

# Outline

- **1. Docker**
  - **1.1. Homework: Docker basics**
- **2. Postgres**
  - 2.1. Running Postgres in Docker with pgcli
  - 2.2. Ingesting Data Into Postgres
  - 2.3. Running Postgres and pgAdmin with Docker
  - 2.4. Dockerizing the data ingestion
  - 2.5. Running Postgres and pgAdmin with Docker-Compose
  - 2.6. Homework: using SQL
- **3. Terraform**
  - 3.1. Creating and configuring a project on Google Cloud Platform (GCP))
  - 3.2. Configure and Deploy with Terraform
  - 3.3. Homework: Terraform

## 1. Docker

### 1.1. Homework: Docker basics

- **Question 1. Knowing docker tags**

Run the command to get information on Docker

```
docker --help
```

Now run the command to get help on the "docker build" command:

```
docker build --help
```

Do the same for "docker run".

Which tag has the following text? - *Automatically remove the container when it exits*

- `--delete`
- `--rc`
- `--rmc`
- `--rm`

We can search for the command using pipe operator `|` and `grep` command to filter the output:

```
docker run --help | grep "Automatically"
```

**output:**

```
--rm                Automatically remove the container when it exits
```

- **Question 2. Understanding docker first run**

Run docker with the python:3.9 image in an interactive mode and the entrypoint of bash. Now check the python modules that are installed (use `pip list`).

What is version of the package *wheel*?

- **0.42.0**
- 1.0.0
- 23.0.1
- 58.1.0

Use `--entrypoint=bash` to override the default entrypoint of the container and run the Bash shell, allowing for manual operations like installing Python packages. The entrypoint is the command that Docker runs by default when the container starts.

```
docker run -it --entrypoint=bash python:3.9
```

After run the cl, inside of the container, run `pip list` to check the python modules that are installed:

```
root@1434e4eadd55:/# pip list
Package    Version
-----
```

```
pip          23.0.1
setuptools  58.1.0
wheel        0.42.0
```

## 2. Postgres

### Dataset:

We'll use the green taxi trips from September 2019:

```
wget https://github.com/DataTalksClub/nyc-tlc-data/releases/download/green/green_tripdata_2019-09.csv.gz
```

and the dataset zones:

```
wget https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv
```

### 2.1. Running Postgres in Docker with pgcli

The image we'll use is called `postgres`. We'll use the latest version, which is `13`. We need to configure the following environment variables:

- `POSTGRES_USER` - the username for the database
- `POSTGRES_PASSWORD` - the password for the database
- `POSTGRES_DB` - the name of the database

We need to use the flags `-e` and `-v` to set environment variables and mount volumes, respectively. The `-v` format is `-v host_path:container_path`, and PostgreSQL by default is listen on port 5432. Also, with `-p` flag we can expose the port from the container to the host. The format is `-p host_port:container_port`, and the default data directory in docker is `/var/lib/postgresql/data`. The full command is:

```
docker run -it \
  -e POSTGRES_USER="marcosbenicio" \
  -e POSTGRES_PASSWORD="0102" \
  -e POSTGRES_DB="ny_taxi" \
  -v $(pwd)/taxi-trip-postgres:/var/lib/postgresql/data \
  -p 5432:5432 \
  postgres:13
```

The necessity to specify a volume is because the data is stored in the container, and when the container is removed, the data is lost. So, we need to mount a volume to persist the data.

To easily run this command, we can create a `run-postgres.sh` file with the docker command and turn into a executable file with:

```
chmod +x run-postgres.sh
```

Then, we can run the executable file with:

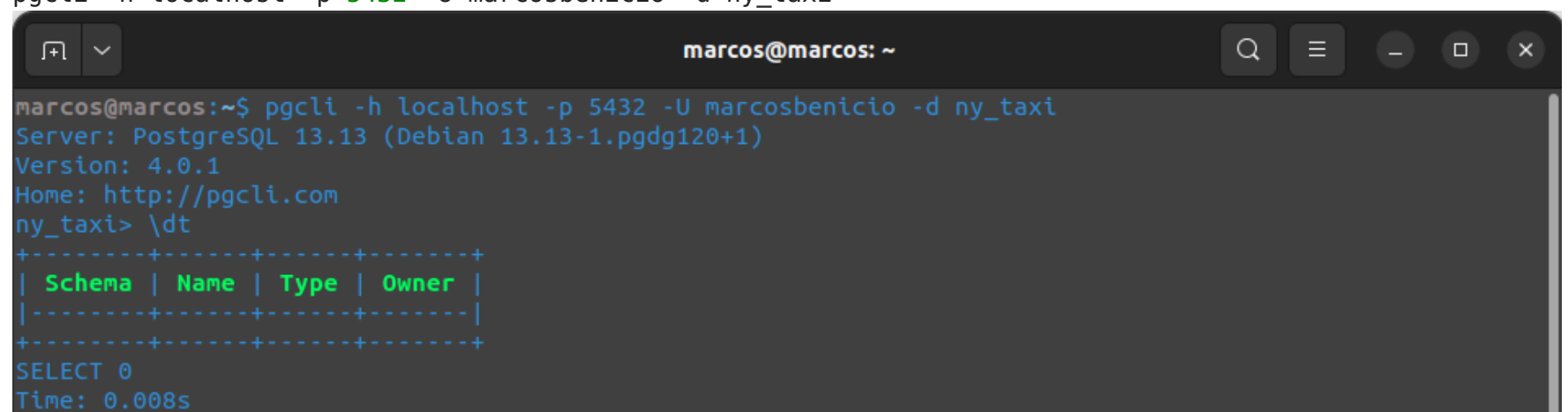
```
./run-postgres.sh
```

To interact with the database from terminal, we can install `pgcli`:

```
pip install pgcli
```

and connect to the database:

```
pgcli -h localhost -p 5432 -U marcosbenicio -d ny_taxi
```



```
marcos@marcos:~$ pgcli -h localhost -p 5432 -U marcosbenicio -d ny_taxi
Server: PostgreSQL 13.13 (Debian 13.13-1.pgdg120+1)
Version: 4.0.1
Home: http://pgcli.com
ny_taxi> \dt
+-----+-----+-----+-----+
| Schema | Name | Type | Owner |
+-----+-----+-----+-----+
SELECT 0
Time: 0.008s
```

### 2.2. Ingesting Data Into Postgres

To ingest data into Postgres, we can use the sql alchemy library. First, we need to install it:

```
pip install sqlalchemy
```

Then, we can use the following code to ingest the data:

```
engine = create_engine('postgresql://username:password@localhost:port/database_name')
```

```
In [ ]: from sqlalchemy import create_engine
import pandas as pd

# Create a connection to the database
engine = create_engine('postgresql://marcosbenicio:0102@localhost:5432/ny_taxi')


df_taxi_trip = pd.read_csv('data/green-taxi-trip-2019-09.csv',
                           parse_dates=['lpep_pickup_datetime', 'lpep_dropoff_datetime'],
                           low_memory=False)

# Create a table in database from the pandas dataframe
df_taxi_trip.to_sql(name = "green_taxi_trip", con = engine, if_exists = "replace")

# Create a ddl schema from the pandas dataframe
print(pd.io.sql.get_schema(df_taxi_trip, name = "green_taxi_trip", con = engine))
```

```
CREATE TABLE green_taxi_trip (
  "VendorID" FLOAT(53),
  lpep_pickup_datetime TIMESTAMP WITHOUT TIME ZONE,
  lpep_dropoff_datetime TIMESTAMP WITHOUT TIME ZONE,
  store_and_fwd_flag TEXT,
  "RatecodeID" FLOAT(53),
  "PULocationID" BIGINT,
  "DOLocationID" BIGINT,
  passenger_count FLOAT(53),
  trip_distance FLOAT(53),
  fare_amount FLOAT(53),
  extra FLOAT(53),
  mta_tax FLOAT(53),
  tip_amount FLOAT(53),
  tolls_amount FLOAT(53),
  ehail_fee FLOAT(53),
  improvement_surcharge FLOAT(53),
  total_amount FLOAT(53),
  payment_type FLOAT(53),
  trip_type FLOAT(53),
  congestion_surcharge FLOAT(53)
)
```

