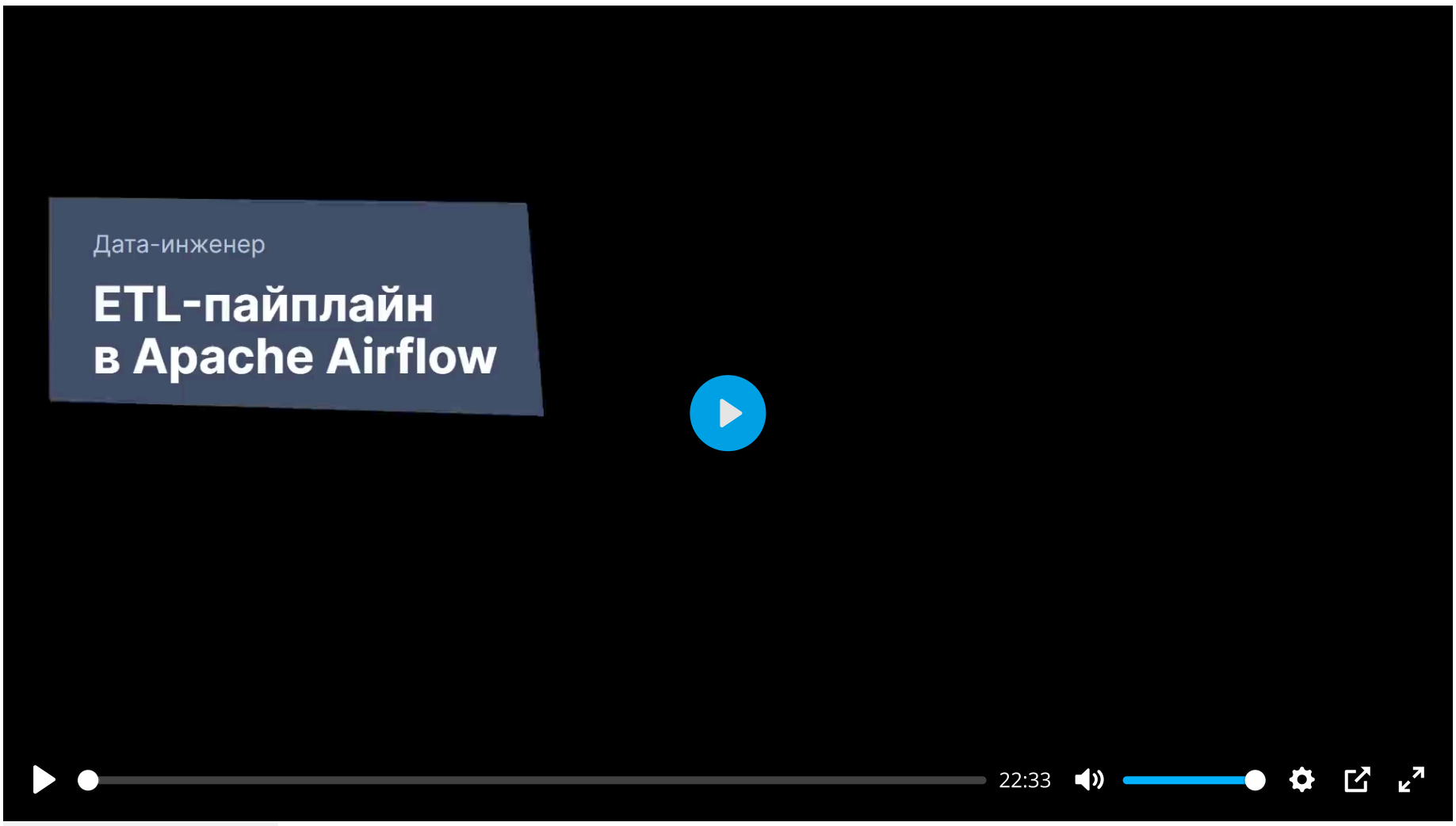


Презентация к уроку 8.3.2

Текстовая расшифровка видео:

ETL-ПАЙПЛАЙН В APACHE AIRFLOW

План:

- Первый пайплайн;
- Создание DAG;
- Tasks;
- Создание таблицы;
- Создание соединения;
- Создание оператора сенсора;
- Соединение с API;
- Извлечение данных из API (Extract);
- Обработка юзеров. PythonOperator;
- Функция обработки данных _process_user (Transform);
- Хранение пользователей (Load);
- Оператор store_user и DAG.

Первый пайплайн

Этапы:

- Сначала мы выгружаем строчки с рандомными юзерами из некого API.
- Эта информация будет преобразовываться и нормализовываться с помощью команды «JSON Normalize».

- Складываем информацию в базы данных.

Появление информации с юзером в базе данных означает, что все успешно.

Повторим:

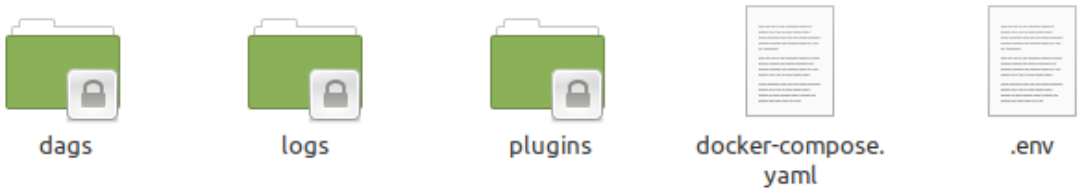
В данном пайплайне мы будем выгружать рандомно генерирующиеся юзеры, преобразовывать их и строить в базе данных.

В качестве примера базы данных станет MetaStore, но в продакшне лучше не использовать это для хранения каких-либо данных. MetaStore и Базу Данных необходимо разделять.

Первый шаг:

- **Extract** – это Extract External JSON Data;
- **Transform** – это нормализация (JSON Normalize);
- **Load** – это загрузка CSV в Postgres Database.

