Stock Data Pipeline Report

This report provides a comprehensive overview of the components and functionality of the provided stock data pipeline. The pipeline is designed to periodically fetch stock market data from an external API and store it in a PostgreSQL database using Apache Airflow for orchestration.

Dockerfile

The Dockerfile is a script that automates the creation of a Docker image. This image serves as the base for the Airflow services, ensuring they have all the necessary dependencies to run the data pipeline.

Code Explanation

```
# Use a base image for Airflow. This image already has Python installed.

FROM apache/airflow:2.7.0
```

- FROM apache/airflow:2.7.0: This line specifies the base image for the Docker build. It uses a pre-built Airflow image from Docker Hub, which already includes Python and the Airflow framework.

```
# Set the AIRFLOW_HOME environment variable

ENV AIRFLOW_HOME=/opt/airflow
```

- ENV AIRFLOW_HOME=/opt/airflow: This command sets an environment variable named AIRFLOW_HOME to /opt/airflow. This is the directory where Airflow stores its configurations, DAGs, and logs within the container.

```
# Install system dependencies.

USER root

RUN apt-get update && apt-get install -y \
  libpq-dev \
  && rm -rf /var/lib/apt/lists/*
```

- USER root: Switches the user context to root to allow system-level package installation.
- RUN apt-get update && apt-get install -y libpq-dev: This command updates the package list and installs libpq-dev, which is a library required for the psycopg2 Python package to connect to a PostgreSQL database.
- && rm -rf /var/lib/apt/lists/*: Cleans up the package lists to reduce the final image size.

```
# Copy the requirements file into the container.
```

```
USER airflow

COPY requirements.txt .
```

- USER airflow: Switches the user back to airflow for security reasons, as running as root is not recommended for application processes.
- COPY requirements.txt .: This copies the requirements.txt file from the local directory (where the Dockerfile is located) into the current directory of the container.

```
# Install Python dependencies from the requirements.txt file.

RUN pip install --no-cache-dir -r requirements.txt
```

- RUN pip install --no-cache-dir -r requirements.txt: This command installs all the Python packages listed in requirements.txt. The --no-cache-dir flag prevents caching of the downloaded packages, further reducing the image size.

docker-compose.yml

This file defines and configures the multi-container Docker application. It sets up the database, Airflow webserver, scheduler, and an initialization service, linking them all together.

Code Explanation

```
# This docker-compose.yml file defines the services for a new stock data pipeline.

# It sets up a PostgreSQL database and all the necessary components for an Airflow instance.


services:

  postgres:

    # Use the official PostgreSQL image.

    image: postgres:14

    container_name: stock-postgres

    # Define environment variables for the PostgreSQL database.

    environment:

      - POSTGRES_USER=stock_user

      - POSTGRES_PASSWORD=stock_password

      - POSTGRES_DB=stock_data

    # Map a different port to avoid conflict with the previous project.

    ports:
```

```
    - "5434:5432"
  # Use a named volume to persist the database data.
  volumes:
    - stock_postgres_data:/var/lib/postgresql/data
  # Health check to ensure the service is ready.
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U stock_user"]
    interval: 5s
    timeout: 5s
    retries: 5
```

- services: This key defines all the services (containers) that will run.
- postgres: Defines a service named postgres.
- image: postgres:14: Uses the official PostgreSQL version 14 image.
- container_name: stock-postgres: Assigns a readable name to the container.
- environment: Sets environment variables for the PostgreSQL container, defining the database name, user, and password.
- ports: - "5434:5432": Maps port 5434 on the host machine to port 5432 inside the container, allowing you to connect to the database from your host.
- volumes: - stock_postgres_data:/var/lib/postgresql/data: Creates a named volume to persist the database data, so it won't be lost when the container is stopped or removed.
- healthcheck: Configures a health check to verify that the database is ready for connections.

```
airflow-webserver:
  # Build the image using the provided Dockerfile.
  build: .
  container_name: stock-airflow-webserver
  # Map a different port for the Airflow UI.
  ports:
    - "8082:8080"
  # The command to run the Airflow webserver.
  command: webserver
```

```
    # Set environment variables, including the database connection details.
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=false
      - AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://stock_user:stock_password@postgres/stock_data
      - AIRFLOW_VAR_STOCK_API_KEY=${STOCK_API_KEY}
    # Mount the dags and data folders to the container.
    volumes:
      - ./dags:/opt/airflow/dags
      - ./data:/opt/airflow/data
    # Set the webserver's dependencies.
    depends_on:
      postgres:
        condition: service_healthy
      airflow-init:
        condition: service_completed_successfully
    # Health check to ensure the webserver is running.
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8082/health"]
      interval: 10s
      timeout: 10s
      retries: 5
```

- airflow-webserver: Defines the Airflow webserver service.
- build: .: Instructs Docker Compose to build the image using the Dockerfile in the current directory.
- ports: - "8082:8080": Maps host port 8082 to the container's port 8080, where the Airflow UI is exposed.
- command: webserver: Runs the Airflow webserver command.
- environment: Sets environment variables, including the database connection string and the API key from the host machine.