We can check the existing tables in the database from the terminal using the `\dt` command or a query:



The screenshot shows a terminal window with the prompt `ny_taxi>`. The user enters the command `\dt`, which returns a table of database objects. The output is as follows:

Schema	Name	Type	Owner
public	green_taxi_trip	table	marcosbenicio

Below the table, the terminal shows the execution of a SQL query: `SELECT 1`, with a time of `0.004s`. The user then enters `SELECT table_name FROM information_schema.tables WHERE table_schema = 'public';`, which returns a single row: `green_taxi_trip`. The time for this query is `0.005s`. The prompt `ny_taxi>` is shown again at the bottom.

In PostgreSQL, the information about the tables, including their structure, columns, and other metadata, is stored in a kind of system catalogs within the `information_schema` and `pg_catalog` schemas. These are special schemas that hold metadata about the database.

To remove a table from our database, we can use the `DROP TABLE` command:

```
DROP TABLE green_taxi_trip;
```

## 2.3. Running Postgres and pgAdmin with Docker

It's not convenient to use the pgcli for data exploration and querying. Instead, we can use pgAdmin, which is a web-based interface for managing PostgreSQL databases. We can download the image of pgAdmin from docker hub:

```
docker pull dpage/pgadmin4
```

Then, we can run the docker container with the following environment variables, similar to the postgres container:

```
docker run -it \
  -e PGADMIN_DEFAULT_EMAIL="marcosbenicio@admin.com" \
  -e PGADMIN_DEFAULT_PASSWORD="0102" \
  -p 8080:80 \
  dpage/pgadmin4
```

Again, to easily run this command, we can create a run-pgadmin.sh file with the docker command and turn into a executable file with:

```
chmod +x run-pgadmin.sh
```

Then, we can run the executable file with:

```
./run-pgadmin.sh
```

This will download the image if it's not already downloaded and run the container with the defined environment variables.

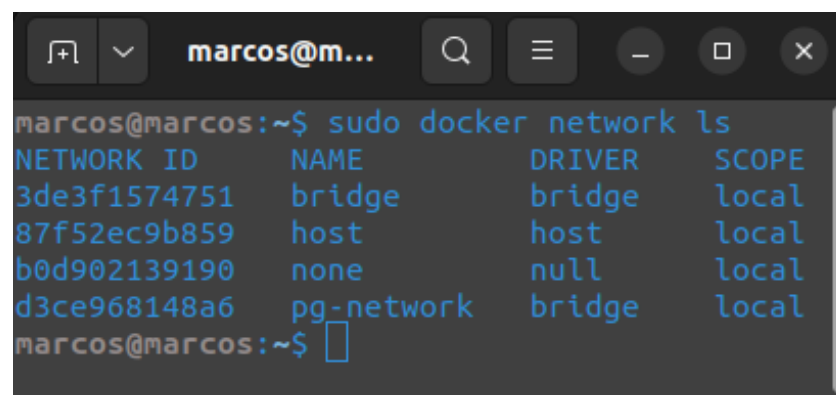
Now we need to create a connection between the postgres container and the Pgadmin container. To do that, we need to create a network between the containers. We can do that with the following docker command:

```
docker network create pg-network
```

where `pg-network` is the name give to our network that will connect the postgres to Pgadmin. We can list all the networks with the command:

```
docker network ls
```

The result is the following



```
marcos@marcos:~$ sudo docker network ls
NETWORK ID      NAME             DRIVER          SCOPE
3de3f1574751    bridge          bridge         local
87f52ec9b859    host            host           local
b0d902139190    none           null           local
d3ce968148a6    pg-network      bridge         local
marcos@marcos:~$
```

There are three networks by default and one that we created. The default networks are:

- Bridge Network: This is the default network that Docker containers connect to if no other network is specified.
- Host Network: When you use this network, your container shares the network stack of the host.
- None Network: This is a null network, used when we don't want the container to have networking capabilities. Containers attached to this network are completely isolated.

Now, let's remove all containers created before with `docker rm -f <container-id>` and run again the postgres and the pgadmin container in this new network with the following commands:

- For postgres, is exactly the same command as before with the addition of the `--network <network-name>` flag and `--name` flag to give a name to the container. This name is how PgAdmin will recognize the postgres container. Also, we change the flag `-it` to `-d` to run the container in the background.

```
docker run -d \
  -e POSTGRES_USER="marcosbenicio" \
  -e POSTGRES_PASSWORD="0102" \
  -e POSTGRES_DB="ny_taxi" \
  -v $(pwd)/taxi-trip-postgres:/var/lib/postgresql/data \
  -p 5432:5432 \
  --network=pg_network
  --name pg-database
  postgres:13
```

- For pgadmin, we specify again the same network `pg-network`. The name flag is less important here, because the connection is made from the postgres container to the pgadmin container.

```
docker run -d \
  -e PGADMIN_DEFAULT_EMAIL="marcosbenicio@admin.com" \
  -e PGADMIN_DEFAULT_PASSWORD="0102" \
  -p 8080:80 \
  --network=pg-network \
  --name pgadmin-interface \
  dpage/pgadmin4
```

Note that if the PostgreSQL container was started previously without the `POSTGRES_DB` environment variable, or if the database was deleted, it will not recreate the database or the table. In such cases, the database and the table must be created manually with the following SQL command:

```
CREATE DATABASE ny_taxi;
```

Again, to easily run this command, we can create a `postgres-pgadmin-connection.sh` file with both docker commands and turn into a executable file with:

```
chmod +x postgres-pgadmin-connection.sh
./postgres-pgadmin-connection.sh
```

Let's inspect our network with the command:

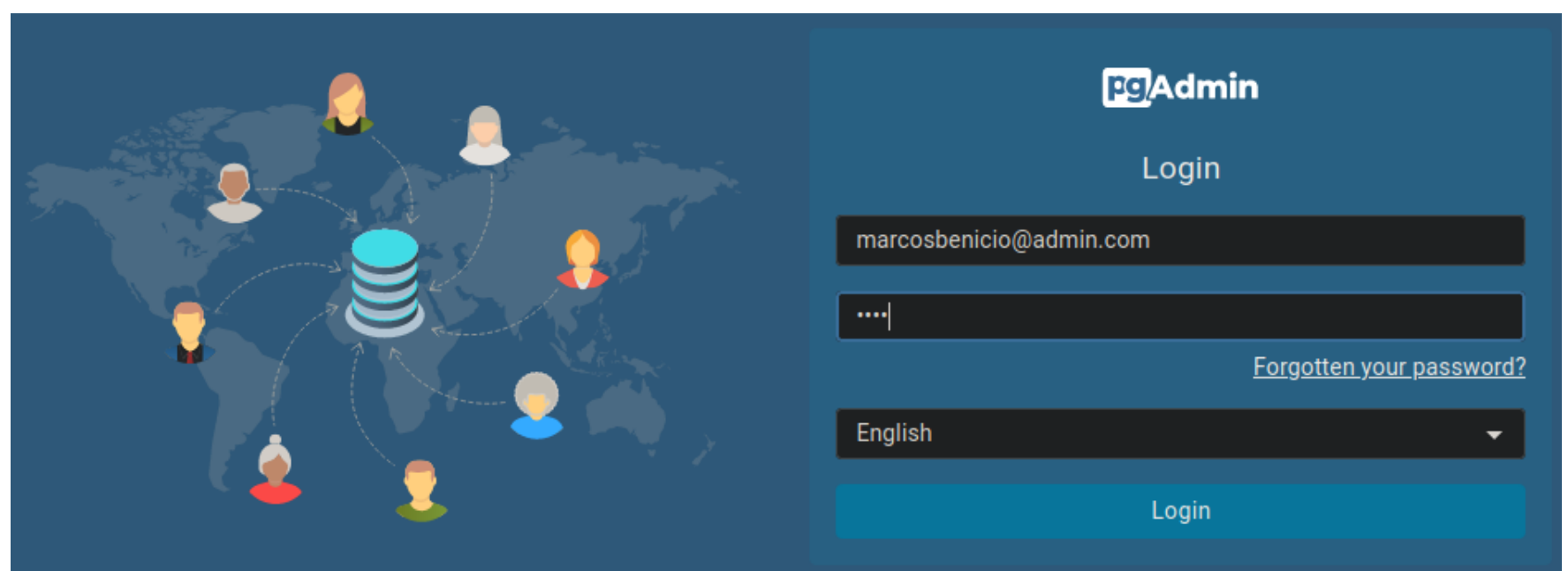
```
docker network inspect pg-network
```

The result is the following:

```
marcos@marcos:~$ sudo docker network inspect pg-network
[
  {
    "Name": "pg-network",
    "Id": "d3ce968148a6648262e589698e11562f543b0e2ac672d2c35d6ed519c4d6cf2c",
    "Created": "2024-01-27T13:03:06.234325179-03:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.20.0.0/16",
          "Gateway": "172.20.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "0b9abe85466d5e14ce5a291c774f884fbaa6724e0cba4da620a79fd0944270c1": {
        "Name": "pgadmin-interface",
        "EndpointID": "a77e7bd0b3ccadf294d91a580223e734130cc59f1d3538eedaa4d316d79cc382",
        "MacAddress": "02:42:ac:14:00:03",
        "IPv4Address": "172.20.0.3/16",
        "IPv6Address": ""
      },
      "91bbfaf24ac7858585d84fd83d250b01e1ad6c56b0e977cd22e7bc5b20d58ba4": {
        "Name": "pg-database",
        "EndpointID": "f7c083ae861f278ede57aa79101dfe59aea88aed47c052beab8e53974bcb933c",
        "MacAddress": "02:42:ac:14:00:02",
        "IPv4Address": "172.20.0.2/16",
        "IPv6Address": ""
      }
    },
    "Options": {},
    "Labels": {}
  }
]
marcos@marcos:~$
```

We can see that both the containers are connected to the network `pg-network`, as shown inside the containers variable.

Now we can access our localhost on port 8080 with `http://localhost:8080/` via browser:



After login, we can register a new server with host `pg-database` (the name of the postgres container) as show bellow. The username and password are the same as defined in the environment variables. The port is 5432, the default port of postgres.

The screenshot shows a 'Register - Server' dialog box with the 'Connection' tab selected. The fields are as follows:

- Host name/address: pg-database
- Port: 5432
- Maintenance database: postgres
- Username: marcosbenicio
- Kerberos authentication?: ☐
- Password: ....
- Save password?: ☐
- Role: (empty)
- Service: (empty)

At the bottom, there are three buttons: 'Close', 'Reset', and 'Save'.

Then, we can save and access the database.

## 2.4. Dockerizing the data ingestion

Another way to ingest the data into postgres is creating a python script. This way the process of download and ingest data into postgres can be automated. The script file is the following:

**ingest-data.py**

```
import os
import argparse
from sqlalchemy import create_engine
import pandas as pd
import gzip

def main(params):

    try:
        # Get the parameters
        user = params.user
        password = params.password
        host = params.host
        port = params.port
        database_name = params.database_name
        table_name = params.table_name
        url = params.url
        file_name = params.file_name
        file_extension = params.file_extension

        # Download csv file from url
        if os.system(f'wget {url} -O {file_name}') != 0:
            raise Exception(f"Failed to download file from {url}")

        # Check the file extension and process
        if file_extension == '.csv':
            df = pd.read_csv(file_name, low_memory=False)

        elif file_extension == '.gz':
            with gzip.open(file_name, 'rb') as f:
                df = pd.read_csv(f, low_memory=False)

        else:
            raise Exception(f"File extension {file_extension} not supported")

        # Create a connection to the database
        engine = create_engine(f'postgresql://{user}:{password}@{host}:{port}/{database_name}')

        # Create a table in database from the pandas dataframe
        df.to_sql(name=table_name, con=engine, if_exists="replace")
```



```

    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":

    parser = argparse.ArgumentParser(description='Ingest CSV data to postgres')

    parser.add_argument("--user", help= 'user name for postgres' )
    parser.add_argument("--password", help= 'password for postgres' )
    parser.add_argument("--host", help= 'host for postgres')
    parser.add_argument("--port", help= 'port for postgres')
    parser.add_argument("--database_name", help= 'database name for postgres')
    parser.add_argument("--url", help= 'url for file csv or .gz file to ingest')
    parser.add_argument("--table_name", help= 'Table name to pass data')
    parser.add_argument("--file_name", help= 'Name to save data')
    parser.add_argument("--file_extension", help= 'Extension of the file to ingest (csv or
.gz)')

    args = parser.parse_args()
    main(args)

```

---

Now to automatically download and ingest the data with the script create the docker file:

### Dockerfile

```

FROM python:3.9.1

RUN apt-get update && apt-get install -y wget
RUN pip install pandas sqlalchemy pycopg2 pyarrow

WORKDIR /app

# Copy the script directly into /app
COPY ["ingest-data.py", "./"]

# Execute ingest-data.py when the container starts
ENTRYPOINT [ "python", "ingest-data.py" ]

```

Then, build the docker image containing the python script with the following command:

```
docker build -t ingest-data:v01 .
```

Now that we build the docker image, we can run the container with the same network as the Postgres and Pgadmin containers. To do so, add the `--network <network-name>` flag to the docker run command with the defined variables from our script. The command can be written inside a executable file as:

### ingest-data-postgres.sh

```

URL1="https://github.com/DataTalksClub/nyc-tlc-data/releases/download/green/green_tripdata_2019-09.csv.gz"
URL2="https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv"

# Define parameters
USER="marcosbenicio"
PASSWORD="0102"
HOST="pg-database"
PORT="5432"
DATABASE_NAME="ny_taxi"
TABLE_NAME1="green_taxi_trip"
TABLE_NAME2="taxi_zone_lookup"
FILE_NAME1="green-tripdata-2019-09"
FILE_NAME2="taxi-zone-lookup"
FILE_EXTENSION1=".gz"
FILE_EXTENSION2=".csv"

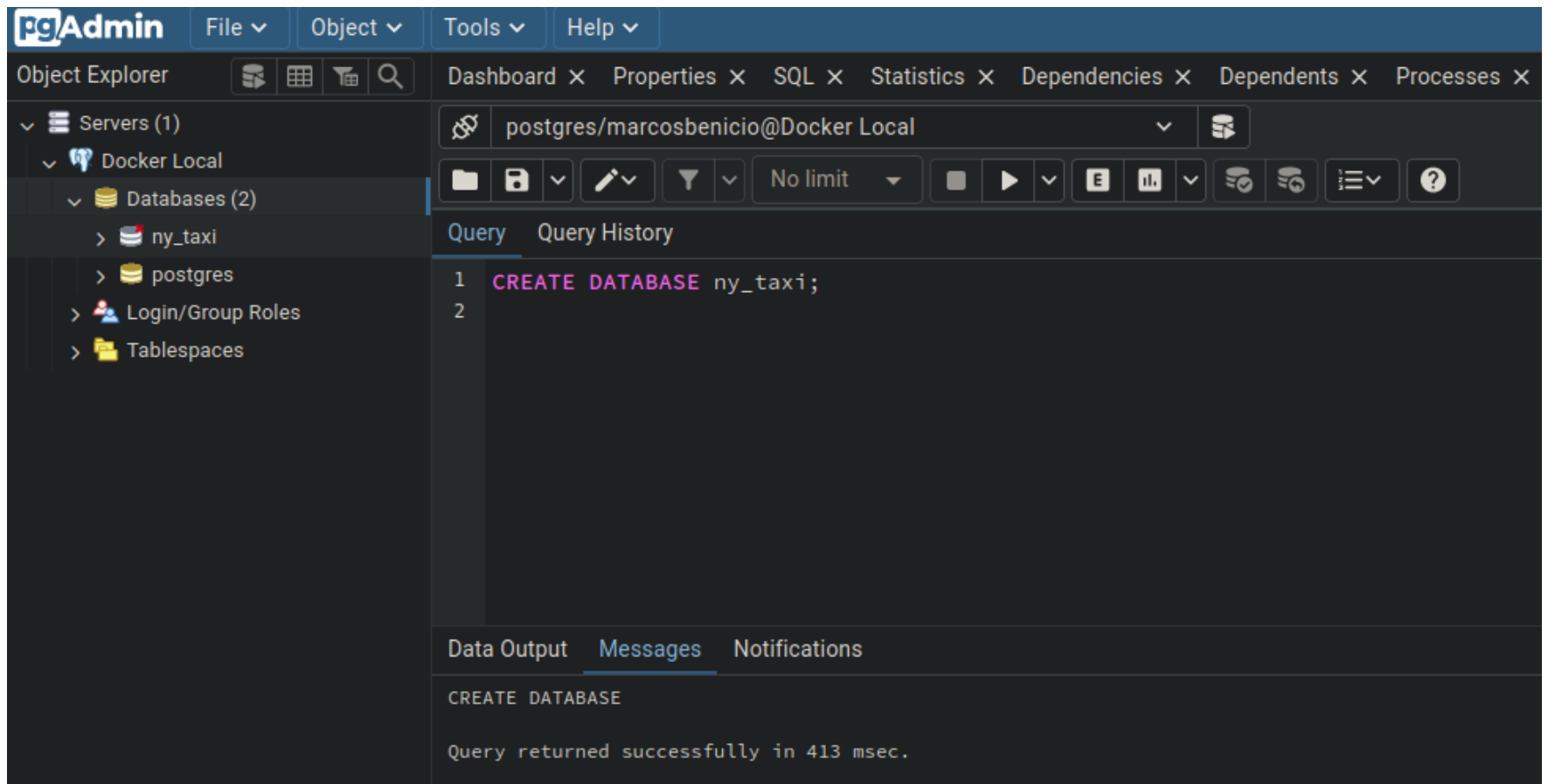
# Ingest first dataset
docker run -it --rm \
  --network pg-network \
  ingest-data:v01 \
    --user=${USER} \
    --password=${PASSWORD} \
    --host=${HOST} \
    --port=${PORT} \
    --database_name=${DATABASE_NAME} \
    --table_name=${TABLE_NAME1} \
    --url=${URL1} \
    --file_name=${FILE_NAME1} \
    --file_extension=${FILE_EXTENSION1}

```

```
# Ingest second dataset
docker run -it --rm \
  --network pg-network \
  ingest-data:v01 \
    --user=${USER} \
    --password=${PASSWORD} \
    --host=${HOST} \
    --port=${PORT} \
    --database_name=${DATABASE_NAME} \
    --table_name=${TABLE_NAME2} \
    --url=${URL2} \
    --file_name=${FILE_NAME2} \
    --file_extension=${FILE_EXTENSION2}
```

The `docker run` command create a container every time we run it, so is needed to add the `--rm` flag to remove the container after the execution. This is useful to avoid the creation of multiple containers.

If the table was dropped before, first recreate the database and table as in the following:



Finally, run the executable file with:

```
chmod +x ingest-data-postgres.sh
./ingest-data-postgres.sh
```

## 2.5. Running Postgres and pgAdmin with Docker-Compose

Instead of doing all the steps above, we can use docker-compose to run the container with a single file. With Compose, we use a YAML file to configure our application's services without the need of the `postgres-pgadmin-connection.sh` or the `ingest-data.sh`. The docker-compose files are the following:

### docker-compose.yml

```
services:
  pg-database:
    image: postgres:13
    environment:
      - POSTGRES_USER=marcosbenicio
      - POSTGRES_PASSWORD=0102
      - POSTGRES_DB=ny_taxi
    volumes: # mount a volume to persist the data
      - "/taxi-trip-postgres:/var/lib/postgresql/data:rw"
    ports: # default port for postgres
      - "5432:5432"

  pgadmin:
    image: dpage/pgadmin4
    environment:
      - PGADMIN_DEFAULT_EMAIL=marcosbenicio@admin.com
      - PGADMIN_DEFAULT_PASSWORD=0102
    ports:
      - "8080:80"
    networks: # connect to the same network as postgres
      - default
```



```
networks: # create a network to connect the containers
  default:
    external:
      name: pg-network
```

---

#### **docker-compose.ingest.yaml**

```
services:
  data-ingest-1:
    image: ingest-data:v01
    command: > # it's not necessary to use an "=" or ":" sign here to pass the variables
      --user marcosbenicio
      --password 0102
      --host pg-database
      --port 5432
      --database_name ny_taxi
      --table_name green_taxi_trip
      --url https://github.com/DataTalksClub/nyc-tlc-
data/releases/download/green/green_tripdata_2019-09.csv.gz
      --file_name green-tripdata-2019-09
      --file_extension .gz
    networks:
      - default

  data-ingest-2:
    image: ingest-data:v01
    command: >
      --user marcosbenicio
      --password 0102
      --host pg-database
      --port 5432
      --database_name ny_taxi
      --table_name taxi_zone_lookup
      --url https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv
      --file_name taxi-zone-lookup
      --file_extension .csv
    networks:
      - default

networks:
  default:
    external:
      name: pg-network
```

---

If we do not specify the network, docker-compose will create a new network for us for each docker-compose file. To run the docker-compose files, both to start the database service and ingest data, we need to connect to the same network as the postgres container. To do so, we need to create the network before running the docker-compose files if the network does not exist. To so do run the following command:

```
docker network create pg-network
```

Then, we can first run the `docker-compose.yaml` file to start our database services:

```
docker-compose up -d
```

and after started the containers, we can run the `docker-compose.ingest.yaml` file to ingest the data:

```
docker-compose -f docker-compose.ingest.yaml up
```

To stop the containers, we can type the following command:

```
docker-compose down
```

## 2.6. Homework: using SQL

- **Question 3. Count records**

How many taxi trips were totally made on September 18th 2019?

