Docker Zero to One: Step by Step Data Ingestion in Postgres by Docker Containerization & Docker Compose

Table of Contents

1) Introduction to Docker

Docker is a tool that helps developers create, share, and run applications in a consistent way. It packages everything an application needs into a single unit called a container, making it easy to run on any system with Docker installed. For data teams, Docker ensures that everyone uses the same setup, which is great for sharing and scaling tasks like data analysis or machine learning.

Use Cases

Docker containers are used to package and run applications in a portable and isolated environment. Here are a few examples of how they can be used:

- **Web Applications**: Docker packages web apps, making them easy to deploy and scale
- **Database Management Systems**: Run and manage databases like MySQL or MongoDB easily.
- **Data Processing and Analysis**: Share and run data workflows seamlessly across different environments.
- **Machine Learning**: Package and deploy machine learning models consistently.
- **DevOps**: Containers can be used for CI/CD (Continuous Integration and Continuous Delivery) pipeline as well, for example, by packaging the application and its dependencies in a container, and then using tools like Jenkins or Travis CI to test, build, and deploy the container to a production environment.

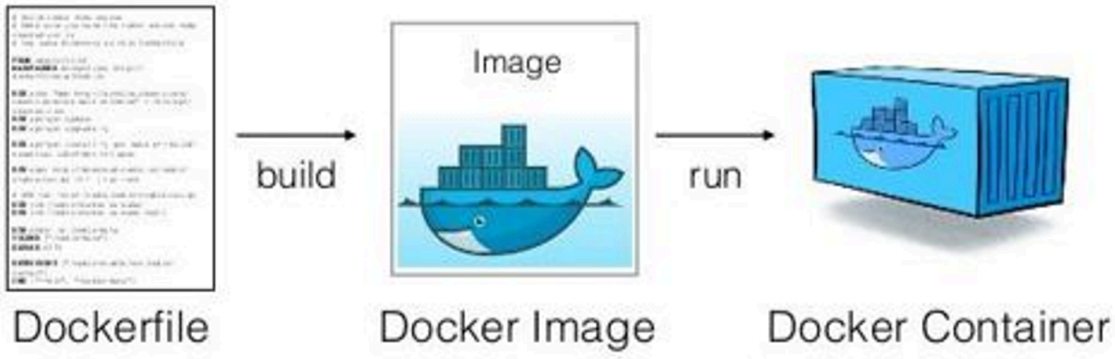
These are just a few examples of the many ways that Docker containers can be used. In general, containers are a powerful tool for packaging, deploying, and scaling applications in a consistent and reproducible way.

1.1) Setting up Docker

Setting up Docker involves the following steps:

1. **Install the Docker Engine**: Download and install Docker for your operating system.
2. **Verify the Installation**: Run docker version in your terminal to check if Docker is installed correctly.
3. **Create a Dockerfile**:Write a script (Dockerfile) that tells Docker how to build your application. To create a new Dockerfile, you can use a text editor and then save it with the name "Dockerfile" (without any file extension)
4. **Build the Image**: Use **'docker build'** to create an image from your Dockerfile.
5. **Run the Container**: After building the image, you can use the **'docker run'** command to start a container from the image. The command takes the image name as an argument, and it will start the container with the environment and dependencies that are defined in the Dockerfile.
6. **Interact with the Container**: Once the container is running, you can use the **'docker exec'** command to start a shell session inside the container. This allows you to run commands and interact with the application that is running inside the container.

These steps get you started with Docker, making it easier to manage, share, and scale your applications and workflows.



1.2) Running basic commands in GitBash after Docker installation

i) docker version

```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ docker version
Client:
 Cloud integration: v1.0.35+desktop.10
 Version:           25.0.3
 API version:       1.44
 Go version:        go1.21.6
 Git commit:        4debf41
 Built:             Tue Feb  6 21:13:02 2024
 OS/Arch:           windows/amd64
 Context:           default

Server: Docker Desktop 4.27.2 (137060)
 Engine:
  Version:          25.0.3
  API version:      1.44 (minimum version 1.24)
  Go version:       go1.21.6
  Git commit:       f417435
  Built:            Tue Feb  6 21:14:25 2024
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.6.28
  GitCommit:        ae07eda36dd25f8a1b98dfbf587313b99c0190bb
 runc:
  Version:          1.1.12
  GitCommit:        v1.1.12-0-g51d5e94
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

ii) docker run hello-world

```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

iii) docker run -it ubuntu bash

```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ docker run -it ubuntu bash
the input device is not a TTY.  If you are using mintty, try prefixing the command with 'winpty'
```

NOTE:

When running Docker commands on Windows, you may need to use the winpty command to run them in Git Bash. This is because the Docker client that runs on Windows communicates with the Docker daemon using the Docker for Windows service, which is not compatible with the Git Bash terminal.

winpty is a Windows version of the pty (pseudo-terminal) used in UNIX-like systems. It allows the Docker client to interact with the Windows console, allowing the user to run Docker commands in Git Bash as if they were running in a native Windows console window.

When you run a docker command in Git Bash without winpty, the command may fail or produce unexpected results, because Git Bash does not have the capability to interact with the Windows console.

Using winpty command before the docker command allows Git Bash to communicate with the Windows console and execute the command correctly. For example, instead of running docker run -it ubuntu, you would run winpty docker run -it ubuntu.

It's worth noting that this is a Windows-specific issue and is not required on other operating systems like Linux or Mac.

iv) winpty docker run -it ubuntu bash

```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ winpty docker run -it ubuntu bash
root@deb597360155:/#
```

Now that we added winpty in front of the command, the command works. Let's see what it does below:

The command **docker run -it ubuntu bash** is used to start a new container based on the ubuntu image and run a **bash** shell inside it. The **docker run** command is used to start a new container from an image.

The **-i** flag stands for "interactive" and it keeps the stdin open, even if not attached. The **-t** flag stands for "tty", it allocates a pseudo-TTY, this makes the container to have a terminal-like interface.

When you run this command, Docker will first check if you already have the ubuntu image on your machine. If not, it will download the image from the public Docker registry, called Docker Hub.

Once the image is downloaded, Docker will create a new container from the image and start it. The **bash** command is passed as an argument to the **docker run** command, which tells Docker to run a **bash** shell inside the container. (As you can see from the image above)

The bash shell will be running in an interactive mode, which means you can run commands inside the container and see their output in the terminal. For example, you can run the command **ls** to see the list of files in the container's file system, or **apt-get install** to install additional packages.

When you're done interacting with the container, you can exit the bash shell by typing **exit** or **Ctrl+D**. This will stop the container, but the image will remain on your machine for later use.

1.3) Building a Docker image and running a Container

i) code .

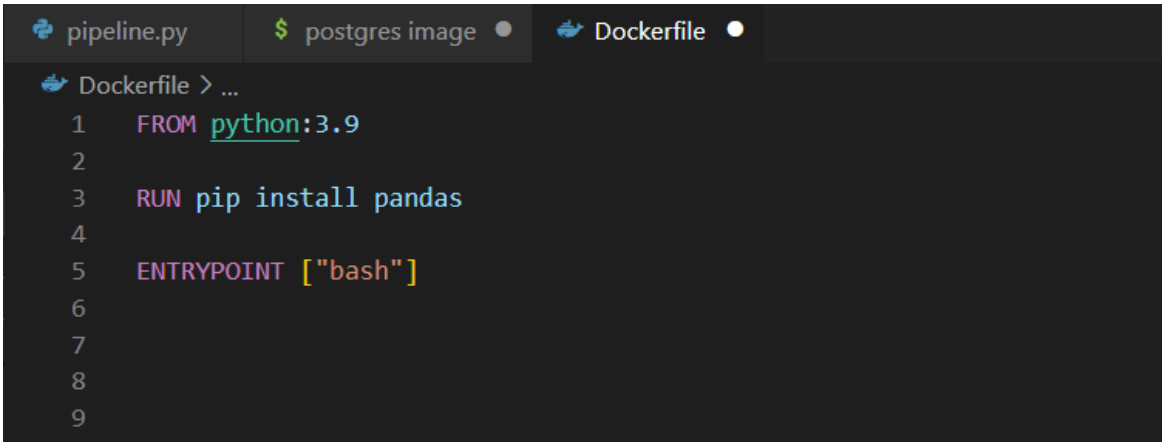
```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ code .

aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$
```

The command code . is used to open the current working directory in Visual Studio Code (VSCode), a popular code editor developed by Microsoft. The code command is the command line interface (CLI) for VSCode, and it allows you to open a folder or file in the editor directly from the command line.

The . at the end of the command is a shorthand for the current directory, so the command code . opens the current working directory in the VSCode editor. This can be useful when you want to quickly open a project or folder in VSCode without having to navigate through the file explorer.

It's worth noting that the code command is specific to VSCode and it's not a built-in command in the operating system. To use it, you need to have VSCode installed on your machine and the command line interface should be added to your system PATH during the installation.

ii) Define instructions for creating a docker image in the opened VSCode Editor



The above code is simple example of a Dockerfile, which is a script that defines the instructions for creating a Docker image. This Dockerfile is creating an image based on the official python image version 3.9 and it has three instructions:

1. **FROM python:3.9**: This instruction specifies the base image for the new image. In this case, the base image is python:3.9, which is an official image of Python version 3.9 provided by Docker.
2. **RUN pip install pandas**: This instruction runs a command inside the container to install the pandas library using pip. This will ensure that the pandas library is available in the image.
3. **ENTRYPOINT ["bash"]**: This instruction sets the default command that is executed when a container is started from this image. In this case, it is set to run the bash command, which will start a shell session inside the container. This allows the user to interact with the container and run commands inside it.

