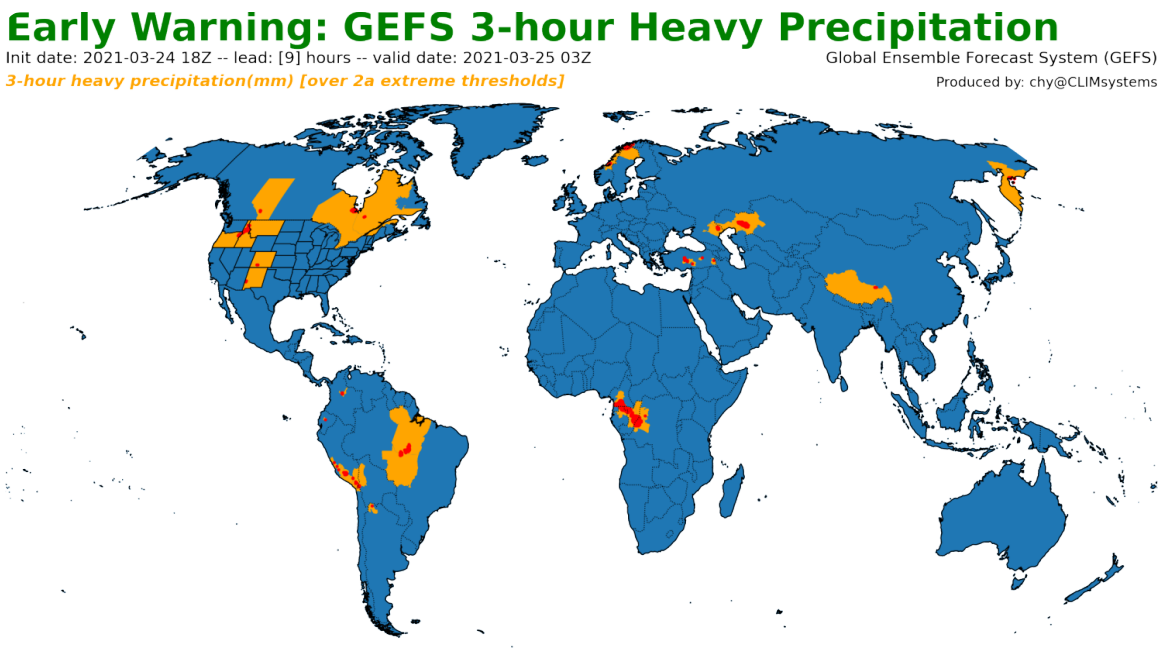
Build A Simple Point-in-Polygon Web API Using FastAPI

Chonghua Yin
Climate Scientist | Data Scientist | Programmer

On year ago, I built an early warning service, which applied the point-in-polygon (PIP) technique to identify the regions (in polygons) facing 3-hour heavy precipitation risks (in points) identified from real-time GEFS forecasts with latitude and longitude coordinates. A demonstration is displayed in the following image. The core algorithm is to use point data to find the area it belongs to. In essence, it is a reverse geocoding process, which converts the latitude and longitude coordinates to a readable address.



In our service, we only care about national and provincial administrative boundaries just as the image shows. It is a simplified reverse geocoding process. Therefore, we developed the algorithm by ourselves instead of applying those commercial geocoding services (such as Google Map API). Our algorithm is based on R-Tree, which basically satisfy our current requirements for severe weather early warning. In fact, it is quite easy to wrap it into web APIs. Thus, it can be accessed and applied by other similar services or products that require a simplified Reverse Geocoding service.

In this tutorial, let's build a simple reverse geocoding API using FastAPI and other python packages. **FastAPI** is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints. The speed of the framework is compared with other competitors like NodeJS or Go. Also is fast to

code (you'll see this in a moment), easy to understand, intuitive and robust.

It is worth noting that this is not an extensive FastAPI tutorial and we will only use some simple functions provided by FastAPI to mainly stay on our simple reverse geocoding functions.

1. Prepare World Map Data

You can download latest world map from **GADM** or from **Natural Earth**. They are shapefiles and contain many properties. We can use **GeoPandas** to open it and remove redundant information to only keep three properties of provincial administrative name (State), country name (Country), and geometry (polygons). Then save the data to the Parquet format, which is a popular columnar storage format to store geospatial vector data (point, lines, polygons) in Apache Parquet and has faster reading and writing speed than the Shapefile format. You may also have to the function of "dissolve" to aggregate finer administrative boundaries to provincial level or you can do it with geospatial software such as ArcMap or QGIS.

