# **Introduce Docker (….Week 1….)**

1) Create DOCKERFILE

In the first line: We load the already existing image of python with tag <3.9> from Docker Hub.

Then we run all other commands we wish.

In ENTRYPOINT we say that: When we do docker run, we want docker to do whatever is inside there, i.e., python, and

pipeline.py

And that is how we take the default image and based on that we create a new one.

2) Create a **data pipeline** pipeline.py

Write your code and do whatever you want.

3) Create Docker IMAGE and run: **docker build -t vision .**

Build the docker file to create the image: vision is the name of the docker image.

"Vision is the name, dashboards are the game!" - Efsta

4)\* Run the image: docker run -it vision *(This step is not required for the project – just for you to see what is happening if you perform this step manually)*

After vision add parameters if needed in the code.

# **Ingest Data to Postgres from Host Machine manually!**

**(At a later step we dockerize these actions for the data pipeline)**

5) **Run Postgres in Docker =** Create **docker-compose.yaml** (see details below for this file)

This yaml file will set-up the connection and the container of the Postgres database.

We specify the name of the image for postgres, we created manually a folder in our local system that is for the dataset. We called it *ny\_taxi\_postgres\_data* andwe will map this to a folder in Postgres (mounting).

Later that we will run the docker image for postgres (in step 6), we will see that postgres will create and install some stuff there. Maybe it is some internal representation for the database.

6)\* Run the docker image for postgres (the name of the image is postgres:13). The “*volumes*” is a way of mapping the folder that we have in the host machine to the folder in the container. Postgres is a database and needs to keep files in a file system. However, Docker does not keep the state so we want to define this volume to remember where is the data. *(This step is not required for the project – just for you to see what is happening if you perform this step manually)*

docker run -it \  
 -e POSTGRES\_USER="root" \  
 -e POSTGRES\_PASSWORD="root" \  
 -e POSTGRES\_DB="ny\_taxi" \  
 -v $(pwd)/ny\_taxi\_postgres\_data:/var/lib/postgresql/data \  
 -p 5432:5432 \  
 postgres:13

7)\* Access this database manually from your local machine just to check the connection to the Postgres is ok. *(This step is not required for the project – just for you to see what is happening if you perform this step manually)*

Run a **cli client** for accessing the database. We use **pgcli** which is a library in python. So, in another terminal, we must pip3 install pgcli , if we don’t have pgcli already installed.

Run:

python3.10 -m pip install --upgrade pip

pip3 install pgcli

pip3 install "psycopg[binary,pool]"

**pgcli -h localhost -p 5432 -u root -d ny\_taxi**

If connection is successful: select \* from information\_schema.schemata

Run: \dt 🡪 to see the list of all the tables

8) We wget the dataset locally and we explore it a bit. To see how many samples, we have in the dataset run this: **wc -l yellow\_tripdata\_2021-01.csv**

9) Install **sqlalchemy** and connect to Postgres to **send the dataset in Postgres in batches**.

First, we need **to generate the compatible DDL schema for Postgres**, which means that we convert this dataframe to DDL: pd.io.sql.get\_schema(df, name=”yellow\_taxi\_data”)

First install:

pip install sqlalchemy

pip install psycopg2-binary

**create\_engine()** from sqlalchemy: we need to create a connection to postgres and

**pd.io.sql.get\_schema()**: we will generate the statement of the schema that is specific for Postgres.

We create the table, and we **send the data in chunks using an iterator**. Once this is done, if we go back to step 7 (that can be omitted) we can check manually what has happened in the Postgres.

Run this: SELECT count(1) FROM yellow\_taxi\_data;

See that data has been sent to Postgres.

# **Connect pgAdmin and Postgres**

10) **Install pgAdmin using Docker**: is a GUI tool to interact with the Postgres database.

So, before we used **pgcli**, but it is not convenient to use pgcli for data exploration and querying.

🡪**pgAdmin** is more convenient. Since, we have docker, we don’t need to install it, but we can just pull an image from Docker Hub (or create one by ourselves) that contains the tool.

Run pgAdmin in Docker:

docker run -it \

-e PGADMIN\_DEFAULT\_EMAIL="admin@admin.com" \

-e PGADMIN\_DEFAULT\_PASSWORD="root" \

-p 8080:80 \

dpage/pgadmin4

We map our port 8080 which is on our machine, to a port 80 on the container.

**pgAdmin** is running, is listening to requests on port 80 and we map this port to our host machine port 8080. Hence, all the requests we will send to port 8080, they will be sent to port 80 on the container.

Open localhost:8080 from your browser, and see you now have pgAdmin. Login with the credentials just above.

11) Register new Server (our Postgres database) in pgAdmin:

In Register – Server tab, we give the information for the Postgres database (see Jupyter notebook file: localhost, root, root). But we get a connection error because Postgres is in another container from pgAdmin.

