Cartelis Follow

Nov 22, 2021 · 9 min read · Listen

Launching a docker-based modern open-source data stack

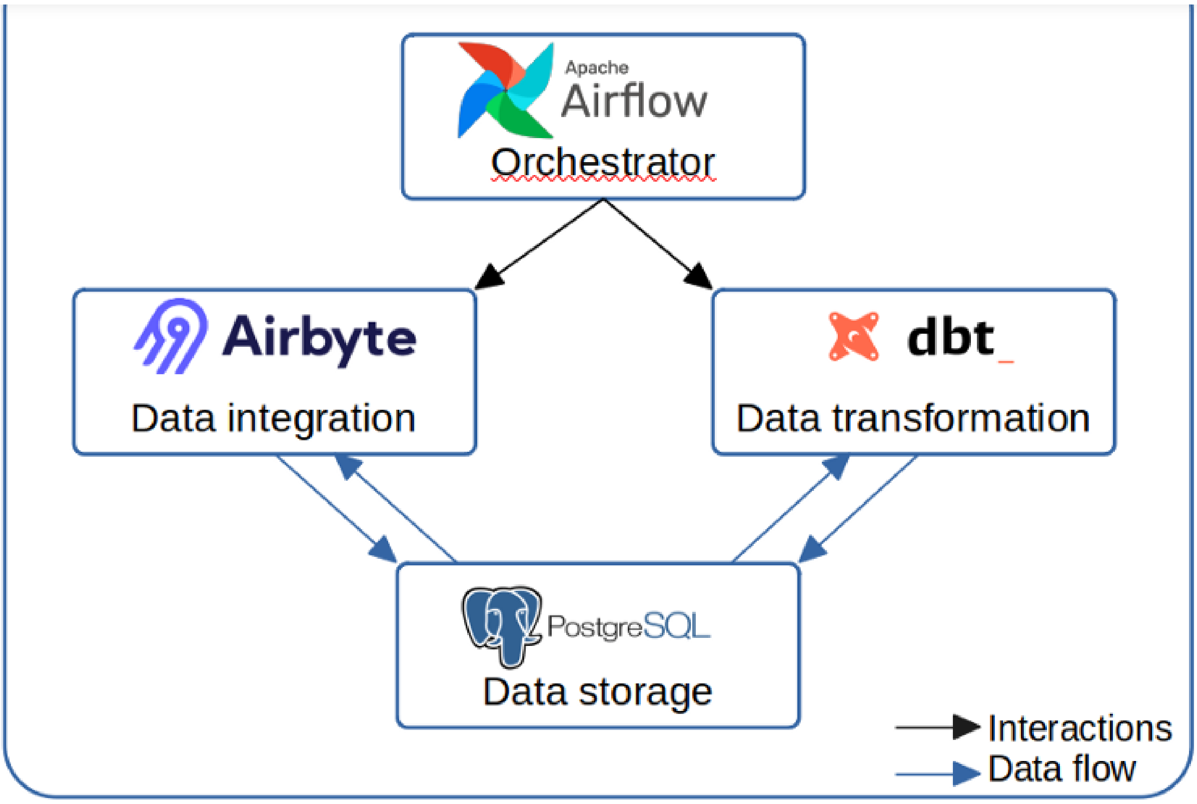
Introduction

Context

We previously presented a way to use several different open-source solutions to create a minimal, modern data stack capable of orchestrating data integrations and data manipulation tasks. In a data ecosystem where one might find sophisticated data integration tools at an understandably high cost, open-source solutions are a welcome alternative for small businesses, freelance analysts or even academic researchers, for whom budgets may significantly limit their capacity to implement more costly solutions.

Thus, the implementation of this open-source stack leverages some of the best existing solutions:

- Airflow. task orchestration
- Airbyte: data integration
- PostgreSQL: data storage
- dbt: data manipulation



Proposed data stack structure

However, for those having read Part I, the stack we built suffered three rather glaring drawbacks:

1- Complex installation:

Building the stack from scratch required the installation of four separate solutions with the possibility of encountering errors in any one of them made it a difficult process for novices, and potentially for experienced individuals unfamiliar with the suggested solutions.

2- Version dependence:

Linked to the previous point is the compatibility of each solution with each other and other software on the host system which can change depending on individual software updates.

3- Operating system dependence:

In this case, the stack should be just as easily installed on other operating systems without having to dig into the minutiae of different installation processes.

This article proposes to solve these issues by leveraging yet another open-source framework: Docker.

Installation back-end

The solution we propose is to implement this stack such as to make it make OS-independent and resilient to future framework development by relying on dockerisation.

Each of these services, with the exception of dbt, will be installed in a separate Docker container from which it will run and interact with selected local folders and other containers.

Installation

Requirements

Our stack can be launched locally or remotely provided the instance satisfies the following requirement

- Cores: 2
- RAM: 4GB (recommended minimum RAM for the latest version of Airflow)
- Memory: 12GB minimum (without accounting for downloaded data with Airbyte)
- Docker (https://docs.docker.com/engine/install/ubuntu/) and Docker Compose (https://docs.docker.com/compose/install/) installed
- Python 3.6+ (not including 3.9, for compatibility with dbt)

The rest of the installation process is done either in a local terminal or on a webserver when indicated.

While there may be no intensive programming involved in this tutorial, a good

Project structure

NB :

- **While this implementation was originally meant to be hosted on a remote server, it will work on a local computer.**

The project can be cloned from Gitlab

```
git clone https://gitlab.com/nicolas217/modern-open-source-data-
stack.git
```

The root folder contains :

- setup.sh: the main installation Shell script which installs Docker, its requirements, and launches the main containers
- setup_jupyter.sh: an installation Shell script which launches a container hosting a python environment and jupyter for script editing
- setup_metabase.sh: an installation Shell script which launches a container hosting a Metabase instance for data visualisation
- docker: folder containing the required files to launch docker containers, notably docker-compose.yaml, dbt models for data manipulation and DAG files for task orchestration

Launching the installation

The installation process is launched by executing the main set up Shell script at the root of the cloned repository:

- Access the data-integration folder if not already in it:

```
cd data-integration
```

- It can then be launched :

```
./setup.sh
```

It can take a little time (5 minutes is normal on weaker servers) for the entire set up to be installed and running. If you wish to follow the progress, you can replace `-d` in the last line of the setup.sh file with `-it` . This will launch docker-compose in an interactive mode which will print out to progress of the installation to the terminal.

