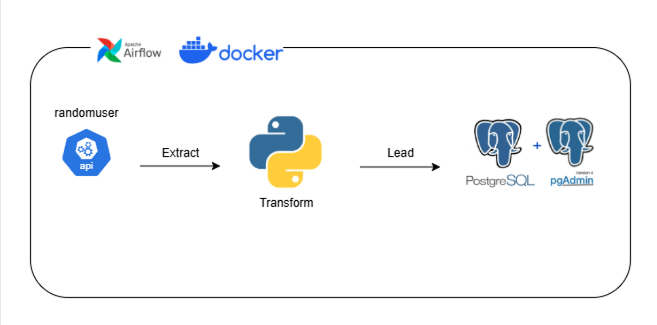
Criação de um pipeline de dados que extrai informações de uma API, transforma os dados com Python e carrega em um banco de dados

Para isso neste projeto vou usar Random User Generator, uma API gratuita e de código aberto para gerar dados aleatórios de usuários. Este tipo de API é ótima para treinar e ver a vulnerabilidade e usabilidade dos dados sem expor dados reais.

A ideia surgiu da necessidade de estruturar dados provenientes de uma API de forma organizada e confiável, garantindo consistência e integridade para futuras análises.

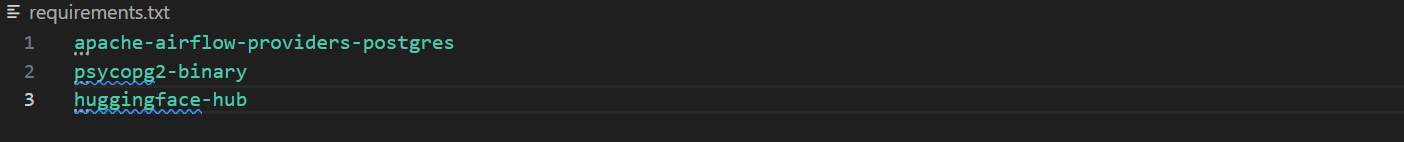
Arquitetura:

O pipeline foi composto por:

* **Coleta automática de dados**: utilização da **API Random User** para obter informação simulada de utilizadores.
* **Armazenamento inicial em base de dados relacional**: ingestão dos dados brutos na base **PostgreSQL**, especificamente na instância criada para o projeto (api\_random\_user).
* **Normalização e modelagem relacional**: criação de tabelas normalizadas diretamente em **SQL**, seguindo uma abordagem de *Data Warehouse*, com dimensões (dim\_user, dim\_address, dim\_login, dim\_id, dim\_picture) e uma tabela fato (fact\_user). Este processo garantiu consistência, eliminação de duplicados e integridade referencial.
* **Execução e validação em ambiente de administração**: utilização do **pgAdmin** para implementar queries de transformação, tratar tipos de dados (date, numeric, timestamp) e validar a qualidade da carga.

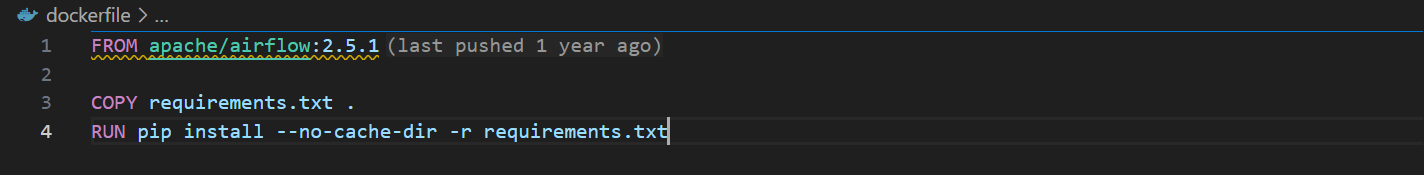
Este projeto ilustra um cenário prático de **ETL com foco em Data Engineering**, no qual dados de origem semi-estruturada são organizados em modelo relacional, servindo como base sólida para análises futuras e construção de dashboards.

O ficheiro **requirements.txt** é fundamental em qualquer projeto Python. Ele define todas as bibliotecas necessárias para que o projeto funcione corretamente. Com ele, qualquer pessoa pode preparar o ambiente de forma rápida e consistente. Este requirements.txt mostra que o projeto combina **orquestração de dados (Airflow)**, **armazenamento e consultas em PostgreSQL** e **modelos de inteligência artificial da Hugging Face**.

