Airflow ⇄ NiFi Orchestration — Project Overview & Runbook

1. Perquisites of the Project

- Docker Desktop Application
- An IDE (Preferably Visual Studio Code)
- WSL integration in case you are on Windows (WSL gets installed when installing Docker Desktop but, in some instances, you might be prompted to upgrade the WSL version. Run the **wsl --upgrade** command to upgrade the WSL version)

2. Tools & technologies

- **Docker:** Docker is like a tiny, self-contained computer for each app. It lets you run software with all its dependencies in a neat box, so it behaves the same on any machine. Docker Compose is a recipe that starts multiple boxes (services) together with one command.
  Docker runs processes in isolated containers using Linux namespaces and cgroups. Images are layered filesystems (UnionFS). Compose defines multi-container apps as YAML (services, volumes, networks, healthchecks). In this project, Compose wires Airflow, PostgreSQL, a one-shot cert exporter, and (in a separate stack) NiFi, all on the same user-defined bridge network.

- **NiFi:** NiFi is a visual data conveyor belt. You drag-and-drop processors to pull data from places, transform it, and push it elsewhere—without writing lots of code.
  NiFi is a flow-based programming platform for data logistics. It provides processors, connections (queues), back pressure, lineage/provenance, and a REST API. We run NiFi over TLS (HTTPS) with a JKS keystore/truststore and Single-User Auth. Airflow controls NiFi via REST:
  - POST /nifi-api/access/token (get bearer token),
  - PUT /nifi-api/flow/process-groups/{id} (RUNNING/STOPPED),
  - GET /.../status (monitor),
  - POST /flowfile-queues/{id}/drop-requests (purge).

- **Airflow:** Airflow is a scheduler for repeatable work. You write Python “recipes” (DAGs) that say what to run and in what order, and Airflow makes sure it happens. Airflow orchestrates Directed Acyclic Graphs of tasks. It has a web UI,

scheduler, triggerer, workers/executor, and a metadata DB. Operators/Tasks are Python callables here. We store NiFi credentials as an Airflow Connection (nifi_default) and use Python requests with REQUESTS_CA_BUNDLE to validate NiFi's HTTPS.

- **Postgres:** Postgres is a reliable, open-source database. Airflow uses it to remember what ran and when, what worked, and what failed. PostgreSQL stores Airflow's metadata (DAG runs, task states, connections, variables, users).

  In Compose we run postgres:13 with a persistent named volume. Airflow connects via SQLAlchemy to postgresql+psycopg2://airflow:airflow@postgres/airflow.

- **TLS, Certificates, and OpenSSL (the cert exporter):** Because NiFi uses HTTPS with a self-signed certificate, Airflow needs a copy of that certificate so it can trust the connection. A tiny helper container grabs NiFi's cert and shares it with Airflow.

  An Alpine container runs openssl s_client -showcerts against nifi:8443 and writes /out/nifi-ca.pem to a named volume (nifi-ca-vol). Airflow mounts that volume read-only and sets REQUESTS_CA_BUNDLE=/opt/shared-certs/nifi-ca.pem, so all TLS handshakes with NiFi validate successfully. No verify=False shortcuts.

- **Docker networks & volumes:** A Docker network lets the containers find each other by name. A Docker volume is a shared folder that survives container restarts.
  Network: airflow-nifi-network (bridge). DNS inside the network resolves nifi to NiFi's container IP.

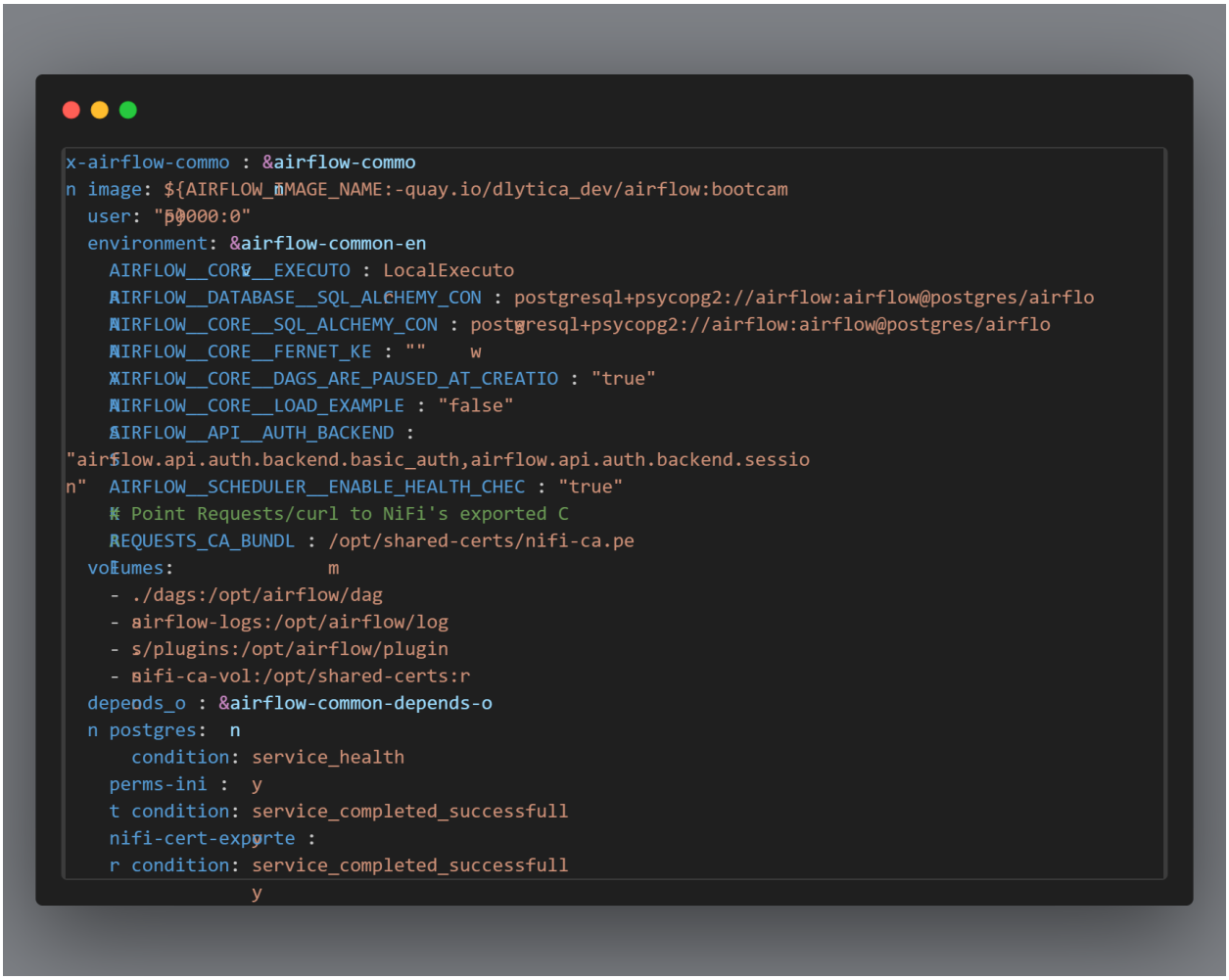
  Volumes:
  - postgres-db-volume → durable Postgres data
  - airflow-logs → persistent Airflow logs (with a perms-init job to chown)
  - nifi-ca-vol → carries /out/nifi-ca.pem from the exporter to Airflow