Then, we must place them in the same network to avoid these connection errors.

So, terminate the 2 containers (the one we run the Postgres, and the one we run the pgAdmin).

12) **Docker Networks: Running Postgres and pgAdmin together in one network**

Run: docker network create **pg-network**

So, we have a network, and then we need to run postgres container in this network.

So, RE-run the updated **Postgres command**:

**docker run -it \**

**-e POSTGRES\_USER="root" \**

**-e POSTGRES\_PASSWORD="root" \**

**-e POSTGRES\_DB="ny\_taxi" \**

**-v $(pwd)/ny\_taxi\_postgres\_data:/var/lib/postgresql/data \**

**-p 5432:5432 \**

**--network=pg-network \**

**--name pg-database \**

**postgres:13**

**AFTER 1st run and once the docker image is created we do this >> docker start pg-database**

*(Note: the name will be how pgAdmin will be able to discover Postgres)*

Run again (on another command line) the **pgcli -h localhost -p 5432 -u root -d ny\_taxi**

And **SELECT count(1) FROM yellow\_taxi\_data;**

Check you still have all your data there.

RE-run the updated **pgAdmin command in the same network (on another cmd line):**

**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**

**AFTER 1st run and once the docker image is created we do this >> docker start pgadmin**

Reload pgAdmin page and try to create the Server now. In Register tab, we give the name that we specified in the network attribute above, i.e., pg-database, and then root, root.

Once you connect, you see the 2 databases. Open ny\_taxi, then Schemas, then Tables. Right click, view data, and see the first 100 rows. It runs this query, and you cannot edit it. But open the Query Tool from Tools and run your own queries.

Graphical user interface, text, application, email

Description automatically generated

# **Dockerizing the Ingestion Script**

13) We should turn the notebook file into a script. And we also need to put/ or call this code in the pipeline.py file. By this way we will have the code for downloading the dataset and populating the database.

So, let’s convert this notebook into a python script:

What worked for me is to install jupyter lab and then also pip3 install ipynb-py-convert and then

ipynb-py-convert upload\_data.ipynb upload\_data.py

But you have to be in the folder of the .ipynb files that you want to convert.

* But we need first to drop the table in Postgres (we already have populated it with the notebook file)
* So, go to the pgAdmin server and: **DROP TABLE yellow\_taxi\_data;**
* If you try to run now the SELECT command you will get an error, because this table does not exist anymore.

Let’s run locally the python script to populate again the database via the pipeline.

>> **python3** ingest\_data.py --user=root --password=root --host=localhost --port=5432 --db=ny\_taxi --table\_name=yellow\_taxi\_trips --url=”<https://github.com/DataTalksClub/nyc-tlc-data/releases/download/yellow/yellow_tripdata_2021-01.csv.gz>”

When it will finish, we can refresh the table in the database in pgAdmin server and see that all data are there (run previous SELECT command in this new table)

So, now that we want to dockerize the script we should install all required packages in Dockerfile. For example, to install pgAdmin we needed the sqlalchemy and the psycopg2. Also, install wget and change the name of the pipeline to ingest\_data.py. Since we renamed the name of the file in our project, we need to update the name in the Dockerfile as well.

So, build again the Dockerfile and now you have the container with the script of ingest\_data.py pipeline, in which python script we get the dataset, we connect to the database in Postgres, and we populate it with our data.

>> docker build -t taxi\_ingest:v001 .

However, to run it we need to put it also in the Docker network, otherwise we cannot connect with the Postgres database. So instead of the python3 command above we need to run this:

>> URL="https://github.com/DataTalksClub/nyc-tlc-data/releases/download/yellow/yellow\_tripdata\_2021-01.csv.gz"

docker run -it \

--network=pg-network \

taxi\_ingest:v001 \

--user=root \

--password=root \

--host=pg-database \

--port=5432 \

--db=ny\_taxi \

--table\_name=yellow\_taxi\_trips \

--url=${URL}

**Note 1**: the network attribute means that we run this docker container in **a docker network**.

**Note 2**: Our host now is not localhost but the name of the container that hosts the pipeline.

In a real-life scenario, as *host* we would have a url from our database in cloud, and a Kubernetes job could execute this command instead of running this manually in our terminal.

# **Run Postgres and pgAdmin with docker-compose**

14) What we did so far is to run the docker commands for Postgres and pgAdmin containers from our terminal. Check the yaml file in the repository and there you will see that you can specify the 2 services we want to run, so as to access **pgdatabase** (the Postgres database) from pgAdmin.

BUT in the jobs here in docker-compose.yaml we do not have to specify a docker network, these services will automatically become part of the same network. So we do not need to create a network or to add a network attribute.

So, let’s stop the containers for Postgres and pgAdmin and run: **docker-compose up -d**

Then go again to pgAdmin server and create a new server again.