NOTE:

A Dockerfile should be saved in the root directory of your application, where the application's code and dependencies are located. The file should be named **Dockerfile** with no file extension (As you can see in the above screenshot). This is the default name expected by the **docker build** command, and using this name will make it easy to find and maintain.

It's worth noting that you can also specify a path to the **Dockerfile** when running the docker build command if it is not located in the current working directory. For example, if you have the Dockerfile in a subdirectory, you can use the **-f** or **--file** flag to specify the path to the Dockerfile, like **docker build -f /path/to/Dockerfile .**

iii) Run the docker build command

```
$ docker build -t test:pandas .
#2 [internal] load .dockerignore
#2 sha256:868b654ee9796e7c26e4574c9aaf8c198c37978331234283702a5cc81c712500
#2 transferring context: 2B 0.0s done
#2 DONE 0.1s

#1 [internal] load build definition from Dockerfile
#1 sha256:0d3fa64dc4064796bcbb9f8cb23078354e62df2080ae52f065f1e5593f13eec3
#1 transferring dockerfile: 105B 0.1s done
#1 DONE 0.1s

#3 [internal] load metadata for docker.io/library/python:3.9
#3 sha256:5ceb849adf4ca2eed629b09d8add870d381c45f6f40f947be68f6595c97a8af8
#3 DONE 0.0s

#4 [1/2] FROM docker.io/library/python:3.9
#4 sha256:3464e991adc12806f3b3cffe9c3cb1519d1d1016109972ce939d50250af91d7c
#4 DONE 0.0s

#5 [2/2] RUN pip install pandas
#5 sha256:a976f00ece0266042901ca21f1f51dd076c353a5ee9401813a0b948dec6e6546
#5 CACHED

#6 exporting to image
#6 sha256:e8c613e07b0b7ff33893b694f7759a10d42e180f2b4dc349fb57dc6b71dcab00
#6 exporting layers done
#6 writing image sha256:12affe65a851ded2765f9898f3199bf0812ed229ac98e9bc4bff86e54b8ab743 done
#6 naming to docker.io/library/test:pandas 0.0s done
#6 DONE 0.1s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

The command **docker build -t test:pandas .** is used to build a new Docker image from the Dockerfile we created.

The **docker build** command takes the current directory (**'.'**) as the build context, which is the location where the Dockerfile and other files and directories that are needed to build the image are located. The **-t** option, or **--tag** option, specifies the name and optionally a tag to the name of the image in the name:tag format. In this case, the command is creating an image with the name **test** and the tag **pandas**.

Once the image is built, you can use the **docker images** command to check the list of images available on your machine and confirm that the new image with the name test and the tag pandas is present. Also, you can use the **docker run** command to start a container from this image and test it.

It's worth noting that you can also use other tags for the image, and you can use the **docker tag** command to add, remove or change the tags of an existing image.

iv) Run docker images command to see if the image is

present

```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ docker images
REPOSITORY       TAG       IMAGE ID       CREATED          SIZE
taxi_ingest      v001      d157c62fd6cf   24 hours ago     1.2GB
<none>           <none>    08a0279bd3d1   4 days ago       1.17GB
test             pandas    24209606474b   4 days ago       1.17GB
<none>           <none>    5e988ff1b8df   4 days ago       1.17GB
<none>           <none>    87a1b69367ca   4 days ago       1.17GB
dpage/pgadmin4   latest    5675b83f2460   2 weeks ago      502MB
dpage/pgadmin4   <none>    f3e7a186b202   6 weeks ago      476MB
<none>           <none>    7c47387c4f49   2 months ago     1.17GB
<none>           <none>    a6fdd1595f0e   2 months ago     1.17GB
<none>           <none>    e6d643728b65   2 months ago     1.17GB
<none>           <none>    8a306af7a151   2 months ago     1.17GB
<none>           <none>    5f2c62f114ad   2 months ago     1.17GB
postgres         latest    d60dc4bd84c0   2 months ago     431MB
postgres         13        6e7ee57f3efd   2 months ago     419MB
ubuntu           latest    3db8720ecbf5   3 months ago     77.9MB
python           3.9       e301b6ca4781   3 months ago     997MB
hello-world      latest    d2c94e258dcb   12 months ago    13.3kB
```

As you can see, the first image is the one we created named as test and it's tag is pandas. One question that arises - Why does it show that the image was created 4 days ago?

It's because we are using an image that was already present on my machine, instead of building a new one (I had already created an image with the same command earlier in the week). Docker keeps a local cache of images that were previously pulled or built on your machine. So when you run the **docker build** command, it checks the local cache for an image with the same name and tag before building a new one.

If an image with the same name and tag is found in the local cache, Docker will use that image instead of building a new one, even if the files in the build context have been modified. This is why the image is showing as created 4 days ago.

You can use the **docker images** command to check the list of images available on your machine, and use the **docker rmi** command to remove an image if you want to build a new one.

Also, you can use the **--no-cache** option with the **docker build** command to force a new build and ignore the cache.
docker build --no-cache -t test:pandas .

NOTE:

If a new image has different instructions but the same name and tag as an already created image from cache, Docker will not automatically rebuild the new image.

As mentioned earlier, Docker uses a local cache of images that were previously pulled or built on your machine, so when you run the **docker build** command, it checks the local cache for an image with the same name and tag before building a new one. If an image with the same

v) Running a container from the built image

```
$ winpty docker run -it test:pandas
root@c07a7b289e87:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@c07a7b289e87:/# python
Python 3.9.16 (main, Jan 18 2023, 02:07:48)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pandas as pd
>>> pd.__version__
'1.5.3'
>>> exit()
root@c07a7b289e87:/# exit
exit
```

The command **docker run -it test:pandas** is used to start a new container from an image. Here we have ran **ls** to list the files and directories inside. Then we switched the terminal to a python one (by simply running **python**) and ran **import pandas as pd** command. This was done to check if pandas was installed as per our instructins from the image. Finally, we checked the version to see if it worked correctly.

Then we ran **exit()** to exit the python console. Then we ran **exit** to exit to exit the container.

NOTE:

If you run the docker run command again with the same image name and tag, it will create a new container, not reopen the previous one.

The **docker run** command creates a new container from an image each time it is run. It starts the container, runs the command specified in the command parameter, and then exits. Once the container exits, it is stopped and its state is lost, including any changes made inside the container.

You can use the **docker ps -a** command to list all the containers (including stopped ones) and look for the container id or name that you want to start.

```
$ docker ps -a
CONTAINER ID   IMAGE         COMMAND   CREATED          STATUS                      PORTS     NAMES
c07a7b289e87   test:pandas   "bash"    14 minutes ago   Exited (0) About a minute ago             intelligent_volhard
```

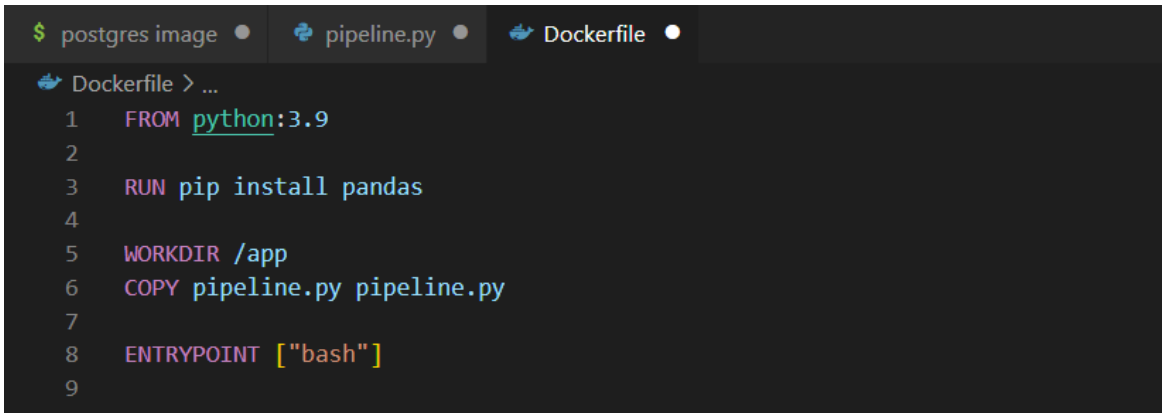
1.4) Running a script inside a container

Now that we know how to build an image and create a container, let's proceed further and run a script inside a container by using the following steps:

i) Create a sample python script which will be used

inside the container

ii) Modifying the Dockerfile to add more instructions



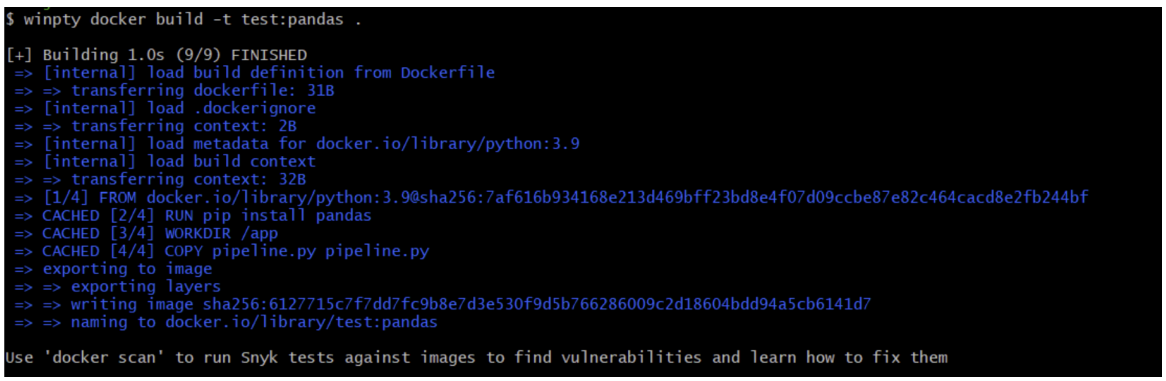
Here we're additionally adding 2 commands - WORKDIR and COPY

The working directory is set to /app, and the python file we created (see above) pipeline.py is copied into that directory.

iii) Building the Docker image

We're keeping the same name:tag combination and running the build command. Since the dockerfile got updated to include **pip install pandas**, you can see description about pandas installation.

It also includes the other two commands that we additionally added - Creation of the working directory app and the Copying of our Python script inside the working directory



iv) Running the container

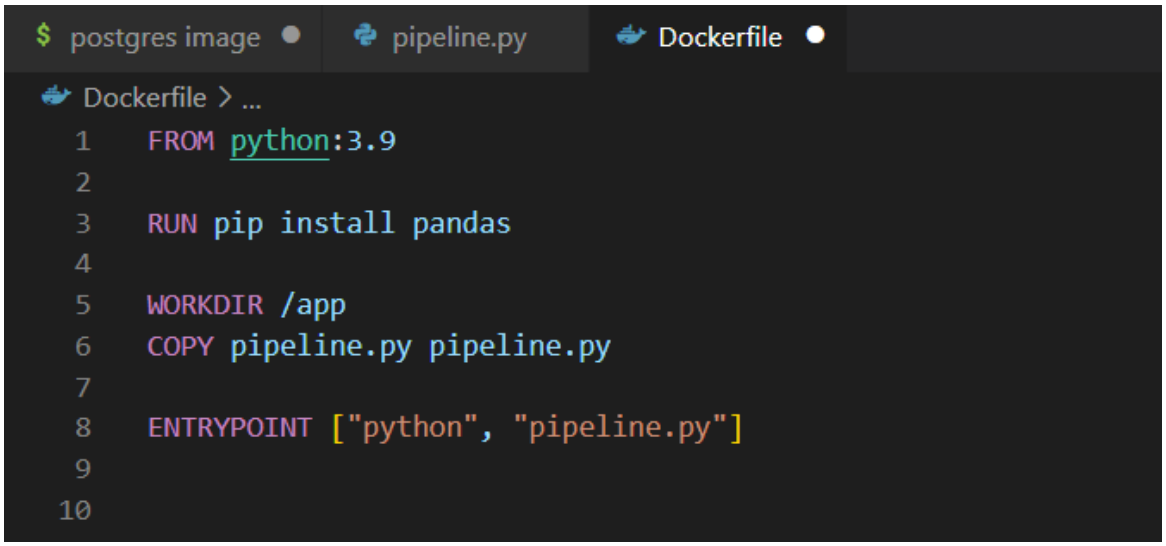
As you can see, when we initiate the run command, the working directory directly shifts to the app directory which we created. When we run **ls**, we can see that the Python script we created (pipeline.py) has been copied as per our instructions. When we run the script, we get the desired output.