Создание DAG



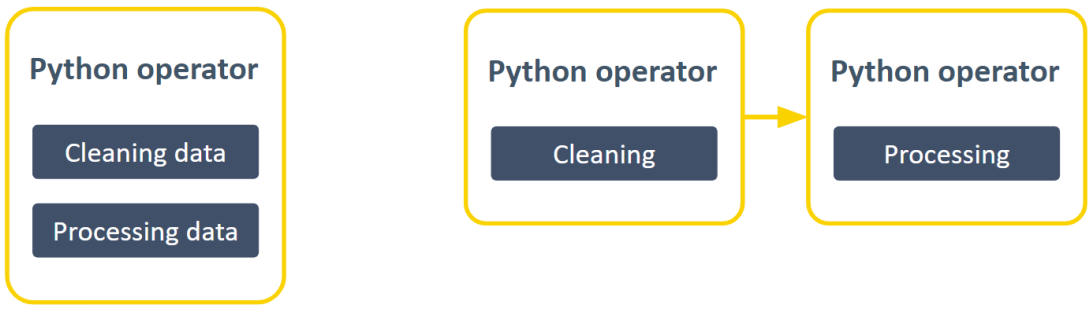
`user_processing.py`

Создадим простой DAG, в котором ничего не происходит, с заглушкой None. Для этого используем менеджер контекста. Мы задаем название, User processing, Start date, Catchup=false и интервал «ежедневный»:

```
from airflow import DAG
from datetime import datetime

with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=False)
    None
```

Затем мы будем создавать Python-оператор.

Tasks



Каждую функцию нужно логично разность по разным таскам. Каждый таск должен выполнять свою задачу по очищению, а затем по процессингу. Это упрощает работу, так как в случае поломки будет проще проследить по логам. Также это поможет избежать «затыка» на всем процессе, если одновременно обрабатываются несколько строчек.

Создание таблицы

Мы будем создавать таблицу с помощью Create table:

```
from airflow import DAG
from airflow.providers.postgres.operators.postgres import PostgresOperator
from datetime import datetime

with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=False)
    create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres', sql='''CREATE
```

Здесь используется postgres operator.

В Airflow много разных операторов – это connections ко внешним сервисам. Это необходимо для того, чтобы использовать внешний API. Операторы уже встроены, поэтому не нужно писать в python какую-то функцию для работы с postgres. Здесь вы указываете в полях Connection ID, а также указываете SQL-запрос, который необходимо сделать.

Создание соединения

Указываем Connection ID в Web-UI. Его параметры, следующие:

id postgres

type: Postgres

Host: postgres

login/password: airflow/airflow

port: 5432

Создание оператора сенсора

Сенсоры – это операторы, которые проверяют какое-то условие на выполнение, когда оно «True», пайплайн идет дальше.

В данном случае сенсор проверяет доступно ли API. Мы проверяем по данным Connection ID доступность данного адреса:

```
from airflow.providers.http.sensors.http import HttpSensor
…
is_api_available = HttpSensor(task_id='is_api_available', http_conn_id='user_api', endpoint='api/')
```

Соединение с API

Сайт Randomuser Me: https://randomuser.me/

Его API способен генерировать рандомные юзеры.

Извлечение данных из API (Extract)

Далее, мы будем извлекать данные, а через json.loads получать некоторую строку с помощью SimpleHttpOperator:

```
from airflow.providers.http.operators.http import SimpleHttpOperator
import json
…
extract_user = SimpleHttpOperator(task_id='extract_user', http_conn_id='user_api', endpoint='api/', me
```

SimpleHttpOperator – это еще один провайдер, который необходим для соединения с Http API.

Обработка юзеров. PythonOperator

Следующий шаг – нормализация юзеров. Будем делать это при помощи PythonOperator, однако из-за того, что функция длинная мы укажем в Python callable некоторое название process user (укажем его позднее):

```
from airflow.operators.python import PythonOperator
from pandas import json_normalize
…
process_user = PythonOperator(task_id='process_user', python_callable=_process_user,)
```

Функция _process_user объявляется отдельно вне менеджера контекста with. Поскольку это служебная функция, указываем с нижним подчеркиванием.

Функция обработки данных _process_user (Transform)

Здесь мы обращаемся к ti. Мы вытаскиваем некоторые значения xcom из предыдущего таска «Extract_user», обрабатываем эти результаты и помещаем в CSV-файл. CSV-файл будет храниться на воркере:

```
def _process_user(ti):
    user = ti.xcom_pull(task_ids="extract_user")
    user = user['results'][0]
    processed_user = json_normalize({ 'firstname': user['name']['first'],'lastname': user['name']['las
        'password': user['login']['password'], 'email': user['email']})
    processed_user.to_csv('/tmp/processed_user.csv', index=None, header=False)
```

Хранение пользователей (Load)

Далее используем Hooks. Это еще один способ соединения Airflow с внешними сервисами:

```
from airflow.providers.postgres.hooks.postgres import PostgresHook
…
```

Postgres Hook – это крючок, который соединяет Airflow с базой Postgres.

Мы указываем наш connection postgres (можно использовать тот же самый, который был указан ранее, когда мы создавали таблицу).