*\*SPACE for fixes: They could have done volume mapping for the ports in the pgadmin job in the yaml file(mount). With this way we could have saved it.*

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

So, we see that we managed to successfully connect to the Postgres database. So, we have the ny\_taxi database, there we can check the Schemas -> Tables the yellow\_taxi\_data. Right click on the table’s name and View/Edit data-> choose the first 100 rows. Hence, you confirm all is good.

To stop this, we do Control+C, and then **docker-compose down** !!!!

Text

Description automatically generated

Run docker compose in detached mode (add attribute -d), which means that we get the terminal back, and hence we do not have to see the logs while docker is running.

Localhost (laptop):

* 5432 => 5432 in pgdatabase
* 8080 => 80 in pgadmin
* 8888 => jupyter notebook

Postgres’s port is exposed to be become accessible from our localhost (in yaml)

# **SQL Refresher**

15) If we have stopped the docker-compose then we need to start this again. BUT we have to re-register the pgAdmin server. Since we did not do the mounting in the yaml file, as we explained above. Once we do this, we can see the tables again, since when we started docker compose it ran again the pipeline that ingests the data.

Now, we add a new table using simply the jupyter notebook. Run the last 3 cells.

Now we have 2 tables, and we want to joint them.

SELECT \*

FROM

yellow\_taxi\_data t,

zones zpu,

zones zdo

WHERE

t."PULocationID"=zpu."LocationID" AND

t."DOLocationID"=zdo."LocationID"

LIMIT 100;

Hence, now we see the zones’ columns in the yellow\_taxi\_data table. Let’s limit the records we see by running this statement:

SELECT

tpep\_pickup\_datetime,

tpep\_dropoff\_datetime,

total\_amount,

CONCAT(zpu."Borough", ' / ', zpu."Zone") AS "pick\_up\_loc",

CONCAT(zdo."Borough", ' / ', zdo."Zone") AS "dropoff\_loc"

FROM

yellow\_taxi\_data t,

zones zpu,

zones zdo

WHERE

t."PULocationID"=zpu."LocationID" AND

t."DOLocationID"=zdo."LocationID"

LIMIT 100;

Hence, we can see the records when these 2 IDs are matching. We can write this query differently:

SELECT

tpep\_pickup\_datetime,

tpep\_dropoff\_datetime,

total\_amount,

CONCAT(zpu."Borough", ' / ', zpu."Zone") AS "pick\_up\_loc",

CONCAT(zdo."Borough", ' / ', zdo."Zone") AS "dropoff\_loc"

FROM

yellow\_taxi\_data t JOIN zones zpu

ON t."PULocationID"=zpu."LocationID"

JOIN zones zdo

ON

t."DOLocationID"=zdo."LocationID"

LIMIT 100;

Then check for records with location ID not in the zones table:

SELECT tpep\_pickup\_datetime,

tpep\_dropoff\_datetime, total\_amount, "PULocationID", "DOLocationID"

FROM yellow\_taxi\_data WHERE “PULocationID” is NULL

Then:

SELECT tpep\_pickup\_datetime,

tpep\_dropoff\_datetime, total\_amount, "PULocationID", "DOLocationID"

FROM yellow\_taxi\_data t WHERE “DOLocationID”

NOT IN (SELECT “LocationID” FROM zones)

**LEFT Join in SQL means** that if there is a key only on the left table that you join with another table on the right, you still keep it. So after JOIN you have all the common keys plus all the extra keys that exist only in the left table.

SELECT

tpep\_pickup\_datetime,

tpep\_dropoff\_datetime,

total\_amount,

CONCAT(zpu."Borough", ' / ', zpu."Zone") AS "pick\_up\_loc",

CONCAT(zdo."Borough", ' / ', zdo."Zone") AS "dropoff\_loc"

FROM

yellow\_taxi\_data t LEFT JOIN zones zpu

ON t."PULocationID"=zpu."LocationID"

LEFT JOIN zones zdo

ON

t."DOLocationID"=zdo."LocationID"

LIMIT 100;

Now let’s **GROUPBY** a certain column and do some aggregations.

Calculate number of trips per day:

SELECT

CAST(tpep\_dropoff\_datetime AS DATE) as "day", total\_amount

FROM yellow\_taxi\_data t

LIMIT 100;

SELECT

CAST(tpep\_dropoff\_datetime AS DATE) as "day",

COUNT(1)

FROM yellow\_taxi\_data t

GROUP BY CAST(tpep\_dropoff\_datetime AS DATE)

LIMIT 100;

To order the results:

SELECT

CAST(tpep\_dropoff\_datetime AS DATE) as "day",

COUNT(1) as “count”

FROM yellow\_taxi\_data t

GROUP BY CAST(tpep\_dropoff\_datetime AS DATE)

ORDER BY “count” DESC;

**Group by multiple fields:**

SELECT

CAST(tpep\_dropoff\_datetime AS DATE) as "day", “DOLocationID”,

COUNT(1) as “count”,

MAX(total\_amount), MAX(passenger\_count)

FROM yellow\_taxi\_data t

GROUP BY

**1,2**

ORDER BY “day” DESC, “DOLocationID” DESC;

So now we can see, for each drop-off location, for each date, how many trips happened, how much money the driver made, and what was the maximum number of passengers.