Docker-compose settings

While a one-click installation tool is convenient, it is useful to understand the underlying processes and settings if customization is required to fit with a pre-existing system.

The core of the installation is the docker-compose.yaml located in the `docker` folder. The contents can be split into four main parts:

- Docker format:

```
version:                                         "3.7"
```

gistfile1.txt hosted with ❤ by GitHub view raw

This instruction indicates to the Docker engine the version of the docker-compose file, which affects how it is interpreted and executed.

- Extension fields:

```
x-logging:                                       &default-logging
  options:
    max-size:                                    "100m"
    max-file:                                    "5"
  driver:                                        json-file

```

```
    AIRFLOW__CORE__EXECUTOR:                    CeleryExecutor
    AIRFLOW__CORE__SQL_ALCHEMY_CONN:            postgresql+psycopg2://airflow:airflow@postgres/
    AIRFLOW__CELERY__RESULT_BACKEND:            db+postgresql://airflow:airflow@postgres/airflo
    AIRFLOW__CELERY__BROKER_URL:                redis://:@redis:6379/0
    AIRFLOW__CORE__FERNET_KEY:                  ''
    AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION: 'true'
    AIRFLOW__CORE__LOAD_EXAMPLES:               'true'
    AIRFLOW__API__AUTH_BACKEND:                 'airflow.api.auth.backend.basic_auth'
  volumes:
    - ./dags:/opt/airflow/dags
    - ./logs:/opt/airflow/logs
    - ./plugins:/opt/airflow/plugins
    - ./dbt:/opt/airflow/dbt

  user:                                         "${AIRFLOW_UID:-50000}:${AIRFLOW_GID:-50000}"
  depends_on:
    redis:
      condition:                                service_healthy
    postgres:
      condition:                                service_healthy
```

Extensions enable the creation of settings and images which can then be used by subsequent services without having to copy them repeatedly.

In the previous code, we define two such extensions: Airbyte with explicitly named parameters and Airflow on the basis of a file named Dockerfile in the same folder as docker-compose.yml.

- Services:

```
services:
  # Airbyte services
  init:
    image:                                      airbyte/init:${VERSION}
    logging:                                    *default-logging
    container_name:                             init
    command:                                    /bin/sh -c "./scripts/create_mount_directories
    environment:
      - LOCAL_ROOT=${LOCAL_ROOT}
    logging:                                    *default-logging
    container_name:                             airbyte-db
    restart:                                    unless-stopped
    environment:
      - POSTGRES_USER=${DATABASE_USER}
      - POSTGRES_PASSWORD=${DATABASE_PASSWORD}
    volumes:
      - db:/var/lib/postgresql/airbyte
  seed:
    image:                                      airbyte/seed:${VERSION}
    container_name:                             airbyte-data-seed
    # Pre-populate the volume if it is empty.
    # See:                                      https://docs.docker.com/storage/volumes/#popul
    volumes:
      - data:/app/seed
  scheduler:
    image:                                      airbyte/scheduler:${VERSION}
    logging:                                    *default-logging
    container_name:                             airbyte-scheduler
    restart:                                    unless-stopped
    environment:
      - WEBAPP_URL=${WEBAPP_URL}
      - WAIT_BEFORE_HOSTS=5
      - WAIT_HOSTS_TIMEOUT=45
      - WAIT_HOSTS=${DATABASE_HOST}:${DATABASE_PORT}
      - DATABASE_USER=${DATABASE_USER}
      - DATABASE_PASSWORD=${DATABASE_PASSWORD}
      - DATABASE_URL=${DATABASE_URL}
      - WORKSPACE_ROOT=${WORKSPACE_ROOT}
      - WORKSPACE_DOCKER_MOUNT=${WORKSPACE_DOCKER_MOUNT}
      - LOCAL_ROOT=${LOCAL_ROOT}
      - LOCAL_DOCKER_MOUNT=${LOCAL_DOCKER_MOUNT}
      - CONFIG_ROOT=${CONFIG_ROOT}
      - TRACKING_STRATEGY=${TRACKING_STRATEGY}
      - AIRBYTE_VERSION=${VERSION}
      - AIRBYTE_ROLE=${AIRBYTE_ROLE:-}
      - TEMPORAL_HOST=${TEMPORAL_HOST}
      - S3_LOG_BUCKET=${S3_LOG_BUCKET}
      - S3_LOG_BUCKET_REGION=${S3_LOG_BUCKET_REGION}
      - AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
      - AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
    volumes:
```

```
    image:                                      airbyte/server:${VERSION}
    logging:                                    *default-logging
    container_name:                             airbyte-server
    restart:                                    unless-stopped
    environment:
      - WEBAPP_URL=${WEBAPP_URL}
      - WAIT_BEFORE_HOSTS=5
      - WAIT_HOSTS_TIMEOUT=45
      - WAIT_HOSTS=${DATABASE_HOST}:${DATABASE_PORT}
      - DATABASE_USER=${DATABASE_USER}
      - DATABASE_PASSWORD=${DATABASE_PASSWORD}
      - DATABASE_URL=${DATABASE_URL}
      - WORKSPACE_ROOT=${WORKSPACE_ROOT}
      - CONFIG_ROOT=${CONFIG_ROOT}
      - TRACKING_STRATEGY=${TRACKING_STRATEGY}
      - AIRBYTE_VERSION=${VERSION}
      - AIRBYTE_ROLE=${AIRBYTE_ROLE:-}
      - TEMPORAL_HOST=${TEMPORAL_HOST}
      - S3_LOG_BUCKET=${S3_LOG_BUCKET}
      - S3_LOG_BUCKET_REGION=${S3_LOG_BUCKET_REGION}
      - AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
      - AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
    ports:
      - 8001:8001
    volumes:
      - workspace:${WORKSPACE_ROOT}
      - data:${CONFIG_ROOT}
  webapp:
    image:                                      airbyte/webapp:${VERSION}
    logging:                                    *default-logging
    container_name:                             airbyte-webapp
    restart:                                    unless-stopped
    ports:
      - 8000:80
    environment:
      - AIRBYTE_ROLE=${AIRBYTE_ROLE:-}
      - AIRBYTE_VERSION=${VERSION}
      - API_URL=${API_URL:-}
      - IS_DEMO=${IS_DEMO:-}
      - PAPERCUPS_STORYTIME=${PAPERCUPS_STORYTIME:-}
      - FULLSTORY=${FULLSTORY:-}
      - TRACKING_STRATEGY=${TRACKING_STRATEGY}
      - INTERNAL_API_HOST=${INTERNAL_API_HOST}
    ports:
      - 7233:7233
    environment:
      - DB=postgresql
      - DB_PORT=${DATABASE_PORT}
      - POSTGRES_USER=${DATABASE_USER}
      - POSTGRES_PWD=${DATABASE_PASSWORD}
      - POSTGRES_SEEDS=${DATABASE_HOST}
      - DYNAMIC_CONFIG_FILE_PATH=config/dynamicconfig/development.yaml
    volumes:
      - ./temporal/dynamicconfig:/etc/temporal/config/dynamicconfig

  # Backend database
  postgres:
      image:                                    postgres:13
      environment:
        POSTGRES_USER:                          airflow
        POSTGRES_PASSWORD:                      airflow
        POSTGRES_DB:                            airflow
      volumes:
        - postgres-db-volume:/var/lib/postgresql/data
      healthcheck:
        test:                                   ["CMD", "pg_isready", "-U", "airflow"]
        interval:                               5s
        retries:                                5
      restart:                                  always
      ports:
              - 5400:5432

  # Airflow services
  redis:
      image:                                    redis:latest
      container_name:                           airflow-redis
      ports:
        - 6379:6379
      healthcheck:
        test:                                   ["CMD", "redis-cli", "ping"]
        interval:                               5s
        timeout:                                30s
        retries:                                50
      restart:                                  always
  airflow-webserver:
      <<:                                       *airflow-common
      healthcheck:
        test:              ["CMD", "curl", "--fail", "http://localhost:80
        interval:          10s
        timeout:           10s
        retries:           5
      restart:             always
  airflow-scheduler:
      <<:                  *airflow-common
      container_name:      airflow-scheduler
      command:             scheduler
      healthcheck:
        test:              ["CMD-SHELL", 'airflow jobs check --job-type S
        interval:          10s
        timeout:           10s
        retries:           5
      restart:             always
  airflow-worker:
      <<:                  *airflow-common
      #build:              .
      container_name:      airflow-worker
      command:             celery worker
      healthcheck:
        test:
          - "CMD-SHELL"
          - 'celery --app airflow.executors.celery_executor.app inspect ping -d "celery@$${HOS
        interval:          10s
        timeout:           10s
        retries:           5
      restart:             always

  airflow-init:
      <<:                  *airflow-common
      container_name:      airflow-init
      command:             version
      environment:
        <<:                *airflow-common-env
```