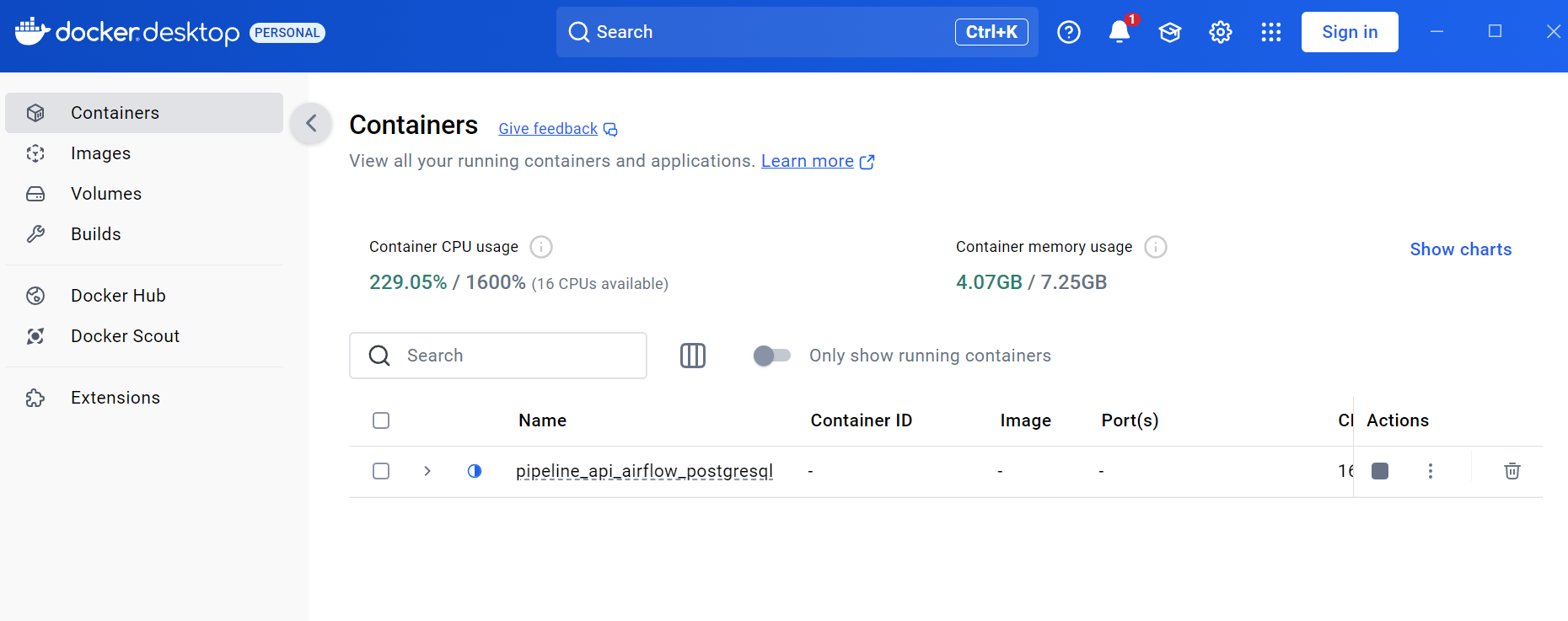


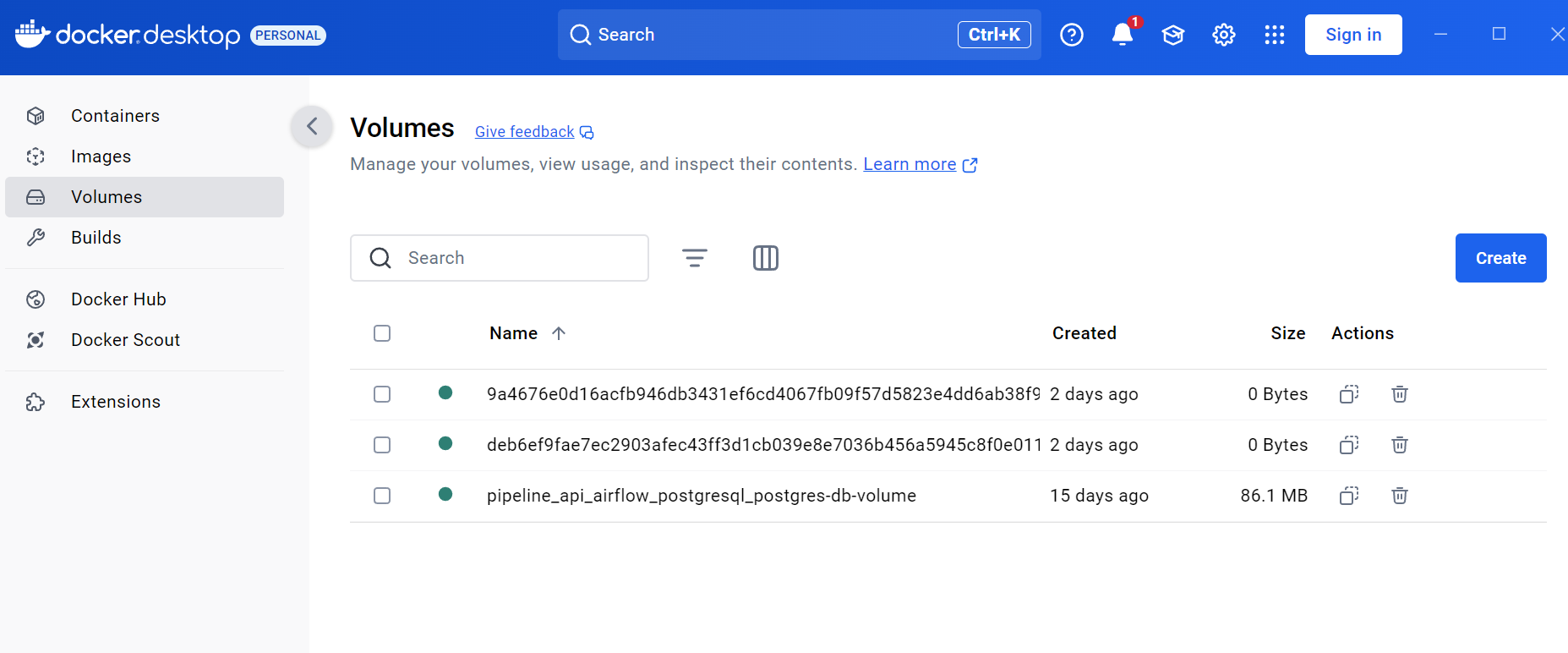
O **Dockerfile** é um ficheiro usado para personalizar o ambiente do **Airflow dentro do Docker**. Ele é útil quando os teus **DAGs precisam de bibliotecas Python extras** que não vêm instaladas na imagem oficial do Airflow. A função dele é manter o ambiente isolado dentro do Docker.