Hook имеет встроенный метод «Copy_expert». Здесь мы пишем SQL-запрос, копируем user из поступающего потока с разделителем «запятая», получаем эти данные из CSV-файла:

```
def _store_user():
    hook = PostgresHook(postgres_conn_id='postgres',)
    hook.copy_expert(sql="COPY users FROM stdin WITH DELIMITER as ',' ", filename='/tmp/processed_user
```

//В менеджере контекста:

```
store_user = PythonOperator(task_id='store_user', python_callable=_store_user)
```

Оператор store_user и DAG

На этом моменте мы получаем полностью готовую программу:

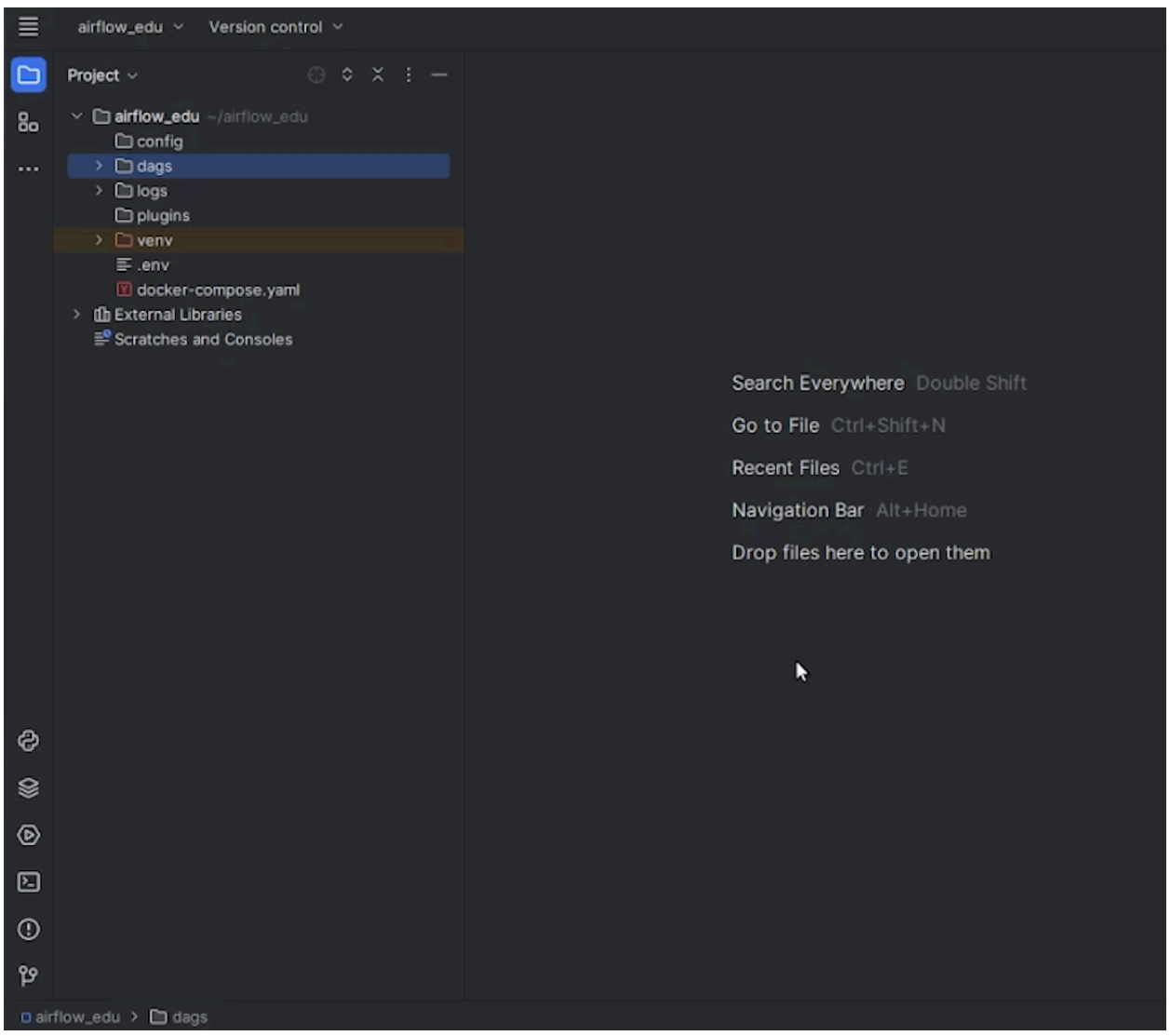
```
with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=False)
    create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres', sql='''
        CREATE TABLE IF NOT EXISTS
        users(firstname TEXT NOT NULL, lastname TEXT NOT NULL,  country TEXT NOT NULL,
        username TEXT NOT NULL, password TEXT NOT NULL, email TEXT NOT NULL); ''')
    is_api_available = HttpSensor(task_id='is_api_available', http_conn_id='user_api', endpoint='api/'
    exctract_user = SimpleHttpOperator(task_id='extract_user', http_conn_id='user_api', endpoint='api/
    process_user = PythonOperator(task_id='process_user', python_callable=_process_user,)
    store_user = PythonOperator(task_id='store_user', python_callable=_store_user)
```

Создадим первый пайплайн. Переходим в директорию DAG's (на данный момент у нас ее нет):

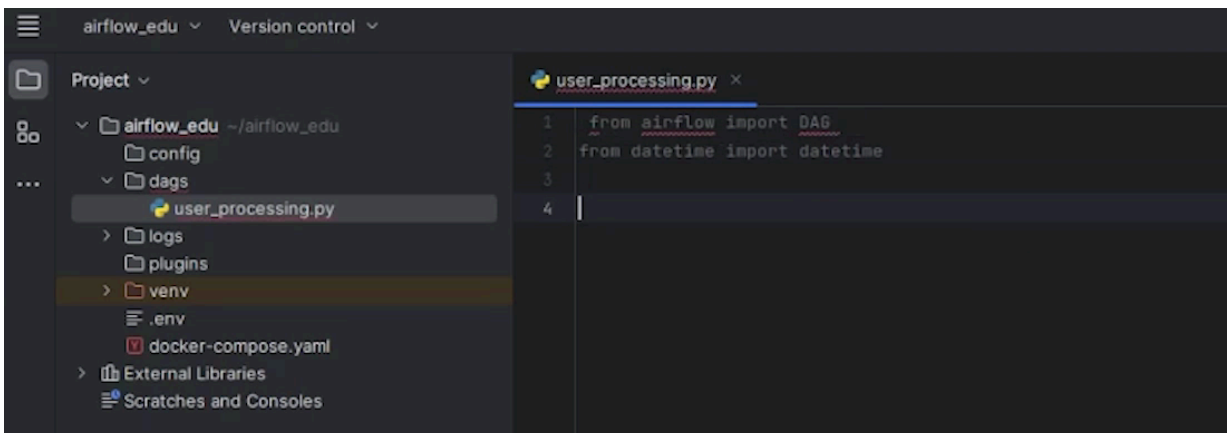
```
asya@asya-Aspire-XC-886:~/airflow_edu$ cd dags
asya@asya-Aspire-XC-886:~/airflow_edu/dags$ ls
asya@asya-Aspire-XC-886:~/airflow_edu/dags$ nano user_processing.py
```