The contents of the services configure the required containers for the three main solutions we set up here: Airbyte, Airflow, and a PostgreSQL database. The file provides the following container details:

1- Container image

4- Container health checks

5- Environment variables

6- Port mapping

- Mounted volumes:

```
volumes:
  workspace:
     name:                                         ${WORKSPACE_DOCKER_MOUNT}
  data:
     name:                                         ${DATA_DOCKER_MOUNT}
  db:
     name:                                         ${DB_DOCKER_MOUNT}
  postgres-db-volume:
```

gistfile1.txt hosted with ❤ by GitHub view raw

The remaining yaml code details which volumes are mounted onto the containers so that local files are available within the containers both for reading and writing.

Installing dbt

The T(ransformation) role in our ETL setup is played by dbt and it needs to be installed on the only container which will actually handle data transformation : airflow-worker.

To access the airflow-worker container:

- Print out the different running Docker containers:

```
sudo docker ps
```

- Identify the container ID of the relevant container and access it:

```
sudo docker exec -it <container-id> /bin/bash
```

```
pip install dbt==0.19.0 # (there is a newer version but we work with
this one at the time of writing)
```

- Install dbt requirements:

```
sudo apt-get install git libpq-dev python-dev python3-pip
sudo apt-get remove python-cffi
sudo pip install — upgrade cffi
pip install cryptography~=3.4
```

Creating an Airbyte connection

As previously, Airbyte is our data integration solution and, at this stage, still requires manual configuration.

