Outline

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
```

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('posgresql://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:

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:

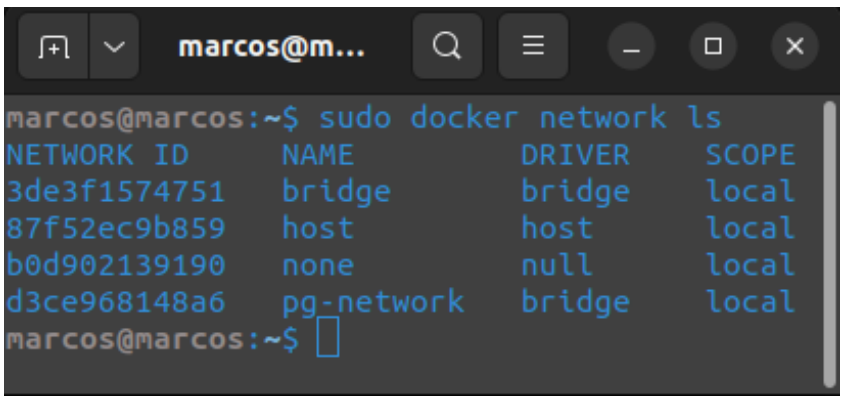
```
chmod +x run-pgadmin.sh
```
Then, we can run the executable file with:

```
./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



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;