```
$ winpty docker run -it test:pandas
root@dc08d04b3b85:/app# pwd
/app
root@dc08d04b3b85:/app# ls
pipeline.py
root@dc08d04b3b85:/app# python pipeline.py
Pandas installation was successful!
root@dc08d04b3b85:/app#
```

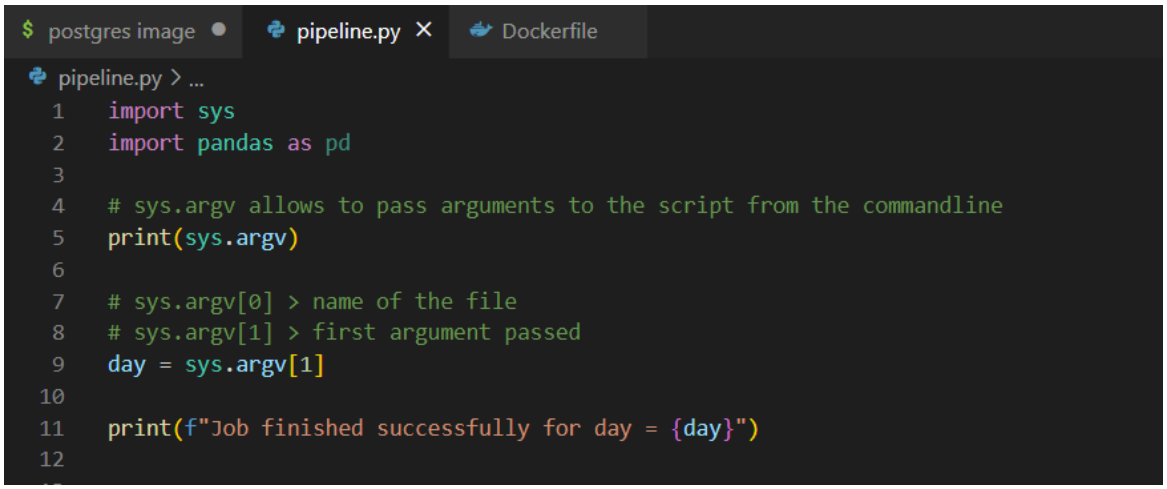
1.5) Passing arguments into a Container

Instead of doing the script running manually, let's pass all of it as arguments inside the docker file and make it run automatically

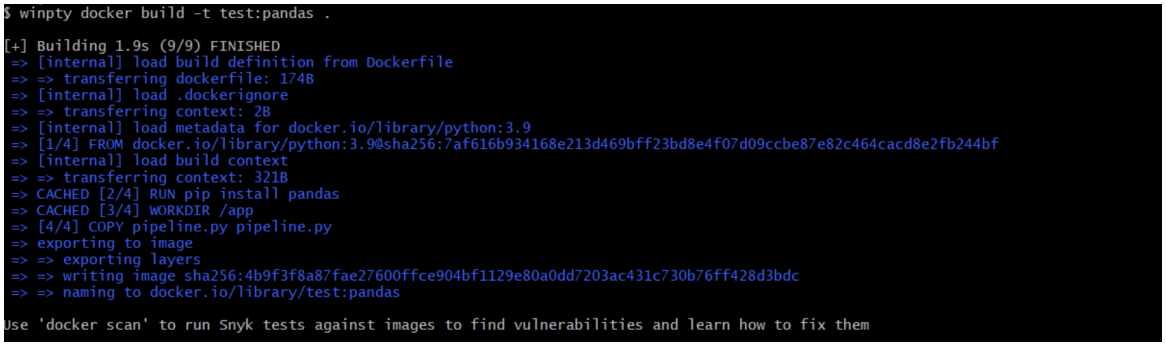
i) Modifying the docker file



ii) Updating the Python script to add some scheduling configuration

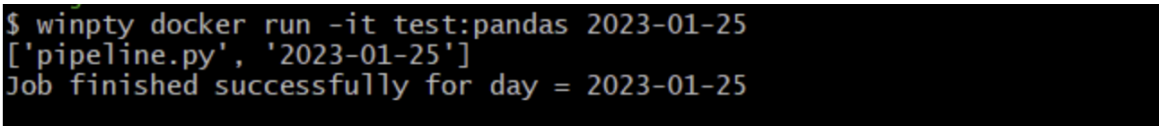


iii) Building the Docker image



iv) Running the Container

As we can see, when we ran the container, it automatically ran the Python script as instructed. Also, since we passed the date argument into the run command, it was used in the Python script as instructed and printed.



1.6) Note on Docker Containers

When you exit a Docker container, the container stops running but it is not automatically removed from the system. By default, the container's file system, network settings, and runtime state are all preserved. This allows you to start the container again later and pick up where you left off.

However, you can configure the container to automatically remove itself when it exits by using the **--rm** option when running the **docker run** command. This will remove the container and its associated resources (e.g., file system, network settings) from the system.

Additionally, you can use the **docker container prune** command to remove all stopped containers, which can help to reclaim disk space.

It's worth noting that when you remove a container, any data or changes made to the container's file system that have not been saved to a volume will be lost. To preserve data between container restarts or deletions, you can use volumes to mount data from the host machine or other containers.

2) Ingesting NY Taxi Data to Postgres

i) Creating a folder (ny_taxi_postgres_data) in your

working directory to store the container data

ii) Creating and running a postgres:13 image