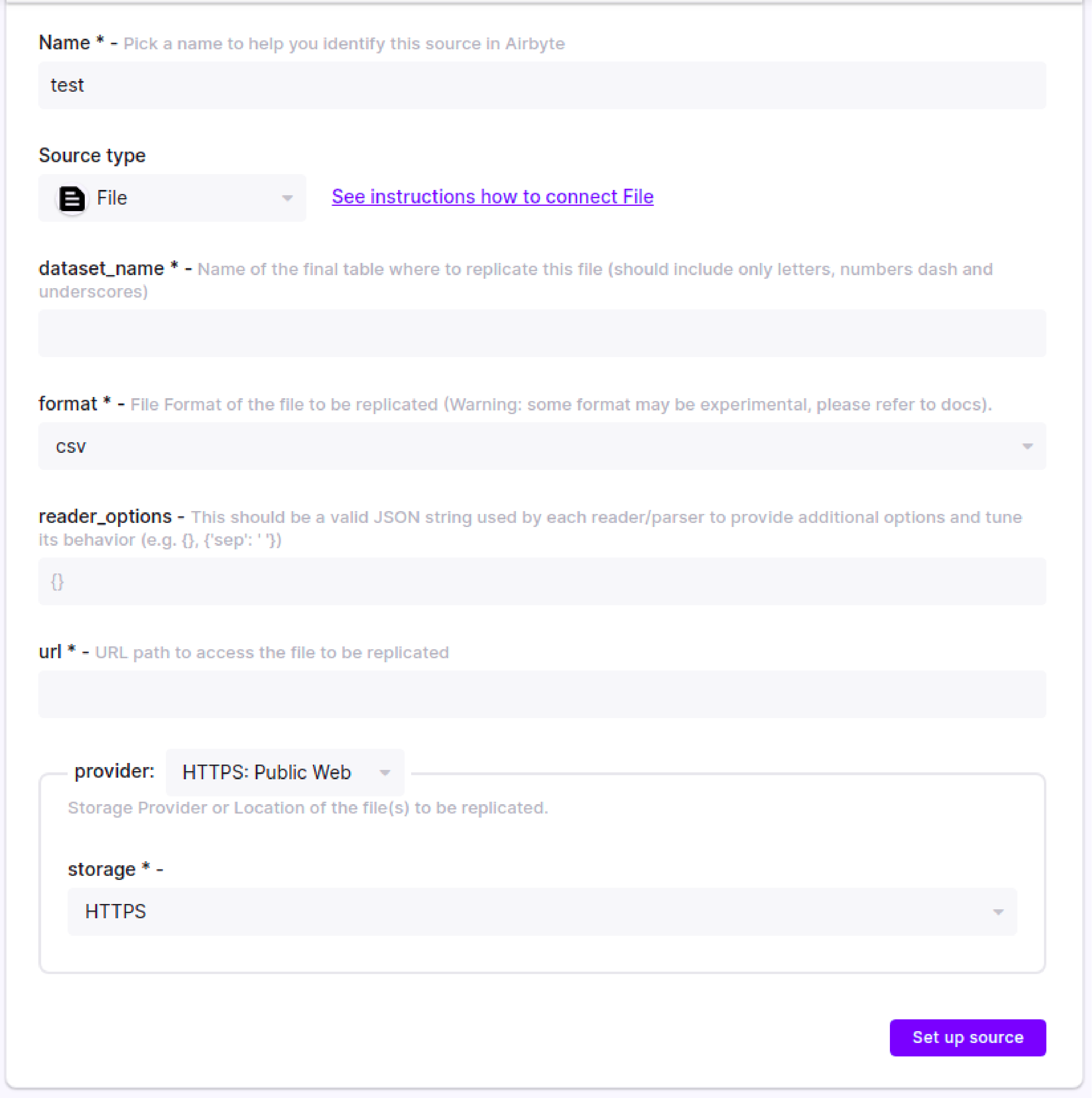
1- Access the Airbyte webserver: `<IP address>:8000` in a browser or `localhost:8000` for a local installation

2- Sign up

3- Skip the onboarding

4- Select "new source" from the Sources tab

5- Enter a name and choose a source type, *File* in our example et wait for the corresponding settings fields to load



Name * - Pick a name to help you identify this source in Airbyte

test

Source type

File

See instructions how to connect File

dataset_name * - Name of the final table where to replicate this file (should include only letters, numbers dash and underscores)

format * - File Format of the file to be replicated (Warning: some format may be experimental, please refer to docs).

csv

reader_options - This should be a valid JSON string used by each reader/parser to provide additional options and tune its behavior (e.g. {}, {'sep': ' '})