3. Project Overview

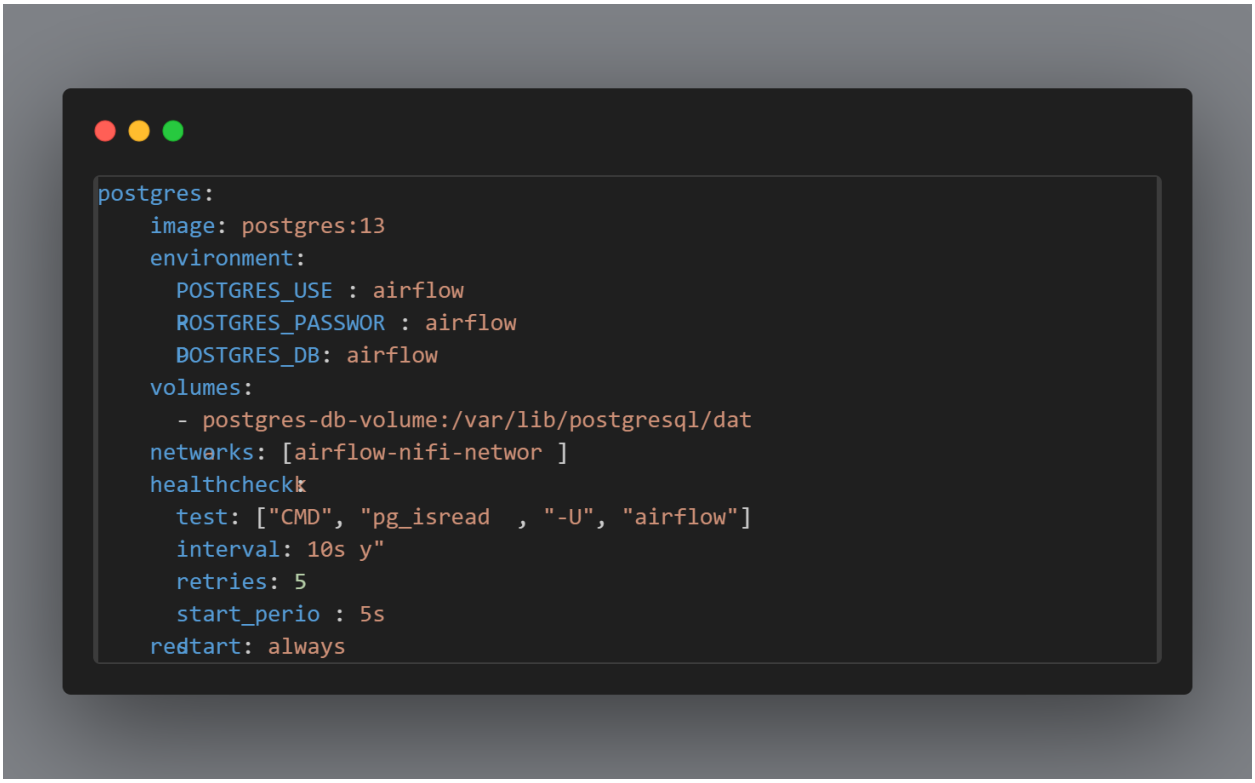
- Apache NiFi and Apache Airflow are set to be up and running via Docker as individual containers.
- The NiFi container consists of one service:
    1. Nifi: For the NiFi UI and metadata, mounting of JKS files, creation of network.

```
version: "3.9"

services:
  nifi:
    image: apache/nifi:2.5.
    contain@r_nam : nifi
    hostname: nifi
    ports:
      - "8443:8443"
    environment:
      SINGLE_USER_CREDENTIALS_USERNAM : nifi
      SINGLE_USER_CREDENTIALS_PASSWOR : nifi1234567
      D                                8
      # Bind/advertise host+port for HTTP
      NIFI_WEB_HTTPS_HOS : nifi
      NIFI_WEB_HTTPS_POR : 8443
      T
      # Use the pre-made JKS files (mounted belo
      NIFI_SECURITY_KEYSTOR : /opt/nifi/certs/keystore.jk
      NIFI_SECURITY_KEYSTORE_TYP : JKS
      NIFI_SECURITY_KEYSTORE_PASSW : nifi1234567
      NIFI_SECURITY_KEY_PASSW : nifi1234567
      D                        8
      NIFI_SECURITY_TRUSTSTOR : /opt/nifi/certs/truststore.jk
      NIFI_SECURITY_TRUSTSTORE_TYP : JKS
      NIFI_SECURITY_TRUSTSTORE_PASSW : nifi1234567
      D                               8
      # >= 12 char
      NIFI_SENSITIVE_PROPS_KE : MySuperSecretKey_202
      Y                        5
    volumes:
      # container path with the keystore/truststor
      e ./nifi-certs/nifi:/opt/nifi/certs:r
      - o/persist/conf:/opt/nifi/nifi-current/con
      - nifi-logs:/opt/nifi/nifi-current/log
      - nifi-flowfile-repo:/opt/nifi/nifi-current/flowfile_repositor
      - nifi-content-repo:/opt/nifi/nifi-current/content_repositor
      - nifi-provenance-repo:/opt/nifi/nifi-current/provenance_repositor
      - nifi-state:/opt/nifi/nifi-current/stat
      - e/nifi-drivers:/opt/nifi/nifi-current/lib/custo
        m
    networks:
      - airflow-nifi-networ
        k
networks:
  airflow-nifi-networ :
  k name: airflow-nifi-networ
    external: false

volumes:
  nifi-logs:
  nifi-flowfile-rep :
  nifi-content-rep :
  nifi-provenance-rep :
  nifi-stat :
  e
```

- The Airflow container consists of multiple services:


x-airflow-commo : &airflow-commo
n image: ${AIRFLOW_IMAGE_NAME:-quay.io/dlytica_dev/airflow:bootcam
user: "50000:0"
environment: &airflow-common-en
AIRFLOW__CORE__EXECUTO : LocalExecuto
AIRFLOW__DATABASE__SQL_ALCHEMY_CON : postgresql+psycopg2://airflow:airflow@postgres/airflo
AIRFLOW__CORE__SQL_ALCHEMY_CON : postgresql+psycopg2://airflow:airflow@postgres/airflo
AIRFLOW__CORE__FERNET_KE : "" w
AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATIO : "true"
AIRFLOW__CORE__LOAD_EXAMPLE : "false"
AIRFLOW__API__AUTH_BACKEND :
"airflow.api.auth.backend.basic_auth,airflow.api.auth.backend.sessio
n" AIRFLOW__SCHEDULER__ENABLE_HEALTH_CHEC : "true"
Point Requests/curl to NiFi's exported C
REQUESTS_CA_BUNDL : /opt/shared-certs/nifi-ca.pe
volumes: m
- ./dags:/opt/airflow/dag
- airflow-logs:/opt/airflow/log
- s/plugins:/opt/airflow/plugin
- nifi-ca-vol:/opt/shared-certs:r
depends_o : &airflow-common-depends-o
n postgres: n
condition: service_health
perms-ini : y
t condition: service_completed_successfull
nifi-cert-exporte :
r condition: service_completed_successfull
y