```
winpty docker run -it \
    -e POSTGRES_USER="root" \
    -e POSTGRES_PASSWORD="root" \
    -e POSTGRES_DB="ny_taxi" \
    -v "/c/Users/aksha/Documents/Study/DE_Zoomcamp/Week1/ny_taxi_postgres_data:/var/lib/postgresql/data" \
    -p 5432:5432 \
    postgres:13
```

The above command uses the **docker run** command to start a new container based on the **postgres:13** image. The **-it** option runs the container in interactive mode, which allows you to interact with the container's command prompt.

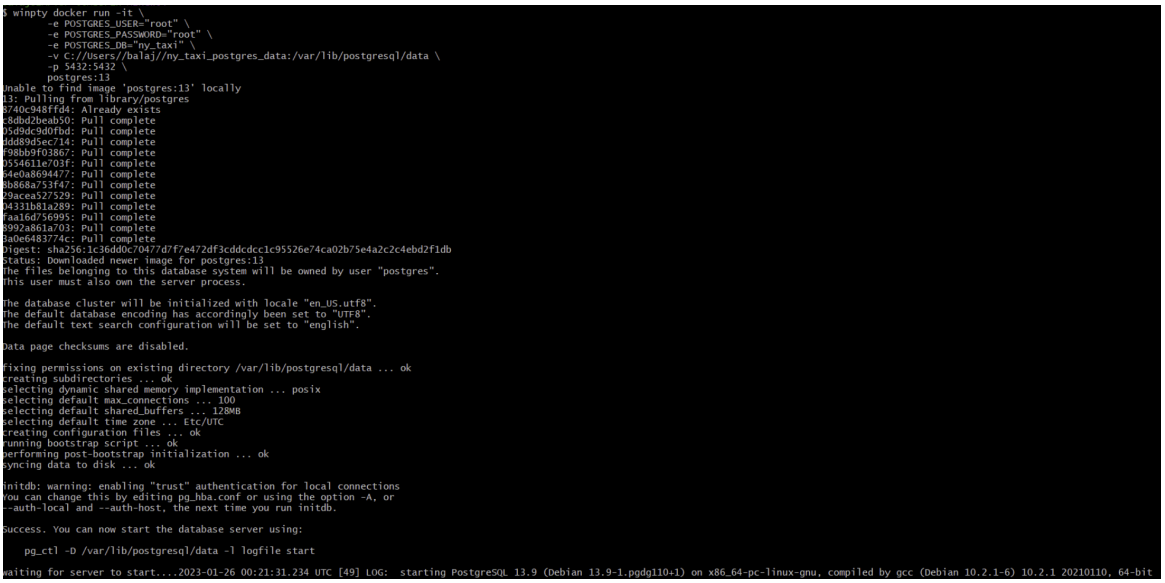
The **-e** option is used to set environment variables inside the container. In this case, it sets the following environment variables:

- POSTGRES_USER to "root"
- POSTGRES_PASSWORD to "root"
- POSTGRES_DB to "ny_taxi"

The **-v** option is used to mount a volume from the host machine to the container. In this case, it mounts the directory /c/Users/aksha/Documents/Study/DE_Zoomcamp/Week1/ny_taxi_postgres_data from the host machine to /var/lib/postgresql/data inside the container. This allows the data stored in the Postgres database to persist even if the container is removed.

The **-p** option is used to map a container's port to a port on the host machine. In this case, it maps port 5432 inside the container to port 5432 on the host machine. This allows you to connect to the Postgres database running in the container from applications running on the host machine.

As mentioned earlier, The **winpty** command is used to run the **docker run** command in a Windows environment which allows the interaction with the container's terminal.



As you can see above, when we run the command, postgres:13 image is downloaded and all the relevant files are loaded into the container. This would mean that we would be able to access these files in the **/c/Users/aksha/Documents/Study/DE_Zoomcamp/Week1/ny_taxi_postgres_data** location we had passed in the command.

The reason is that our command uses the -v option to mount a volume from the host machine to the container. The host path /c/Users/aksha/Documents/Study/DE_Zoomcamp/Week1/ny_taxi_postgres_data is being mounted to /var/lib/postgresql/data inside the container. This means that any files or directories created inside /var/lib/postgresql/data inside the container will be visible at /c/Users/aksha/Documents/Study/DE_Zoomcamp/Week1/ny_taxi_postgres_data on the host machine.

When the container runs, the Postgres server inside the container writes its data files, including the databases and their tables, to the /var/lib/postgresql/data directory. Since this directory is mounted to the host machine, any files created in this directory inside the container will be visible in the host machine at the specified host path, in this case /c/Users/aksha/Documents/Study/DE_Zoomcamp/Week1/ny_taxi_postgres_data. This allows you to persist data even if the container is stopped or removed.

In other words, the files in the ny_taxi_postgres_data folder of your local machine are the files that are created by postgres inside the container when it runs, because of the volume mount, the files are visible on the host machine.

iii) Accessing the Database (postgres) inside the container

Now that we have created a postgres database, the next step is to access it. To do that we need to install pgcli.

Step 1:

Open a new Gitbash window and run **pip install pgcli**

```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ pip install pgcli
Requirement already satisfied: pgcli in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (4.0.1)
Requirement already satisfied: pgspecial>=2.0.0 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (2.1.1)
Requirement already satisfied: click>=4.1 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (8.1.7)
Requirement already satisfied: Pygments>=2.0 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (2.17.2)
Requirement already satisfied: prompt-toolkit<4.0.0,>=2.0.6 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (3.0.43)
Requirement already satisfied: psycopg>=3.0.14 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (3.1.18)
Requirement already satisfied: sqlparse<0.5,>=0.3.0 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (0.4.4)
Requirement already satisfied: configobj>=5.0.6 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (5.0.8)
Requirement already satisfied: pendulum>=2.1.0 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (3.0.0)
Requirement already satisfied: cli-helpers>=2.2.1 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from cli-helpers[styles]>=2.2.1->pgcli) (2.3
Requirement already satisfied: setproctitle>=1.1.9 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pgcli) (1.3.3)
Requirement already satisfied: tabulate>=0.9.0 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from tabulate[widechars]>=0.9.0->cli-helpers>=2
->pgcli) (0.9.0)
Requirement already satisfied: colorama in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from click>=4.1->pgcli) (0.4.6)
Requirement already satisfied: six in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from configobj>=5.0.6->pgcli) (1.16.0)
Requirement already satisfied: python-dateutil>=2.6 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pendulum>=2.1.0->pgcli) (2.9.0.post0)
Requirement already satisfied: tzdata>=2020.1 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pendulum>=2.1.0->pgcli) (2024.1)
Requirement already satisfied: time-machine>=2.6.0 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from pendulum>=2.1.0->pgcli) (2.14.1)
Requirement already satisfied: wcwidth in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from prompt-toolkit<4.0.0,>=2.0.6->pgcli) (0.2.13)
Requirement already satisfied: typing-extensions>=4.1 in c:\users\aksha\appdata\local\programs\python\python312\lib\site-packages (from psycopg>=3.0.14->pgcli) (4.11.0)
```

Step 2:

Run the command **winpty pgcli -h localhost -p 5432 -u root -d ny_taxi**, which would prompt for a password. Enter the password we created (root, in our case). This would create a connection with the database.

```
Password for root:
Server: PostgreSQL 13.9 (Debian 13.9-1.pgdg110+1)
Version: 3.5.0
Home: http://pgcli.com
```

This command connects to a Postgres server running on the localhost, on port 5432, using the username "root" and the database "ny_taxi" where,

- -h flag specifies the hostname or IP address of the Postgres server,
- -p flag specifies the port number,
- -u flag specifies the username,
- -d flag specifies the database name.

If the Postgres server is running and accessible on the local machine on port 5432, and the user "root" exists and has the correct password, you should be able to connect to the "ny_taxi" database. Once the connection is established, you will be presented with the pgcli prompt and you can start executing SQL commands.

Step 3:

Run codes to ensure there's a connection to the database and check if it's working.

- Run the command **\dt** to see a list of tables available in our database - This returned empty as we have not populated the database with data
- Run the command **SELECT 1;** to check if it returns a value

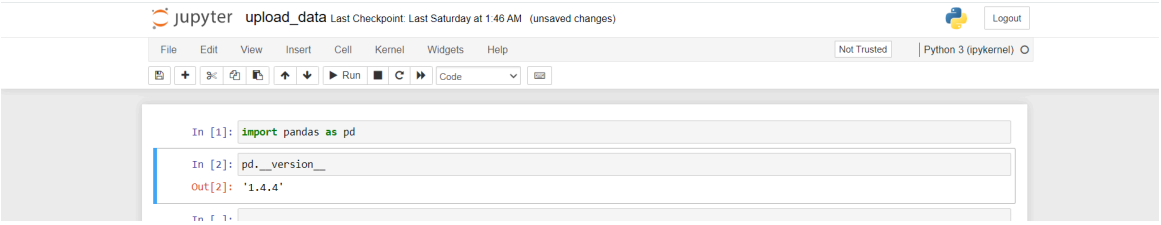
```
Password for root:
Server: PostgreSQL 13.9 (Debian 13.9-1.pgdg110+1)
Version: 3.5.0
Home: http://pgcli.com
root@localhost:ny_taxi> \dt
+--------+------+------+-------+
| Schema | Name | Type | Owner |
|--------+------+------+-------|
+--------+------+------+-------+
SELECT 0
Time: 0.211s
root@localhost:ny_taxi> SELECT 1;
+----------+
| ?column? |
|----------|
| 1        |
+----------+
SELECT 1
Time: 0.009s
root@localhost:ny_taxi>
```

iv) Creating a Jupyter Notebook

Install jupyter notebook in your system by running **pip install jupyter** and then run the command **jupyter notebook** to open the jupyter notebook terminal.

Create a new jupyter notebook and open it. I have named the notebook as **upload_data**

You can then run a few commands to check if it is up and running.



v) Downloading the data in Jupyter and doing some pre-processing

Step 1:

Download the data from the Github location hosted by DatatalksClub and Amazon S3-

Green Taxi Data - "https://github.com/DataTalksClub/nyc-tlc-data/releases/download/green/green_tripdata_2019-01.csv.gz" (https://github.com/DataTalksClub/nyc-tlc-data/releases/download/green/green_tripdata_2019-01.csv.gz")
Taxi Zones - "https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv" (https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv")

And then verify that the data has been loaded correctly.