{}

url * - URL path to access the file to be replicated

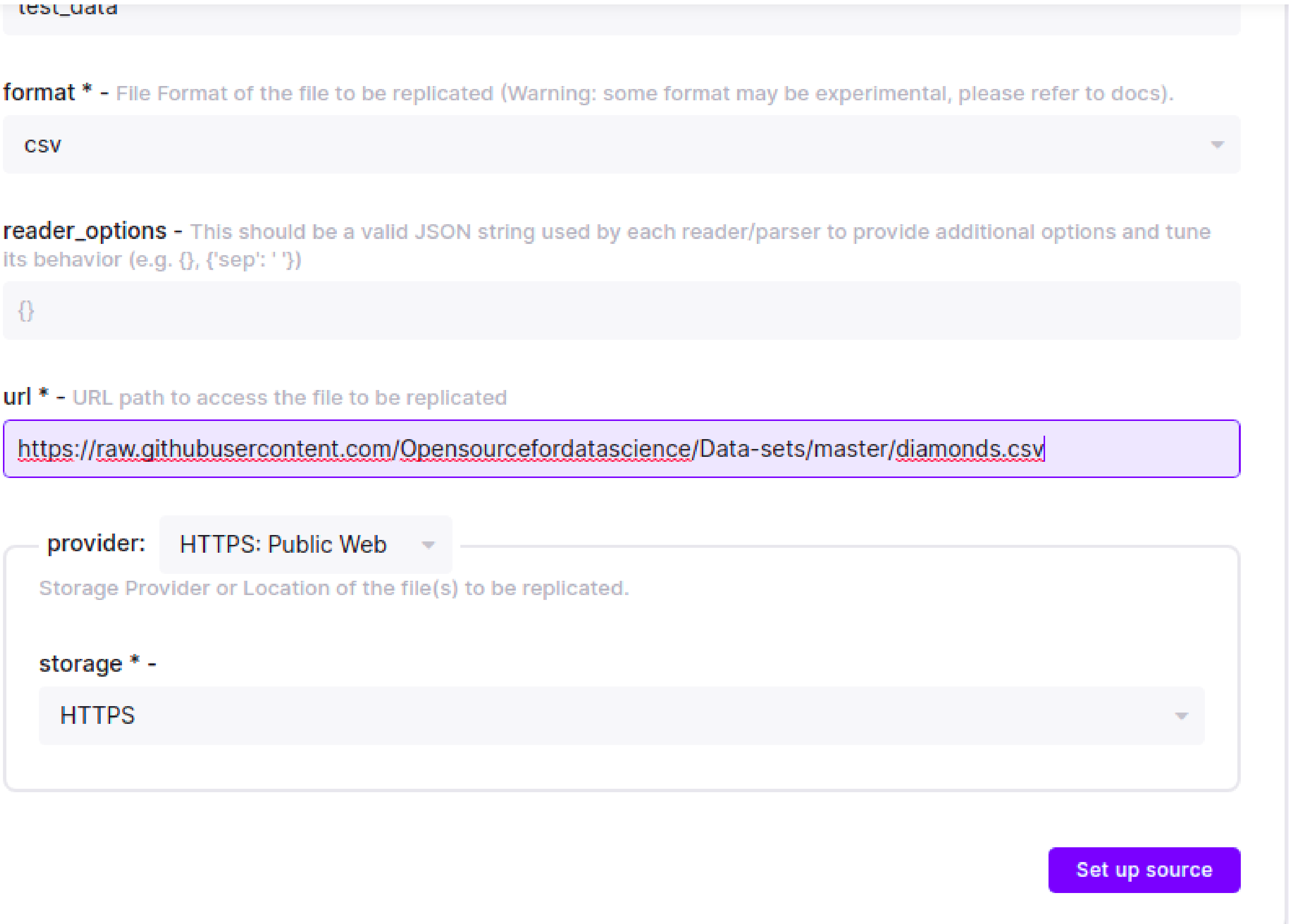
provider: HTTPS: Public Web

Storage Provider or Location of the file(s) to be replicated.

storage * -

HTTPS

6- Fill in the fields, namely the dataset name and the source url, in our example: https://raw.githubusercontent.com/Opensourcefordatascience/Data-sets/master/diamonds.csv



test_data

format * - File Format of the file to be replicated (Warning: some format may be experimental, please refer to docs).

csv

reader_options - This should be a valid JSON string used by each reader/parser to provide additional options and tune its behavior (e.g. {}, {'sep': ' '})

{}

url * - URL path to access the file to be replicated

https://raw.githubusercontent.com/Opensourcefordatascience/Data-sets/master/diamonds.csv

provider: HTTPS: Public Web

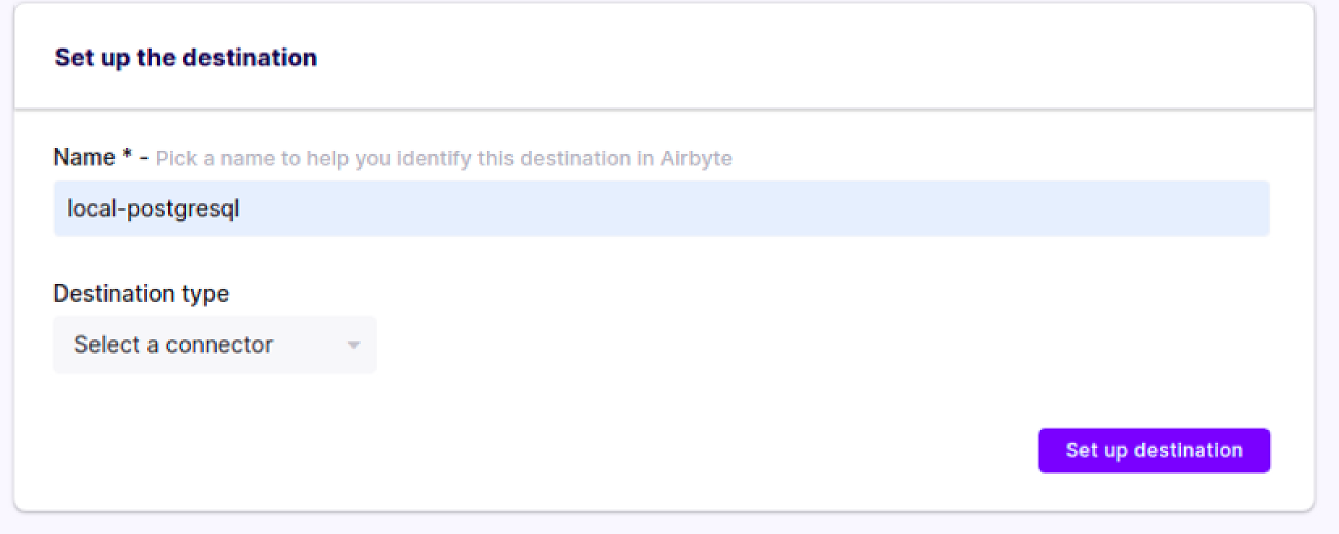
Storage Provider or Location of the file(s) to be replicated.

storage * -

HTTPS

7- Go to the Destination tab of the Airbyte webserver and select New Destination

8- Fill in a name an select a connector type, Postgres in our example, and again wait for the relevant setting fields to load