1. Postgres: Used for storing the metadata for the airflow environment + the actual data that is being fetched.

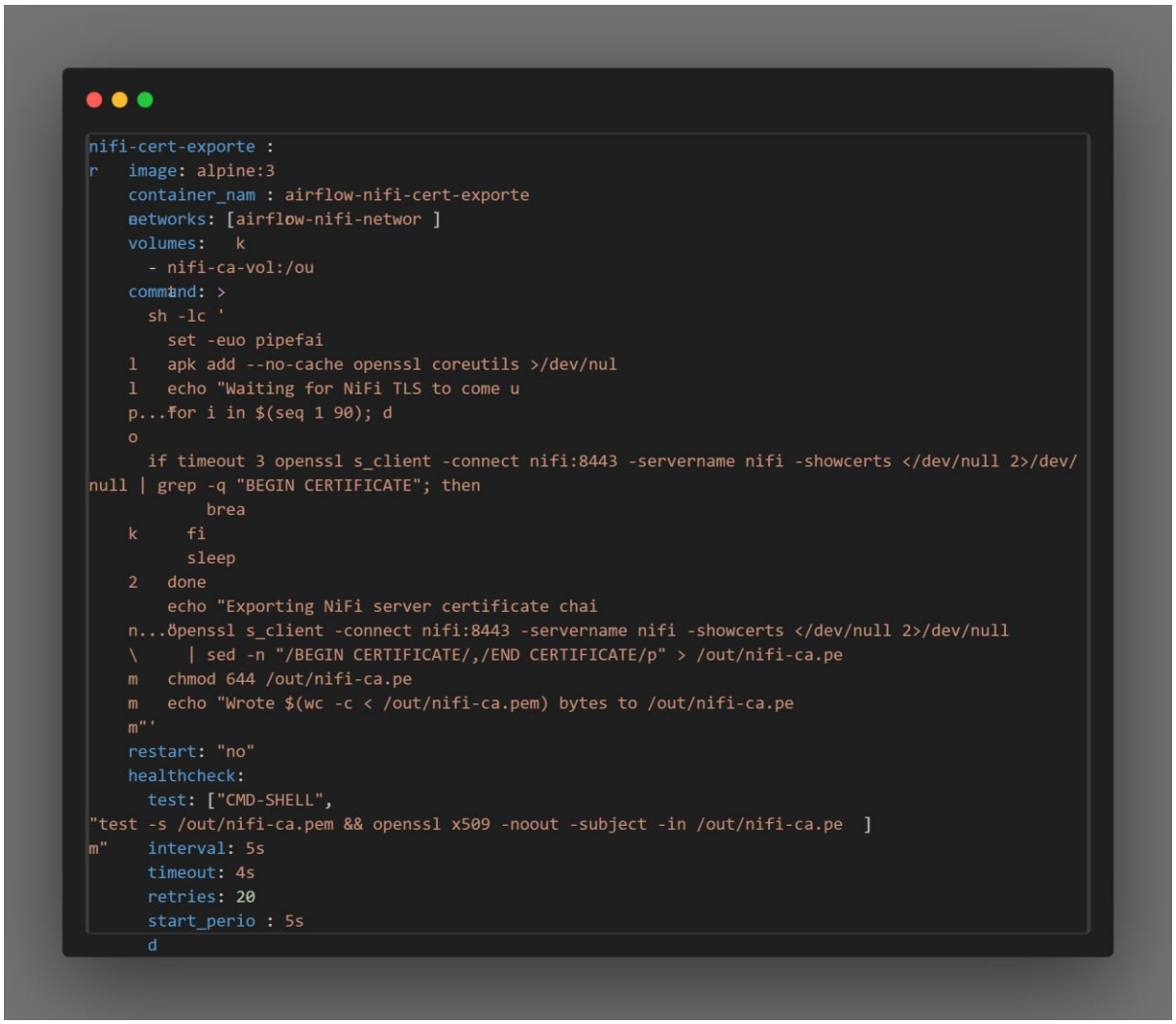


```
postgres:
    image: postgres:13
    environment:
      POSTGRES_USE : airflow
      ROSTGRES_PASSWOR : airflow
      DOSTGRES_DB: airflow
    volumes:
      - postgres-db-volume:/var/lib/postgresql/dat
    networks: [airflow-nifi-networ ]
    healthcheck
      test: ["CMD", "pg_isread  , "-U", "airflow"]
      interval: 10s y"
      retries: 5
      start_perio : 5s
    redtart: always
```

2. Perms-init: Used for fixing the file permissions on the logs volume before Airflow starts. It is a one-shot helper, i.e., it runs only once throughout the container's lifespan.

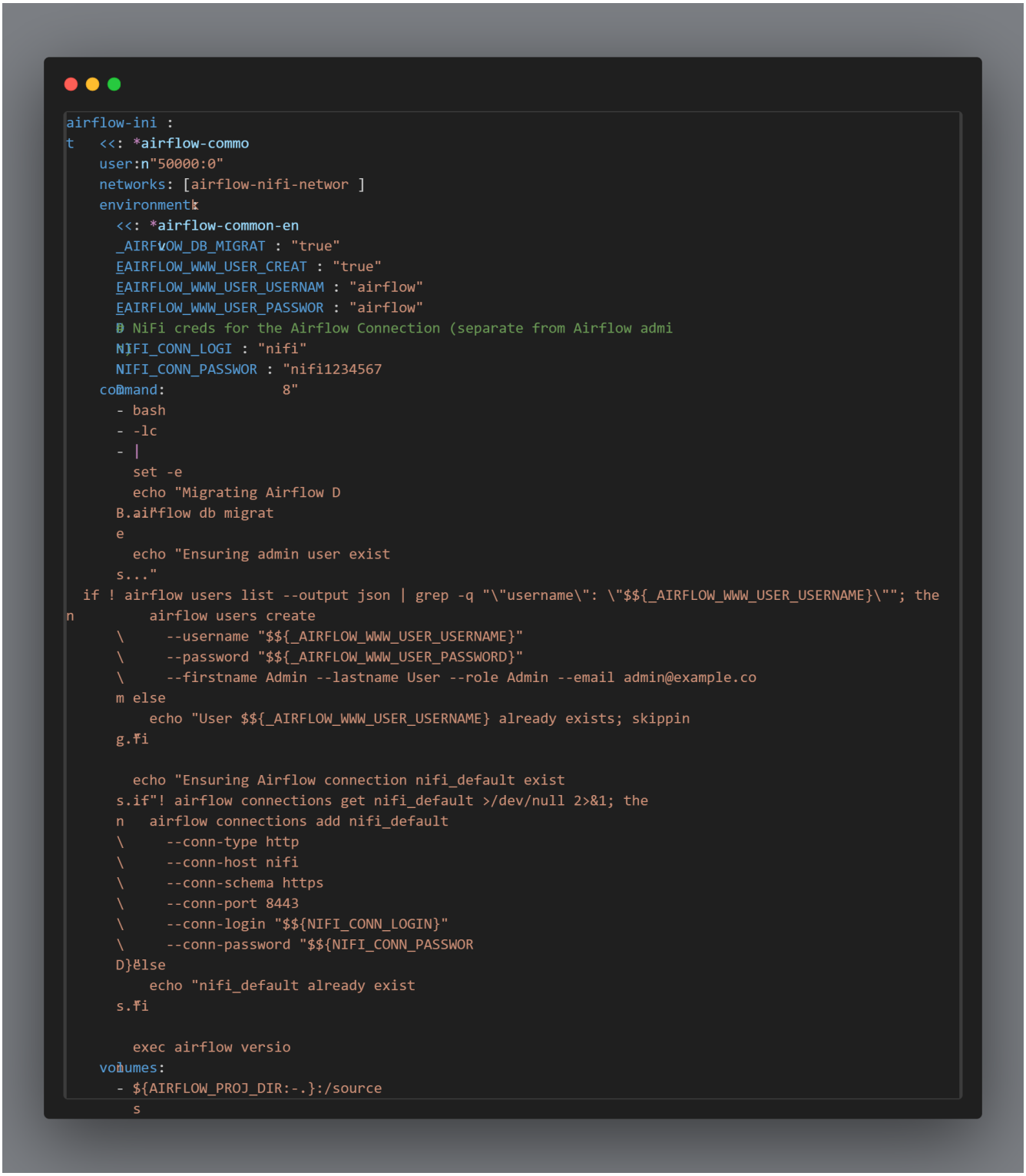
```
perms-ini :
t   image: alpine:3.20
    user: "0:0"
    command: ["/bin/sh","-lc","chown -R 50000:0 /logs && chmod -R u+rwX,g+rwX /log  ]
    volumes:                  s"
      - airflow-logs:/log
    restart: "no"
    networks: [airflow-nifi-networ ]
               k
```