```

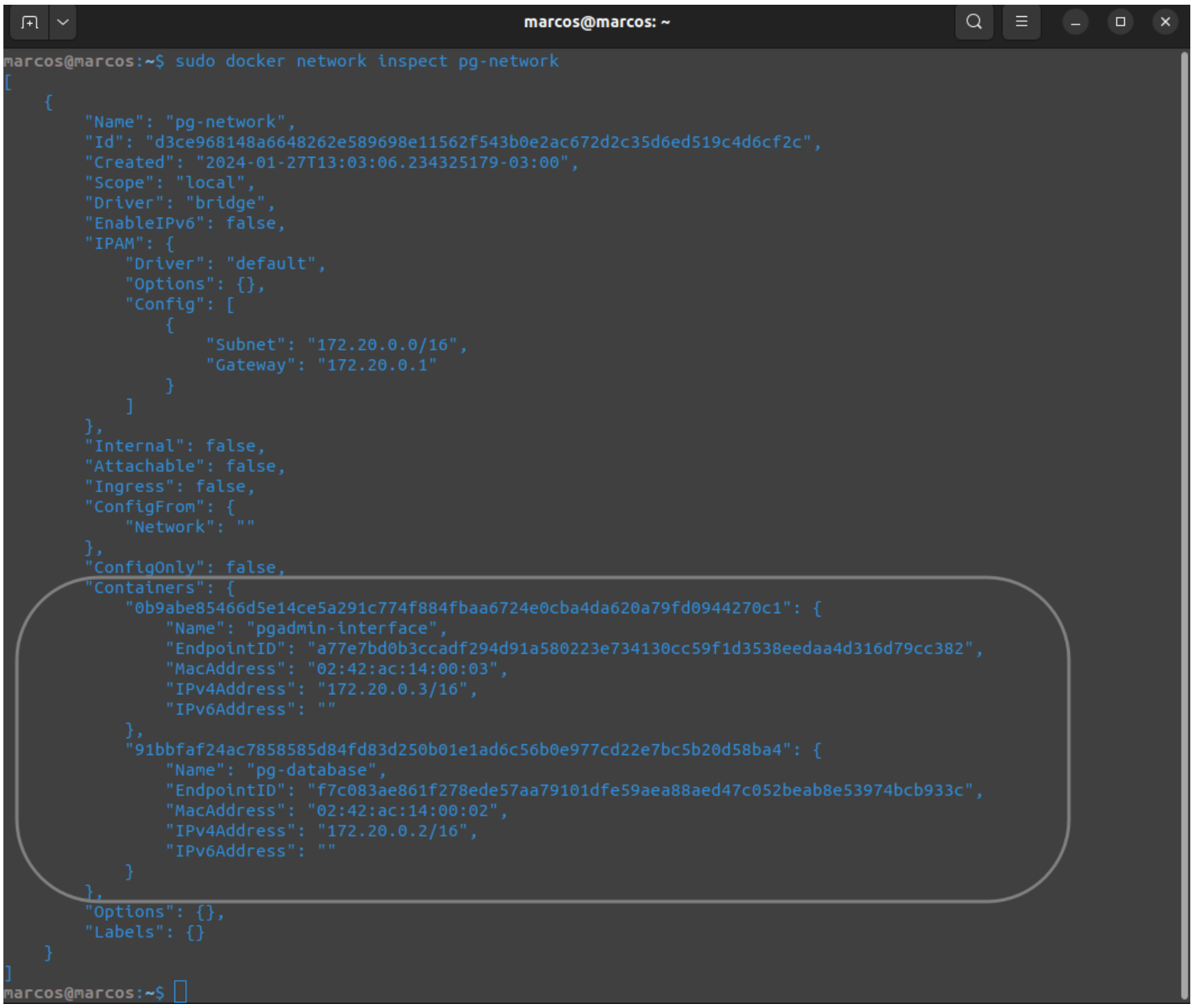
Again, to easily run this command, we can create a `postgres-pgadmin-connection.sh` file with both docker commands and turn into a executable file with:

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

Let's inspect our network with the command:

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

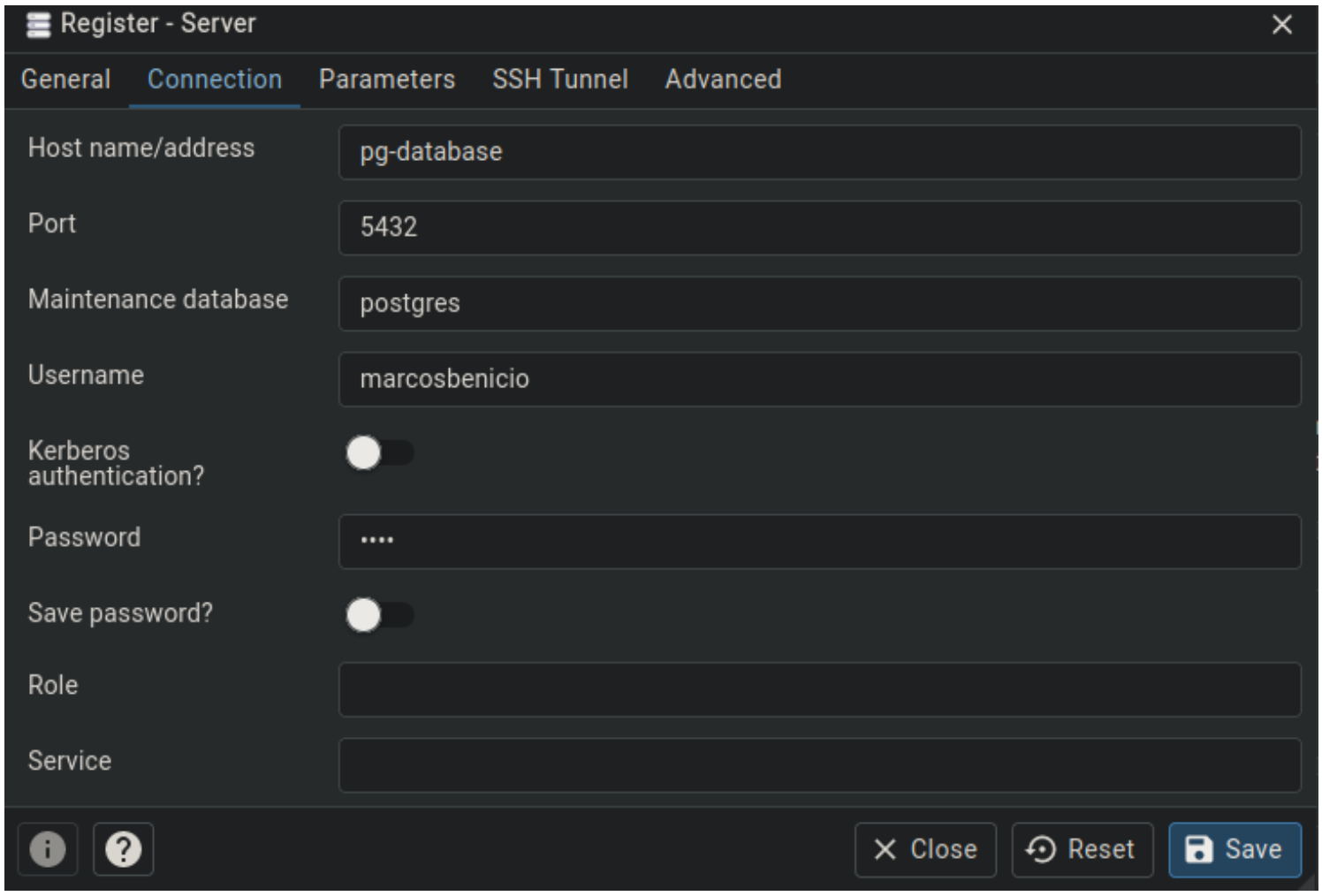
The result is the following:



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.