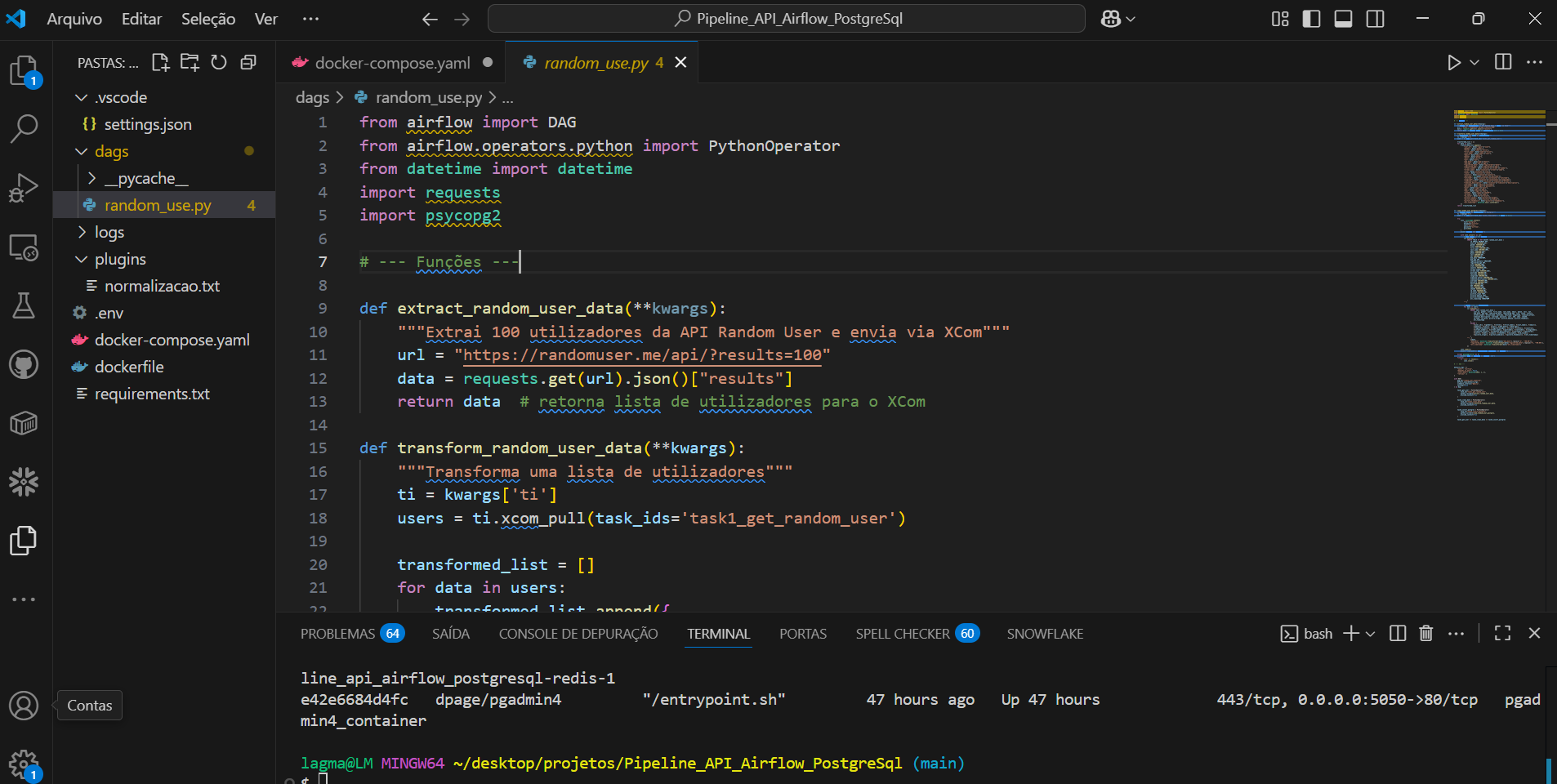
Um Dockerfile simples tem apenas 3 instruções:



Assim, quando executas docker compose up, o Docker constrói a imagem já com todas as dependências e inicia o Airflow preparado para as DAGs. O **Dockerfile é o passo que transforma o Airflow "padrão" no Airflow "à nossa medida"**, incluindo todas as bibliotecas extra que os teus projetos precisam.







O código apresentado define um pipeline **ETL (Extract, Transform, Load)** utilizando o **Apache Airflow** como orquestrador de tarefas. Ele tem como objetivo consumir dados da API pública **Random User**, transformá-los num formato estruturado e carregá-los para uma tabela no **PostgreSQL**.

O pipeline está organizado em três fases principais:

1. **Extração (Extract)**  
   A primeira tarefa, extract\_random\_user\_data, realiza uma requisição à API https://randomuser.me/api/, pedindo 100 utilizadores aleatórios. Os dados retornados em formato JSON são enviados para o mecanismo de comunicação do Airflow (**XCom**) para que possam ser utilizados pelas próximas etapas.
2. **Transformação (Transform)**  
   A segunda tarefa, transform\_random\_user\_data, lê os dados brutos da etapa anterior e reorganiza-os num dicionário padronizado, facilitando a carga na base de dados.
   * São extraídos atributos como nome, email, morada, localização geográfica, credenciais de login, entre outros.
   * Converte-se o campo postcode para string (garantindo consistência).
   * É adicionado um campo adicional chamado etl\_timestamp, que regista o momento exato em que a transformação ocorreu, útil para auditoria e rastreabilidade.
3. **Carga (Load)**  
   A terceira tarefa, load\_random\_user\_postgres, conecta-se a uma instância PostgreSQL utilizando a biblioteca **psycopg2**.
   * Se a tabela random\_user\_data ainda não existir, ela é criada com todas as colunas necessárias.
   * Cada registo é inserido na base de dados. Durante este processo, os campos de data são convertidos para o tipo TIMESTAMP do PostgreSQL.
   * Ao final, a transação é confirmada com commit() e a conexão é encerrada.