3. Nifi-cert-exporter: Used for exporting NiFi's HTTPS server certificate into a shared volume so Airflow can trust NiFi over TLS automatically, allowing for the communication between the Airflow and NiFi's services. The service requires NiFi to be running and it is a one-shot helper as well.



```
nifi-cert-exporte :
r   image: alpine:3
    container_nam : airflow-nifi-cert-exporte
    networks: [airflow-nifi-networ ]
    volumes:   k
      - nifi-ca-vol:/ou
    command: >
      sh -lc '
        set -euo pipefai
    l   apk add --no-cache openssl coreutils >/dev/nul
    l   echo "Waiting for NiFi TLS to come u
    p...for i in $(seq 1 90); d
    o
      if timeout 3 openssl s_client -connect nifi:8443 -servername nifi -showcerts </dev/null 2>/dev/
null | grep -q "BEGIN CERTIFICATE"; then
            brea
    k     fi
          sleep
    2   done
        echo "Exporting NiFi server certificate chai
    n...openssl s_client -connect nifi:8443 -servername nifi -showcerts </dev/null 2>/dev/null
    \     | sed -n "/BEGIN CERTIFICATE/,/END CERTIFICATE/p" > /out/nifi-ca.pe
    m   chmod 644 /out/nifi-ca.pe
    m   echo "Wrote $(wc -c < /out/nifi-ca.pem) bytes to /out/nifi-ca.pe
    m"'
    restart: "no"
    healthcheck:
      test: ["CMD-SHELL",
"test -s /out/nifi-ca.pem && openssl x509 -noout -subject -in /out/nifi-ca.pe  ]
m"    interval: 5s
      timeout: 4s
      retries: 20
      start_perio : 5s
      d
```

4. Airflow-init: Used for initializing the airflow environment. Also, ensures the existence of admin user and establishes connection to nifi_default in case it doesn't exist.



```
airflow-ini :
t   <<: *airflow-commo
    user:n"50000:0"
    networks: [airflow-nifi-networ ]
    environmentk
      <<: *airflow-common-en
      _AIRFvOW_DB_MIGRAT : "true"
      EAIRFLOW_WWW_USER_CREAT : "true"
      EAIRFLOW_WWW_USER_USERNAM : "airflow"
      EAIRFLOW_WWW_USER_PASSWOR : "airflow"
      D NiFi creds for the Airflow Connection (separate from Airflow admi
      NIFI_CONN_LOGI : "nifi"
      NIFI_CONN_PASSWOR : "nifi1234567
    command:              8"
      - bash
      - -lc
      - |
        set -e
        echo "Migrating Airflow D
      B.air"flow db migrat
      e
        echo "Ensuring admin user exist
      s..."
  if ! airflow users list --output json | grep -q "\"username\": \"${_AIRFLOW_WWW_USER_USERNAME}\""; the
n         airflow users create
      \     --username "${_AIRFLOW_WWW_USER_USERNAME}"
      \     --password "${_AIRFLOW_WWW_USER_PASSWORD}"
      \     --firstname Admin --lastname User --role Admin --email admin@example.co
      m else
          echo "User ${_AIRFLOW_WWW_USER_USERNAME} already exists; skippin
      g.fi

        echo "Ensuring Airflow connection nifi_default exist
      s.if"! airflow connections get nifi_default >/dev/null 2>&1; the
      n   airflow connections add nifi_default
      \     --conn-type http
      \     --conn-host nifi
      \     --conn-schema https
      \     --conn-port 8443
      \     --conn-login "${NIFI_CONN_LOGIN}"
      \     --conn-password "${NIFI_CONN_PASSWOR
      D}else
          echo "nifi_default already exist
      s.fi

        exec airflow versio
    volumes:
      - ${AIRFLOW_PROJ_DIR:-.}:/source
        s
```

5. Airflow-webserver: For serving the UI of airflow on port 8080. Provides the DAGs dashboard, allows for triggering of the DAGs manually, checking logs, Managing connections, variables, pools, and user authentication.

```
airflow-webserve :
r   <<: *airflow-commo
    command: webserver
    ports: ["8080:8080"]
    networks: [airflow-nifi-networ ]
    restart: always
    depends_o :
    n <<: *airflow-common-depends-o
      airflow-ini :
      t condition: service_completed_successfull
                   y
```

6. Airflow-scheduler: For monitoring the DAGs and deciding what tasks need to run and when. It looks at the time-based schedules, task dependencies, and sensor completions. Once it figures out a task is ready, it pushes that task into the queue.

```
airflow-schedule :
r   <<: *airflow-commo
    command: scheduler
    networks: [airflow-nifi-networ ]
    restart: always
    depends_o :
    n <<: *airflow-common-depends-o
      airflow-ini :
      t condition: service_completed_successfull
                   y
```

7. Airflow-triggerer: For waking tasks up when external conditions are met.

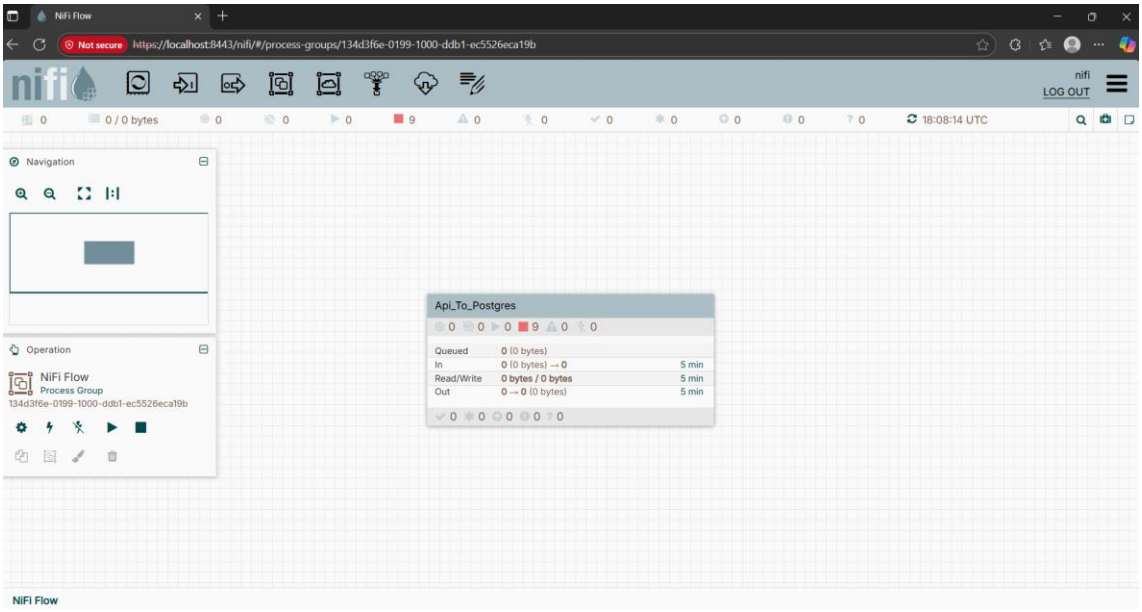
```
airflow-triggere :
r <<: *airflow-commo
  command: triggerer
  networks: [airflow-nifi-networ ]
  restart: always
  depends_o :
  n <<: *airflow-common-depends-o
    airflow-ini :
    t condition: service_completed_successfull
                 y
```