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 psycopg2 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:

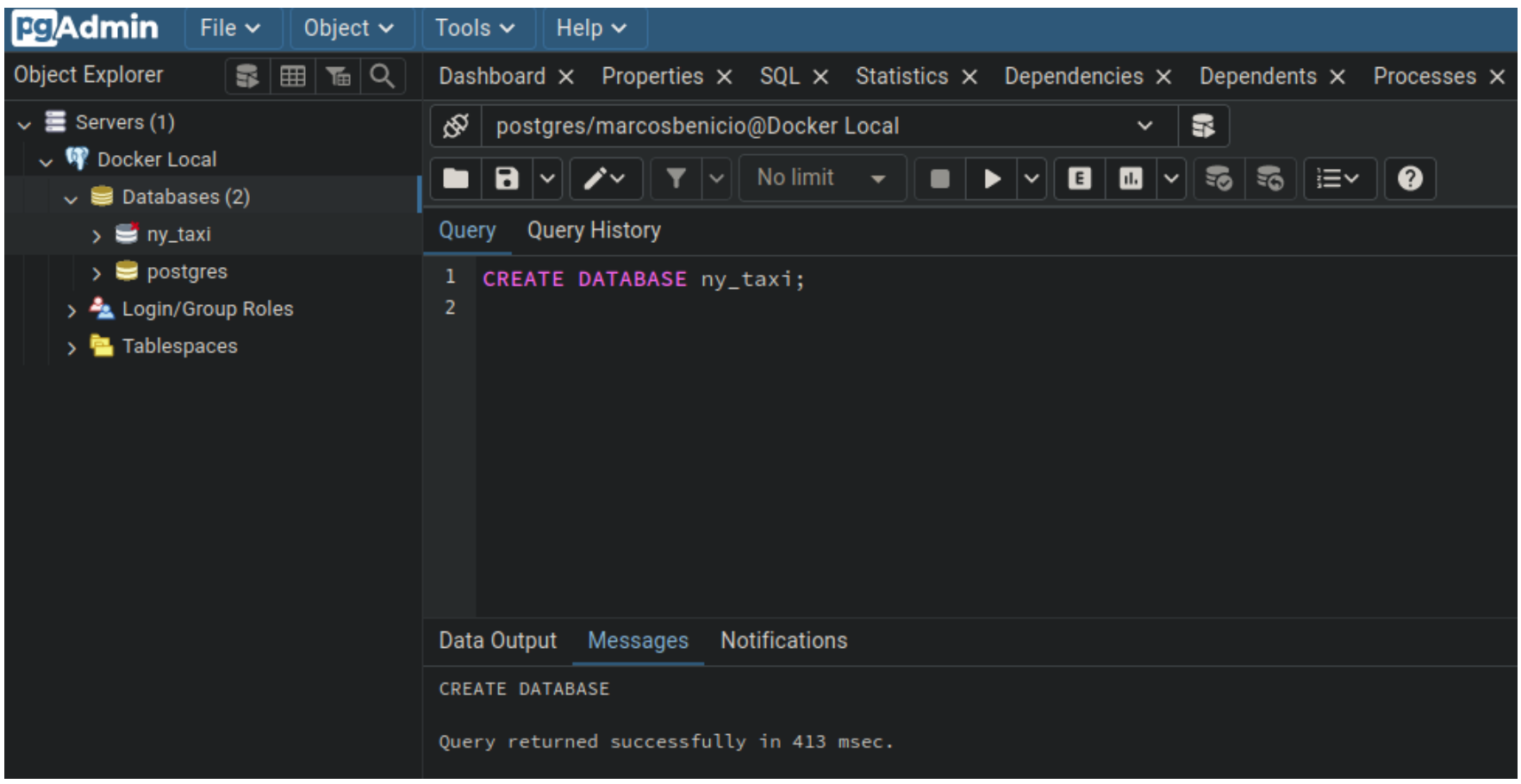
ingest-data-postgres.sh

```
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.yaml

```
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**

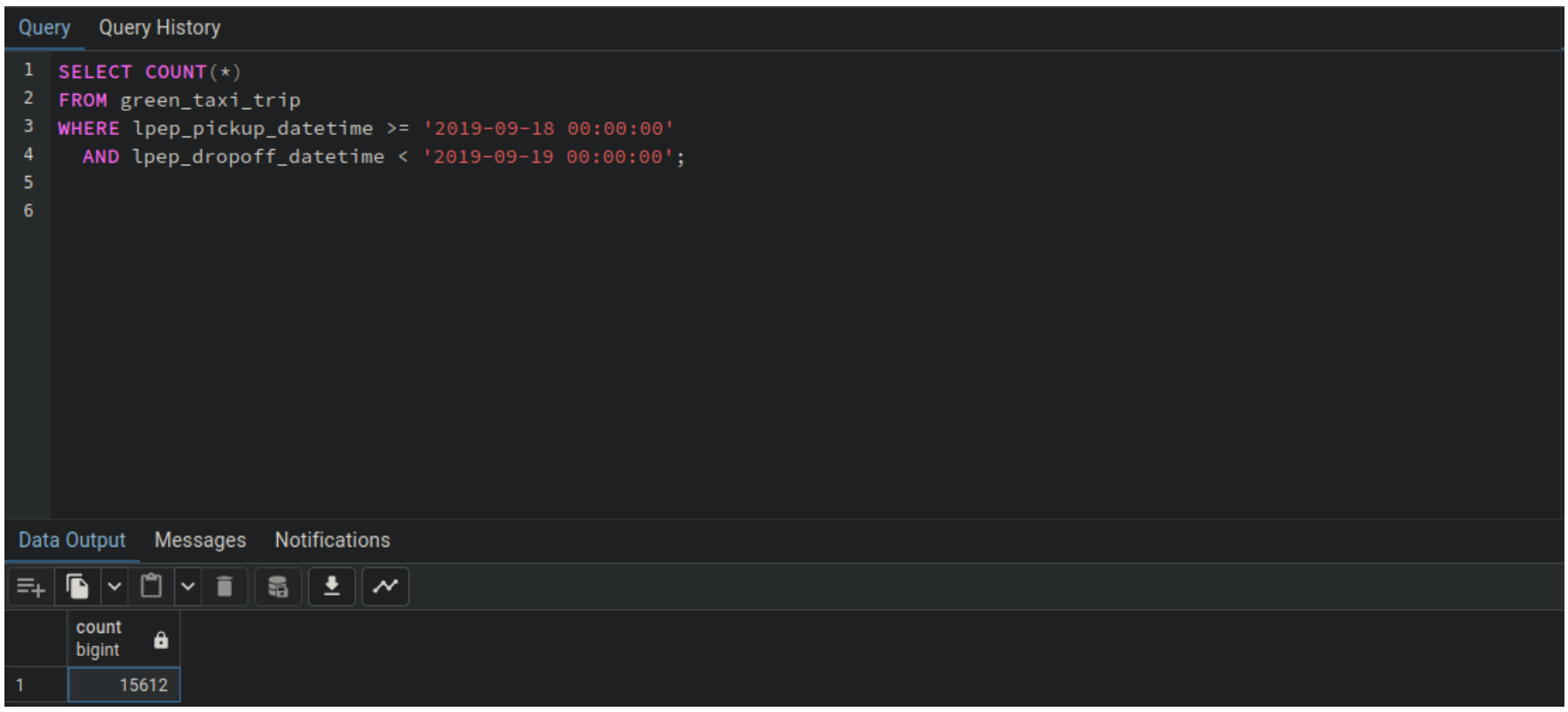
How many taxi trips were totally made on September 18th 2019?