### Estrutura da DAG

A DAG chama-se **random\_user\_pipeline** e tem as seguintes configurações:

* **Execução diária** (@daily).
* **Início a 1 de janeiro de 2025**.
* **Reexecução automática em caso de falha** (1 retry).
* **Ordem de execução**:

task1\_get\_random\_user → task2\_clean\_data → task3\_insert\_postgres.

### Conclusão

Este pipeline exemplifica bem a utilização do Airflow para orquestrar processos de ETL:

* A **extração** automatiza a coleta de dados de uma API pública.
* A **transformação** garante consistência e qualidade nos dados antes de carregá-los.
* A **carga** integra os dados num repositório relacional (PostgreSQL), possibilitando análises posteriores.

Execução do Docker com os ficheiros todos já feitos :

**Verificar se o Docker está a correr**

docker ps

Mostra os containers em execução.

**Subir os serviços (iniciar Airflow + Postgres + Redis + etc.)**

docker compose up -d

**Parar os serviços**

docker compose down

Para e remove todos os containers definidos no docker-compose.yaml.

**Reconstruir a imagem (se alteraste o Dockerfile ou requirements.txt)**

docker compose build

E depois voltas a subir:

docker compose up -d

Resumindo o fluxo típico para trabalhares com o Airflow no Docker:

1. docker compose build (se tiveres Dockerfile personalizado).
2. docker compose up -d (sobe os serviços).
3. Abrir o Airflow no navegador: <http://localhost:8080>.
4. docker compose down (quando quiseres parar).

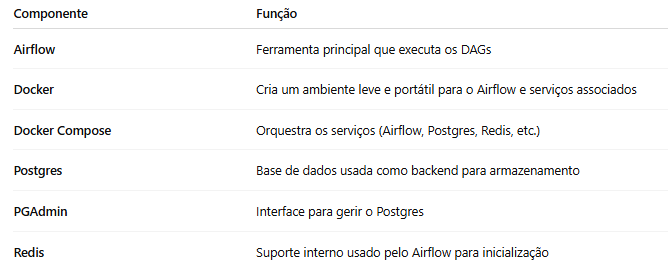
#### 1. O que é o Airflow

O Apache Airflow é uma ferramenta usada para orquestração de workflows, baseada em **DAGs (Directed Acyclic Graphs)**.

* **DAGs**: representam fluxos de tarefas que seguem uma ordem específica. Não há ciclos, ou seja, uma tarefa não pode voltar para a anterior.
* **Pipelines ETL**: esta estrutura torna o Airflow ideal para processos de Extração, Transformação e Carga (ETL), seguindo a sequência Extrair → Transformar → Carregar.

#### 2. Arquitetura do Airflow com Docker

A forma mais simples de instalar e executar o Airflow é com **Docker** e **Docker Compose**. A arquitetura típica inclui:



#### 3. Funcionamento de um DAG no Airflow

Os DAGs são definidos em ficheiros **Python**.  
Exemplo de um pipeline ETL que extrai dados da **API** e carrega para uma base Postgres:

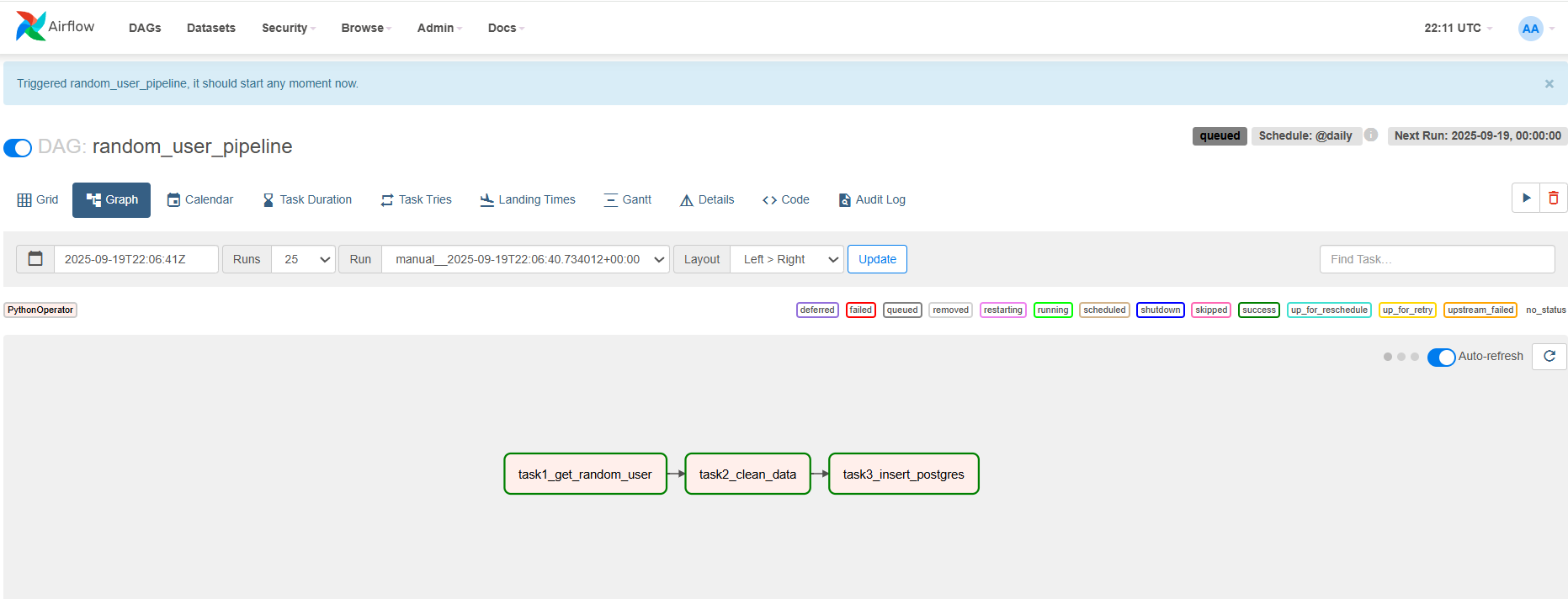
1. **Extract Task** - obtém dados (ex: 50 modelos mais recentes), envia os brutos para o **XCOM**.
2. **Transform Task** - recebe dados do XCOM, remove duplicados, trata nulos, normaliza colunas, envia dados limpos de volta ao XCOM.
3. **Load Task** - lê dados transformados do XCOM e insere no Postgres através de um **Postgres**, com conexão configurada no Airflow UI.

* **XCOM**: mecanismo de troca de dados entre tarefas.
* **PythonOperator**: define tarefas em Python dentro do DAG.
* **Conexões (Admin > Connections)**: configuram credenciais e parâmetros para bases externas (ex: host do Postgres no Docker, nome da base pg\_db).

#### 4. Gestão de Dependências

Para incluir bibliotecas necessárias no ambiente Airflow (ex: psycopg2, huggingface):

* **requirements.txt**: lista de pacotes Python.
* **Dockerfile**: copia o requirements.txt e instala dependências com pip.
* **docker-compose.yaml**: ajustado para usar a build personalizada, garantindo que a imagem do Airflow tenha todas as dependências.



### PostgreSQL e PG Admin no Pipeline Airflow

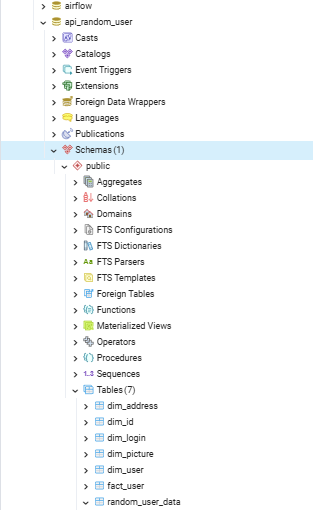
O **PostgreSQL (Psql/Postgress)** e o **PG Admin** são componentes essenciais para hospedar e gerir a base de dados utilizada pelo Apache Airflow em pipelines ETL.