4. Project Flow

You spin up Apache NiFi and Apache Airflow in Docker (go into their respective folders using the cd command and run docker compose up --build command).

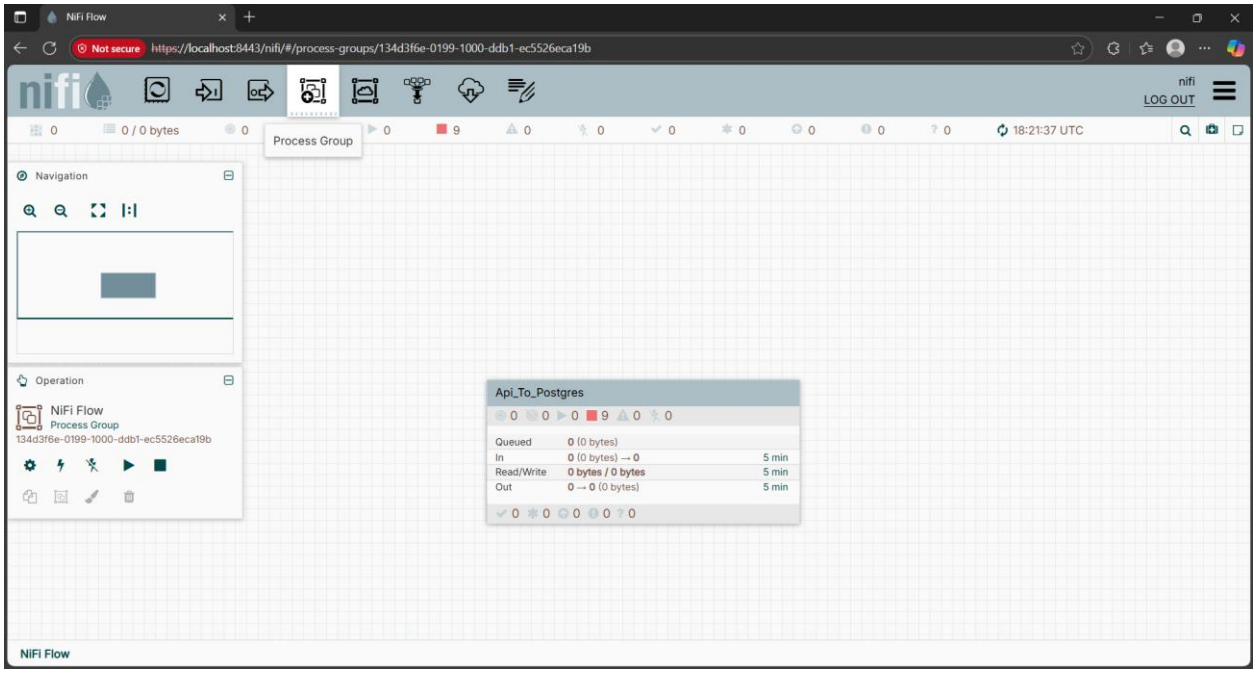
The initial run may take some time to start all the containers as the image needs to be downloaded as well. But after the download is successful you should be able to access the NiFi and Airflow UI on https://localhost:8443/nifi/ and http://localhost:8080/home respectively.

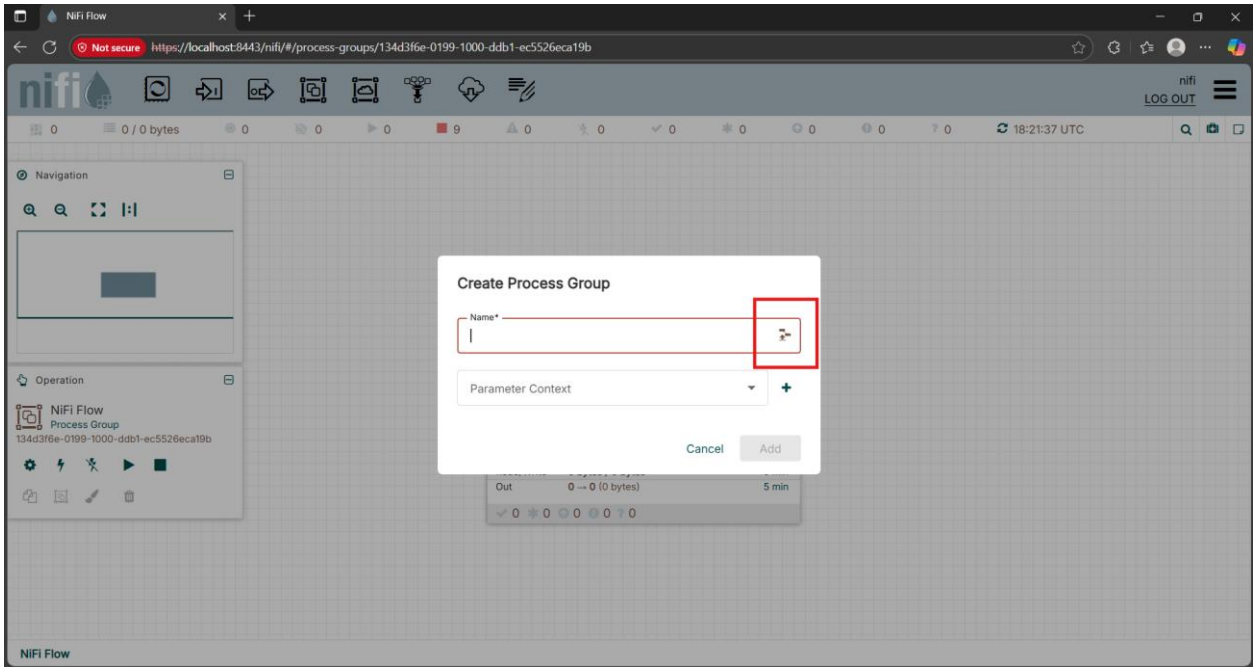
NiFi UI:



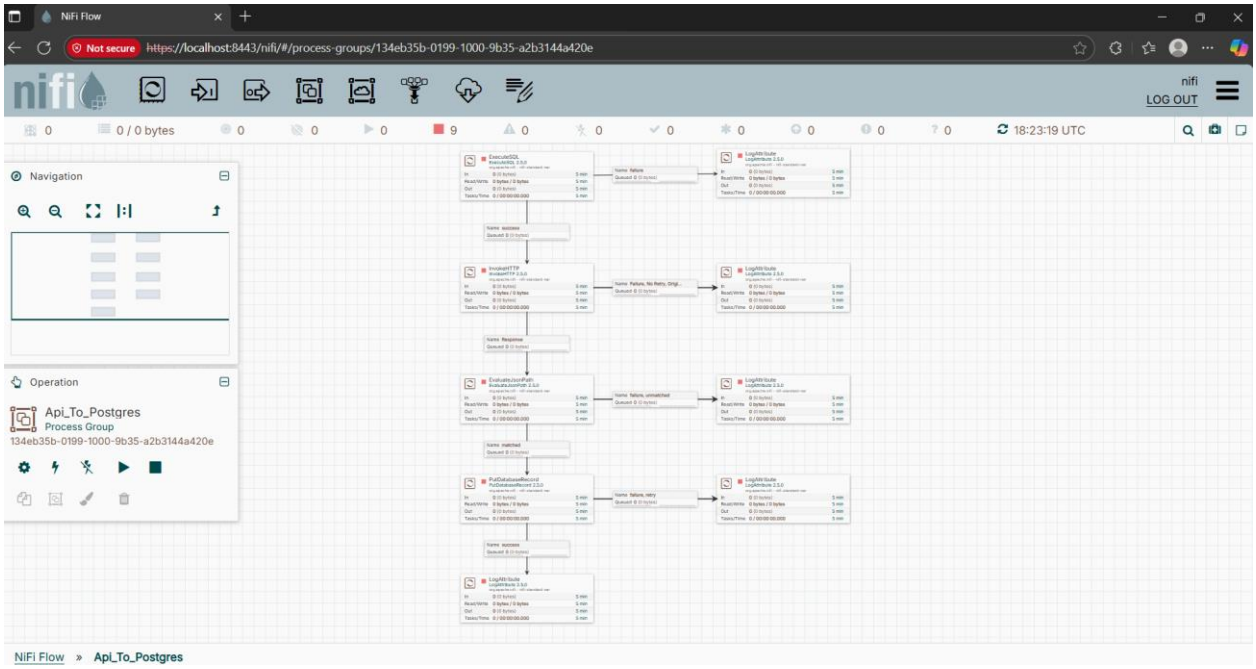
You should see a Process Group available in the NiFi UI and a DAG available in the Airflow UI.

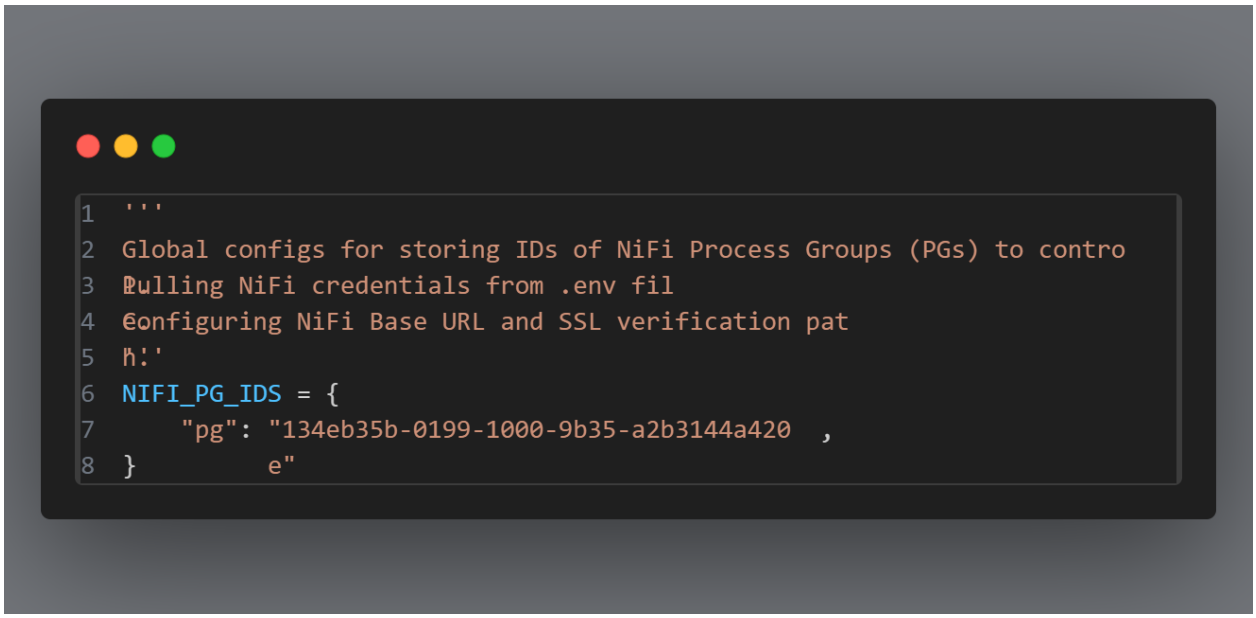
In case you can't see the Process Group, you can by dragging a Process Group and exporting the Api_To_Postgres.json file available in the .zip file.





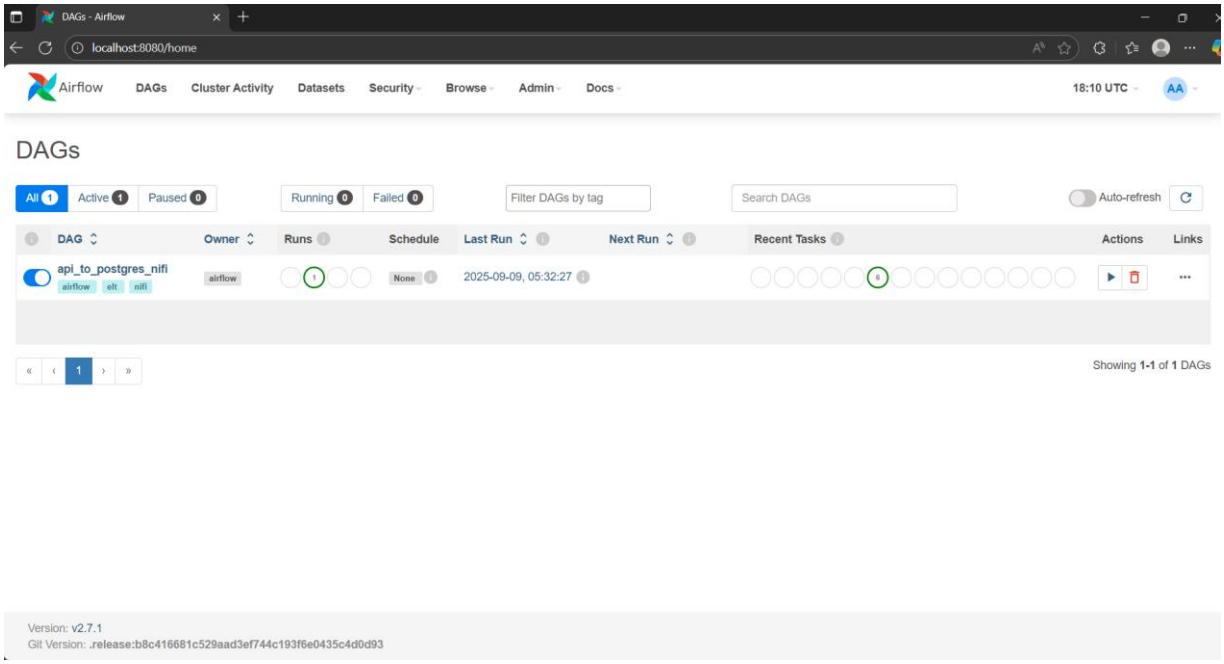
You can also view the Processors within the Process Group by double tapping to head inside the Process Group.