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



- **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

- **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"

Query | Query History

```sql
SELECT DATE(lpep_pickup_datetime), SUM(total_amount) AS sum_total_amount, taxi_zone_lookup."Borough"
FROM green_taxi_trip
INNER JOIN taxi_zone_lookup
ON green_taxi_trip."PULocationID" = taxi_zone_lookup."LocationID"
WHERE DATE(lpep_pickup_datetime) = '2019-09-18'
GROUP BY taxi_zone_lookup."Borough", DATE(lpep_pickup_datetime)
HAVING SUM(total_amount) >= 50000;
```

Data Output | Messages | Notifications

	date date	sum_total_amount double precision	Borough text
1	2019-09-18	96333.23999999947	Brooklyn
2	2019-09-18	92271.29999999843	Manhattan
3	2019-09-18	78671.70999999892	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

Query | Query History

```sql
SELECT
    pickup_zone_lookup."Zone" AS pickup_zone,
    dropoff_zone_lookup."Zone" AS dropoff_zone,
    MAX(tip_amount) AS largest_tip
FROM green_taxi_trip
INNER JOIN taxi_zone_lookup AS pickup_zone_lookup
    ON green_taxi_trip."PULocationID" = pickup_zone_lookup."LocationID"
INNER JOIN taxi_zone_lookup AS dropoff_zone_lookup
    ON green_taxi_trip."DOLocationID" = dropoff_zone_lookup."LocationID"
WHERE DATE(lpep_pickup_datetime) BETWEEN '2019-09-01' AND '2019-09-30'
    AND pickup_zone_lookup."Zone" = 'Astoria'
GROUP BY lpep_pickup_datetime, pickup_zone_lookup."Zone", dropoff_zone_lookup."Zone"
ORDER BY largest_tip DESC
LIMIT 1;
```

Data Output | Messages | Notifications

	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.

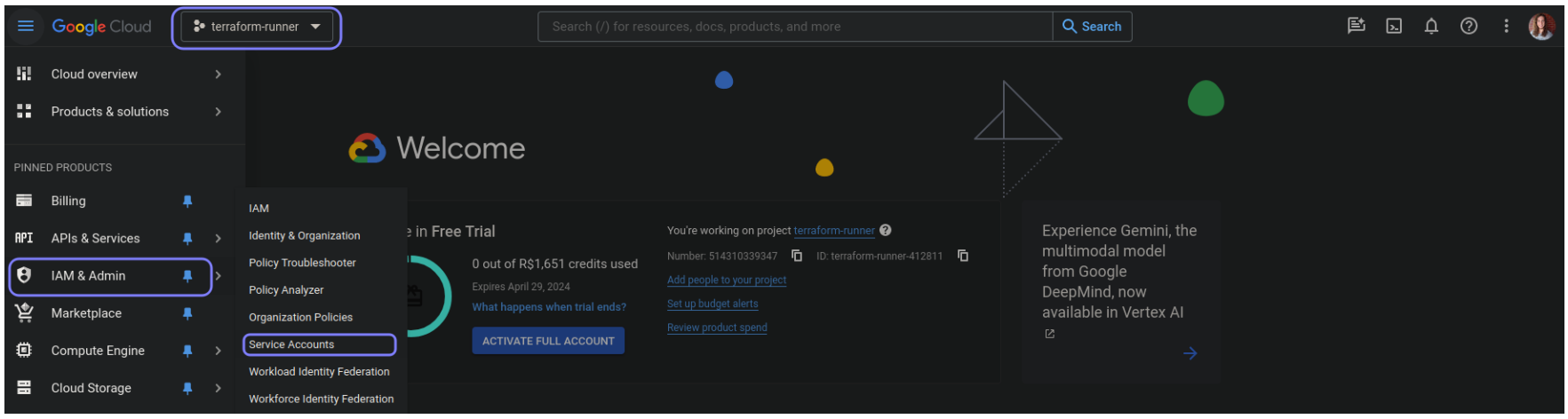
Modify the files as necessary to create a GCP Bucket and Big Query Dataset.

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.