# Terraform and GCP

16) is an open-source tool for provisioning infrastructure resources, similar to CloudFormation. It helps you managing configuration files in source control to maintain an ideal provisioning state for testing and production environments.

**Provisioning infrastructure** involves tasks such as installing servers, configuring networks, installing software, etc.

As we know, IaC is a framework that helps you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version and reuse. So, it is something like a Git version Control but for infrastructure.

This also helps you to manage the infrastructure with configuration files rather than through a GUI.

So, you can configure how to deploy an entire cluster of resources.

**Terraform state** is a feature that allows you to track all resources’ changes.

**Installations:**

Based on this **Terraform client** installation: <https://www.terraform.io/downloads>

Run these:

**brew tap hashicorp/tap**

**brew install hashicorp/tap/terraform**

**GCP - Cloud Provider account**:

Set up GCP with this one: <https://github.com/DataTalksClub/data-engineering-zoomcamp/blob/main/week_1_basics_n_setup/1_terraform_gcp/2_gcp_overview.md>

(Note) Google Service Categories:

Compute, Management, Networking, Storage & Databases, Big Data, Identity & Security, ML

We create a new account (or we login to an existing one) – we set-up the billing account and we create a new project. Then we set-up the IAM & Admin roles.

IAM & Admin > **Service Account**: is an account which is used by an application to make authorized API calls.

For example, you can attach a service account to a Compute Engine instance so that applications running on that instance can authenticate as the service account. Then, **you can grant the service account IAM roles to let the service account—and, by extension, applications on the instance—access Google Cloud resources.**

Once new service account has been created then we select the 3 dots in Actions > Manage keys > Create new key > and then it automatically downloads to your computer the new key.

Then set-up Google SDK.

brew install python@3.11

brew install --cask google-cloud-sdk

export GOOGLE\_APPLICATION\_CREDENTIALS="<path/to/your/service-account-authkeys>.json"

# Refresh token/session, and verify authentication

gcloud auth application-default login

Check that all is good by running:

* gcloud -v

So, now my local is ready to interact with GCP.

We need to **create 2 resources in our infrastructure with Terraform:**

* Google Cloud Storage: Data Lake
* BigQuery: Data Warehouse

**Grant IAM roles to the service account**:

We are granting 2 types of storage rules, one is for the bucket itself and one for the objects inside the bucket. By ‘Admin’ it means that it has all the ownership level positions (i.e., create, update, delete, write, read, etc.).

In production we usually use custom roles, we do not use these ones, and we associate specific permissions to specific resources.

Graphical user interface, text, application, email

Description automatically generated

**Enable APIs:**

When the local environment interacts with the cloud, it does not interact directly with the resource. So, we have some APIs as enablers of the communication.

So, one is for IAM itself and one for IAM credentials.

**Be sure you are on the right project when you grant permissions and enable APIs.**

1. Enable these APIs for your project:
   * <https://console.cloud.google.com/apis/library/iam.googleapis.com>
   * <https://console.cloud.google.com/apis/library/iamcredentials.googleapis.com>
2. Please ensure GOOGLE\_APPLICATION\_CREDENTIALS env-var is set.

export GOOGLE\_APPLICATION\_CREDENTIALS="<path/to/your/service-account-authkeys>.json"

# Create GCP Infrastructure with Terraform

17) In our terraform files, we define configurations of all our resources.

Notes for **main.tf** and **variables.tf**:

* required\_providers: This is where the terraform registry is picking publicly available providers. So later on, when we define a resource (i.e., BigQuery, etc.) it will be imported from this particular library. This is like you import a library in python and then you implement/call some functions from this library in your code.
* **provider**{} section: terraform relies on **plugins** called providers to interact with SAS providers, cloud providers, and other APIs. **This adds a set of predefined data sources and resource types.** For example, in order to implement a cloud storage bucket, there is already a predefined module definition, and it has a specification for the arguments. This is specific to each different provider.
* provider’s section - credentials: we can pass a credentials file in the credentials attribute.
* resource: is something like a physical component, e.g., a server or a storage bucket. It contains arguments, like machine sizes, disk image names, etc.
* What are the variables? Variables are something equivalent to constants. They are generally passed at runtime. The mandatory runtime arguments do NOT have a default value in their declaration.
* They recommend in the training to use the same region for all your resources. If you have some resources in one region and some other resources in another region, when these resources will cross-communicate that will cost some money. However, within regional communication, it is cheaper or even free.