Создаем **user_processing.py**

Сохраним пустой файл. Чтобы проще было работать, открываем PyCharm и в нем открываем файл (он хранится в папке «DAG's»):



Вначале добавляем необходимые нам импорты: импорт DAG и datetime:



С помощью контекста или стандартного конструктора всегда нужно делать «From airflow import DAG»:

```python
from airflow import DAG
from datetime import datetime

with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=Fal
    None
```

Создадим DAG с заглушкой None. Здесь мы указываем все параметры DAG (user processing, startdate, scheduler interval, catchhup=False).

Зададим функцию «Create table». **Create table** – наш postgres-оператор. ID таска здесь – Create table (первый параметр Task ID), второй – connection id postgres, а также сам запрос, где мы задаем поля таблицы и их типы. Таблица называется «Users»:

```
Project
airflow_edu ~/airflow_edu
  config
  dags
    user_processing.py
  logs
  plugins
  venv
  .env
  docker-compose.yaml
External Libraries
Scratches and Consoles
```

```python
from airflow import DAG
from datetime import datetime

with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=Fa

    create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres',
                                    sql='''CREATE TABLE IF NOT EXISTS users(firstname TEXT NOT NU
```

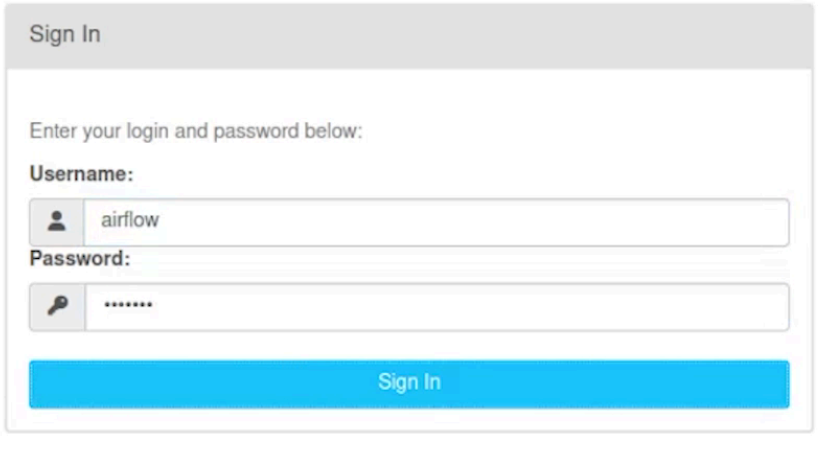
Postgres-оператор тоже нужно будет добавить в импорты:

```python
from airflow import DAG
from datetime import datetime
from airflow.providers.postgres.operators.postgres import PostgresOperator

with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=Fal


    create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres',
                                    sql='''CREATE TABLE IF NOT EXISTS users(firstname TEXT NOT NUL
```