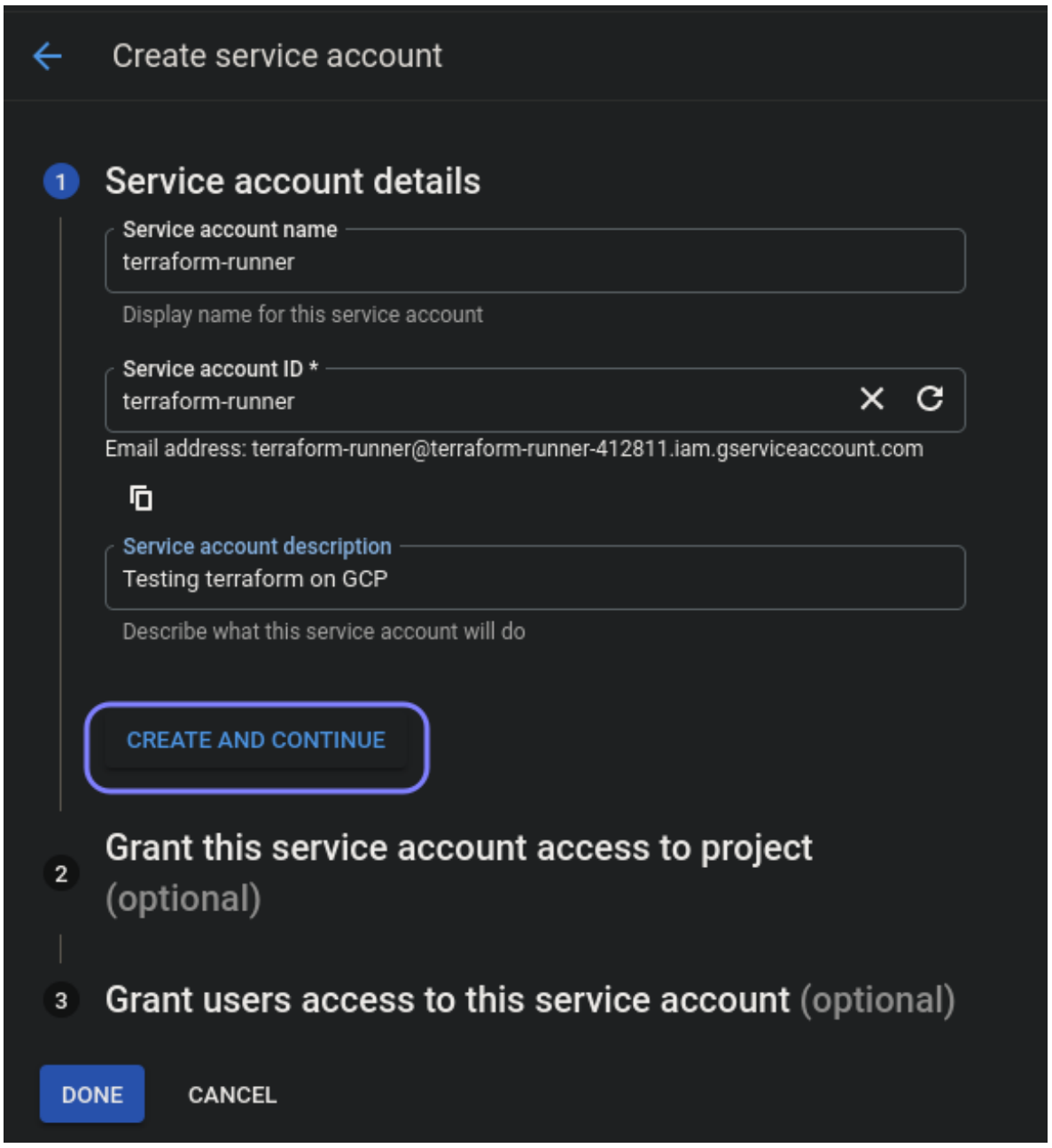
Terraform is a infrastructure management, while Docker is for deployment and management.

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

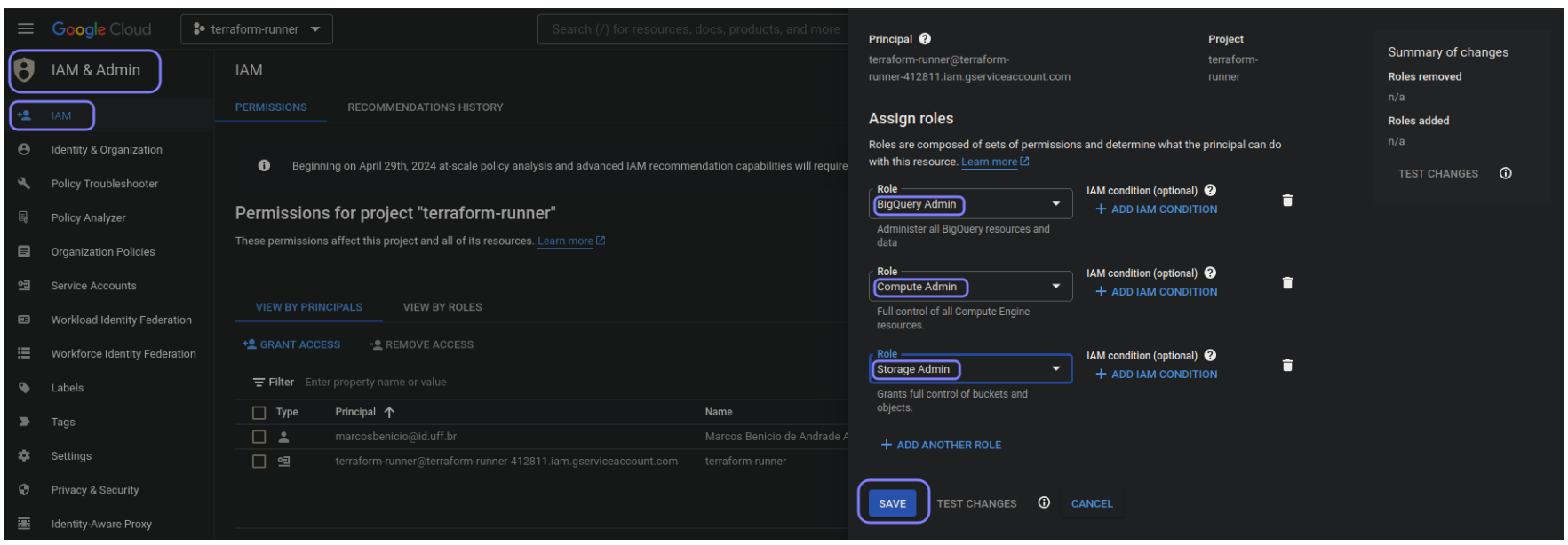
To use terraform, we first need to create a project in GCP. To do so, we need to go to the GCP console and create a new project. After created the project, we can create our service. Go to `1AM & Admin > Service Accounts` and create a new service account by clicking in `+ CREATE SERVICE ACCOUNT`.