Tip: started and finished on 2019-09-18.

Remember that `lpep_pickup_datetime` and `lpep_dropoff_datetime` columns are in the format timestamp (date and hour+min+sec) and not in date.

- 15767
- **15612**
- 15859

- 89009

QueryQuery History

123456

```
SELECT COUNT(*)
FROM green_taxi_trip
WHERE lpep_pickup_datetime >= '2019-09-18 00:00:00'
AND lpep_dropoff_datetime < '2019-09-19 00:00:00';
```

Data OutputMessagesNotifications

count

bigint

1

15612

- Question 4. Largest trip for each day

Which was the pick up day with the largest trip distance Use the pick up time for your calculations.

- 2019-09-18
- 2019-09-16
- 2019-09-26
- 2019-09-21

QueryQuery History

12345

```
SELECT lpep_pickup_datetime, trip_distance
FROM green_taxi_trip
WHERE trip_distance = (SELECT MAX(trip_distance) FROM green_taxi_trip);
```

Data OutputMessagesNotifications

lpep\_pickup\_datetime

text

trip\_distance

double precision

1

2019-09-26 19:32:52

341.64

- Question 5. Three biggest pick up Boroughs

Consider lpep\_pickup\_datetime in '2019-09-18' and ignoring Borough has Unknown

Which were the 3 pick up Boroughs that had a sum of total\_amount superior to 50000?

- "Brooklyn" "Manhattan" "Queens"
- "Bronx" "Brooklyn" "Manhattan"
- "Bronx" "Manhattan" "Queens"
- "Brooklyn" "Queens" "Staten Island"

QueryQuery History

12

```
1 SELECT DATE(lpep_pickup_datetime), SUM(total_amount) AS sum_total_amount, taxi_zone_lookup."Borough"
2 FROM green_taxi_trip
3 INNER JOIN taxi_zone_lookup
4 ON green_taxi_trip."PULocationID" = taxi_zone_lookup."LocationID"
5 WHERE DATE(lpep_pickup_datetime) = '2019-09-18'
6 GROUP BY taxi_zone_lookup."Borough", DATE(lpep_pickup_datetime)
7 HAVING SUM(total_amount) >= 50000;
8
9
10
11
12
```

Data OutputMessagesNotifications

	date date	sum_total_amount double precision	Borough text
1	2019-09-18	96333.23999999947	Brooklyn
2	2019-09-18	92271.299999999843	Manhattan
3	2019-09-18	78671.709999999892	Queens

• Question 6. Largest tip

For the passengers picked up in September 2019 in the zone name Astoria which was the drop off zone that had the largest tip? We want the name of the zone, not the id.

Note: it's not a typo, it's tip , not trip

- Central Park
- Jamaica
- JFK Airport
- Long Island City/Queens Plaza

QueryQuery History

14

```
1 SELECT
2     pickup_zone_lookup."Zone" AS pickup_zone,
3     dropoff_zone_lookup."Zone" AS dropoff_zone,
4     MAX(tip_amount) AS largest_tip
5 FROM green_taxi_trip
6 INNER JOIN taxi_zone_lookup AS pickup_zone_lookup
7     ON green_taxi_trip."PULocationID" = pickup_zone_lookup."LocationID"
8 INNER JOIN taxi_zone_lookup AS dropoff_zone_lookup
9     ON green_taxi_trip."DOLocationID" = dropoff_zone_lookup."LocationID"
10 WHERE DATE(lpep_pickup_datetime) BETWEEN '2019-09-01' AND '2019-09-30'
11     AND pickup_zone_lookup."Zone" = 'Astoria'
12 GROUP BY lpep_pickup_datetime, pickup_zone_lookup."Zone", dropoff_zone_lookup."Zone"
13 ORDER BY largest_tip DESC
14 LIMIT 1;
```

Data OutputMessagesNotifications

	pickup_zone text	dropoff_zone text	largest_tip double precision
1	Astoria	JFK Airport	62.31

### 3. Terraform

In this section homework we'll prepare the environment by creating resources in GCP with Terraform.

In your VM on GCP/Laptop/GitHub Codespace install Terraform. Copy the files from the course repo [here](#) to your VM/Laptop/GitHub Codespace.

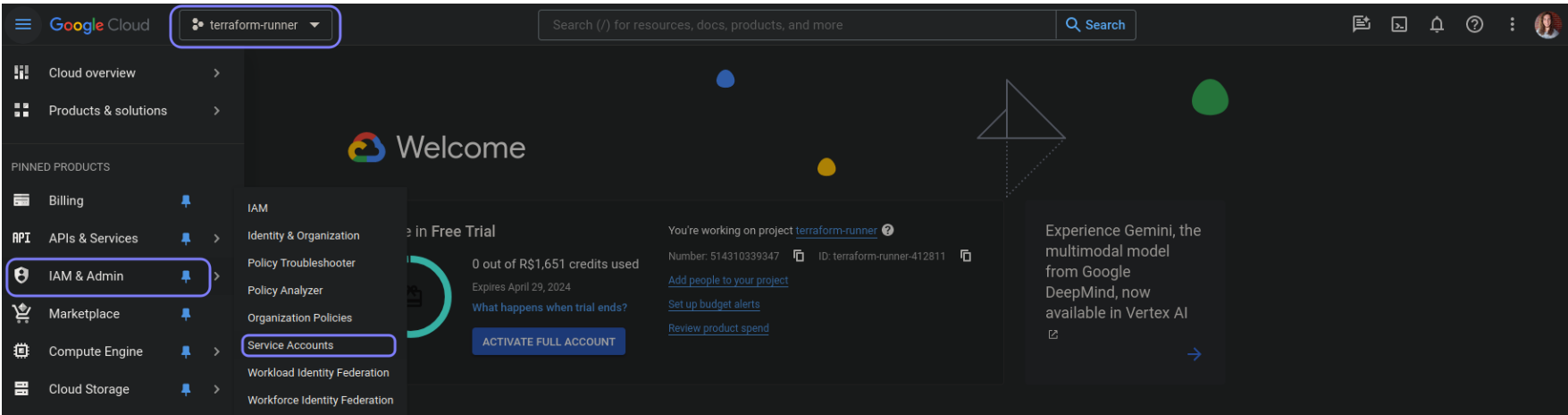
Modify the files as necessary to create a GCP Bucket and Big Query Dataset.

Terraform is an infrastructure as code (IaC) tool used for building, changing, and versioning infrastructure. It manages the infrastructure itself (servers, databases, networks, etc.) in cloud providers (like AWS, Azure, GCP) and on-premises. They use a special program language called HashiCorp Configuration Language (HCL) to define the infrastructure. It is used to manage the lifecycle of infrastructure resources, including creation, modification, and deletion. Terraform operates at the infrastructure level, managing the platform on which applications run.

Terraform is a infrastructure management, while Docker is for deployment and management.

### 3.1. Creating and configuring a project on Google Cloud Platform (GCP)

To use terraform, we first need to create a project in GCP. To do so, we need to go to the [GCP console](#) and create a new project. After created the project, we can create our service. Go to `IAM & Admin > Service Accounts` and create a new service account by clicking in `+ CREATE SERVICE ACCOUNT`.