(Once you create the 2 .tf files may your editor suggest you install plugins for this file extension. Accept and install them)

**Terraform commands**:

* terraform **init**: initialize & install all plugins (e.g., the google provider).
* terraform **plan**: match changes against the previous state.
* terraform **apply**: apply changes to the cloud.
* terraform **destroy:** remove your stack from the cloud.

Once you **run the init command** you see some new files in your system:

* The .terraform is like a package manager like conda or pip. It basically packages all of our code in a more compressed fashion. If you go into that directory, you see the terraform states and the providers.

Once you **run the plan command,** we see the only mandatory variable that we defined (the one with no default values). A variable for a project variable needs to be mandatory because we do not want everyone to overlap each other’s configurations if there are shared resources. Hence, it is better to use your own project, and this is why we can make the project variable a mandatory variable, so as to pass the project id at runtime. Therefore, it asks for the project id, so copy-paste it from GCP to the command line. And here you go! You now see the actions or else the changes that will be applied. Hence, we confirm the 2 new resources that will be created.

Now that we approved the changes, let’s apply them! Run the next command for terraform apply.

If you encounter authentication problems, you might need to run this:

gcloud auth application-default login

Then run again all commands.

Graphical user interface, text, application

Description automatically generated

Now we see in GCP our new resources.

Graphical user interface, text, application

Description automatically generated

**Now we saw how to create an infrastructure cluster on the cloud using Terraform.**

# **Workflow Orchestration (…Week 2…)**

18) **Data Lake**: is a central repository that holds raw, structured, semi-structured, and unstructured data. The data is stored in its native format.

There is no predefined schema, and the data is stored in a flat architecture, using object storage or Hadoop Distributed File System (HDFS).

The idea is to ingest data as quickly as possible and to make it accessible to other team members.

For faster access, you associate your data with some meta-data.

A **data lake** allows for Extract, Load, and Transform (**ELT**) processes, where data is first loaded into the data lake in its raw format, and then transformed as needed for specific use cases. This approach enables data scientists and analysts to explore and analyze the data without the need for predefining a schema.

Data lakes are more flexible in terms of data governance policies, and data access is often decentralized, allowing different teams and users to access the data for different purposes.

19) **Data Warehouse**:

A data warehouse is designed to store structured data that **has been preprocessed**, cleaned, and transformed into a schema. The data is usually stored in a relational database and is used for business intelligence (BI) reporting.

A data warehouse stores **structured data** in **tables**, using a predefined schema that defines the data types, relationships, and constraints.

A data warehouse typically involves Extract, Transform, and Load (**ETL**) processes, which transform the data from various sources into a common format and load it into a relational database. The data is then processed using SQL queries, OLAP cubes, or BI tools.

Data warehouses often have **strict data governance policies** to ensure data accuracy, consistency, and security. The data is typically controlled and managed by a central IT team.

20) **Workflow Orchestration** means governing your data flows in a way that respects orchestration roles in your business logic.

A workflow orchestration tool allows you to turn any code into a workflow that you can schedule, run, and observe.

**Workflow orchestration is taking care of the delivery. So, that really means the execution of the workflow run.** This ensures that your products are getting packaged as desired, shipped at the schedule that you specified and with the right delivery type.

Graphical user interface, text, application, chat or text message

Description automatically generated

Many products/tools have additional features that include some form of orchestration. For example, **Snowflake has a feature for a data pipeline, which is great only if you want to orchestrate Snowflake**.

But a workflow orchestration tool will give you the ability to orchestrate data flow from all the different tools you have.

Diagram

Description automatically generated with medium confidence

https://github.com/Balajirvp/DE-Zoomcamp/blob/main/Week%202/Detailed%20Week%202%20Notes.ipynb

# Prefect as our Workflow Orchestration Tool

21) We need to orchestrate the data pipeline script (ingest\_data.py) with **Prefect**. Then we create a new table in Postgres.

Theory

**Prefect** is a modern open-source **workflow orchestration tool** for coordinating all of your data tools. Orchestrate and observe your data flow using Prefect's open-source Python library. **Scheduling**, **executing**, and **visualizing your data workflows** are the main functionalities.

So, with Prefect, you can add Observability to your infrastructure.

**Main Concepts:**

1. **Task =** A unit of work in a Prefect flow. It can perform a variety of operations, such as executing a Python function, running a shell command, or querying a database.
2. **Flow =** A collection of Tasks that defines a workflow.
3. **Subflow:** In Prefect, you can create a sub-flow by using the @flow decorator to define a flow and then using the Flow class to instantiate a task from the flow. This task can then be added to another flow just like any other task.

**Orion UI:** a graphical interface for managing and visualizing Prefect flows, tasks, and runs. With the Orion UI, users can easily monitor and manage the execution of Prefect flows, view the status of individual tasks, and explore the data and outputs of completed runs.