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

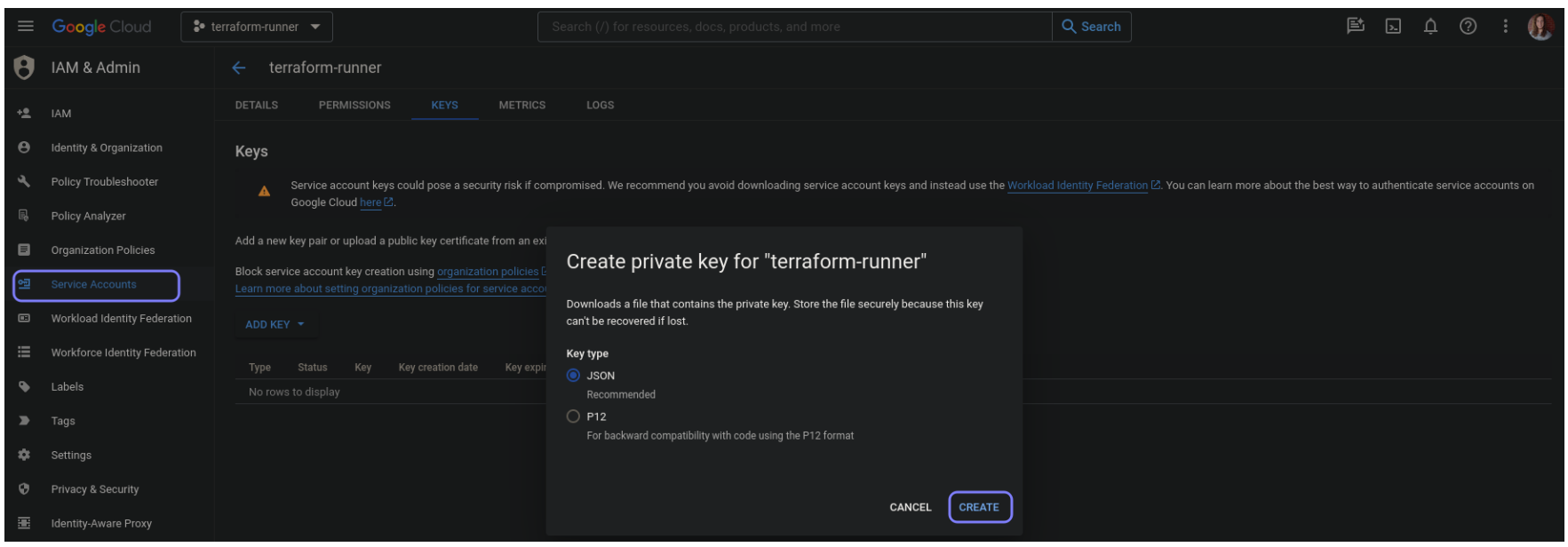


We can grant the service account access to the project by assign roles on the optional step show in the image. If we forgot to assign roles, we can do it later by going to the left menu on `1AM` and clicking on the edit principle button (the pencil icon):



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.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 clound 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