The following window will appear to create the service account. Give a name to the service account and click on create:

←

Create service account

1

Service account details

Service account name

terraform-runner

Display name for this service account

Service account ID \*

terraform-runner

✕ ↺

Email address:

terraform-runner@terraform-runner-412811.iam.gserviceaccount.com

Service account description

Testing terraform on GCP

Describe what this service account will do

CREATE AND CONTINUE

2

Grant this service account access to project (optional)

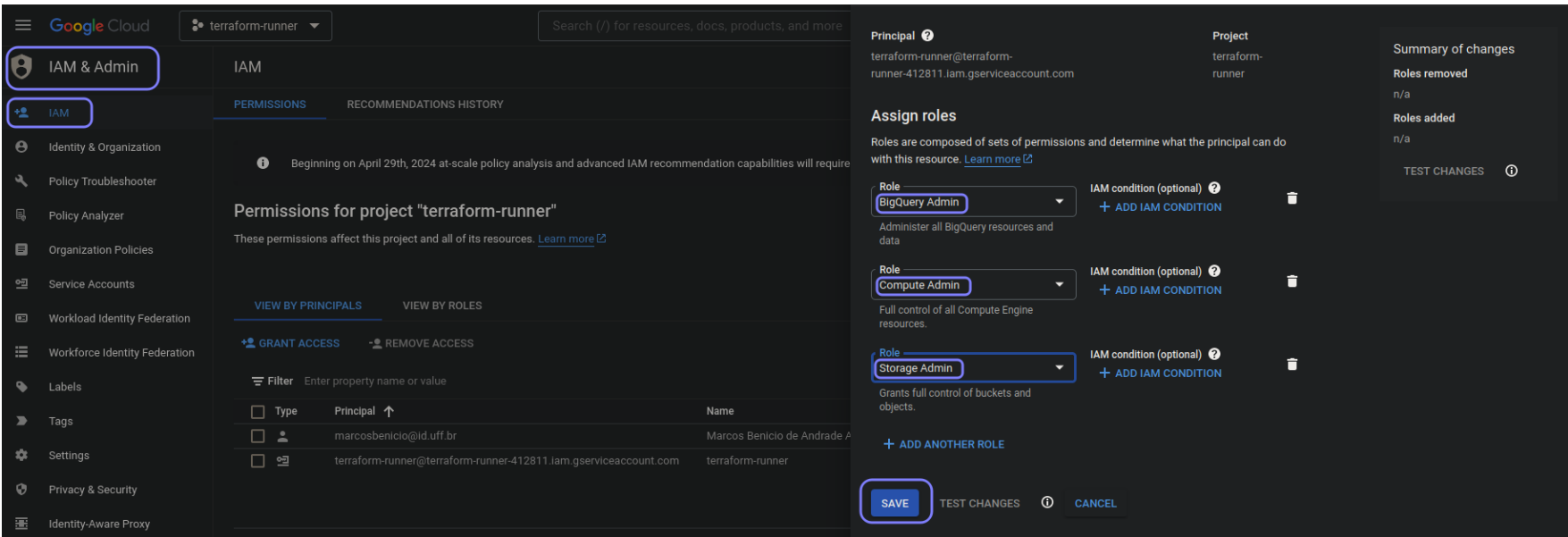
3

Grant users access to this service account (optional)

DONE

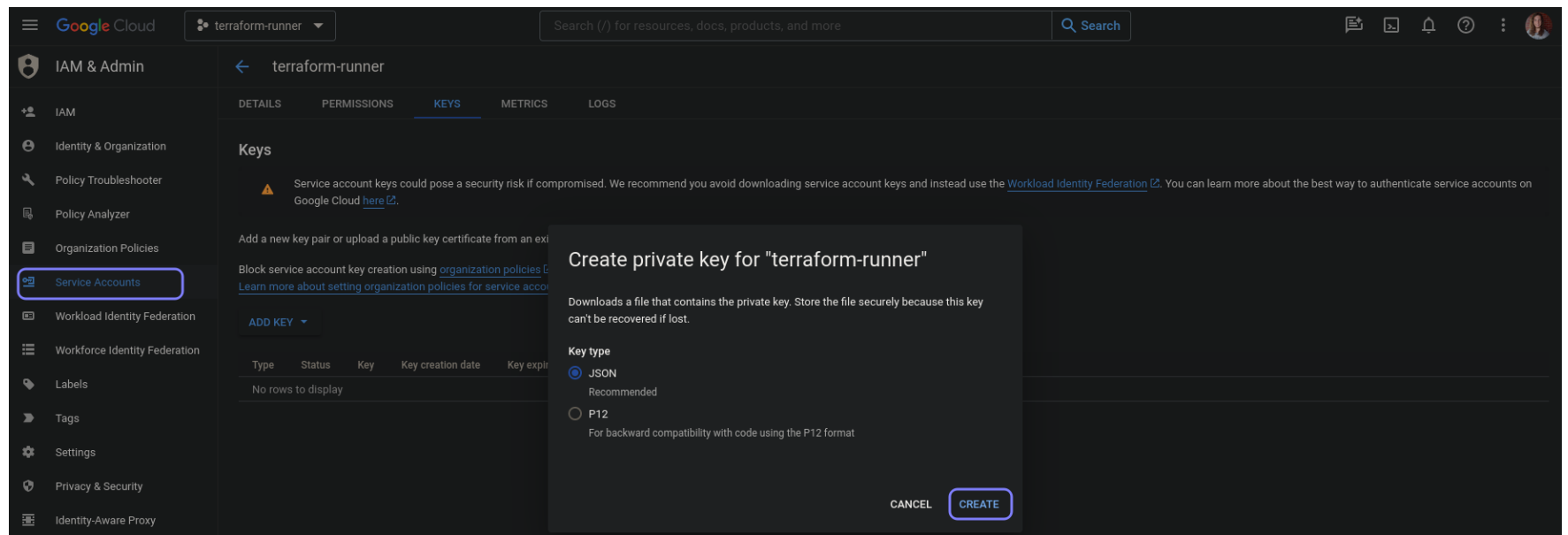
CANCEL

We can grant the service account access to the project by assign roles on the optional step show in the image. If we forgot to assign roles, we can do it later by going to the left menu on `IAM` and clicking on the edit principle button (the pencil icon):



To access from our local machine to the GCP project, we need to create a key for the service account. To do so, go to `Service Accounts > Actions` and select `Manage Keys` to generate ssh keys. Then, click on `ADD KEY > Create new key` and select

the JSON format.



This will download a JSON file with the credentials for the service account. We can rename the file to `terraform-runner-credentials.json` and move it to the same directory as the terraform files.

## 3.2. Configure and Deploy with Terraform

To configure the [GCP provider](#) in Terraform, add the following code to the `main.tf` file:

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
      version = "5.14.0"
    }
  }
}

provider "google" {
  credentials = "../keys/credentials.json" # Path to the JSON file credentials.
  project = "terraform-runner-412811" # ID of the Google Cloud project.
  region = "southamerica-east1"
}
```

If necessary, we can automatically format the code with the command:

```
terraform fmt
```

Now, we need to take the project ID from the GCP console and replace `my-project-id`. For this project the ID is `terraform-runner-412811`. Also, we can change the region to somewhere more close to our country if necessary. Because I'm from Brazil is reasonable to change to `southamerica-east1` instead of the default `us-central1`. The `terraform` block defines the required providers for your Terraform project, while `provider "google"` block configures the Google Cloud provider with specific settings like credentials, project ID, and the default region for resources.

Now, we continue working on the `main.tf` file by adding the `google_storage_bucket` resource block. To do so we can check a example for [GCP cloud storage bucket](#). Is only needed to change the name of the bucket and the location. The final code is the following:

```
resource "google_storage_bucket" "demo-bucket" {
  name      = "terraform-runner-412811-bucket" # needs to be unique name across all GCP
  location = "southamerica-east1"
  force_destroy = true

  lifecycle_rule {
    condition {
      age = 1 # In days
    }
    action {
      type = "AbortIncompleteMultipartUpload"
    }
  }
}
```

The `lifecycle_rule` block is used to define a lifecycle rule for the bucket. In this case, we define a rule to abort incomplete multipart uploads after 1 day. This is useful to avoid unnecessary charges for incomplete uploads.

One last block that we need is the `google_bigquery_dataset` resource block. To do so we can check a example for [GCP BigQuery dataset](#). Is only needed to change the name of the dataset and the location. The final code is the following:

```
resource "google_bigquery_dataset" "demo_dataset" {
  dataset_id = "demo_dataset"
  location = "southamerica-east1"
}
```

The `dataset_id` is the name of the dataset. The `location` is the location where the dataset will be created. The default location is `US`. The location can't be changed after the dataset is created.

We construct a simple terraform file just to create a bucket to deploy a dataset into GCP BigQuery. Before running the code, we can create another file named `variable.tf` to define variables used in the `main.tf` file. This is useful to avoid hardcoding values in the code.

#### `variable.tf`

```
variable "project" {
    description = "default project name that Terraform will use"
    default = "terraform-runner-412811"
}

variable "region" {
    description = "Sets the default region for the infrastructure deployment"
    default = "southamerica-east1"
}

variable "location" {
    description = "Indicates the location for the resources. Often similar to or the same as
the region."
    default = "southamerica-east1"
}

variable "bq_dataset_name" {
    description = "Names the BigQuery Dataset"
    default = "demo_dataset"
}

variable "gcs_bucket_name" {
    description = "Name of a Google Cloud Storage (GCS) bucket."
    default = "terraform-runner-412811-bucket"
}

variable "gcs_storage_class" {
    description = "Determines the storage class of the GCS bucket"
    default = "STANDARD"           # common values: STANDARD, NEARLINE, COLDLINE
}
```

---

Each variable has a name, a type, and a default value.

#### `main.tf`

```
terraform {
    required_providers {
        google = {
            source = "hashicorp/google"
            version = "5.14.0"
        }
    }
}

provider "google" {
    credentials = file(var.credentials) # load content of JSON file credentials.
    project     = var.project           # ID of the Google Cloud project.
    region      = var.region            # Region where resources will be deployed.
}

resource "google_storage_bucket" "demo-bucket" {
    name          = var.gcs_bucket_name # needs to be unique name across all GCP
    location      = var.location
    force_destroy = true

    lifecycle_rule {
        condition {
            age = 1 # In days
        }
        action {
            type = "AbortIncompleteMultipartUpload"
        }
    }
}

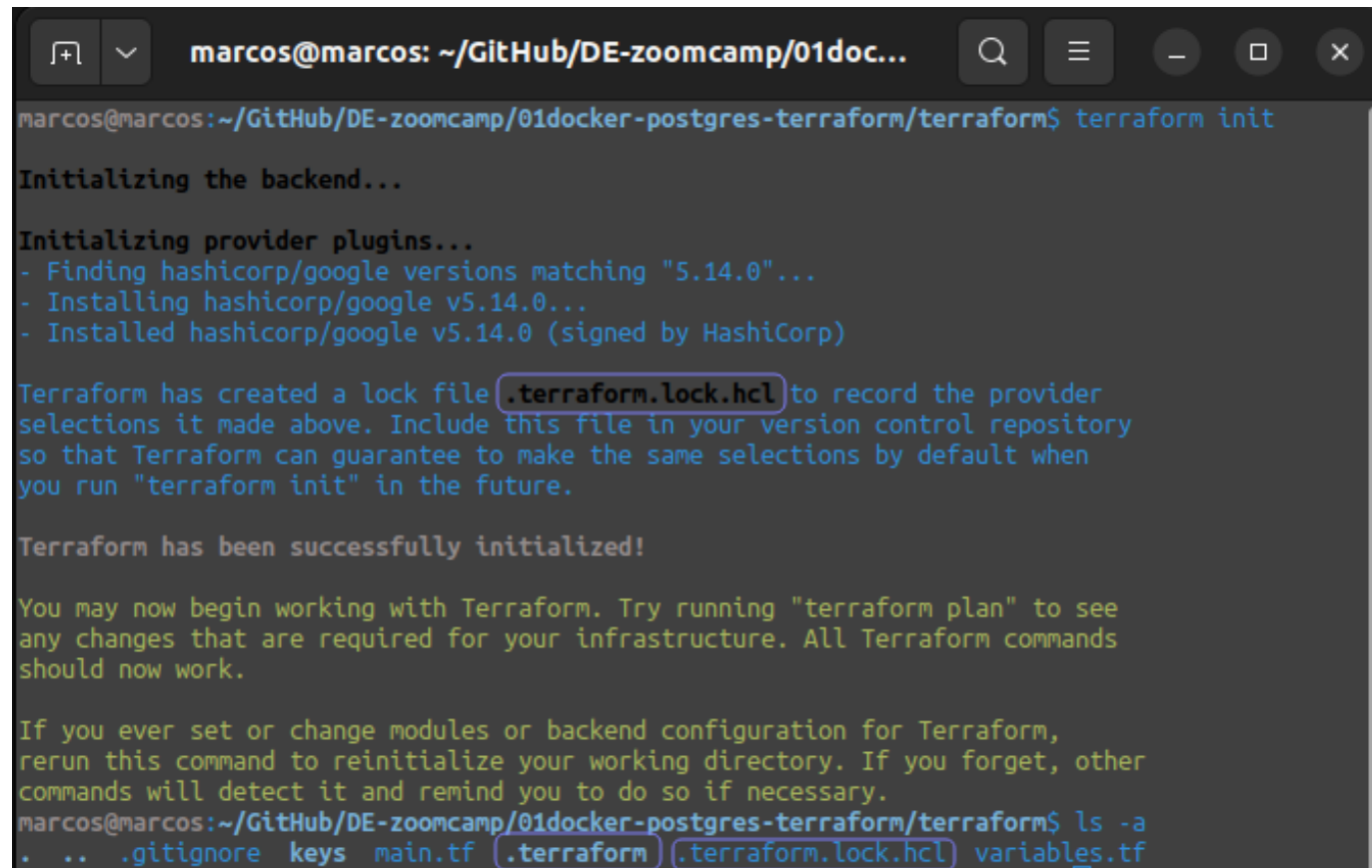
resource "google_bigquery_dataset" "demo_dataset" {
    dataset_id = var.bq_dataset_name
    location   = var.location
}
```



Now, we can run the terraform commands to initialize the project, plan the deployment, and apply the changes. To initialize the project, run the following command:

```
terraform init
```

this will create a lock file `.terraform.lock.hcl` and a hidden directory `.terraform`. The lock file is used to track the versions of the providers used in the project. The hidden directory contains the plugins and modules used in the project.

A terminal window titled 'marcos@marcos: ~/GitHub/DE-zoomcamp/01docker-postgres-terraform/terraform\$' shows the execution of 'terraform init'. The output includes 'Initializing the backend...', 'Initializing provider plugins...', a list of actions for finding and installing hashicorp/google v5.14.0, and a confirmation that the lock file '.terraform.lock.hcl' has been created. It concludes with 'Terraform has been successfully initialized!' and instructions to run 'terraform plan'. A final 'ls -a' command shows the directory contents, including the newly created '.terraform' directory and '.terraform.lock.hcl' file.

```
marcos@marcos:~/GitHub/DE-zoomcamp/01docker-postgres-terraform/terraform$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/google versions matching "5.14.0"...
- Installing hashicorp/google v5.14.0...
- Installed hashicorp/google v5.14.0 (signed by HashiCorp)

Terraform has created a lock file ".terraform.lock.hcl" to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
marcos@marcos:~/GitHub/DE-zoomcamp/01docker-postgres-terraform/terraform$ ls -a
.  ..  .gitignore  keys  main.tf  .terraform  .terraform.lock.hcl  variables.tf
```

To display what will be created we can run the following command:

```
terraform plan
```

We can see that the plan will create a bucket and a dataset in GCP:

```
marcos@marcos: ~/GitHub/DE-zoomcamp/01docker-postgres-terraform/terraform
marcos@marcos:~/GitHub/DE-zoomcamp/01docker-postgres-terraform/terraform$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# google_bigquery_dataset.demo_dataset will be created
+ resource "google_bigquery_dataset" "demo_dataset" {
+   creation_time           = (known after apply)
+   dataset_id              = "demo_dataset"
+   default_collation       = (known after apply)
+   delete_contents_on_destroy = false
+   effective_labels        = (known after apply)
+   etag                    = (known after apply)
+   id                      = (known after apply)
+   is_case_insensitive      = (known after apply)
+   last_modified_time       = (known after apply)
+   location                 = "southamerica-east1"
+   max_time_travel_hours    = (known after apply)
+   project                  = "terraform-runner-412811"
+   self_link                = (known after apply)
+   storage_billing_model    = (known after apply)
+   terraform_labels         = (known after apply)
+ }

# google_storage_bucket.demo-bucket will be created
+ resource "google_storage_bucket" "demo-bucket" {
+   effective_labels        = (known after apply)
+   force_destroy          = true
+   id                     = (known after apply)
+   location                = "SOUTHAMERICA-EAST1"
+   name                   = "terraform-runner-412811-bucket"
+   project                 = (known after apply)
+   public_access_prevention = (known after apply)
+   rpo                    = (known after apply)
+   self_link              = (known after apply)
+   storage_class           = "STANDARD"
+   terraform_labels        = (known after apply)
+   uniform_bucket_level_access = (known after apply)
+   url                     = (known after apply)

+   lifecycle_rule {
+     action {
+       type = "AbortIncompleteMultipartUpload"
+     }
+     condition {
+       age = 1
+       matches_prefix = []
+       matches_storage_class = []
+       matches_suffix = []
+       with_state = (known after apply)
+     }
+   }
+ }

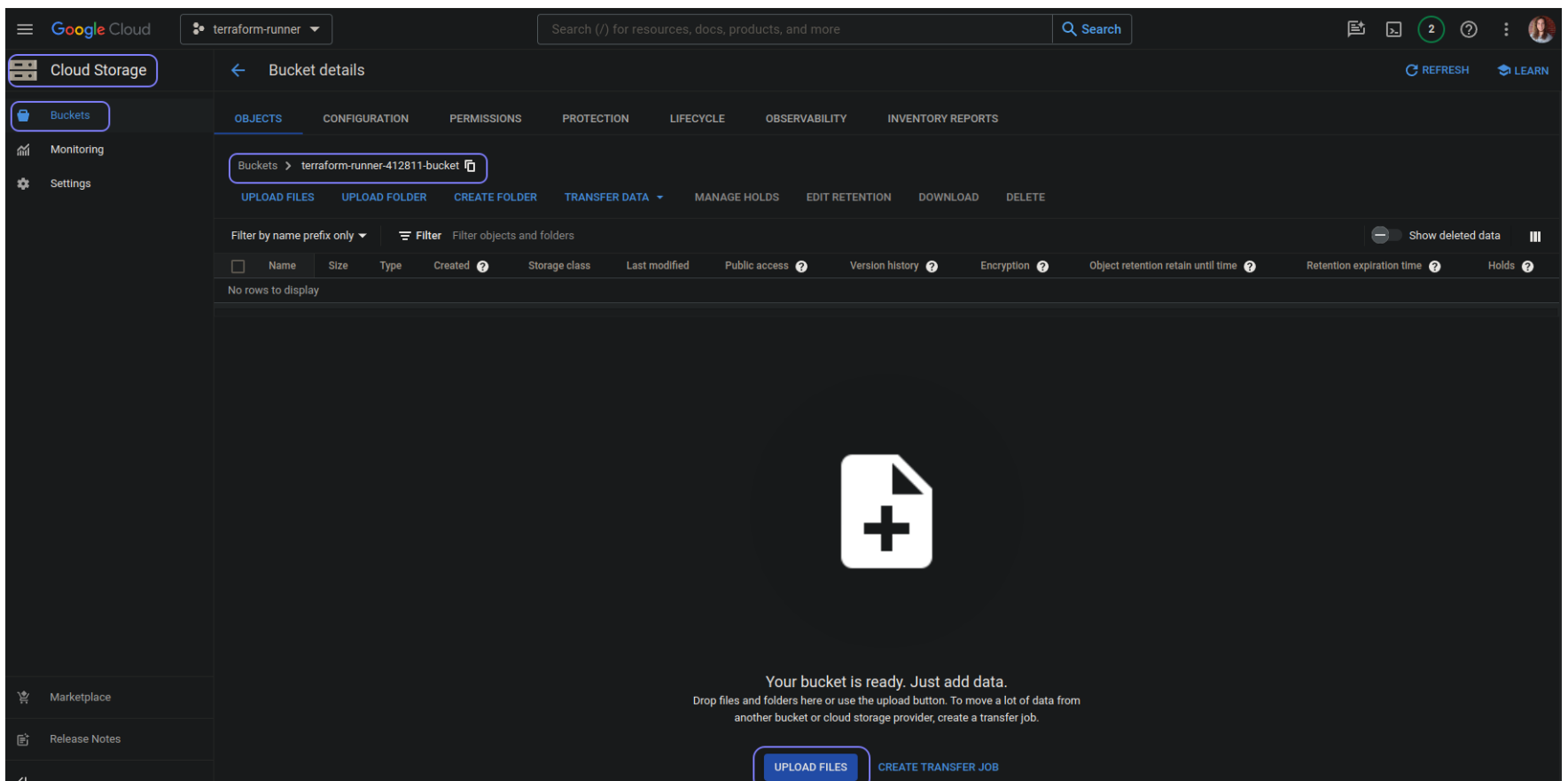
Plan: 2 to add, 0 to change, 0 to destroy.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
marcos@marcos:~/GitHub/DE-zoomcamp/01docker-postgres-terraform/terraform$
```

And finally, we can deploy the resources with the following command to create what was planned. If an Error 403 is returned, is need to enable the BigQuery API in GCP console by going into `APIs & Services > Enabled APIs & Services` and enabling `BigQuery API` .

terraform apply

After the deployment, we can check the resources created in GCP console and upload our dataset to the bucket.



To destroy the resources created, we can run the following command:

terraform destroy

### 3.3. Homework: Terraform

- **Question 7. Creating Resources**

After updating the main.tf and variable.tf files run:

```
terraform apply
```

Paste the output of this command:

Plan: 2 to add, 0 to change, 0 to destroy.

```
Do you want to perform these actions?  
  Terraform will perform the actions described above.  
  Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
google_bigquery_dataset.demo_dataset: Creating...  
google_storage_bucket.demo-bucket: Creating...  
google_storage_bucket.demo-bucket: Creation complete after 2s [id=terraform-runner-412811-bucket]  
google_bigquery_dataset.demo_dataset: Creation complete after 2s [id=projects/terraform-runner-412811/datasets/demo_dataset]
```

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```