Blue Onion Labs Take Home Test

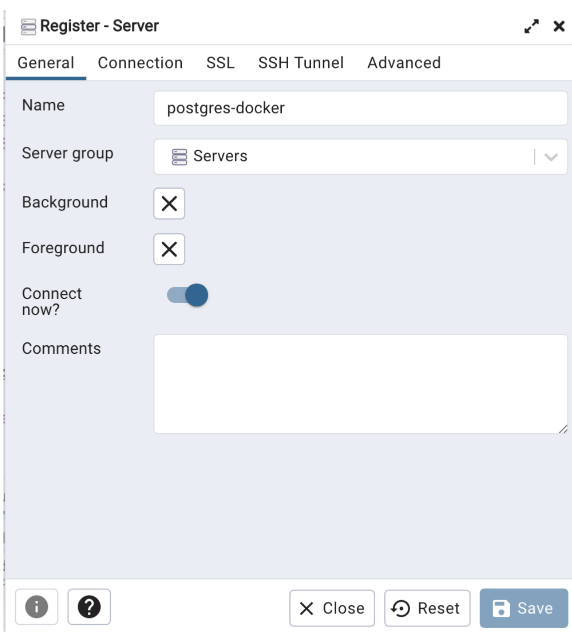
The Task (Part 1)

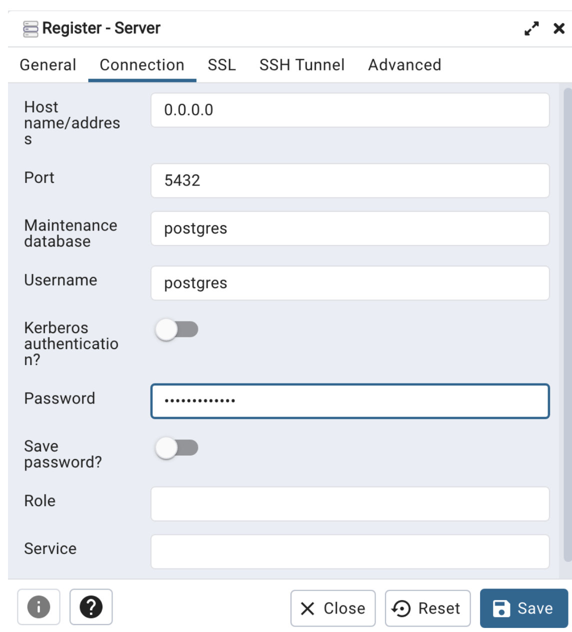
I'll use a Docker container with a basic version of Postgresql. To make it work, we need first to install Docker on our computer. Then, after running docker client, we'll type:

```
docker run --name postgres-docker --rm -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=4y7sV96vA9wv46VR -e PGDATA=/var/lib/postgresql/data/pgdata -v
/tmp:/var/lib/postgresql/data -p 5432:5432 -it postgres:14.1-alpine
```

We are using **postgres-docker** as a name for the container, setting up the credentials and the folder that will store data. We are using **postgres:14:1-alpine image**, which if not present, will be downloaded.

Once we had the docker container running, we'll be able to connect to some postgres client, like **pgAdmin**, the one I used for this assessment. These are the credentials we must set when registering a new server:

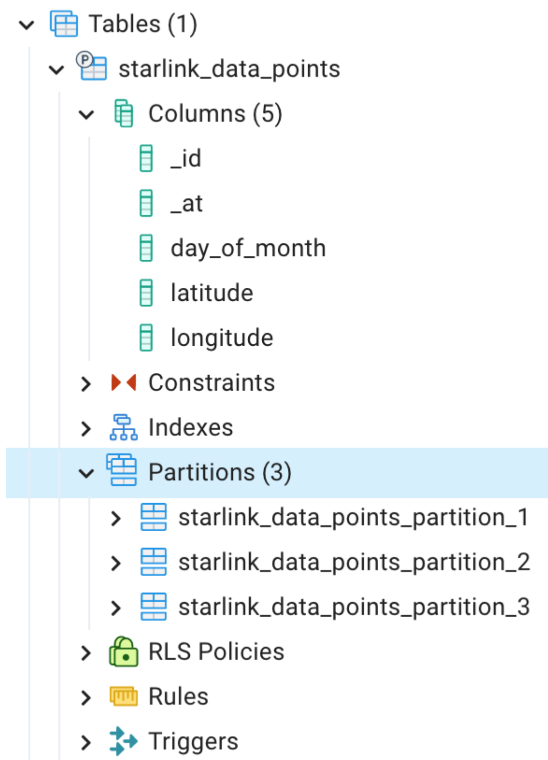




After connecting, we'll land in the public schema, which is empty. For this exercise, I decided to create a new schema, called **blue_onion**.

The Task (Part 2)

For this purpose, I'll create a table in the schema **blue_onion**, called **starlink_data_points**. The table structure will be very simple:



As you can see, I made three partitions, one for each third of month (from days 1-10, 10-20 and 20-31), so insertions can be optimized. The pipeline for populating our table is written in the jupyter notebook located at /jupyters/blue_onion_test.ipynb, under the subtitle **The Task (Part 2).** Basically, I'm reading the *json* file, filtering the columns I need and writing them to our Postgres database.

The Task (Part 3)

For this part, I'm using a library to interact with Postgres, sqlalchemy . It uses pyscopg2 internally among many others and has a better interface to do things like configuring connections, inserting and querying. I'm writing a small function in Python and a little test:

```
user, password, host, port, db, schema = ('postgres','4y7sV96vA9wv46VR','0.0.0.0','5432','postgres','blue_onion')
```

```
def last_known_position(_id,data_time_point):
    stringyfied_timestamp = datetime.strftime(data_time_point,'%Y/%m/%d %H:%M:%S')

    conn_string = f"postgresql://{user}:{password}@{host}:{port}/{db}"
    conn = create_engine(conn_string).connect()
    res = conn.execute(
    f"""
    select _id, latitude, longitude
    from blue_onion.starlink_data_points
    where _id = '{_id}' and _at =
    (
    select max(_at) from blue_onion.starlink_data_points
    where _id = '{_id}' and _at <= '{stringyfied_timestamp}'
    )
    """)

    try:
        last_position_row = res.first()
        last_position = (last_position_row[1],last_position_row[2])
        return last_position
    except:
        return (-1,-1)
```

```
timestamp_str = '2021-01-26 07:00:00'
_id = '5eed7714096e59000698563e'
data_time_point = datetime.strptime(timestamp_str, '%Y-%m-%d %H:%M:%S')
last_position = last_known_position(_id,data_time_point)

print(f'lets see the last position: {last_position}')
```

As it can be shown in the figure the function last_known_position connects to the database and tries to get the closest date with data. Error handling was not developed, and the credentials were copy and pasted raw. But I think the idea can be seen clearly.

Bonus Task (Part 4)

For this bonus task I will show a diagram and explain the idea of the solution, so I can eventually dive deeper into that.

First, we can download the haversine library, which allows us to calculate a distance between two points knowing only latitude and longitude coordinates.

Second, the function will be closest_satellite(latitude,longitude,time). It will take coordinates and time, and will return a tuple: (_id, latitude,longitude,time). We will calculate distances between the input point and the satellites' positions located in our table using haversine function, as it is shown in the tutorial:

Usage

Calculate the distance between Lyon and Paris

```
from haversine import haversine, Unit

lyon = (45.7597, 4.8422) # (lat, lon)
paris = (48.8567, 2.3508)

haversine(lyon, paris)
>> 392.2172595594006  # in kilometers

haversine(lyon, paris, unit=Unit.MILES)
>> 243.71250609539814  # in miles

# you can also use the string abbreviation for units:
haversine(lyon, paris, unit='mi')
>> 243.71250609539814  # in miles

haversine(lyon, paris, unit=Unit.NAUTICAL_MILES)
>> 211.78037755311516  # in nautical miles
```

We'll order the results by distance and time_diff(). So, if there are two snapshots where the position of satellite was the same, we'll keep the more recent one.