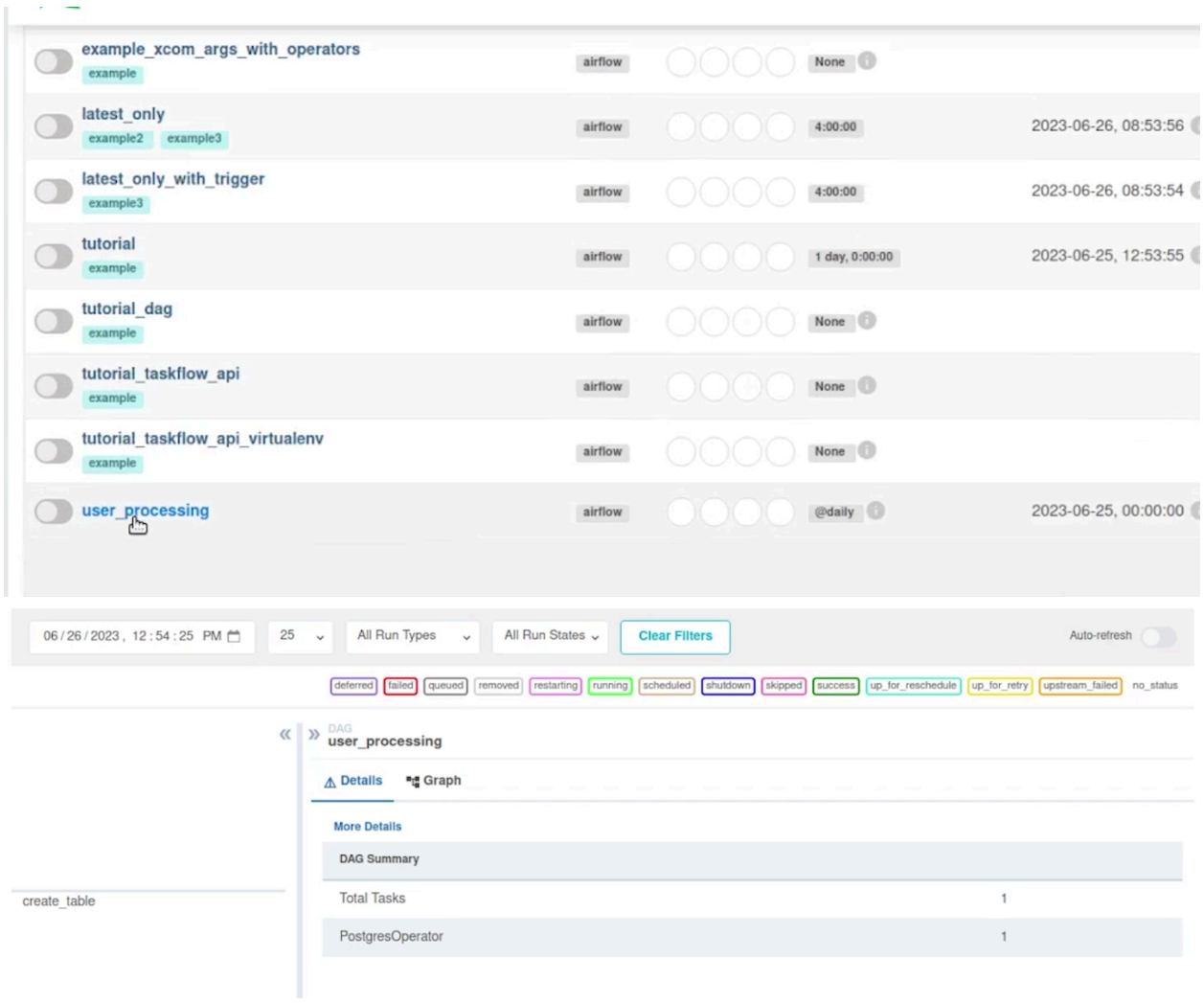
Запускаем Instance docker по уже известной команде «docker compose up -d». Ожидаем некоторое время:

```
asya@asya-Aspire-XC-886:~/airflow_edu$ cd dags
asya@asya-Aspire-XC-886:~/airflow_edu/dags$ ls
asya@asya-Aspire-XC-886:~/airflow_edu/dags$ nano user_processing.py
asya@asya-Aspire-XC-886:~/airflow_edu/dags$ cd ..
asya@asya-Aspire-XC-886:~/airflow_edu$ docker compose up -d
[+] Building 0.0s (0/0)
[+] Running 8/8
 ✔ Network airflow_edu_default                Created
 ✔ Container airflow_edu-postgres-1           Healthy
 ✔ Container airflow_edu-redis-1              Healthy
 ✔ Container airflow_edu-airflow-init-1       Exited
 ✔ Container airflow_edu-airflow-scheduler-1  Started
 ✔ Container airflow_edu-airflow-worker-1     Started
 ✔ Container airflow_edu-airflow-triggerer-1  Started
 ✔ Container airflow_edu-airflow-webserver-1  Started
asya@asya-Aspire-XC-886:~/airflow_edu$
```

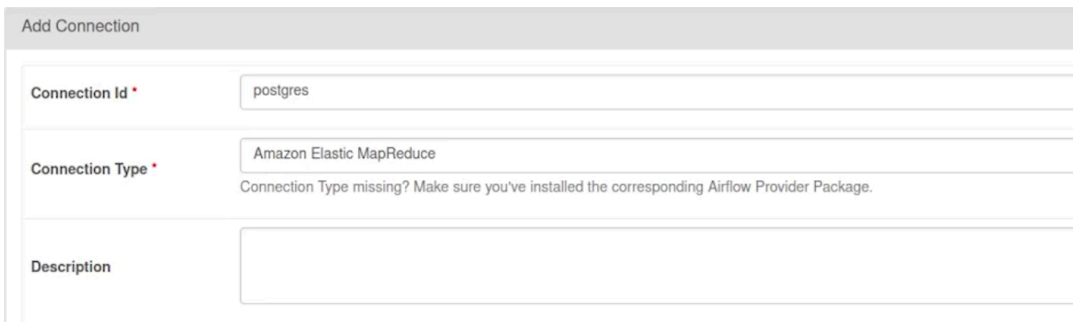
Заходим в Web-UI, где username и пароль – **airflow**:

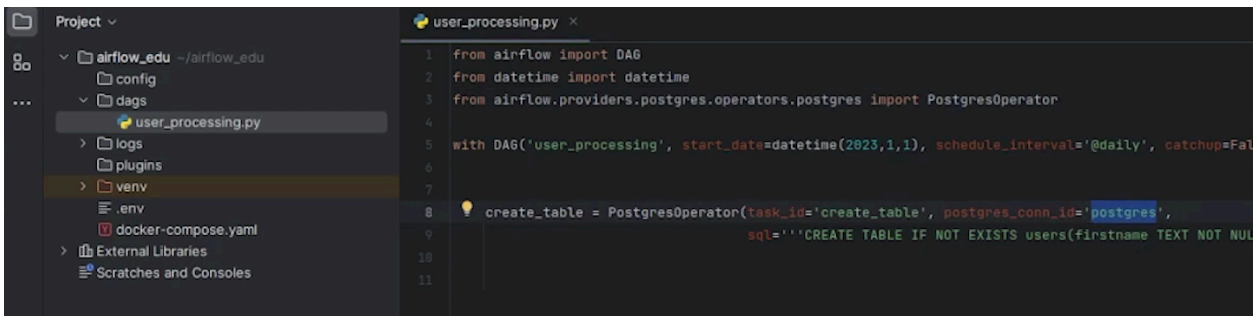


Среди большого количества примеров должен находиться **user_processing.py**:

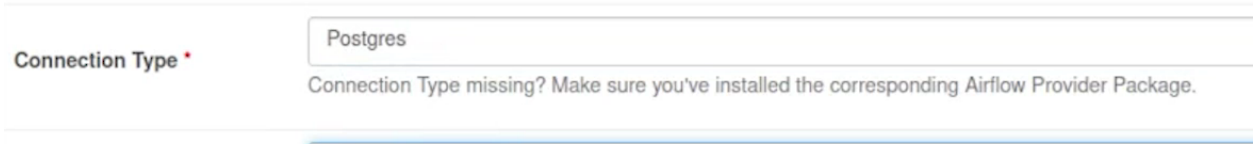


Добавляем connection к postgres. Прописываем такой же connection ID, какой мы прописали в коде postgres. Проверяем, чтобы написание было идентичным:





Далее, connection type:

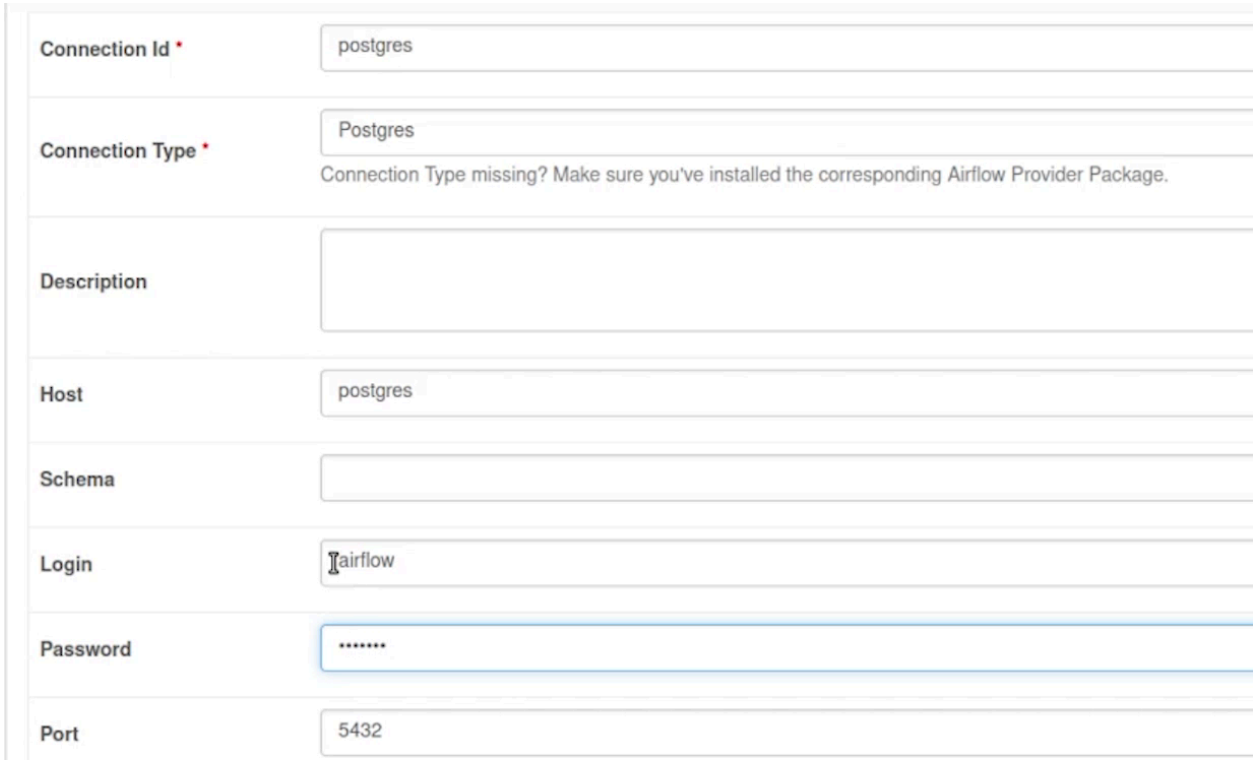


Description лучше указывать, так как все connections всех DAG’s хранятся здесь. Это необходимо, чтобы не запутаться.

В Host указываем postgres, поскольку в нашей сетке Docker Compose данный контейнер называется «Postgres».

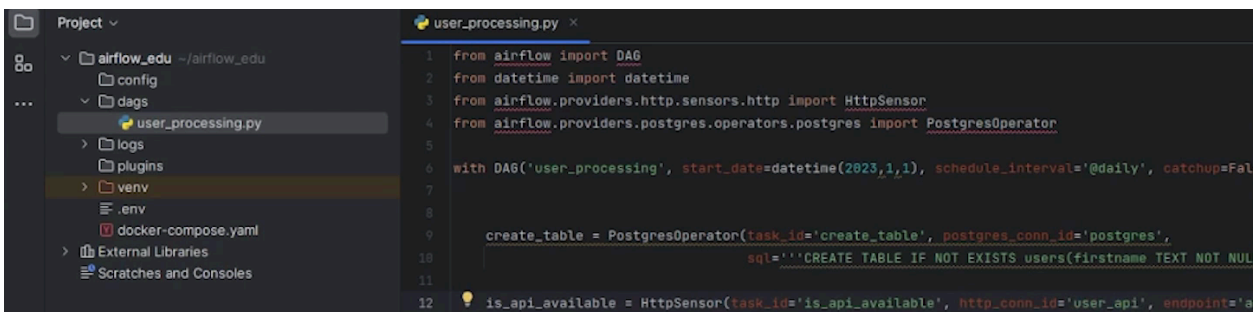
Логин и пароль – **airflow**

В Port по умолчанию – **5432**:



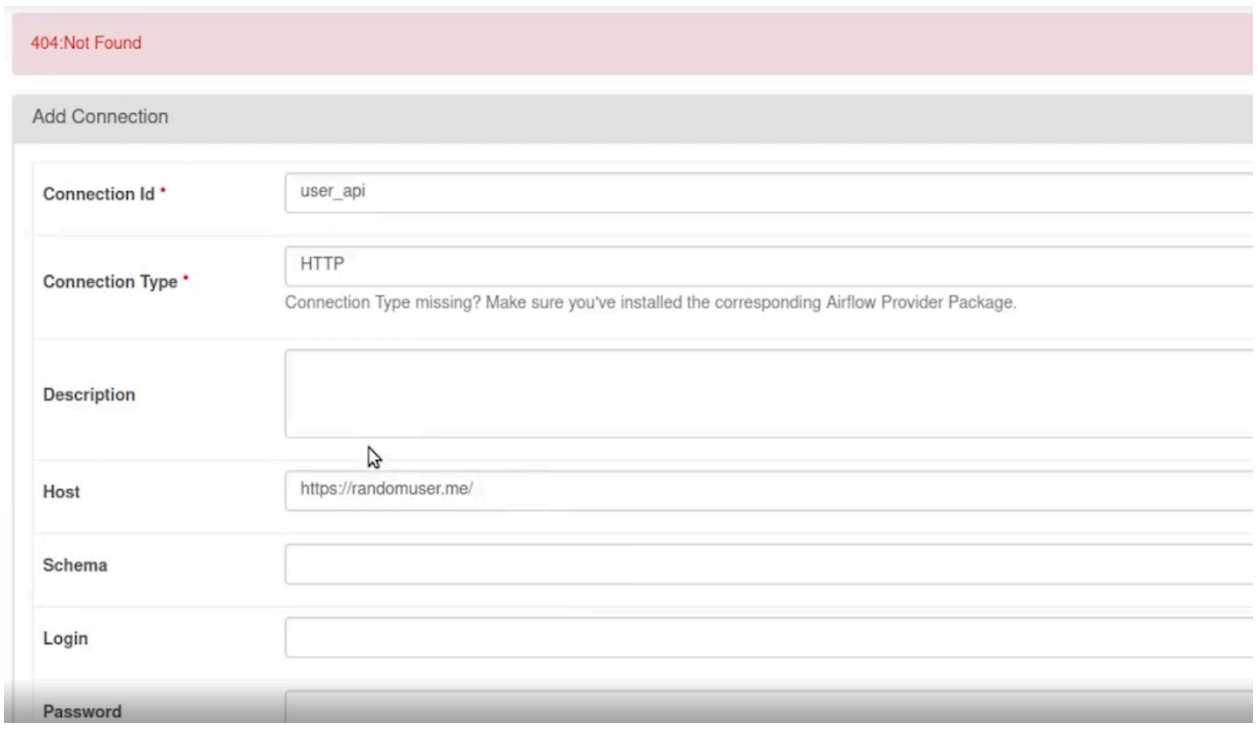
Тестируем. Успешно!

Когда connection добавлен, create table должен заработать, но у нас есть и другие операции. Сейчас мы импортируем https-сенсор для того, чтобы проверить API. Указываем ID «`is_api_available`», connection ID «`user_api`» и endpoint «`ap/`» (в какое место сайта необходимо зайти):



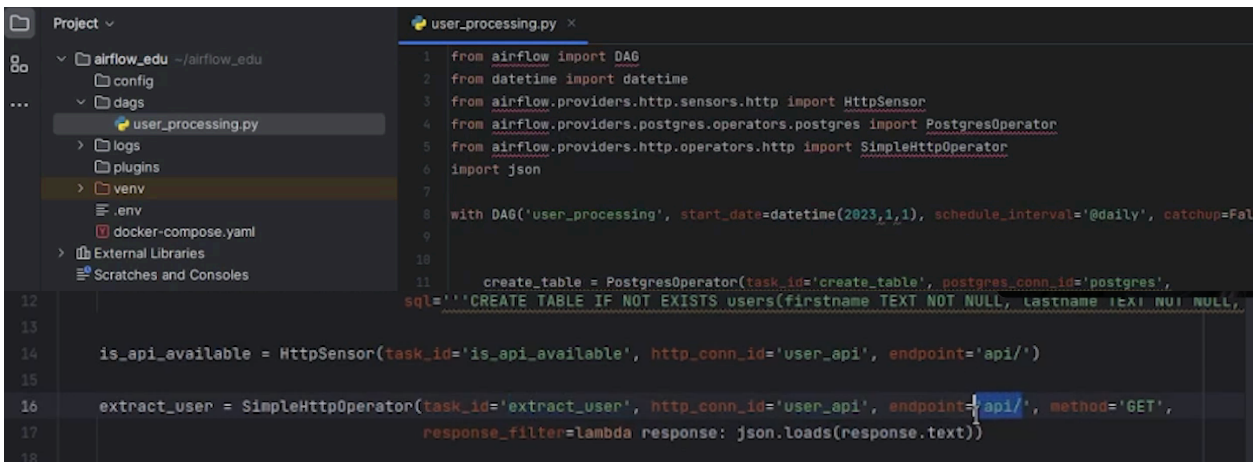
Добавляем аналогичным способом: такой же connection ID как в коде; название – «HTTP» (connection type должен быть именно таким). Копируем connection ID и вставляем.

Важно: connection ID может не проходить, но в коде все сработает корректно:

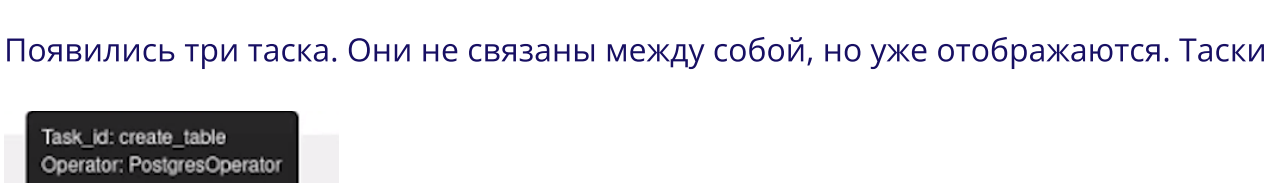


После того, как мы проверили доступность API, необходимо получить данные. Для этого мы используем SimpleHttpOperator и библиотеку JSON для преобразования данных. Библиотека JSON полезна, когда вы работаете с API. Данные почти всегда находятся в этом формате.