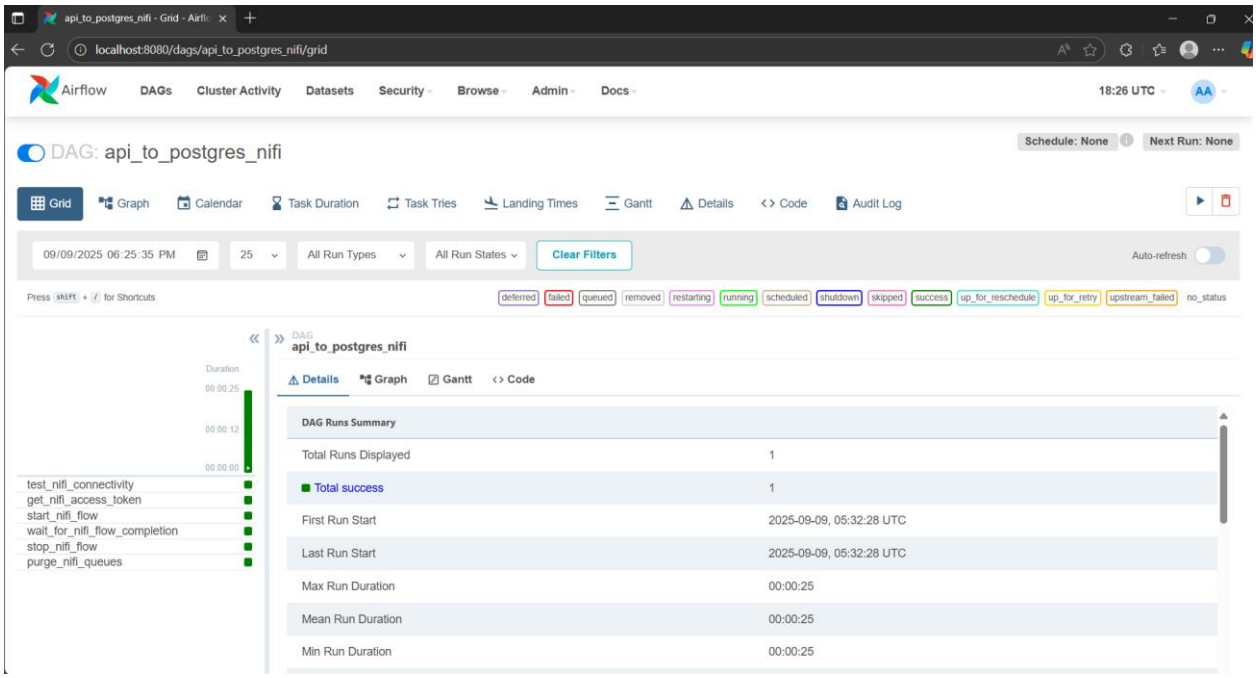


Make sure the Process Group ID in the DAG code is the same as the DAG code available on NiFi.

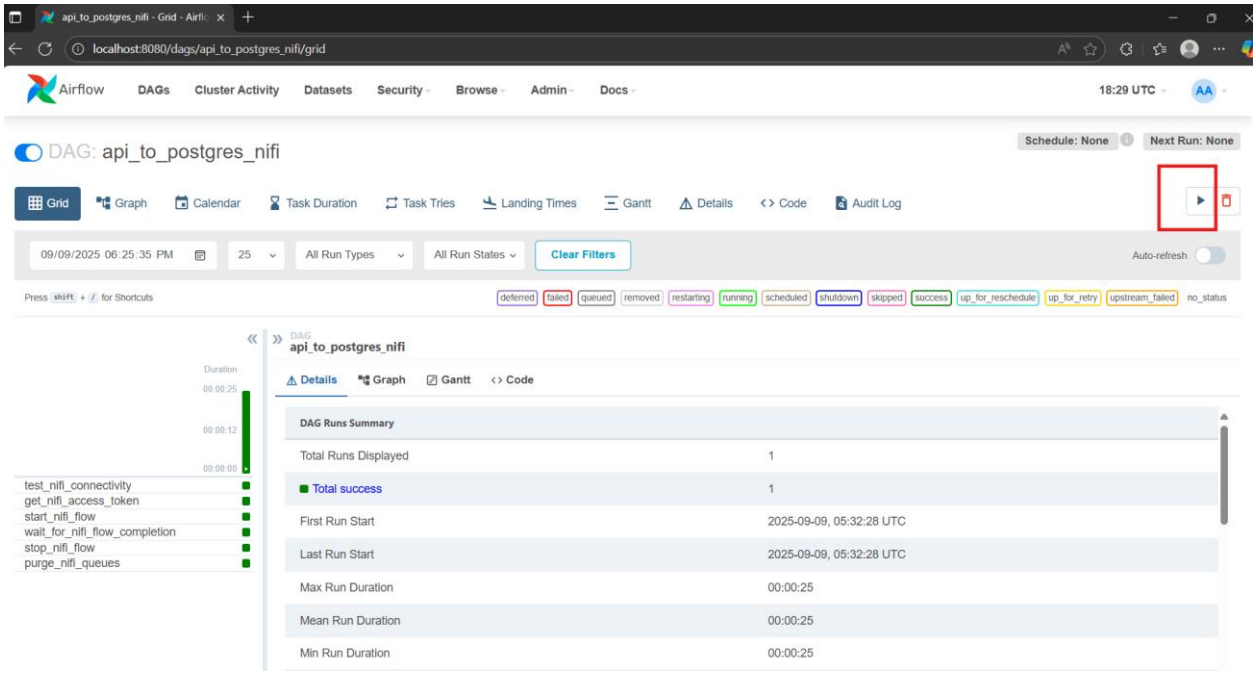
Airflow UI:



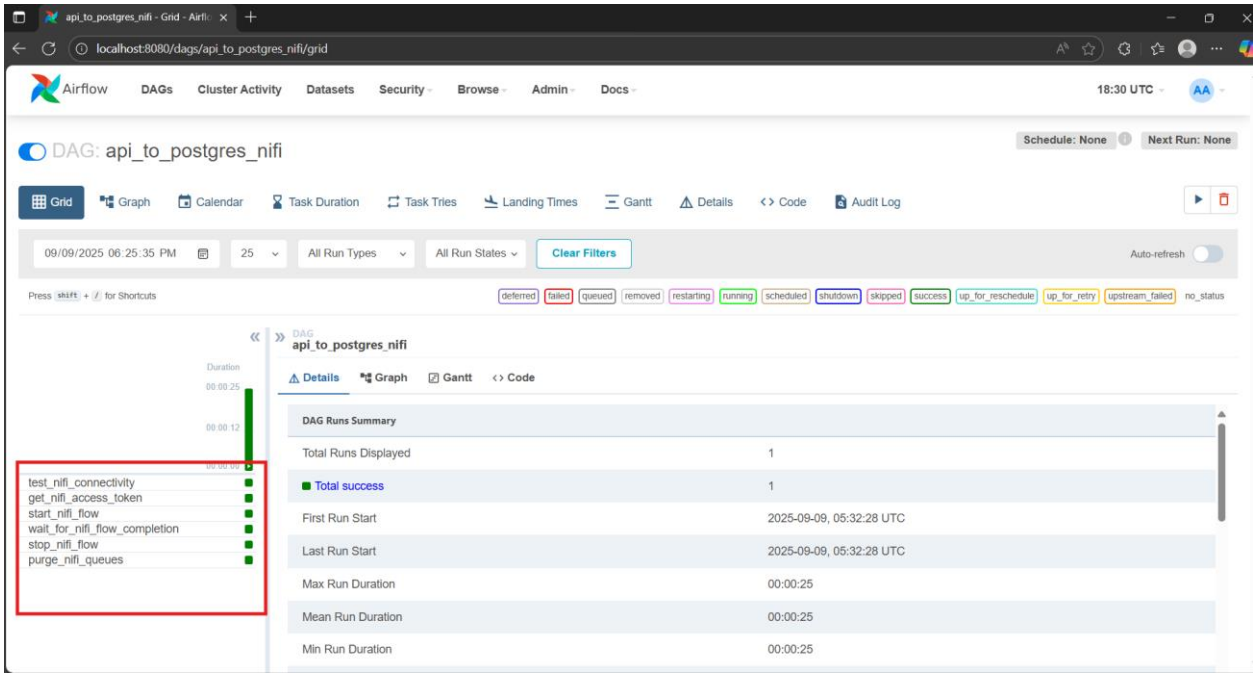
Double-click on the DAG name in Airflow to view the details of the DAG.



