We're building a simulated data pipeline for a smart electric vehicle (EV) factory—much like a Tesla production line. The idea is to simulate sensor data from various factory stations (chassis, battery, paint, quality), stream that data via Kafka, and eventually process/store it (with Postgres) and expose it through an API (FastAPI).

What We Did

1. **Initialized a Git Repository:**
   - **Files Updated:**
     - `.gitignore`
     - `README.md`
   - **Purpose:**
     We set up a local Git repository to keep our project organized and version-controlled. The `.gitignore` ensures we don't commit unnecessary or sensitive files, while the `README.md` gives a brief description of our project.
2. **Configured Docker Compose for Essential Services:**
   - **Files Updated:**
     - `docker-compose.yml`
   - **Purpose:**
     We defined services for:
     - **Zookeeper & Kafka:** For data streaming. Kafka is our message broker that will handle the sensor data.
     - **Postgres:** For eventual data storage.
     - **Simulator:** Our service that simulates sensor data from multiple factory stations.
   - We set environment variables (like `KAFKA_ADVERTISED_LISTENERS`) to ensure that our services can communicate within Docker's network correctly.
3. **Created the Data Simulator Service:**
   - **Files Updated/Created:**
     - `services/data_simulator/requirements.txt` (lists `kafka-python` as a dependency)
     - `services/data_simulator/simulator.py` (contains the code that simulates data for different factory stations)
   - **Purpose:**
     The simulator script generates random sensor data for four factory stations (chassis, battery, paint, quality) and publishes each message to its respective Kafka topic. This simulates the real-world data that might be produced on an EV production line.
4. **Dockerized the Simulator:**
   - **Files Updated/Created:**
     - `services/data_simulator/Dockerfile` (defines how to build a Docker image for the simulator)

        - Updated `simulator.py` to use an environment variable for the Kafka broker address and to include a retry loop to wait for Kafka to be ready.
        - Modified `docker-compose.yml` to include the simulator service.
    - **Purpose:**
      Dockerizing the simulator ensures that it runs in a consistent environment and can be managed alongside our other services (Kafka, Postgres, etc.) via Docker Compose. The changes in the simulator code (waiting for Kafka) help it to handle startup timing issues—making sure Kafka is fully up before it tries to send data.

Airflow:

In this phase, we'll use Airflow to create a simple ETL (Extract, Transform, Load) pipeline that will read the data our simulator is sending to Kafka and then store it into our PostgreSQL database.

Airflow DAG for ETL:
We'll create an Airflow DAG (Directed Acyclic Graph) that will:

1. **Extract** data from Kafka (consume messages from topics like `chassis_topic`, `battery_topic`, etc.).
2. **Transform** the data if needed (for now, you can pass it through as is).
3. **Load** the data into our Postgres database.

STEPS to get Airflow working in 8080 localhost

1. Adding Airflow to Docker Compose

- **What We Did:**
  Added an Airflow service block in `docker-compose.yml` with settings for the executor, database connection, volume for DAGs, and a command to run Airflow in standalone mode.
- **Key Settings:**
    - **Environment Variables:**
        - `AIRFLOW__CORE__EXECUTOR=LocalExecutor`
        - `AIRFLOW__CORE__FERNET_KEY` **(we generated a key with Python)**
        - `AIRFLOW__DATABASE__SQL_ALCHEMY_CONN` **(connection string to Postgres)**
    - **Command: Set to `standalone` so Airflow starts its webserver and scheduler.**
    - **Port Mapping: Exposed port 8080 to access the web UI.**

---

2. Error: Environment Variable Parsing Issue

- **Problem:**
  Received an error like `unexpected type []interface {}` because the Fernet key contained special characters.
- **Solution:**
  Quoted the Fernet key in the YAML file or switched to a key-value mapping format for environment variables.

---

3. Error: Airflow CLI Without a Command

- **Problem:**
  Airflow's CLI printed its help message and exited because no command (like `webserver` or `standalone`) was specified.
- **Solution:**
  Added `command: standalone` in the Airflow service so it would run all necessary components in one container.

---

4. Login Issue in Airflow Web UI

- **Problem:**
  The default login (`admin`/`admin`) did not work.
- **Solution:**
  Entered the Airflow container using `docker-compose exec airflow bash` and either updated the existing admin user's password or created a new admin user using the Airflow CLI commands:

We created new user:

```
airflow users create --username newadmin --password newadmin --role
Admin ...
```

Summary

- Integrated Airflow into our Docker Compose setup.
- Fixed YAML issues by quoting the Fernet key.
- Ensured Airflow starts correctly by using `command: standalone`.
- Resolved login issues by updating/creating an admin user inside the container.

ETL DAG in Airflow:

A scheduled task (DAG) that:

1. **Extracts** data from Kafka (from our topics like `chassis_topic`, `battery_topic`, etc.).
2. **Transforms** the data if needed (for this demo, we can simply pass it through).
3. **Loads** the data into a Postgres table.

Why?
This mimics a real-world pipeline: sensor data produced by our simulator goes into Kafka, and then an ETL process picks it up and stores it for later querying or reporting.

Problem with login:

→ docker-compose exec airflow bash

→ check users → airflow users list

→ if no user, create: airflow users create \

  --username newadmin \

  --firstname New \

  --lastname Admin \

  --role Admin \

  --email newadmin@example.com \

  --password newadmin

→ EXIT

→ docker-compose restart airflow

What We Did After Airflow Was Connected

1. **Created and Enhanced the ETL Pipeline (Airflow DAG):**
    - **Goal:** Consume sensor data from multiple Kafka topics, transform it (e.g., round numeric values, add a processing timestamp), and insert it into a Postgres table (`sensor_data`).
    - **Key Steps:**
        - Wrote a DAG in `services/airflow/dags/factory_etl.py` that:
            - Subscribes to topics like `chassis_topic`, `battery_topic`, `paint_topic`, and `quality_topic`.
            - Applies a transformation function to the JSON data.
            - Inserts the transformed data into Postgres.
    - **Error Encountered:**
        - **ModuleNotFoundError: No module named 'kafka'**
          Solution: We built a custom Airflow Docker image to install `kafka-python` and `psycopg2-binary` (by creating a Dockerfile in `services/airflow/Dockerfile` and updating `docker-compose.yml` to use the custom build).
2. **Built a FastAPI Service to Serve Data:**
    - **Goal:** Provide an API endpoint (`/sensor-data`) that queries the Postgres table and returns the sensor data in JSON.
    - **Key Steps:**
        - Created a FastAPI app in `services/fastapi_app/main.py` that:
            - Connects to Postgres (using the Docker service name `postgres`).
            - Defines the endpoint `/sensor-data` to return the latest 100 records.
        - Tested the endpoint with a browser or command-line tool.

Commands to View JSON Data:
bash
Copy
```
curl http://localhost:8000/sensor-data
```

- This should display your sensor data in JSON format.

3. **Set Up a Dashboard for Data Visualization:**
    - **Goal:** Connect Tableau directly to your Postgres database.
    - **Steps:**
        - In Tableau, select the PostgreSQL connector.
        - Use these connection details:
            - **Server:** `localhost`
            - **Port:** `5432`

        - **Database:** `factory_db`
        - **Username:** `factory_user`
        - **Password:** `factory_pass`
    - Then choose the `sensor_data` table and build your visualizations.

```
Connect to PostGres:

docker-compose exec postgres psql -U factory_user -d factory_db
```

SELECT COUNT(*) FROM sensor_data;