Let's take the GADM data as an example. Some basic code looks like:

```
import geopandas as gp

shp_file = "data/gadm404_Level1.shp"

df_states = gpd.read_file(shp_file)
df_states = df_states[df_states.NAME_0!="Antarctica"]
df_states = df_states.rename({'NAME_1': "State", 'NAME_0': 'Co
df_states[['Country', 'State', 'geometry']].to_parquet('data/g
```

Then we can load the data into memory as:

```
def load_glb_shp()
    gpd.options.use_pygeos = False
    shp_file = 'data/gadm404_Level1.parquet'
    return gpd.read_parquet(shp_file)


gdf_glb = load_glb_shp():
```

2. Develop Web APIs with FastAPI

This part applies quite a few python packages.

```
import uvicor
from typing import List
from pydantic import BaseModel
from fastapi import FastAPI, HTTPException, status
from fastapi.middleware.cors import CORSMiddleware
import geopandas as gpd
from shapely.geometry import Pointn
```

2.1 Let's setup API server first

```
cpu_count = 4
API_LINK = '127.0.0.1:8000'


app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
    allow_credentials=True,
)


if __name__ == '__main__':
    host, port = API_LINK.split(':')
    uvicorn.run(
        "main:app",
        host=host,
        port=int(port),
        factory=False,
        reload=False,
        loop='asyncio',
        workers=cpu_count,
    )
```

Also add a welcome page

```
@app.get("/"
async def welcome() -> dict:
    return dict(message="Welcome to use point in polygon servi
                        "Check geoinfo at the location of lati
                ))
```

Let's put the code into main.py and run it

```
python main.py
```

You can see the following image. Congratulations! You already finish the first step.

```
// http://127.0.0.1:8000/

{
  "message": "Welcome to use point in polygon service.\nCheck geoinfo at the location
of latitude and longitude"
}
```

2.2 Reverse Geocoding

Our APIs only focus on two Reverse Geocoding functions:

- pip

Single point query. This API returns the State and Country where the point is inside.

- pips

Multi-point query. This API only return the States and Countries where the points are inside without the points over ocean.

2.2.1 Single Point Query (pip)

The key input is latlng, where lat is for latitude while lng is for longitude.

- longitude: Geospatial coordinate, longitude value in degrees. Valid value is a real number and in the range [-180, +180].

- latitude: Geospatial coordinate, latitude value in degrees. Valid value is a real number and in the range [-90, +90].

Note: *I put latitude and longitude into a single string of latlng as it is easier to copy and paste a testing point from Google Map :).*

```
@app.get('/pip/'
async def api_pip(latlng: str = ''):
    latlng = [eval(i) for i in latlng.replace(' ', '').split()
    loc_lon = latlng[0][1]
    loc_lat = latlng[0][0]


    if (loc_lat > 90.0) or (loc_lat < -90.0):
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="latitude should be between 90 and -90.",
        )


    if (loc_lon > 180.0) or (loc_lon < -180.0):
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="longitude should be between -180 and 180."
        )


    return await pointinpolygon(gdf_glb, loc_lat, loc_lon))
```

The backend real query function is presented as follows:

```
async def pointinpolygon(gdf_globe, loc_lat, loc_lon)
    r_index = gdf_globe.sindex
    pt_latlon = Point(loc_lon, loc_lat)
    idx = r_index.query(pt_latlon, predicate="within").T


    if not idx:
        return {"longitude": loc_lon,
                "latitude": loc_lat,
                "State": "Ocean",
                "Country": "Ocean",
                }
    else:
        df_pt = gdf_globe[['State', 'Country']].iloc[idx].valu
        return {"longitude": loc_lon,
                "latitude": loc_lat,
                "State": df_pt[0],
                "Country": df_pt[1],
                }:
```