#### PostgreSQL (Psql/Postgress)

* **Função no ETL**: Serve como banco de dados backend e destino final (Load) dos dados. Exemplo: extrair dados de modelos de IA, transformar e carregar no Psql.
* **Hospedagem**: Alojado no ambiente Docker e gerido pelo Docker Compose.
* **Conexão Airflow**: É necessário configurar a conexão no Airflow UI (Admin > Connections). O Host deve ser Postgress (com P maiúsculo) para que o Docker encontre o serviço automaticamente, mesmo que o IP mude.
* **Base de Dados**: Criar uma base específica, ex: PG\_DB.
* **Credenciais Padrão**: Username: airflow,
* Password: airflow,
* Porta: 5432.

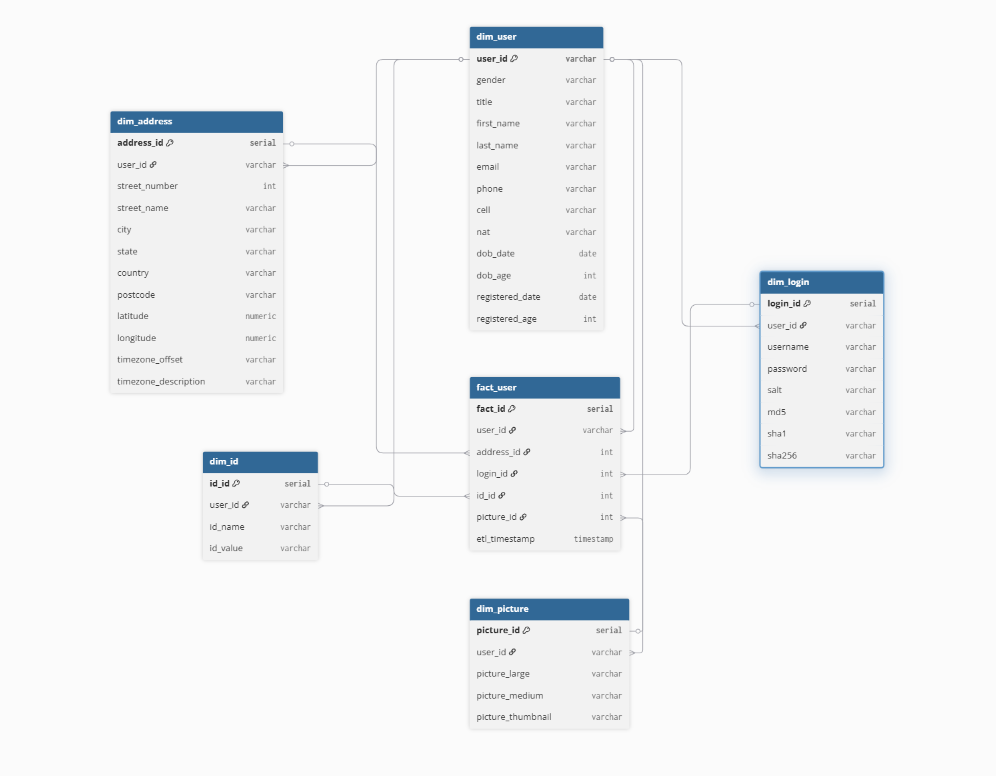
#### PG Admin

* **Função Principal**: Interface para explorar, gerir e consultar os dados no PostgreSQL. Permite verificar a inserção correta dos dados, e para confirmar se todos os modelos foram carregados.
* **Hospedagem em Docker**: Serviço incluído manualmente no docker-compose.yaml, normalmente abaixo do Postgres.
* **Configuração**: Definir credenciais e porta no Docker Compose:
  + Email: admin@admin.com
  + Password: root
  + Porta: 5050
* **Servidor PG Admin**: Após aceder à interface (localhost:5050), adicionar um novo servidor:
  + Nome do Host: postgress
  + Credenciais: Username airflow, Password airflow

Armazenar os dados processados em um banco de dados relacional:

Modelo Lógico:



Modelo Fisico:

