**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:

|  |  |
| --- | --- |
| Graphical user interface, text, application, chat or text message  Description automatically generated | Graphical user interface, text, application  Description automatically generated |

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:

Graphical user interface, application

Description automatically generated with medium confidence

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:

Graphical user interface, text, application, email

Description automatically generated

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:

Graphical user interface, text, application, email

Description automatically generated

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.