Set up the destination

Name * - Pick a name to help you identify this destination in Airbyte

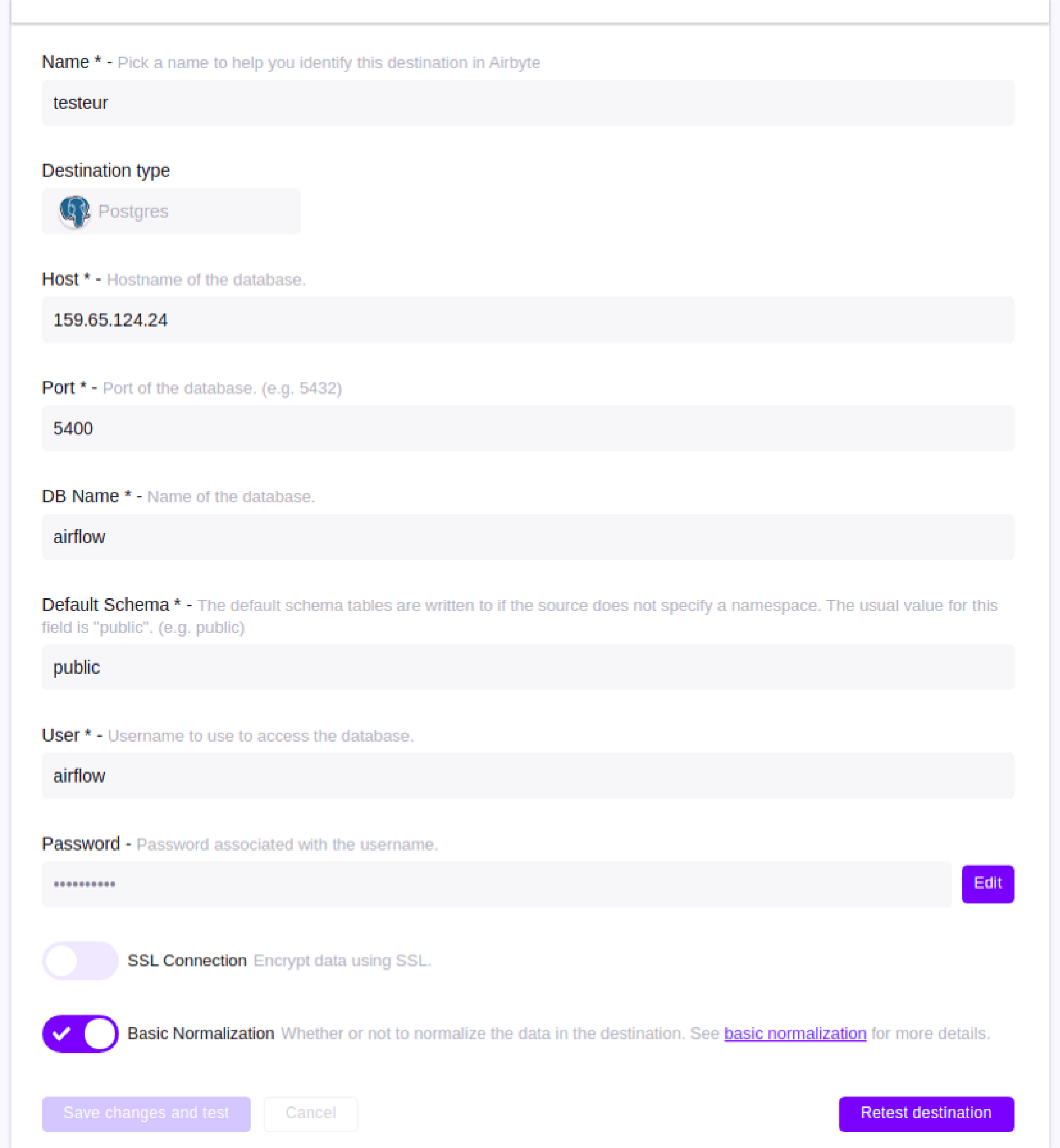
local-postgresql

Destination type

Select a connector

User: airflow

Password: airflow



Name * - Pick a name to help you identify this destination in Airbyte

testeur

Destination type

Postgres

Host * - Hostname of the database.

159.65.124.24

Port * - Port of the database. (e.g. 5432)

5400

DB Name * - Name of the database.

airflow

Default Schema * - The default schema tables are written to if the source does not specify a namespace. The usual value for this field is "public". (e.g. public)

public

User * - Username to use to access the database.

airflow

Password - Password associated with the username.