        }
```

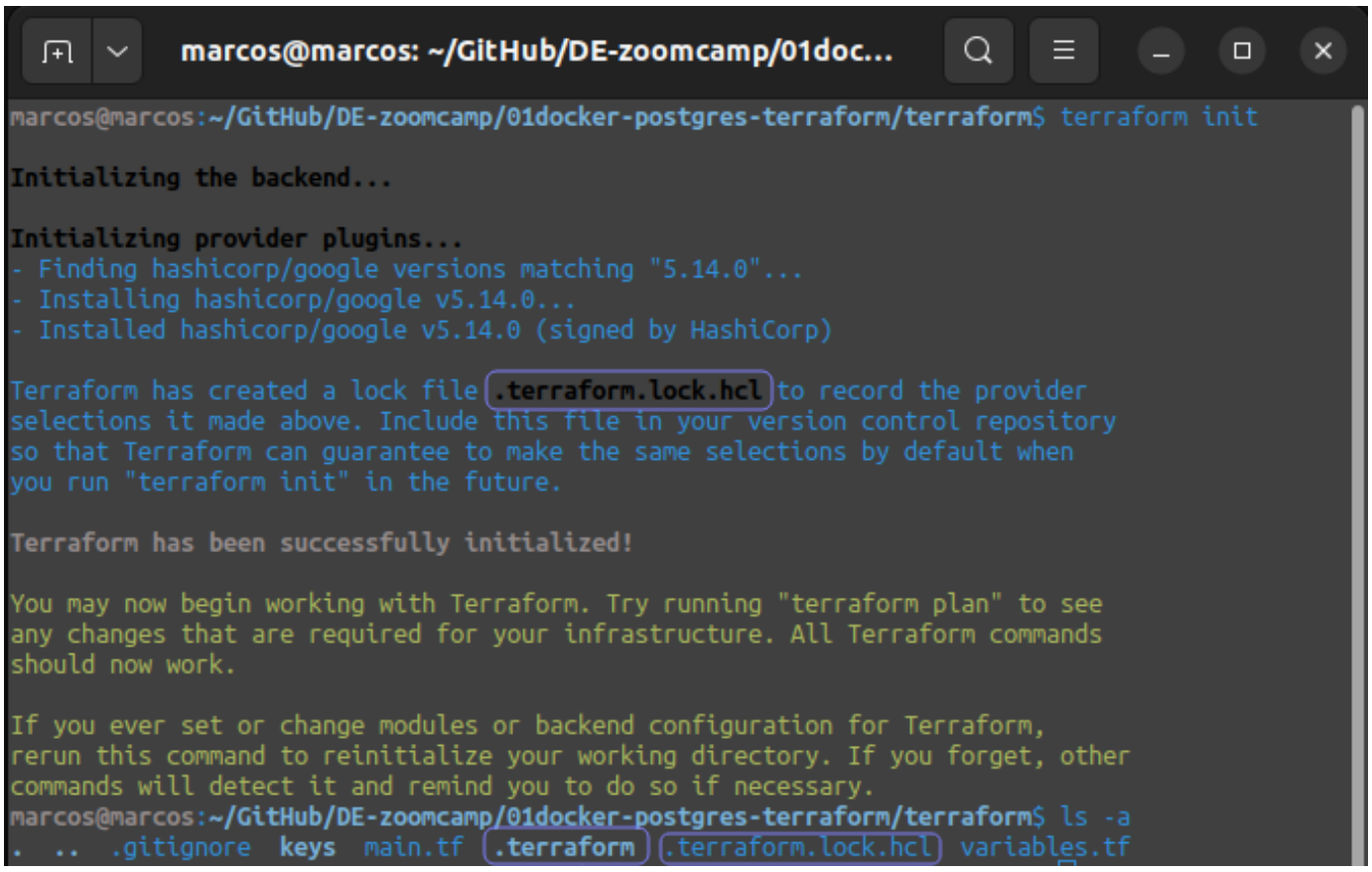
Each variable has a name, a type, and a default value.

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
        }
```

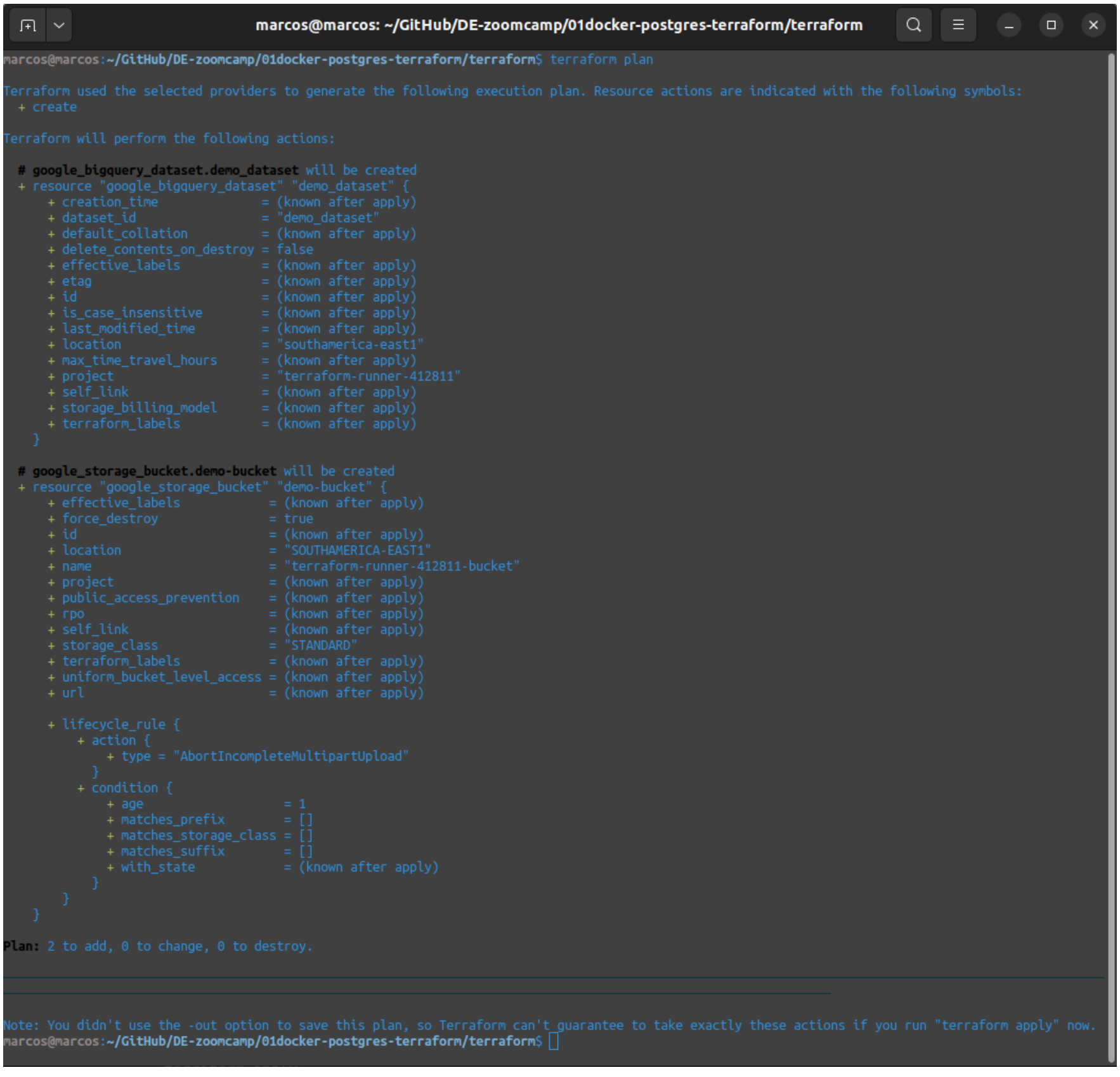
Now, we can run the terraform commands to initialize the project, plan the deployment, and apply the changes. To initialize the project, run the following command:

`terraform init`
this will 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.