- volumes: - ./dags:/opt/airflow/dags: Mounts the local dags folder into the container, so any changes to the DAG files are automatically reflected.
- depends_on: Ensures the webserver only starts after the postgres and airflow-init services are ready.

```
  airflow-scheduler:
    build: .
    container_name: stock-airflow-scheduler
    # The command to run the Airflow scheduler.
    command: scheduler
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=false
      -
AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://stock_user:st
ock_password@postgres/stock_data
      - AIRFLOW_VAR_STOCK_API_KEY=${STOCK_API_KEY}
    volumes:
      - ./dags:/opt/airflow/dags
      - ./data:/opt/airflow/data
    depends_on:
      postgres:
        condition: service_healthy
      airflow-init:
        condition: service_completed_successfully
```

- airflow-scheduler: Defines the Airflow scheduler service, which is responsible for monitoring DAGs and triggering tasks. Its configuration is similar to the webserver's.

```
  # The service to initialize the database and create an Airflow user.
  airflow-init:
    build: .
    container_name: stock-airflow-init
    command: bash -c "airflow db migrate && airflow users create --username airflow --
password airflow --firstname Airflow --lastname Airflow --role Admin --email
airflow@example.com"
```

```
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=false
      - AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://stock_user:stock_password@postgres/stock_data
    volumes:
      - ./dags:/opt/airflow/dags
    depends_on:
      postgres:
        condition: service_healthy
```

- airflow-init: A temporary service that runs a command to initialize the Airflow database and create a default admin user.
- command: Executes two Airflow commands: db migrate to set up the database schema, and users create to add an admin user for the UI.

```
volumes:
  stock_postgres_data:
```

- volumes: Defines the named volume stock_postgres_data used by the postgres service.

stock_api_script.py

This is the core Python script that handles the data fetching and database interaction. It is designed to be executed by an Airflow task.

Code Explanation

```
# This script contains the core logic for the data pipeline:
# 1. Fetching data from a stock market API.
# 2. Connecting to a PostgreSQL database.
# 3. Storing the data in a table.

import requests
import psycopg2
import os
import logging
```

```
import json
from datetime import datetime


# Configure logging for better visibility and debugging.
logging.basicConfig(level=logging.INFO)
```

- These lines import necessary libraries for HTTP requests, PostgreSQL connection, environment variable access, logging, JSON parsing, and handling timestamps.

```
# --- Configuration using Environment Variables ---
DB_HOST = os.getenv('AIRFLOW_VAR_DB_HOST', 'postgres')
DB_NAME = 'stock_data'
DB_USER = 'stock_user'
DB_PASSWORD = 'stock_password'
STOCK_API_KEY = os.getenv('AIRFLOW_VAR_STOCK_API_KEY')
STOCK_SYMBOL = 'AAPL' # Example stock symbol
```

- This section defines configuration variables, retrieving the database host and API key from environment variables passed by Airflow.

```
# --- Database Operations ---


def create_table_if_not_exists():
    """
    Connects to the PostgreSQL database and creates the stock_data table
    if it does not already exist.
    """
    conn = None
    try:
        conn = psycopg2.connect(host=DB_HOST, database=DB_NAME, user=DB_USER, password=DB_PASSWORD)
        cur = conn.cursor()
        logging.info("Successfully connected to the PostgreSQL database.")
```

```
        create_table_sql = """
        CREATE TABLE IF NOT EXISTS stock_data (
            symbol VARCHAR(10) NOT NULL,
            price DECIMAL(10, 2) NOT NULL,
            volume INT NOT NULL,
            timestamp TIMESTAMP NOT NULL,
            PRIMARY KEY (symbol, timestamp)
        );
        """
        cur.execute(create_table_sql)
        conn.commit()
        logging.info("Table 'stock_data' checked/created successfully.")
        cur.close()

    except (Exception, psycopg2.DatabaseError) as error:
        logging.error(f"Error creating table: {error}")
    finally:
        if conn is not None:
            conn.close()
```

- create_table_if_not_exists(): This function connects to the PostgreSQL database using psycopg2 and executes a SQL command to create the stock_data table if it doesn't already exist. It includes error handling and ensures the database connection is closed.

```
def insert_stock_data(symbol, price, volume):
    """
    Inserts a single stock data record into the 'stock_data' table.
    """
    conn = None
    try:
```

```
        conn = psycopg2.connect(host=DB_HOST, database=DB_NAME, user=DB_USER, password=DB_PASSWORD)
        cur = conn.cursor()
        logging.info("Database connection for insertion successful.")

        insert_sql = """
        INSERT INTO stock_data (symbol, price, volume, timestamp)
        VALUES (%s, %s, %s, %s)
        ON CONFLICT (symbol, timestamp) DO NOTHING;
        """

        cur.execute(insert_sql, (symbol, price, volume, datetime.now()))
        conn.commit()
        logging.info(f"Successfully inserted data for {symbol}.")
        cur.close()

    except (Exception, psycopg2.DatabaseError) as error:
        logging.error(f"Error inserting data into table: {error}")
    finally:
        if conn is not None:
            conn.close()
```