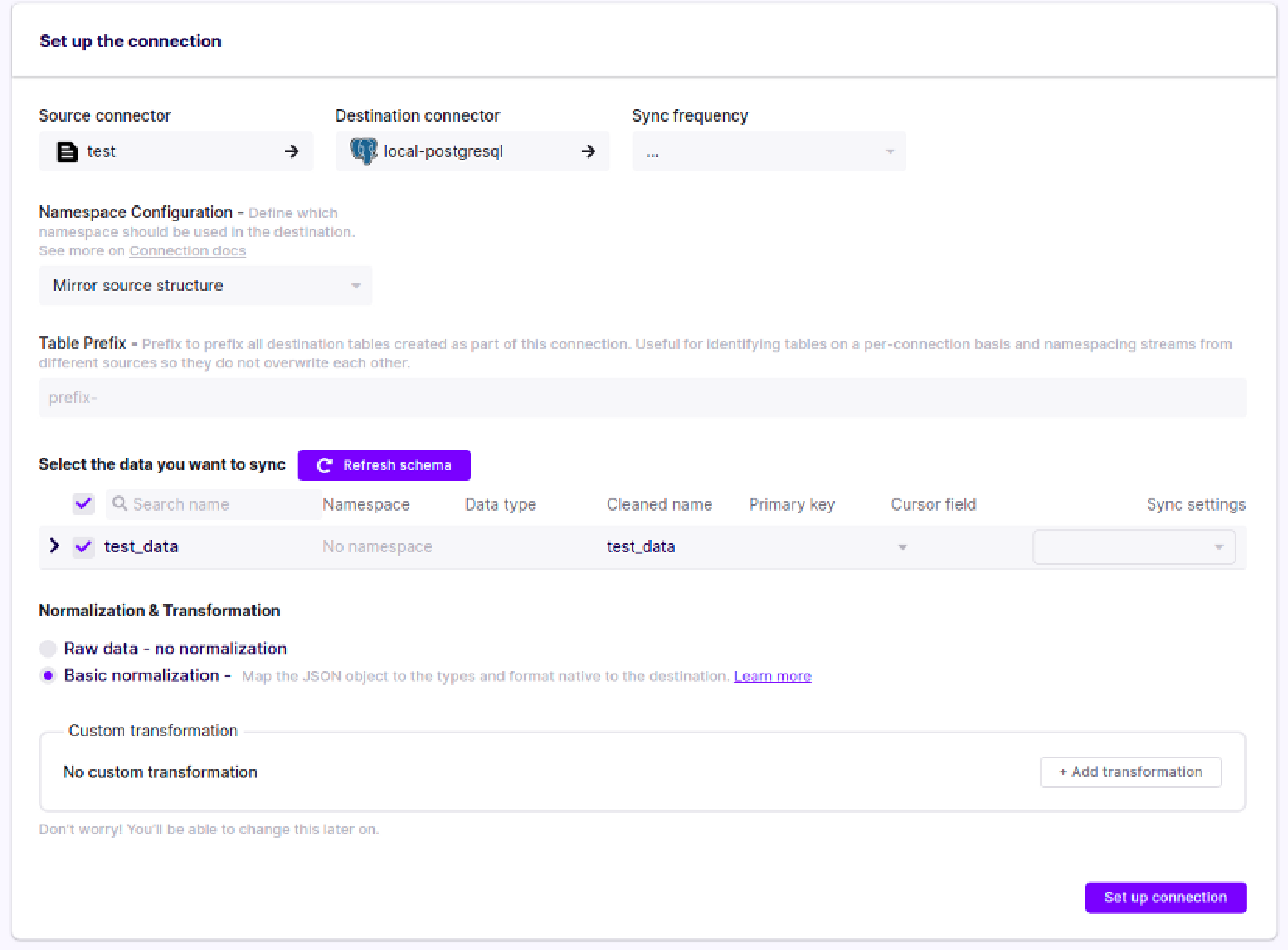
•••••••••••

○ SSL Connection Encrypt data using SSL.

● Basic Normalization Whether or not to normalize the data in the destination. See basic normalization for more details.

10- To create a connection between a source and destination, select the newly created source in the Sources tab, select Add a Destination and choose the Postgres destination recently created.

Sync settings: Fully refresh — Overwrite (for this example)



Set up the connection

Source connector: test

Destination connector: local-postgresql

Sync frequency: ...

Namespace Configuration - Define which namespace should be used in the destination. See more on Connection docs

Mirror source structure

Table Prefix - Prefix to prefix all destination tables created as part of this connection. Useful for identifying tables on a per-connection basis and namespacing streams from different sources so they do not overwrite each other.

prefix-

Select the data you want to sync Refresh schema

☑	Search name	Namespace	Data type	Cleaned name	Primary key	Cursor field	Sync settings
☑	test_data	No namespace		test_data			

Normalization & Transformation

○ Raw data - no normalization

● Basic normalization - Map the JSON object to the types and format native to the destination. Learn more

Custom transformation

No custom transformation

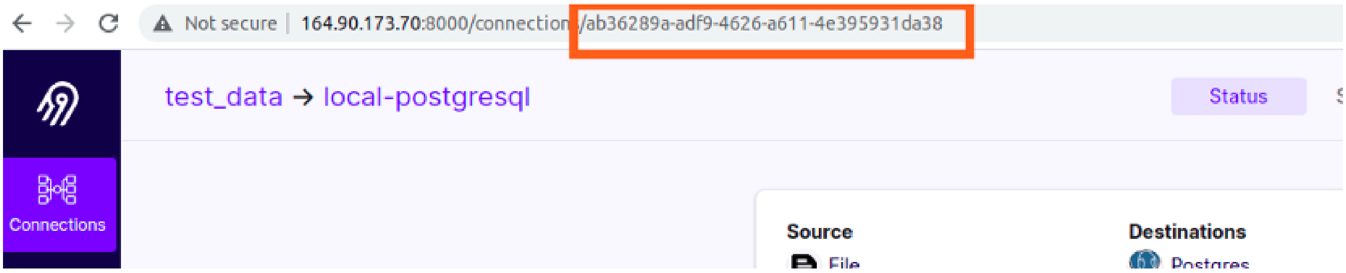
+ Add transformation

Don't worry! You'll be able to change this later on.

Set up connection

12- Set up the connection

13- Save the connection id located in the page url



Not secure | 164.90.173.70:8000/connection/ab36289a-adf9-4626-a611-4e395931da38

test_data → local-postgresql

14- Run a manual synchronisation by selecting Sync to test it

Link Airflow to Airbyte

In the Airflow web browser:

1- Select Connections in the Admin tab of the top menu

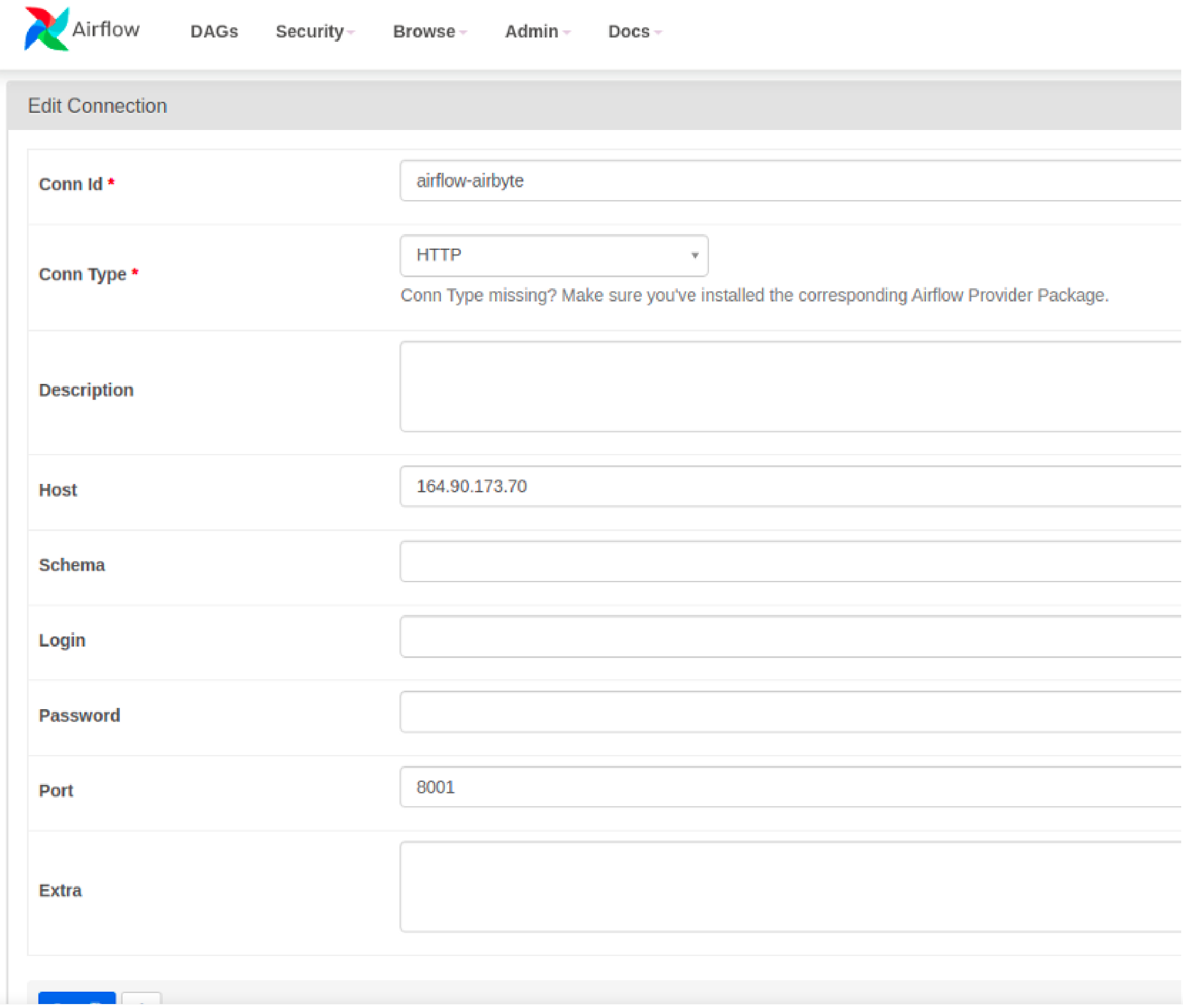
2- Create a new connection with the "plus" sign