Actions

We write python **code AS workflows**, and then it lets us build, run, and monitor our pipeline.

1. I created a new folder in my project for this 2nd week: week2/
2. Ensure that the Docker desktop app is open and running. Run manually the command docker-compose up to be able to communicate with Postgres. Connect to localhost:8888 to see the existing tables in pgAdmin.
3. Create a **venv** in python so that we can install all the relevant libraries without affecting the base environment:
   1. python3 -m venv de-zoomcamp
4. Activate it:
   1. source de-zoomcamp/bin/activate
5. Create a **requirements.txt** file that contains all the relevant libraries which we will be using for this lesson and save it in your working directory.
6. pip install -r requirements.txt
7. **Transform the python script of the data pipeline into a Prefect flow.**
   1. **Create 1 flow: main\_flow()**
   2. **Create 3 tasks:** 
      1. **Extract**
      2. **Transform**
      3. **Load**
8. Once you write your code you can run it: python3 ingest\_data\_flow.py
9. **This should successfully load the data (1 new table – we used another dataset than the one we had in week 1) into Postgres** (1st batch only for simplicity now).
10. **Parameterization and Sub-flows**
    1. Parameterization is when we make a flow run with multiple input parameters, so as to run this flow multiple times with different inputs 🡪 make the flow reusable.
    2. Subflows: make a flow a reusable component that can be used as a task within another flow.
    3. Till now we had 1 main flow and 3 tasks.
    4. We create a new flow (with a flow decorator again) and in the name attribute we call it “Subflow”
11. Orion UI:
    1. This UI hits the server and gets the data flows information that I want to display. Set the endpoint that Prefect uses to communicate with the Prefect API, which is a REST API that provides programmatic access to Prefect flows, tasks, and runs.
    2. prefect config set PREFECT\_API\_URL=http://127.0.0.1:4200/api
    3. By setting the Prefect API URL to <http://127.0.0.1:4200/api>, you are telling Prefect to use a local instance of the Prefect API running at <http://127.0.0.1:4200/api>. This is often used for whenever you want to run the Prefect API on your local machine.
    4. Start the Orion: prefect orion start
    5. Open in your browser: <http://127.0.0.1:4200/api>
12. Blocks - Create a Prefect Block to store our Postgres credentials: With blocks, you are able to **store credentials** for authenticating with any other system you'd like to orchestrate with Prefect. In other words, **blocks allow us to reuse configurations with external services**. **On a Prefect Orion API server, blocks are created in the server's database**. In our Python script ingest\_data\_flow.py instead of hard coding all of the input credentials (URL, user, password, etc.), we can create a block that can store the credentials and can be called directly. So, we can do this in the following way:
    1. In the Blocks, in Orion UI, we create a new block for our PostgreSQL Connector. So, we click on “Add Block”, and then we search and add an **SQLAlchemyConnector**, and fill in the corresponding form.

Graphical user interface, text, application

Description automatically generated

Also, port is 5432, and host is localhost.

Then we see something like this:

Graphical user interface, text, application

Description automatically generated

After the creation of the blog in the UI, we can use it in our code. Import the library for the SQL alchemy connector, and in the task where you load the dataGraphical user interface, text

Description automatically generated

# **ETL with GCP & Prefect (Flow1: Put data to GCP)**

22) **FLOW 1**: We create an ETL pipeline, download the data from the web, clean it, and save it in a parquet file locally. Then, we create blocks in Orion UI to store the GCP credentials.

- We need a new block: In the Prefect Orion UI click on "Blocks" and find the GCS Bucket Block. Add a new GCS Bucket and select Add ‘*Gcp Credentials*’.

- We need to specify the GCP credentials. So, we go to GCP in the main menu go to AIM & Admin, then go down to Service Accounts, then copy and paste the content of the key file (or if you have downloaded it first you go to your file system you find this JSON file) to the Service Account Info field.

- Back in **GCS Bucket** you select the block with the Gcp Credentials you just created. Now the dropdown is not empty.

- Select the “**Create**” button.

- Go to your code and write a new function with a @task decorator to write\_gcs(). This function will write your data to the Google Cloud Storage bucket.

- **Create a folder data/green** before you run your script.

- Run: **prefect block register -m prefect\_gcp**

- Run the pipeline script: python3 ingest\_data\_to\_gcp\_prefect\_flow.py

- Check the new table in Google Cloud storage and

- Check the flow in Orion

Graphical user interface, text, application

Description automatically generated

# **GCP to BigQuery**

23) What we have seen so far is that we have put data from the Web into Google Cloud Storage.