Click on the Trigger Button to start the DAG



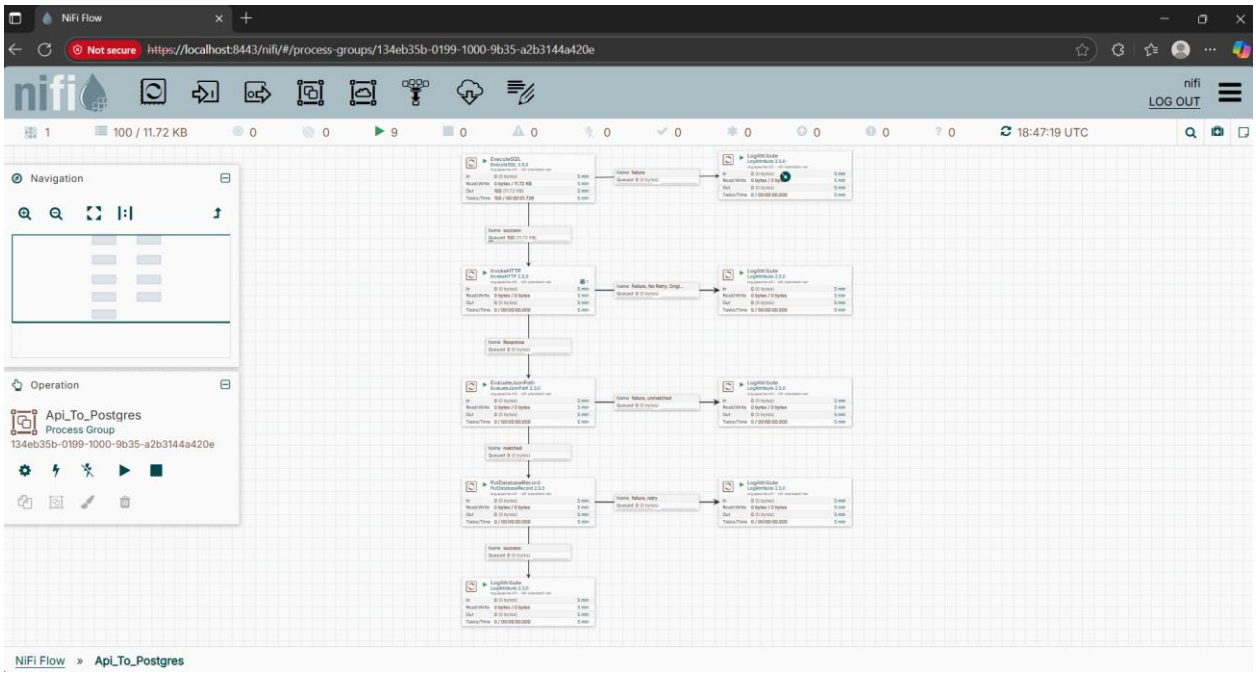
Starting the DAG will trigger the tasks one by one



- test_nifi_connectivity: tests the connection with NiFi to make sure NiFi is reachable.
- get_nifi_access_token: requests a bearer token for authentication.
- start_nifi_flow: starts all the Process Group whose id is provided.
- wait_for_nifi_flow_completion: monitors the Process Group status until stop criteria are met.
- stop_nifi_flow: ensures Process Group is stopped after flow completes.

- purge_nifi_queues: deletes any leftover data in all the queues in the Process Group.

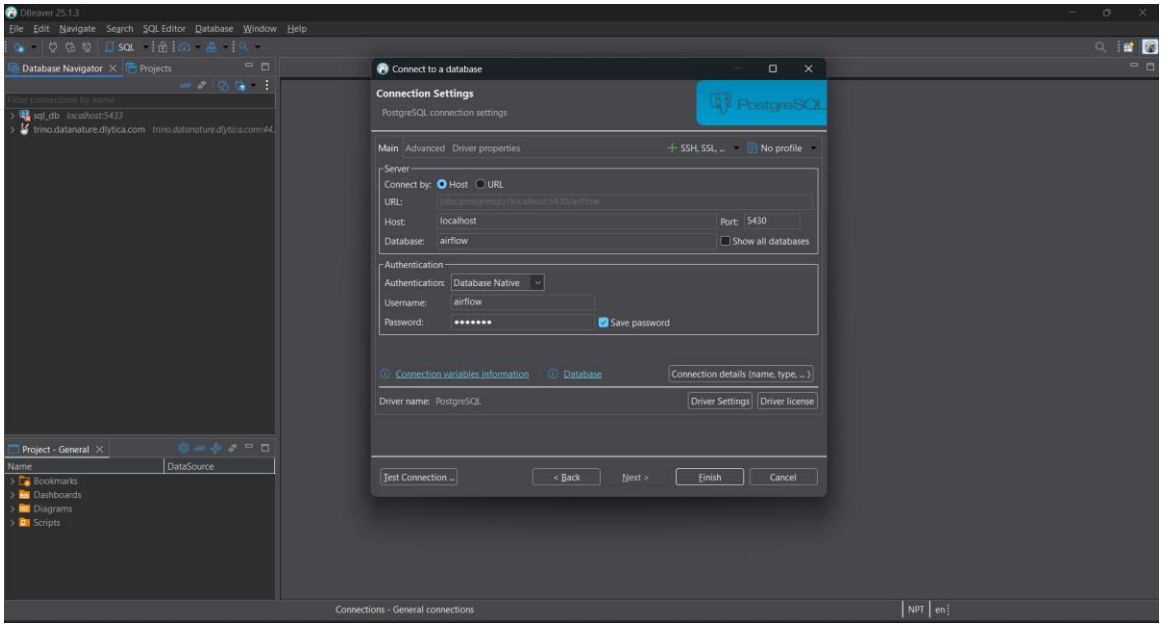
Triggering the DAG leads to all the NiFi processors being started (Marked by green and running)



This also populates the data in the postgres.

You can access the postgres container and cross check using platforms like DBeaver.

1. Create a connection to the postgres environment (the port number might be different in other scenarios. Default for postgres is 5432)



2. Open a query window and execute the query