3- Name the Conn Id "airflow-airbyte" for our example

4- Provide your server IP to the Host field

5- The Airbyte worker port is 8001 in our example

6- Save the connection



Edit Connection

Conn Id *	airflow-airbyte
Conn Type *	HTTP Conn Type missing? Make sure you've installed the corresponding Airflow Provider Package.
Description	
Host	164.90.173.70
Schema	
Login	
Password	
Port	8001
Extra	

```
sudo docker exec -it <container_id> /bin/bash
```

2- Go to the dbt folder for projects which is mounted onto containers:

```
cd /opt/airflow/dbt
```

3- Initiate a new dbt project: `dbt init test_project`

It will create two files:

- profiles.yml (at /root/) which contains the different dbt configurations needed by the dbt projects created
- dbt_project.yml (at /opt/airflow/dbt/test_project/) which contains the configuration specific to the dbt project

Edit the files as follows

- profiles.yml: edit the code and fill it in as follows for a Postgres database, as in our example

Modify the host with your server IP address.

- dbt_project.yml: edit the code and fill in as follows:

Create a test dbt model:

1- Access the folder containing models :

```
cd /opt/airflow/dbt/test_project/models
```

2- Create a new model:

```
touch count_by_cut.sql
```

3- Enter a simple query :

```
SELECT COUNT(*) FROM test_data GROUP BY cut
```

Schedule a task

A template DAG file is already available, cartelis-test which contains two tasks:

- Airbyte data synchronisation
- dbt model run

To run, the python dag file has to be updated with the airflow-airbyte connection created previously.

1- Out of the container, access the folder containing dags:

```
cd ~/data-integration/docker/dags/
```

2- Edit the file test_dag.py as follows:

The only change to make is to replace the airbyte_connection_id parameter in the AirbyteTriggerSyncOperator function with the long ID saved when you created the airbyte connexion between the file and our PostgreSQL data warehouse.

Run the DAG

Now that we have everything set up as required, we can run a full process which synchronizes data from a file online, loads it into a Postgres, and transforms it. To do so:

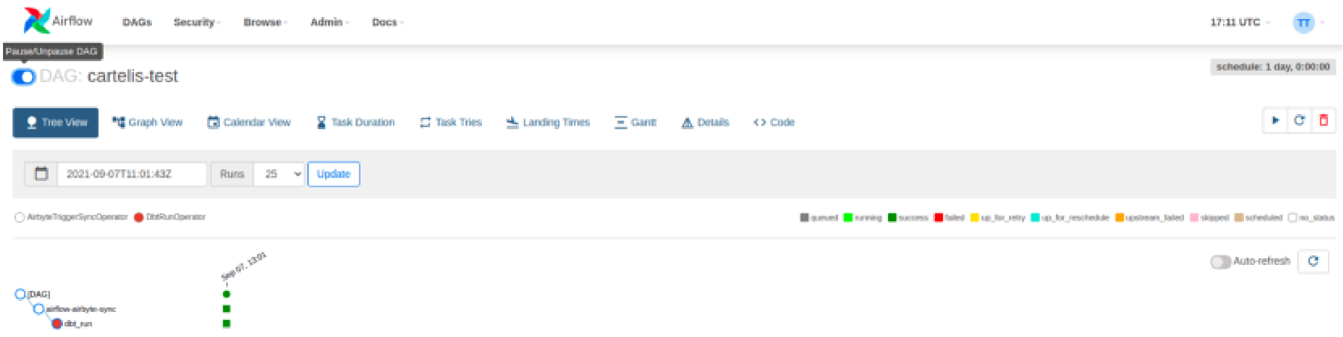
1- Access the Airflow webserver as before : <IP address>:8080

2- In the DAG tab, select the cartelis-test DAG

4- Trigger the DAG by selecting the Play icon in the upper right corner

5- Activate the DAG by clicking on the toggle button next to the DAG name

As configured, the DAG should run immediately, starting with the data synchronization with Airbyte and followed by the data transformation with dbt.



The square indicators next to the different tasks should turn dark green when successfully executed.

Conclusion

Where we initially had three different services to be installed, launched and managed separately, Docker allows us to condense and execute this process into a reproducible setup which can be launched and stopped as needed while reducing the need to worry about operating systems and software version incompatibilities.

What is more, we can further leverage additional Docker containers to add on features as needed such as data visualization with Metabase, or code editing with Jupyter.

Data engineers and data analysts can potentially benefit from this set up when working in environments with few financial resources or businesses with little to no infrastructure facilitating the use and analysis of data.