* We run everything inside a venv
* We orchestrate another script with **Prefect**to **get the data from GCS** and **put it into a Data Warehouse (BigQuery)**
* The main flow is **etl\_gcs\_to\_bq()**
* def**extract\_from\_gcs(),**def**transform\_data(),**def**write\_bq()**
* Find credentials information from the appropriate block in Orion
* Run script, check the new flow in Orion, and check the new data in BigQuery

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

# Deploy with Prefect the “ETL with GCP” flow.

24) Our main flow for uploading data from the web to GCS is not parametrized, since we have hard-coded values for the name of the dataset. What we need to do is to parameterize the main flow so as to avoid hard-coding values. Instead, we can pass those on at the run time.

**How the main flow looks like now**:

Text

Description automatically generated

So, we add parameters in the function of our main flow. We also make this **etl\_web\_to\_gcs()** flow as a **Subflow**. Hence, instead of hard coding one month at a time for each flow run, let’s create a parent flow that is going to trigger this flow maybe multiple X times. So, we are going to get X instances of our single flow all from one **parent flow**. Therefore, from the main function of our Python file, we should call this parent flow.

**How the PARAMETERISED main flow looks like now**:

Text

Description automatically generated

Just a reminder, this flow is built upon the other web-to-GCS flow we have already built. That flow does include utilizing that GCS block with the required credentials. This also means that we do have Orion the API and UI running.

To run the script, make sure you have started manually Orion, and you have activated your venv.

* source de-zoomcamp/bin/activate
* prefect orion start
* python3 parameterised\_etl\_web\_to\_gcs.py

Check how the flow runs in Orion.

**However, we need to schedule running the flows instead of manually triggering them.**

**=> Deployment with Prefect is what we need !!!**

A deployment in Prefect is a server-side concept that encapsulates a flow allowing it to be scheduled and triggered via the API.

**Prefect Deployment** refers to the process of deploying a Prefect workflow to a production environment so that it can be run on a schedule or triggered by external events. There are several ways to deploy a Prefect workflow, depending on the specific requirements of your use case.

Here are some of the common deployment methods for Prefect workflows:

**Prefect Cloud**: Prefect Cloud is a managed service provided by OpenAI that allows you to run Prefect workflows in a scalable, secure, and managed environment. Prefect Cloud provides a web-based interface for managing workflows, as well as a REST API for programmatic access.

**Prefect Server**: Prefect Server is a self-hosted platform that provides the same functionality as Prefect Cloud, but it can be run on-premise or in your own cloud environment. Prefect Server provides a web-based interface for managing workflows, as well as a REST API for programmatic access.

**Prefect CLI**: The Prefect CLI can be used to run Prefect workflows from the command line. This is useful for testing workflows during development or for running workflows on a schedule using a task scheduler like cron or Windows Task Scheduler.

**Prefect API**: The Prefect API can be used to start and manage Prefect workflows programmatically. This is useful for integrating Prefect workflows into other systems or for triggering workflows based on external events.

In general, the deployment method you choose will depend on the specific requirements and constraints of your use case. For example, if you need to run Prefect workflows in a secure, managed environment, you may want to use Prefect Cloud or Prefect Server. If you need to run workflows on a schedule or integrate workflows into other systems, you may want to use the Prefect CLI or the Prefect API.

**Create Deployment Package:**

**Let’s see how to do this with CLI (command-1) :**

prefect deployment build parameterized\_etl\_web\_to\_gcs.py:etl\_parent\_flow -n "Parameterized ETL"

**This command is used to build a Prefect workflow deployment**. It takes a Python file and a workflow object as arguments and in the end, we specify a name for the deployment. This workflow object is actually the entry point to the flow. In the screenshot above you can see that we have 2 flows: the **etl\_parent\_flow()** and the **etl\_web\_to\_gcs()**. So, we need to tell which one is the entry flow, i.e., from which flow it should start.

Once the build command is executed, Prefect will create a deployment package that can be deployed to a Prefect Cloud instance or Prefect Server. This package contains all the necessary information to run the workflow, including the workflow definition, dependencies, etc.

Deployment package 🡪 deploy it to a Prefect Cloud instance or Prefect Server with API/CLI.

We deploy the workflow with Prefect so as to avoid manual steps:

We ran the above command (1) and **this command outputs a yaml** file with all **meta-data** that the deployment needs to know. We specify the parameters {} (For example, we gave the following parameters in the yaml file: parameters: {"color": "yellow", months:[1,2,3], "year":2021}). And then we **run this command**: **prefect deployment apply** etl\_parent\_flow-deployment.yaml

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, application

Description automatically generated

Then, we trigger a **Quick Run from Orion UI**, and after triggering a quick run, it will have a "Scheduled" state under "Flow runs". To deploy this workflow for execution, we need an agent. So, this flow will be triggered now from the UI and not manually from us.

Agent: picks up scheduled workflow runs from Work Queues.