```
In [1]: import pandas as pd
In [2]: pd.__version__
Out[2]: '1.4.4'
In [3]: url = "https://github.com/DataTalksClub/nyc-tlc-data/releases/download/green/green_tripdata_2019-01.csv.gz"
In [4]: df = pd.read_csv(url, nrows = 100)
In [5]: len(df)
Out[5]: 100
In [6]: df.head()
Out[6]:
```

	VendorID	lpep_pickup_datetime	lpep_dropoff_datetime	store_and_fwd_flag	RatecodeID	PULocationID	DOLocationID	passenger_count	trip_distance	fare_a
0	2	2018-12-21 15:17:29	2018-12-21 15:18:57	N	1	264	264	5	0.00	
1	2	2019-01-01 00:10:16	2019-01-01 00:16:32	N	1	97	49	2	0.86	
2	2	2019-01-01 00:27:11	2019-01-01 00:31:38	N	1	49	189	2	0.66	
3	2	2019-01-01 00:46:20	2019-01-01 01:04:54	N	1	189	17	2	2.68	
4	2	2019-01-01 00:19:06	2019-01-01 00:39:43	N	1	82	258	1	4.53	

```
In [8]: taxi_zone = pd.read_csv("https://s3.amazonaws.com/nyc-tlc/misc/taxi+_zone_lookup.csv")
In [9]: len(taxi_zone)
Out[9]: 265
In [10]: taxi_zone.head()
Out[10]:
```

	LocationID	Borough	Zone	service_zone
0	1	EWR	Newark Airport	EWR
1	2	Queens	Jamaica Bay	Boro Zone
2	3	Bronx	Allerton/Pelham Gardens	Boro Zone
3	4	Manhattan	Alphabet City	Yellow Zone
4	5	Staten Island	Arden Heights	Boro Zone

Step 2:

To put the data to postgres using Pandas, you need to first generate a Schema of the dataset, which is an instruction to create a table that specifies the columns and thier data types that you want to have in the table.

Run **print(pd.io.sql.get_schema(df, 'green_taxi_data'))** to generate the schema of the data.

```
In [8]: print(pd.io.sql.get_schema(df, 'green_taxi_data'))

CREATE TABLE "green_taxi_data" (
"VendorID" INTEGER,
  "lpep_pickup_datetime" TEXT,
  "lpep_dropoff_datetime" TEXT,
  "store_and_fwd_flag" TEXT,
  "RatecodeID" INTEGER,
  "PULocationID" INTEGER,
  "DOLocationID" INTEGER,
  "passenger_count" INTEGER,
  "trip_distance" REAL,
  "fare_amount" REAL,
  "extra" REAL,
  "mta_tax" REAL,
  "tip_amount" REAL,
  "tolls_amount" REAL,
  "ehail_fee" REAL,
  "improvement_surcharge" REAL,
  "total_amount" REAL,
  "payment_type" INTEGER,
  "trip_type" INTEGER,
  "congestion_surcharge" REAL
)
```

This shows how the table would look like when created. We still need to establish a connection between pandas and postgres and then run this command to create a schema in postgres. This was run just to show how the code works.

Step 3:

Before creating the connection, we need to do a small update. As can be seen in the above screenshot, the two date columns (**lpep_pickup_datetime**, **lpep_dropoff_datetime**) have their data type as **TEXT**. It would be useful to convert them to Date data type so that the data processing down the line becomes easier.

We can use the Pandas **to_datetime** function to convert datetime text to datetime object. Pass the column name to the function and assign the values to the same column name.

```
In [9]: df.lpep_pickup_datetime = pd.to_datetime(df.lpep_pickup_datetime)
        df.lpep_dropoff_datetime = pd.to_datetime(df.lpep_dropoff_datetime)

In [10]: print(pd.io.sql.get_schema(df, 'green_taxi_data'))

CREATE TABLE "green_taxi_data" (
"VendorID" INTEGER,
  "lpep_pickup_datetime" TIMESTAMP,
  "lpep_dropoff_datetime" TIMESTAMP,
  "store_and_fwd_flag" TEXT,
  "RatecodeID" INTEGER,
  "PULocationID" INTEGER,
  "DOLocationID" INTEGER,
  "passenger_count" INTEGER,
  "trip_distance" REAL,
  "fare_amount" REAL,
  "extra" REAL,
  "mta_tax" REAL,
  "tip_amount" REAL,
  "tolls_amount" REAL,
  "ehail_fee" REAL,
  "improvement_surcharge" REAL,
  "total_amount" REAL,
  "payment_type" INTEGER,
  "trip_type" INTEGER,
  "congestion_surcharge" REAL
)
```

As you can see above, after running the command, the data type has changed to **TIMESTAMP** for both the columns

Please note that the schema generated is not optimized as some more data types could still

vi) Establishing a connection and then putting the Schema into postgresql and loading in the data

Step 1:

Establishing Connection

In [11]:
```
# Pandas uses SQL Alchemy to interact with SQL databases. So you will need to run 'pip install SQLAlchemy' before

from sqlalchemy import create_engine

# Specify the database to be used, based on the docker run command we ran earlier
# postgresql://username:password@localhost:port/dbname


engine = create_engine('postgresql://root:root@localhost:5432/ny_taxi')

# create connection to the database engine to see if everything is working properly

engine.connect()
```

Out[11]:
```
<sqlalchemy.engine.base.Connection at 0x1c39ad7d0d0>
```

In [12]:
```
# Pass the engine variable to get_schema function
# Pandas will execute the schema SQL statement using the engine connection we have defined

print(pd.io.sql.get_schema(df, name="green_taxi_data", con=engine))
```

```
CREATE TABLE green_taxi_data (
        "VendorID" BIGINT,
        lpep_pickup_datetime TEXT,
        lpep_dropoff_datetime TEXT,
        store_and_fwd_flag TEXT,
        "RatecodeID" BIGINT,
        "PULocationID" BIGINT,
        "DOLocationID" BIGINT,
        passenger_count BIGINT,
        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 BIGINT,
        trip_type BIGINT,
        congestion_surcharge FLOAT(53)
)
```

Step 2:

Putting the Schema into postgres and creating a iterator to load the data

Since the entire dataframe is too big to process at once, we will chunk the CSV file into smaller batches. To do that we can use the iterator argument of the pd.read_csv function and declare the chunksize. The iterator argument allows iterating through n number chunks to build up the entire dataset. Using this method doesn't result in a dataframe but a type of iterator.

You can apply the next function to cycle/iter through 100000 rows of chunks of data.

In [16]:
```
df_iter = pd.read_csv(url, iterator = True, chunksize = 100000)
```

In [17]:
```
df_iter
```

Out[17]:
```
<pandas.io.parsers.readers.TextFileReader at 0x1d53cf60670>
```

In [18]:
```
df = next(df_iter)
```

In [19]:
```
len(df)
```

Out[19]:
```
100000
```

Step 3:

Loading the column names into postgres:

We can use to_sql to first load the column names and create the schema in postgres.

```
In [20]: df.head(n=0)
```

Out[20]:

VendorID	lpep_pickup_datetime	lpep_dropoff_datetime	store_and_fwd_flag	RatecodeID	PULocationID	DOLocationID	passenger_count	trip_distance	fare_am

```
In [21]: df.head(n=0).to_sql(name = 'green_taxi_data', con = engine, if_exists='replace')
Out[21]: 0
```

After running this, we can check if the schema got created inside postgres using pgcli. As we can see below, the schema is available and currently doesn't have any records populated.

We ran **\dt** to see the list of tables. To see details about the table run **\d green_taxi_data**

```
Server: PostgreSQL 13.9 (Debian 13.9-1.pgdg110+1)
Version: 3.5.0
Home: http://pgcli.com
root@localhost:ny_taxi> \dt
+--------+-----------------+-------+-------+
| Schema | Name            | Type  | Owner |
|--------+-----------------+-------+-------|
| public | green_taxi_data | table | root  |
+--------+-----------------+-------+-------+
SELECT 1
Time: 0.031s
root@localhost:ny_taxi> \d green_taxi_data
+-----------------------+------------------+-----------+
| Column                | Type             | Modifiers |
|-----------------------+------------------+-----------|
| index                 | bigint           |           |
| VendorID              | bigint           |           |
| lpep_pickup_datetime  | text             |           |
| lpep_dropoff_datetime | text             |           |
| store_and_fwd_flag    | text             |           |
| RatecodeID            | bigint           |           |
| PULocationID          | bigint           |           |
| DOLocationID          | bigint           |           |
| passenger_count       | bigint           |           |
| trip_distance         | double precision |           |
| fare_amount           | double precision |           |
| extra                 | double precision |           |
| mta_tax               | double precision |           |
| tip_amount            | double precision |           |
| tolls_amount          | double precision |           |
| ehail_fee             | double precision |           |
| improvement_surcharge | double precision |           |
| total_amount          | double precision |           |
| payment_type          | bigint           |           |
| trip_type             | bigint           |           |
| congestion_surcharge  | double precision |           |
+-----------------------+------------------+-----------+
Indexes:
    "ix_green_taxi_data_index" btree (index)

Time: 0.047s
root@localhost:ny_taxi> SELECT COUNT(1) FROM green_taxi_data;
+-------+
| count |
|-------|
| 0     |
+-------+
SELECT 1
```

We have successfully loaded the data into the postgres database and now can use it to perform data analysis.

3) Connecting pgAdmin and Postgres

As we can see from the previous screenshot, using pgcli might get difficult as we're not able to see the data clearly and the interface looks clunky. In order to overcome this, we can use pgAdmin.

pgAdmin is a free and open-source administration tool for the PostgreSQL database management system. It provides a graphical user interface for managing and interacting with PostgreSQL databases, allowing users to perform tasks such as creating and modifying

tables, managing users and permissions, and running SQL queries. It is available for Windows, Linux, and macOS.

Step 1

Rather than installing pgAdmin directly we can use docker to install a pgAdmin container.

We will be using the following command -

This command is used to run the pgAdmin tool in a Docker container. The command is starting a new container and maps port 8080 on the host to port 80 in the container.

- The option -it starts the container in interactive mode and assigns a terminal for it.
- The -e option sets environment variables for the container. In this case, it sets the default email and password for the pgAdmin web interface.
- The -p option maps the port 8080 on the host to port 80 in the container, which is the default port for the pgAdmin web interface.
- The last argument dpage/pgadmin4 specifies the image to be used for the container.

Step 2

The command will run pgAdmin in a container, and you will be able to access the pgAdmin web interface by navigating to http://localhost:8080 (http://localhost:8080) in a web browser. Once you login with the default email and password, you will be able to manage your PostgreSQL server via the web interface.

Enter the username and password we passed in our command and you'll be able to login.

Step 3

Next, we need to create a server by entering the details about the postgres server.

But before that, since the container having the pgAdmin and the container having the Postgres server are different, we need to connect them into a single network.

Before we do that, we need to stop both the containers. Then, we need to create a new network and rerun the containers by adding an instruction for the network as you can see below