- insert_stock_data(): This function takes stock data as input, connects to the database, and inserts a new record into the stock_data table. The ON CONFLICT DO NOTHING clause handles cases where a duplicate record might exist, preventing insertion errors.

```
# --- API Interaction and Data Extraction ---

def fetch_and_store_stock_data():
    """
    Main function to orchestrate the data fetching and storage.
    This function will be called by the Airflow task.
```

```
    """
    logging.info("Starting data fetching and storage process...")

    if not STOCK_API_KEY:
        logging.error("STOCK_API_KEY is not set. Please check your environment variables.")
        return

    # Using a placeholder for a real API
    # Replace with a real API like Alpha Vantage
    url = f'https://api.example.com/stock?symbol={STOCK_SYMBOL}&apikey={STOCK_API_KEY}'

    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status()
        data = response.json()
        logging.info("Successfully fetched data from API.")

        # --- Data Extraction ---
        # Assuming the API response has a specific structure.
        price = data.get('price')
        volume = data.get('volume')

        if price is None or volume is None:
            logging.warning("Missing 'price' or 'volume' in API response. Skipping insertion.")
            logging.debug(f"Received data: {data}")
            return

        try:
            price = float(price)
```

```
            volume = int(volume)
        except (ValueError, TypeError) as e:
            logging.error(f"Failed to convert data types: {e}. Raw data: price={price}, volume={volume}")
            return

        insert_stock_data(STOCK_SYMBOL, price, volume)

    except requests.exceptions.RequestException as e:
        logging.error(f"Failed to fetch data from API: {e}")
    except (json.JSONDecodeError, KeyError) as e:
        logging.error(f"Failed to parse API response: {e}")
```

- fetch_and_store_stock_data(): This is the main function. It first checks for the existence of the API key. It then uses the requests library to make a GET request to a placeholder API URL. It includes robust error handling to catch issues with network requests, HTTP status codes, and JSON parsing. Finally, it extracts the relevant price and volume data and calls the insert_stock_data function.

```
if __name__ == "__main__":
    create_table_if_not_exists()
    fetch_and_store_stock_data()
```

- if __name__ == "__main__":: This block ensures that the create_table_if_not_exists() and fetch_and_store_stock_data() functions are called when the script is run directly. This is useful for testing the script independently of Airflow.

stock_data_dag.py

This file defines the Airflow DAG (Directed Acyclic Graph), which orchestrates the pipeline's tasks. The DAG is responsible for scheduling when the Python script should run.

Code Explanation

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime
```

- These lines import the necessary classes from the Airflow library: DAG to define the workflow, BashOperator to execute a shell command, and datetime to specify the start date.

```
with DAG(
    dag_id='stock_data_pipeline_v2',
    start_date=datetime(2023, 1, 1),
    schedule_interval='@daily',
    catchup=False
) as dag:
```

- with DAG(...) as dag:: This is the context manager for defining the DAG.
- dag_id='stock_data_pipeline_v2': A unique identifier for the DAG.
- start_date=datetime(2023, 1, 1): The date from which the DAG should start running.
- schedule_interval='@daily': Specifies the schedule for the DAG to run, in this case, once every day.
- catchup=False: When False, the scheduler will not run any missed past runs of the DAG.

```
    # This task runs the Python script to fetch and store stock data.
    fetch_and_store_data = BashOperator(
        task_id='fetch_and_store_stock_data',
        bash_command='python /opt/airflow/dags/stock_api_script.py',
    )
```

- fetch_and_store_data = BashOperator(...): Defines a task within the DAG.
- task_id='fetch_and_store_stock_data': A unique identifier for the task within the DAG.
- bash_command='python /opt/airflow/dags/stock_api_script.py': The shell command that the BashOperator will execute. This command runs the stock_api_script.py file using the Python interpreter. The /opt/airflow/dags/ path is where the file is mounted in the Docker container.

Dataflow Diagram

The dataflow diagram below illustrates the sequence and flow of data within the stock data pipeline.

1. **Airflow Scheduler**: The process begins with the Airflow scheduler, which is configured to run a DAG daily.
2. **DAG Task**: The DAG contains a single task, fetch_and_store_stock_data, which is triggered by the scheduler.
3. **Python Script Execution**: This task executes the stock_api_script.py file.
4. **API Call**: The Python script uses the requests library to make a call to an external stock market API, passing the STOCK_API_KEY and a stock symbol.
5. **Data Fetch**: The API responds with the requested stock data (e.g., price and volume) in a JSON format.
6. **Data Extraction**: The Python script parses the JSON response and extracts the relevant price and volume data.
7. **Database Connection**: The script establishes a connection to the PostgreSQL database (stock-postgres).
8. **Data Insertion**: The extracted data is then inserted into the stock_data table within the database, along with a timestamp.