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

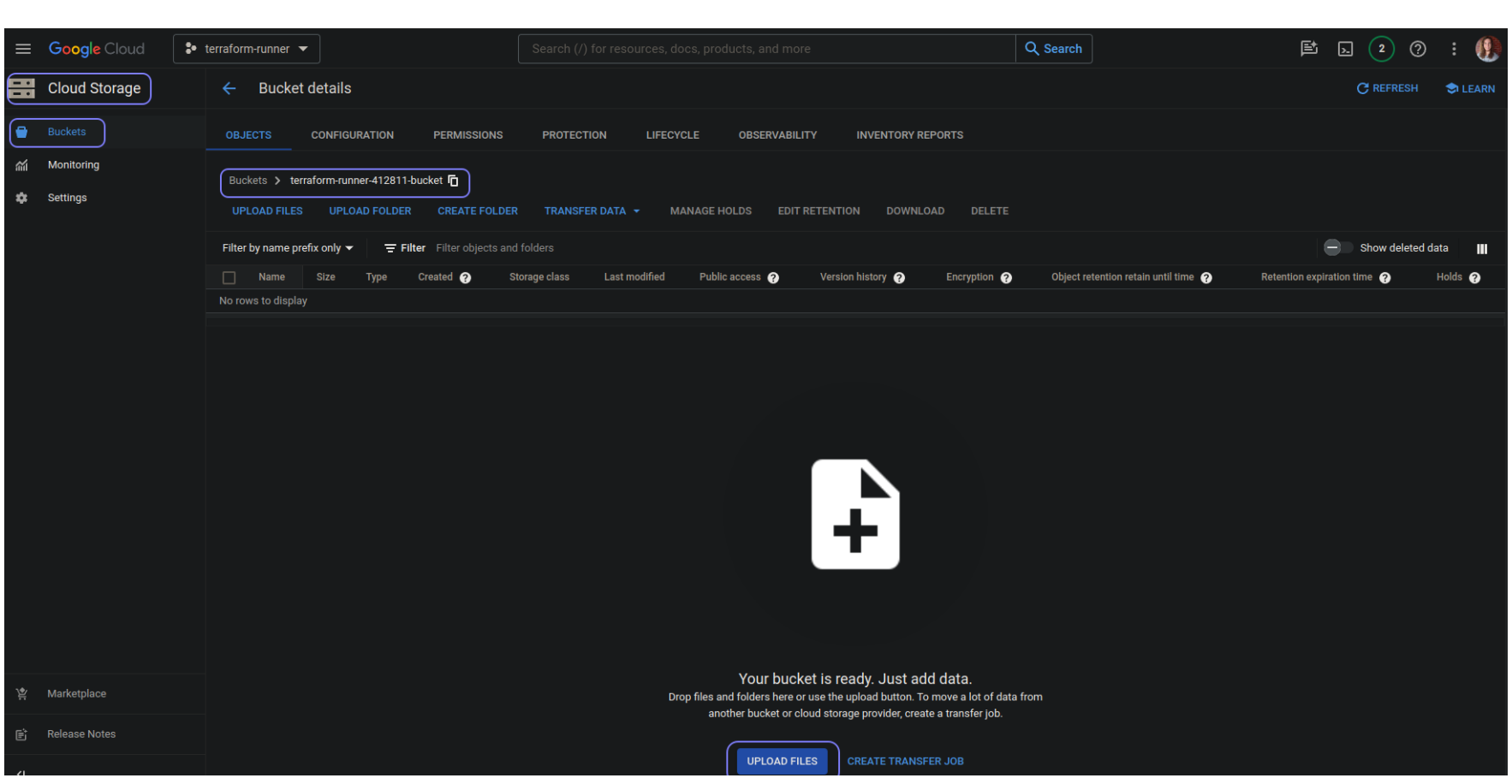
`terraform plan`
We can see that the plan will create a bucket and a dataset in GCP:



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.
```