```
$ docker network create pg-network
78594a7be82fc6c364821d50fdd8223a0a56fc700c39bfb63c084300b5a8ce08
```

```
winpty docker run -it \
    -e POSTGRES_USER="root" \
    -e POSTGRES_PASSWORD="root" \
    -e POSTGRES_DB="ny_taxi" \
    -v "/c/Users/aksha/Documents/Study/DE_Zoomcamp/Week1/ny_taxi_postgres_data:/var/lib/postgresql/data" \
    --network=pg-network \
    --name pg-database \
    -p 5432:5432 postgres
```

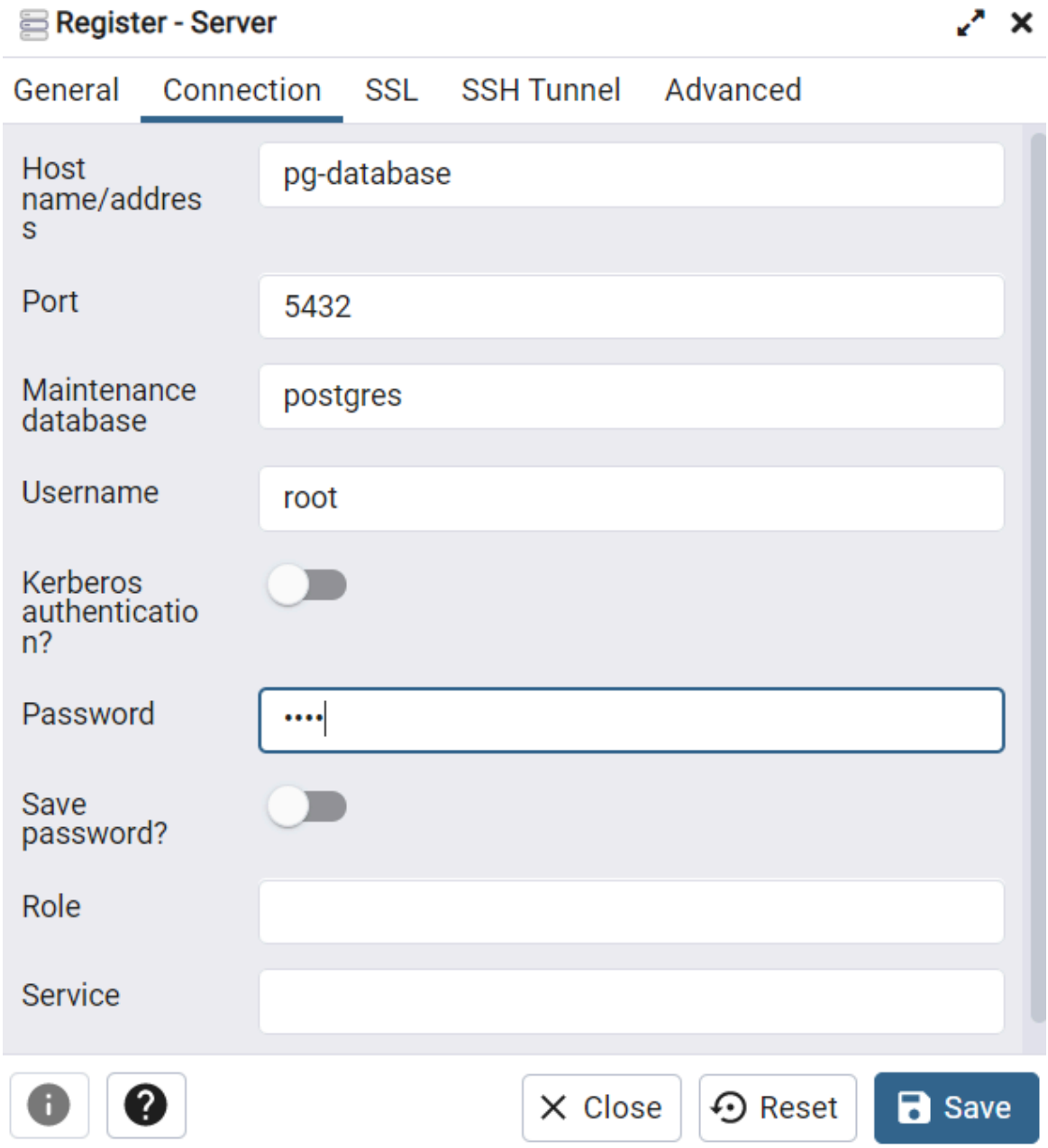
```
Winpty docker run -it \
    -e PGADMIN_DEFAULT_EMAIL="admin@admin.com" \
    -e PGADMIN_DEFAULT_PASSWORD="root" \
    -p 8080:80 \
    --network=pg-network \
    --name pgadmin \
    dpage/pgadmin4
```

As you can see above, we have added 2 new instructions for both -

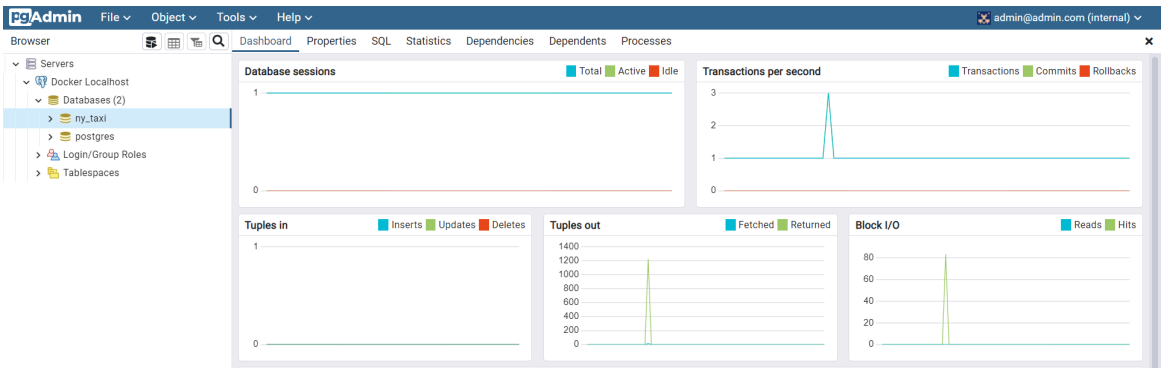
- --network - Here we have given a network name as 'pg-network' so that we can use it to pass a connection between Postgres and pgAdmin
- --name - Similarly, we have given individual names for each of them so that they can be used in the connection

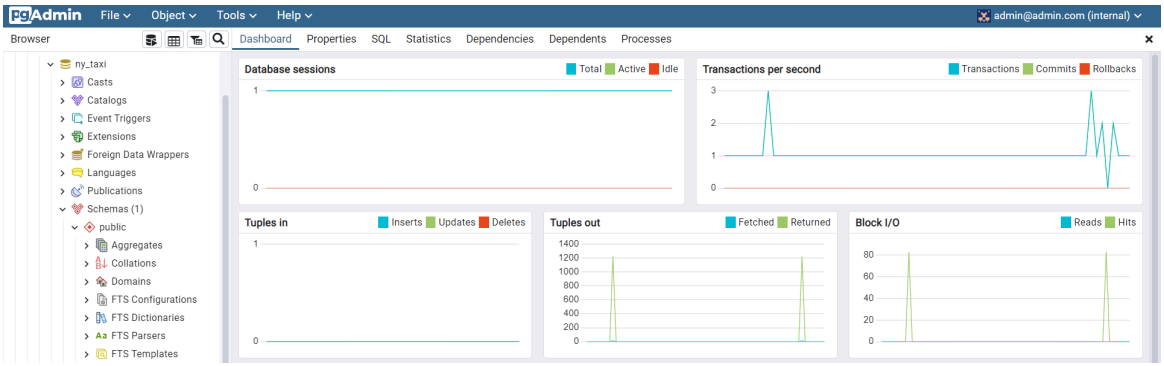
Step 4

Enter the relevant details in the pgAdmin console and create a connection.



We can see that the ny_taxi database is present and that the green_taxi table is available inside that database.





4) Dockerizing the Ingestion Script

Step 1:

Convert the Jupyter notebook into a script by saving it as .py file. Before converting, make relevant changes to the script and remove all the unnecessary commands.

The below screenshot shows the update script. The script is saved as `ingest_data.py` . But before running this, we still need to add a few more commands which you can see in Step 2.

Step 2:

Using `argparse` to parse command line arguments.

The `argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and argparse will figure out how to parse those out of `sys.argv` . The argparse module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

Find below the updated `ingest_data.py` script after adding the parse function.

Ingest_postgresData_Docker.py > ...