Work Queue: a **data structure** that **holds tasks** that are ready to be executed. It acts as a **buffer** between the Prefect Core and the agents, allowing the Core to manage tasks while the agents execute them. Prefect Core is the whole backend open-source platform for Prefect.

We **launch an agent** with the following command. The agent will automatically run our scheduled workflow:

**prefect agent start --work-queue "default"**

**Graphical user interface, text, application, email, Teams

Description automatically generated**

**Graphical user interface, application, Teams

Description automatically generated**

**That was it !!!**

**Trigger flow runs from Prefect deployments, executing them with an Agent pulling from a Work Queue.**

More info:

https://github.com/Balajirvp/DE-Zoomcamp/blob/main/Week%202/Detailed%20Week%202%20Notes.ipynb

# **Schedules & Docker Storage with Infrastructure**

## **Run Prefect Flows on Docker Containers**

25) Scheduling Flows and Running flows in Containers in Docker

If we go and check our deployments in Orion, we can see that we can schedule the deployment (click the button ‘Add’). We can have an Interval Scheduler (i.e., deploy every 5 minutes, or every 10 seconds, etc.). We can also have a Cron Scheduler (i.e., deploy every minute on day 3 of the month).

Graphical user interface, application

Description automatically generated

Instead of scheduling the deployment manually, we can do the scheduling when creating Deployments (via CLI).

Graphical user interface

Description automatically generated

So, if we schedule something manually, our deployment will be run according to our rule as long as we have an agent to pick it up.

Run:

prefect deployment build parameterized\_etl\_web\_to\_gcs.py:etl\_parent\_flow -n etl-with-cron -cron "0 0 \* \* \*" -a

Graphical user interface, application

Description automatically generated

You can pause the deployments; you do not have to run this at a regular base.

You can check the docs with “*prefect deployment –help”* to see how to schedule or pause deployments from CLI (i.e., set-schedule for a given deployment).

## **Run Prefect Flows on Docker Containers**

There are a lot of options where we can store our code, i.e., GitHub, Azure, Gitlab, etc.

We will store our code in a docker image, and put it up on Docker Hub, and then we just run a docker container and the code will be right there.

We can use Docker images to make our workflows production ready. First, we write a Dockerfile for our workflow, then login to Docker Hub, create an image repository, and finally push the image.

* Sign up/or else Sign in in Docker Hub (use your username there).
* Create a repo there online
* Build and push the image there in Docker Hub:

**docker image build --no-cache -t xenie1/de-zoomcamp .**

* Then run: docker login
* Push the image to Docker hub: **docker image push xenie1/de-zoomcamp**

Graphical user interface, text, application, email

Description automatically generated

Verify it via Docker Desktop.

A screenshot of a computer

Description automatically generated with medium confidence

So, till now we got all the code we need in Docker.

Create a Docker Block in Orion UI. We go to Orion 🡪 Blocks 🡪 Docker Container (click Add) 🡪

Graphical user interface, text, application

Description automatically generated

# Make Deployment with Docker and Python file

**Prefect deployment using Python:** write the Python file docker\_deploy.py

Load this new block for Docker Container (de-zoomcamp) in the new python file and then create an instance of this new deployment where you specify your flow name ‘docker-flow’ for example is what we call it.

Then run:

python docker\_deploy.py

🡪 Check in Orion UI this new docker-flow !!!

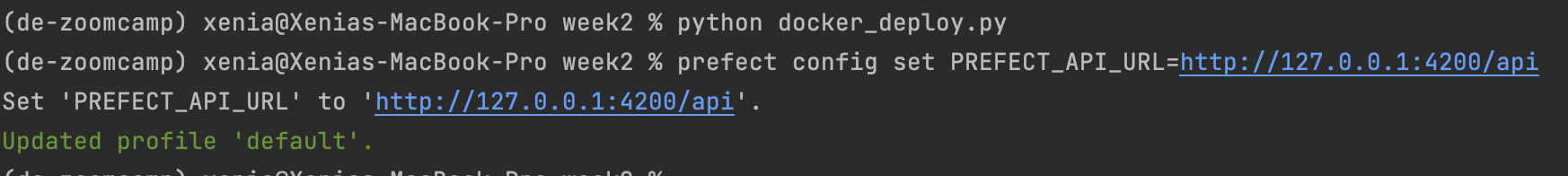
Graphical user interface, application

Description automatically generated

Use a local Orion API server:

prefect config set PREFECT\_API\_URL=http://127.0.0.1:4200/api

🡪 Now the Docker Container is able to interface with the Orion server.



Create and launch Prefect Agent and run workflow from CLI:

prefect agent start -q default

prefect deployment run etl-parent-flow/docker-flow -p "months=[1,2]"

**Notice that this runs in a container and not locally as before !!!**

So, go to Orion UI in Deployments and watch what is happening.

Graphical user interface, application

Description automatically generated

26) Todo

27) Todo

28) Todo

29) Todo

30) Todo