2.2.2 Multi-Point Query (pips)

The key input is two lists- one for latitude and another for longitude

- longitude: Geospatial coordinate, longitude value in degrees. Valid value is a real number and in the range [-180, +180].
- latitude: Geospatial coordinate, latitude value in degrees. Valid value is a real number and in the range [-90, +90].

```
class Item(BaseModel)
    latitude: List[float] = []
    longitude: List[float] = []



@app.post('/pips/')
async def api_pips(args: Item):
    if args.latitude and args.longitude:
        if len(args.latitude) != len(args.longitude):
                raise HTTPException(
```

```
                    status_code=status.HTTP_404_NOT_FOUND,
                    detail="The latitude and longitude should
                )

        results = await pointinpolygons(gdf_glb, args)
        return results
    else:
        return {'Alert': 'Please provide latitude and longitud
```

The backend real query function is presented as follows. To make it robust, you should add some check to make sure latitude and longitude within the proper range. Leave it to you for exercise.

```
async def pointinpolygons(gdf_globe, locs:Item
    loc_lats = locs.latitude
    loc_lons = locs.longitude
    gdf_locs = gpd.GeoSeries(gpd.points_from_xy(loc_lons, loc_


    r_index = gdf_globe.sindex
    idx = r_index.query_bulk(gdf_locs, predicate="within").T


    land_points = gdf_locs.iloc[idx[:, 0]]
    lons = [ew_point.coords.xy[0][0] for ipt, ew_point in enum
    lats = [ew_point.coords.xy[1][0] for ipt, ew_point in enum


    df_land_points = gdf_globe[['State', 'Country']].iloc[idx[
    df_land_points['latitude'] = lats
    df_land_points['longitude'] = lons

    return df_land_points[['longitude', 'latitude', "State", '
```

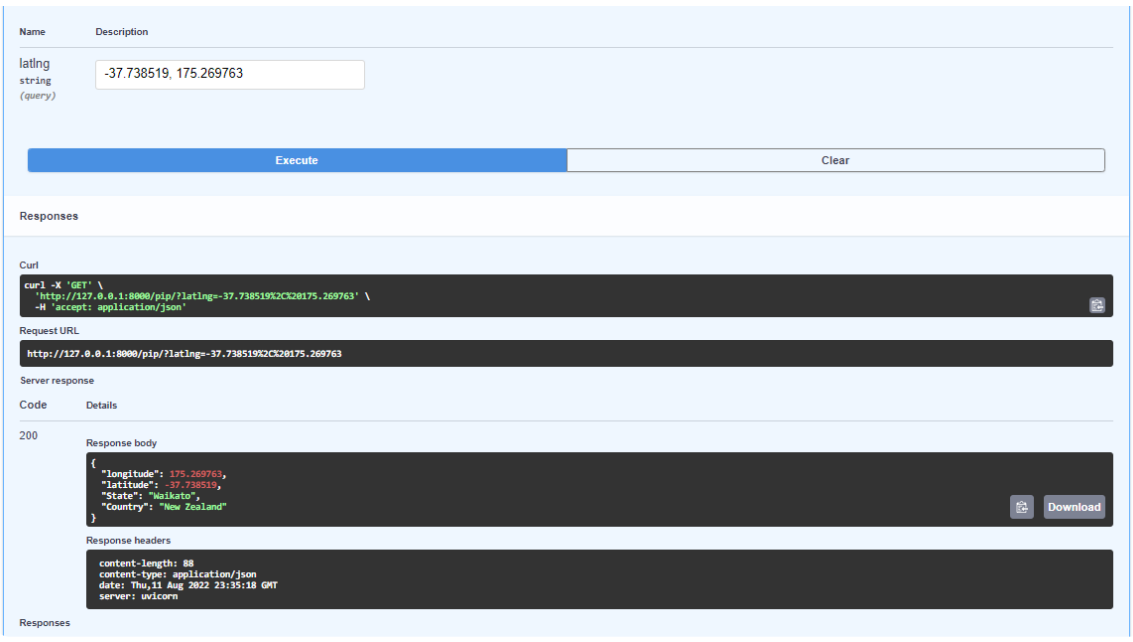
2.3 Showoff Time

We can have a test with the **Swagger UI**. Swagger UI allows anyone — be it your development team or your end consumers — to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption. Of course, you can also use command line tools such as curl to have a test.

Input **http://127.0.0.1:8000/docs**, you can see

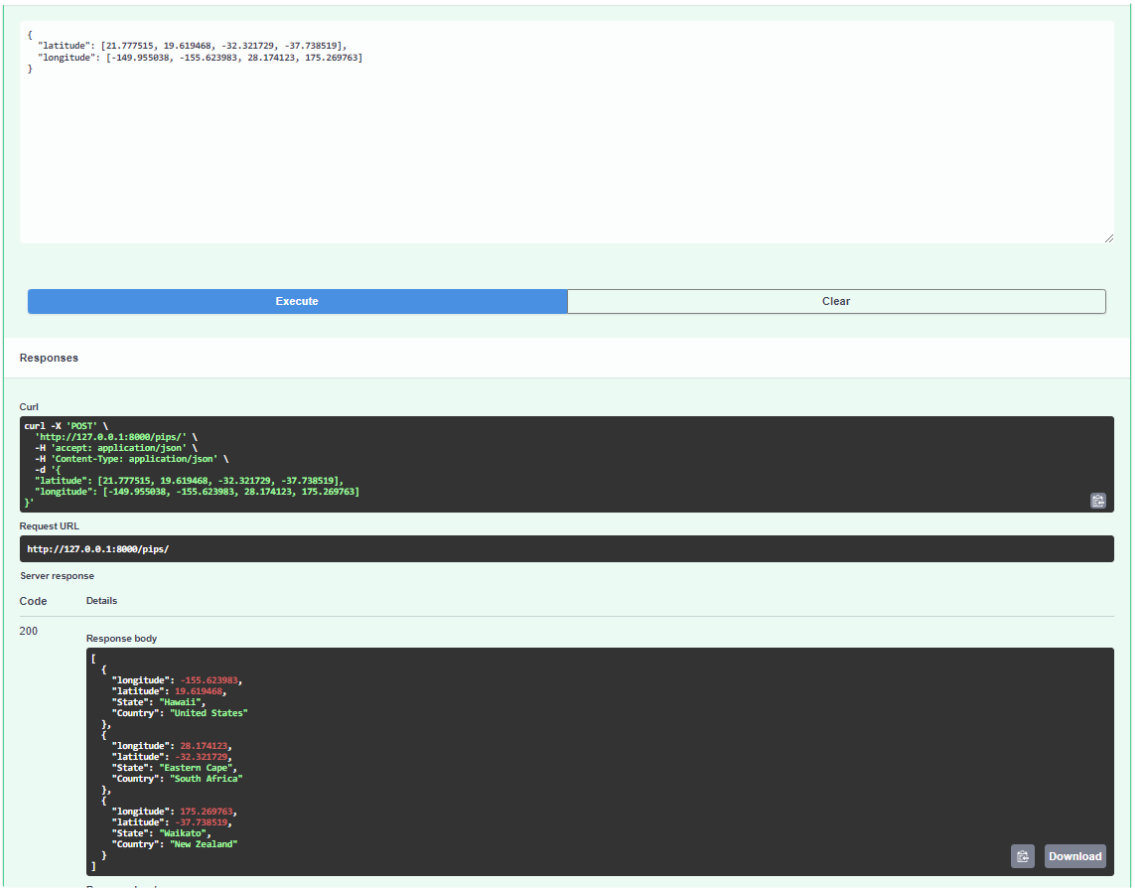
2.3.1 Single Point Query(pip)

let's select a point (-37.738519, 175.269763) in Waikato, New Zealand from the Google Map. copy and paste directly :)



2.3.2 Multi-Point Query(pips)

let's select several points from Google Maps. *It is worth noting that only points on land are returned.*



Summary

Based on FastAPI and other python packages (e.g., geopandas and shapely), it is quite fast and easy to develop and deploy a simple Reverse Geocoding API. Moreover, the Swagger UI provides a very good interface for testing. Although current tutorial just provides a simple prototype and leaves many room for improvements, I think that such an API or the underlying idea could already satisfy basic Reverse Geocoding services in most of cases.

If you like it, you can dig deeper and improve it as you will. For example, you can try other Spatial Indexing methods instead of R-Tree, etc. Hope it is helpful and not only just for fun!

References and resources

https://fastapi.tiangolo.com/

https://geopandas.org/en/stable/

https://swagger.io/tools/swagger-ui/

https://shapely.readthedocs.io/en/stable/manual.html

https://pygeos.readthedocs.io/en/stable/

https://rapidapi.com/blog/api-design-best-practices-principles/

https://docs.microsoft.com/en-us/rest/api/maps/spatial/post-point-in-polygon?tabs=HTTP