```python
import pandas as pd
import os
from sqlalchemy import create_engine
from time import time
import argparse


def main(params):

    username = params.user
    password = params.password
    host = params.host
    port = params.port
    dbname = params.db
    table_name = params.table_name
    url = params.url

    engine = create_engine(f'postgresql+psycopg2://{username}:{password}@{host}:{port}/{dbname}')
    engine.connect()

    df_iter = pd.read_csv(url, iterator=True, chunksize=100000)
    df = next(df_iter)

    df.head(n=0).to_sql(name=table_name, con=engine, if_exists='replace')

    df.tpep_pickup_datetime = pd.to_datetime(df.lpep_pickup_datetime)
    df.tpep_dropoff_datetime = pd.to_datetime(df.lpep_dropoff_datetime)

    for df in df_iter:
        t_start = time()

        df.tpep_pickup_datetime = pd.to_datetime(df.lpep_pickup_datetime)
        df.tpep_dropoff_datetime = pd.to_datetime(df.lpep_dropoff_datetime)

        df.to_sql(name=table_name, con=engine, if_exists='append')

        t_end = time()

        print(f'Inserted a new chunk. Time taken (in s) -  {round(t_end-t_start,2)}')

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 name for postgres")
    parser.add_argument('--port', help="port name for postgres")
    parser.add_argument('--db', help="database name for postgres")
    parser.add_argument('--table_name', help="name of the table where we will write the csv to")
    parser.add_argument('--url', help="url for the csv")

    args = parser.parse_args()
    main(args)


```

In a script, `parse_args()` will typically be called with no arguments, and the `ArgumentParser` will automatically determine the command-line arguments from `sys.argv` .

Step 3:

Run the command `DROP TABLE green_taxi_data;` and `DROP TABLE taxi_zone;` in pgAdmin in order to drop the existing tables so that we can run the Python script we just created which would re-create the same tables.

Run a `SELECT` command to ensure that the table is dropped.

Query Query History

```
1  SELECT * FROM GREEN_TAXI_DATA LIMIT 10;
2
```

Data Output Messages Notifications

```
ERROR:  relation "green_taxi_data" does not exist
LINE 1: SELECT * FROM GREEN_TAXI_DATA LIMIT 10;
                      ^

SQL state: 42P01
Character: 15
```

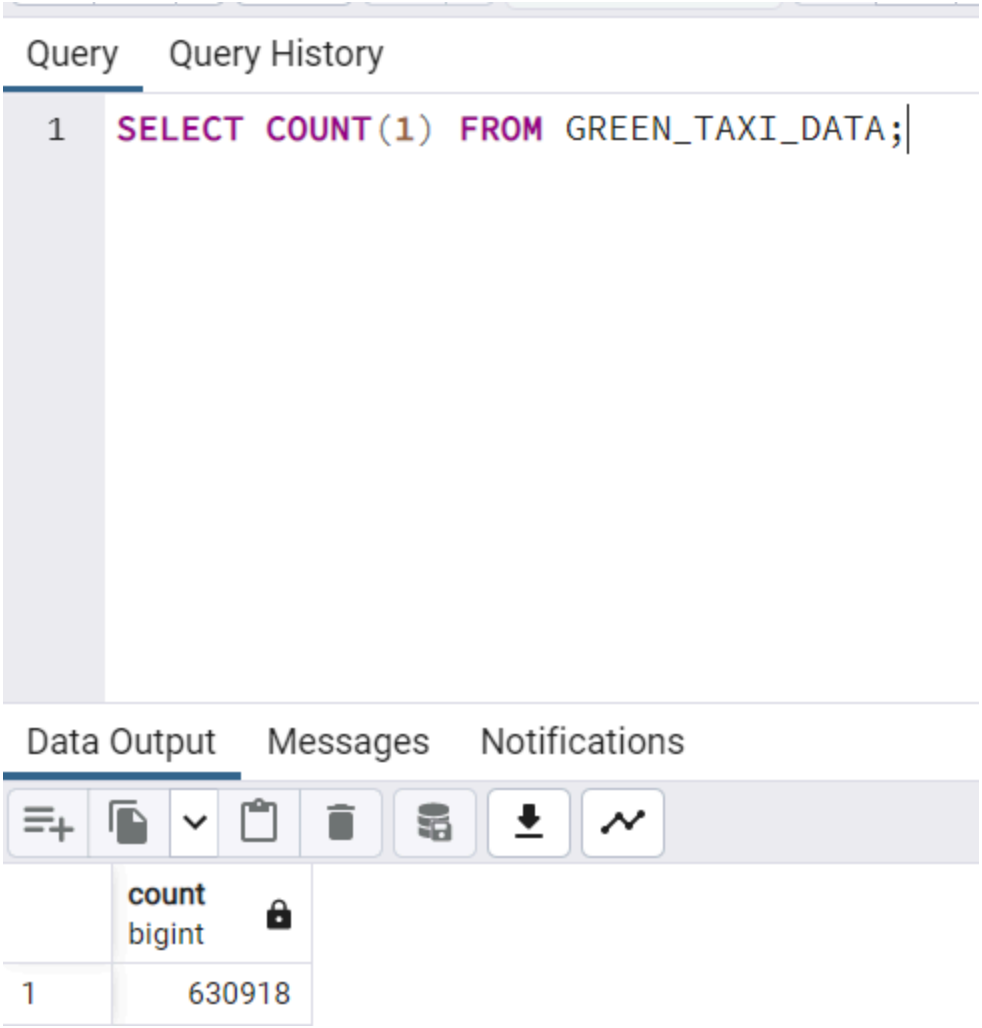
Step 4:

Run the following command to successfully execute the `ingest_data.py` script and upload the data into postgres.

