetch(markers_url)
    const geojson = await response.json()
    return geojson
}

async function render_markers() {
    const markers = await load_markers();
    L.geoJSON(markers)
    .bindPopup((layer) => layer.feature.properties.name)
    .addTo(map);
}

map.on("moveend", render_markers);
```

Testing the populated map

And finally here is our complete map.

In this example, we can see how the markers in a specific map area look.

The loading takes place in a very fluid way, because the number of calls occurs only when the movement on the map stops and therefore the data traffic is reduced to the essentials as well as the rendering of the markers carried out by Leaflet.

You can test the populated web map running this command:

```
$ python3 manage.py runserver
```

Now that the server's running, visit http://127.0.0.1:8000/markers/map/ with your Web browser. You'll see the "Markers map" page, with a full page map and all the markers. It worked!

A map with mountain peaks and spot elevation locations from the free Natural Earth geography map dataset.

Curiosity

If you want to know more about my latest hike to the Monte Amaro peak you can see it on my Wikiloc account: Round trip hike from Rifugio Pomilio to Monte Amaro 🔗.

Conclusion

We have shown an example of a fully functional map, trying to use the least amount of software, without using external services.

This is an advanced map and it uses only GeoDjango, PostGIS and Leaflet to render very large numbers of markers in a fluid and dynamic way.

In future articles we will see how to add many other features:

- shapefile loading to Django models
- markers and popups customization to show additional information
- marker filtering based on relation data
- clustering of the markers to make the loading of data more efficient
- use of Geocoding services to add marker locations starting from the address

Stay tuned.

— Paolo

> **Resources**

- GeoDjango - Django Documentation 🔗
- Leaflet API reference 🔗
- Django REST framework 🔗
- PostGIS — Documentation 🔗

> **Update (2021-11-19)**: update code samples

Meta

19 Jul 2021

Paolo Melchiorre

Article 📝

Django 🦄 · GeoDjango 🌍 · Leaflet 🍃 · Web Map 🗺️ · Python 🐍 · PostGIS 🌍 · PostgreSQL 🐘

≪ DjangoCon Europe...

EuroPython 2021 ≫