```
aksha@Akshay MINGW64 ~/Documents/Study/DE_Zoomcamp/Week1
$ python Ingest_postgresData_Docker.py \
    --user=root \
    --password=root \
    --host=localhost \
    --port=5432 \
    --db=ny_taxi \
    --table_name=green_taxi_data \
    --url "https://github.com/DataTalksClub/nyc-tlc-data/releases/download/green/green_tripdata_2019-01.csv.gz"
```

Step 5:

Check if the data is available in postgres database using pgAdmin.



We can see that the data has been loaded successfully.

Step 6:

Now that we have confirmed the working of the `ingest_data.py` script, we can dockerize the whole thing.

To do that, let's update the Dockerfile. Look at the screenshot below for the changes made.

5) Running Postgres and pgAdmin with Docker Compose

So far, we were creating containers separately and then connected them into a single network. The whole process seemed cumbersome and messy to do. In order to avoid the mess, we can use something known as Docker Compose.

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to configure and run multiple containers, networks, and volumes in a single YAML file. This makes it easy to manage and configure multiple containers as part of a single

application.

Docker Compose uses a `docker-compose.yaml` file to define the services (containers), networks, and volumes that make up an application. The file is written in YAML, which is a human-readable format for specifying configuration settings.

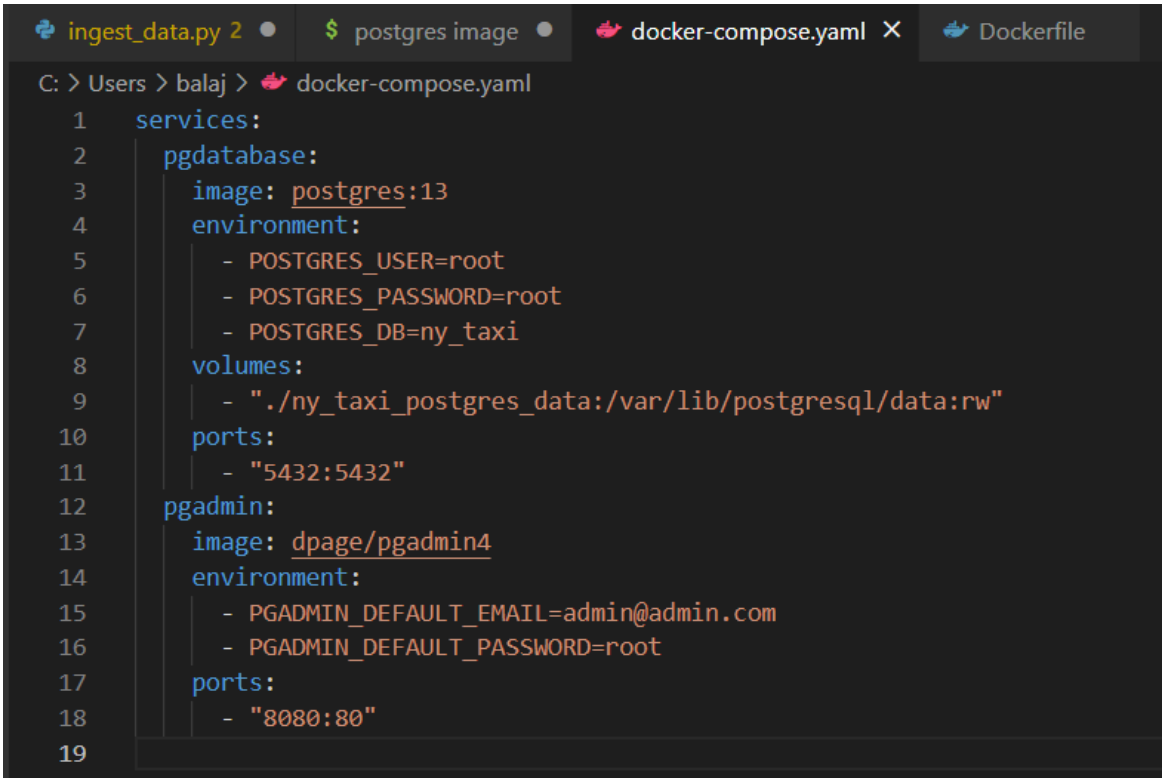
For example, you can use Docker Compose to define a web server container, a database container, and a cache container, and then start all of them with a single command. You can also configure the network and data volumes that the containers need to share.

You can use `docker-compose up` command to start the services defined in the `docker-compose.yaml` file and `docker-compose down` command to stop the services.

Docker Compose also allows you to scale the number of replicas of a service, update the service's configuration, and view the status of the containers running the service. It makes it easy to manage and scale multi-container applications, especially when combined with other Docker tools such as Docker Swarm or Kubernetes.

Step 1

Configure the docker-compose.yaml file as shown below



This is a Docker Compose file that sets up two services: `pgdatabase` and `pgadmin` .

The pgdatabase service uses the `postgres:13` image, which is a pre-built image of PostgreSQL version 13. The environment variables POSTGRES_USER, POSTGRES_PASSWORD, and POSTGRES_DB are set to root, root, and ny_taxi, respectively. This service also uses a volume, which is a way to share files between the host and the container. The volume is defined as "./ny_taxi_postgres_data:/var/lib/postgresql/data:rw" , which means that the local directory

./ny_taxi_postgres_data will be mapped to the directory /var/lib/postgresql/data inside the container, and the container will have read and write access to this directory. The service also maps port 5432 on the host to port 5432 inside the container.

The pgadmin service uses the `dpage/pgadmin4` image, which is a pre-built image of pgAdmin 4. The environment variables PGADMIN_DEFAULT_EMAIL and PGADMIN_DEFAULT_PASSWORD are set to admin@admin.com (mailto:admin@admin.com) and root, respectively. The service maps port 80 on the host to port 80 inside the container.

When you run docker-compose up, it will start the two services defined in the compose file.

Note that the ./ny_taxi_postgres_data directory should be present in the current working directory, otherwise the volume will not be created and the data will not be persistent.

Step 2

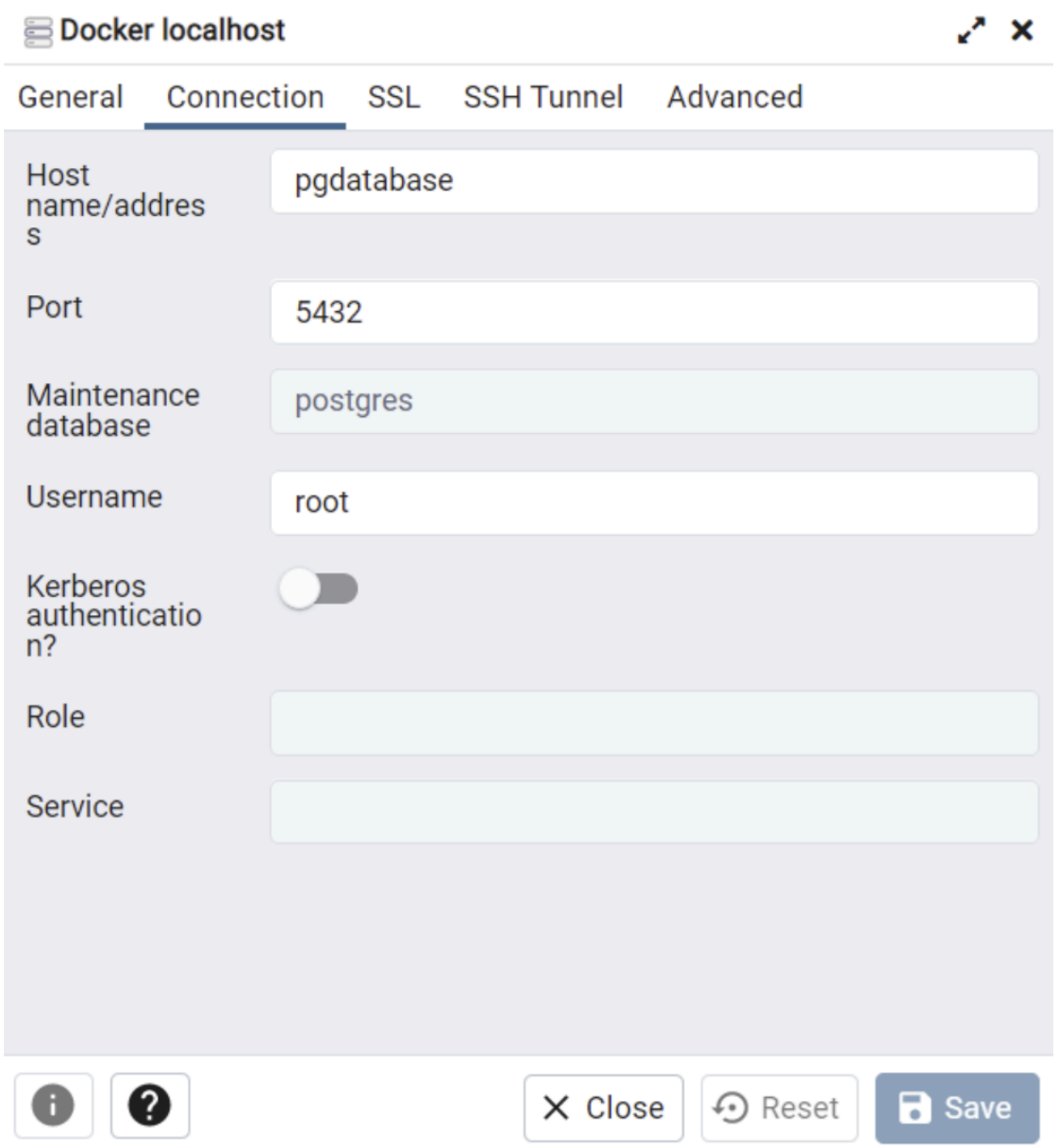
Stop all the containers we have created so far. Use `docker ps` to ensure all the containers are stopped

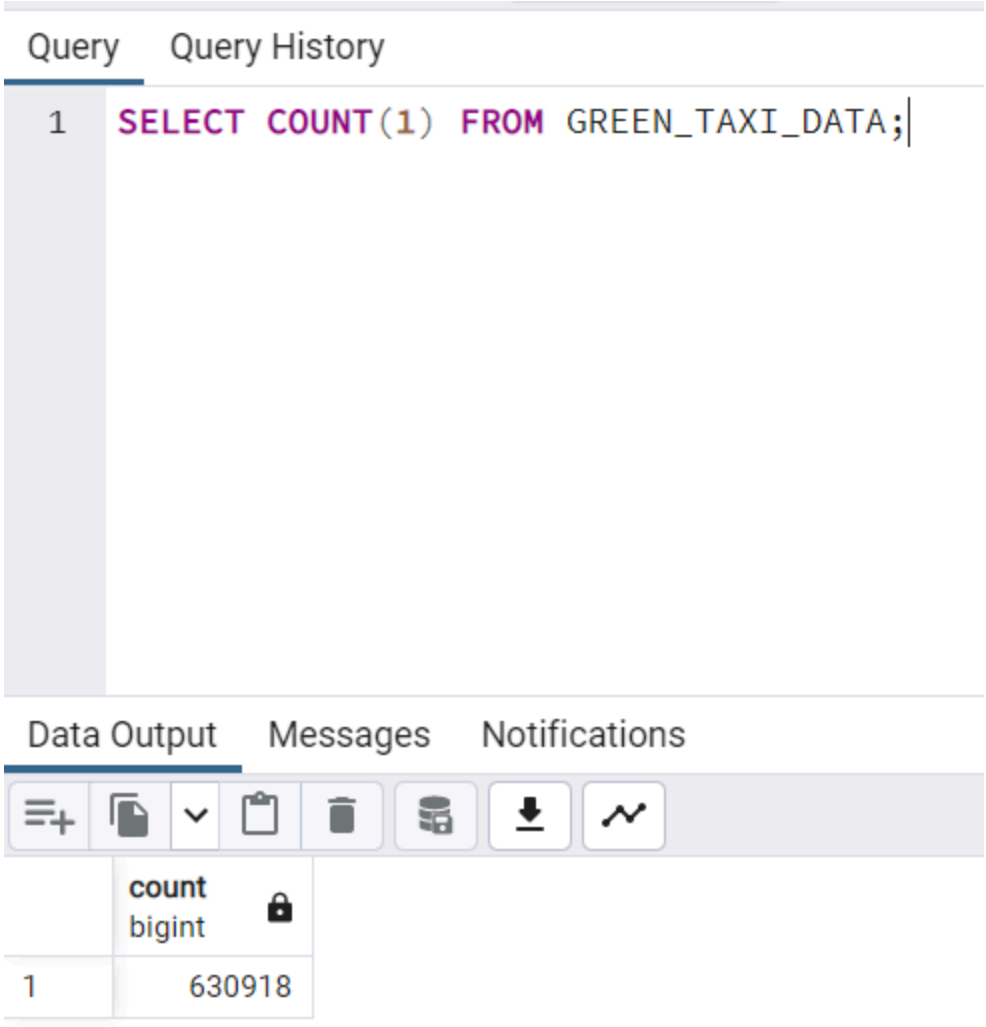
Step 3

Run the `docker-compose up` command to start the services defined in the `docker-compose.yaml` file

Step 4

Refresh pgAdmin web interface, login and then recreate and reconfigure connections to the pgdatabase server.


Docker localhost
General
Connection
SSL
SSH Tunnel
Advanced
Host name/address
pgdatabase
Port
5432
Maintenance database
postgres
Username
root
Kerberos authentication?
Role
Service
Close
Reset
Save



Instead of manually setting up the connection every time in pgAdmin, we can use the below `YAML` script to directly connect. Here, we have added a volumes variable for pgAdmin, which is defined as "pgadmin_conn_data:/var/lib/pgadmin:rw", which means that the local directory pgadmin_conn_data will be mapped to the directory /var/lib/pgadmin inside the container, and the container will have read and write access to this directory.

The volumes section defines named volumes that can be used by other services. The pgadmin_conn_data is a named volume that is used by the pgadmin service.

Note that the ./pgadmin_conn_data directory should be present in the current working directory, otherwise the volume will not be created and the data will not be persistent.