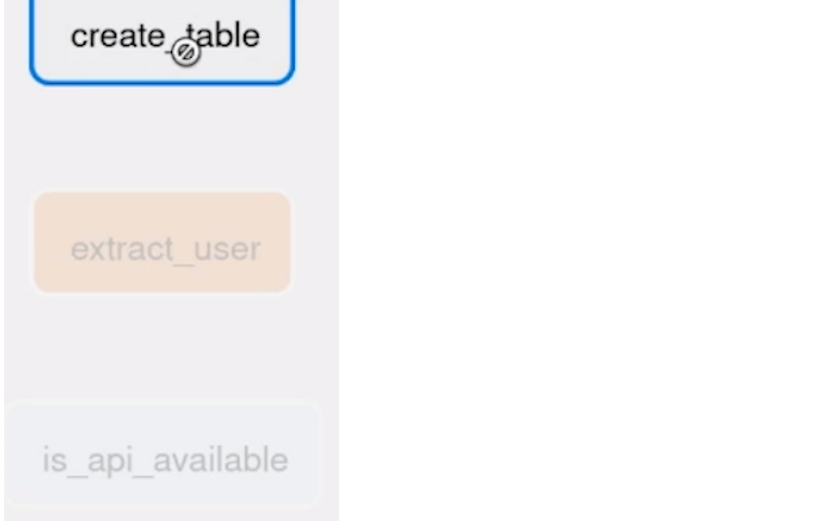
Здесь мы указываем названия тасков, connection ID, endpoint, метод «get» и функцию, преобразовывающую данные:



В данном случае из ответа мы берем text. При помощи JSON преобразовываем строку. Это и есть наш вывод:



Здесь мы можем посмотреть, как выглядит DAG:



Появились три таска. Они не связаны между собой, но уже отображаются. Таски еще не были запущены:

Если вы хотите проверить подтянулся ли код, можно проверить здесь:

Пишем Python-оператор:

```
22      process_user = PythonOperator(task_id='process_user', python_callable=_process_user, )
23
```

Для этого делаем импорт Python-оператор, используем метод процессинга. Здесь мы создаем ссылку на некоторые process user, на некоторую функцию, добавляем функцию до менеджера контекста. Здесь мы используем ti:

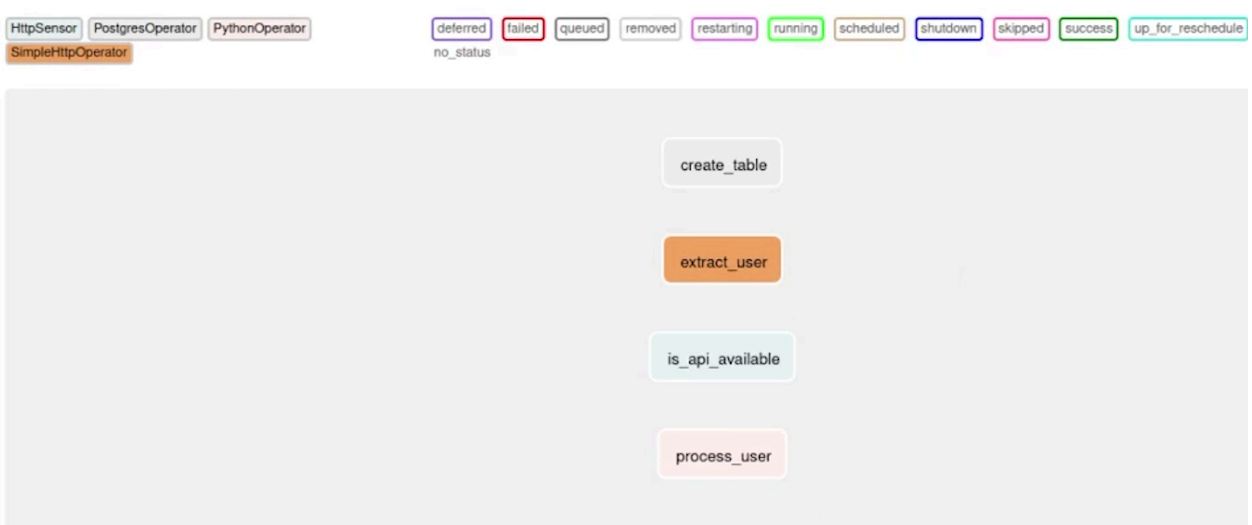
```
10
    1 usage
11  def _process_user(ti):
12      user = ti.xcom_pull(task_ids="extract_user")
13      user = user['results'][0]
```

```
13      user = user['results'][0]
14      processed_user = json_normalize({ 'firstname': user['name']['first'],
15                                        'lastname': user['name']['last'],
16                                        'country': user['location']['country'], 'username': user['login']['username'],
17          'password': user['login']['password'], 'email': user['email']})
18      processed_user.to_csv('/tmp/processed_user.csv', index=None, header=False)
19
20
21  with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=False) as dag:
22
23
24      create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres',
25                                      sql='''CREATE TABLE IF NOT EXISTS users(firstname TEXT NOT NULL, lastname TEXT NOT NULL,
26
27      is_api_available = HttpSensor(task_id='is_api_available', http_conn_id='user_api', endpoint='api/')
28
29      extract_user = SimpleHttpOperator(task_id='extract_user', http_conn_id='user_api', endpoint='api/', method='GET',
30                                        response_filter=lambda response: json.loads(response.text))
31
32      process_user = PythonOperator(task_id='process_user', python_callable=_process_user, )
33
_process_user()  'country'
```

Получаем xcom'ы от предыдущего таска Extract user, то есть некоторую информацию, которую выдал предыдущий таск. Обрабатываем полученные результаты, нормализуем их, помещаем в CSV в указанную директорию tmp без заголовков и индексов:

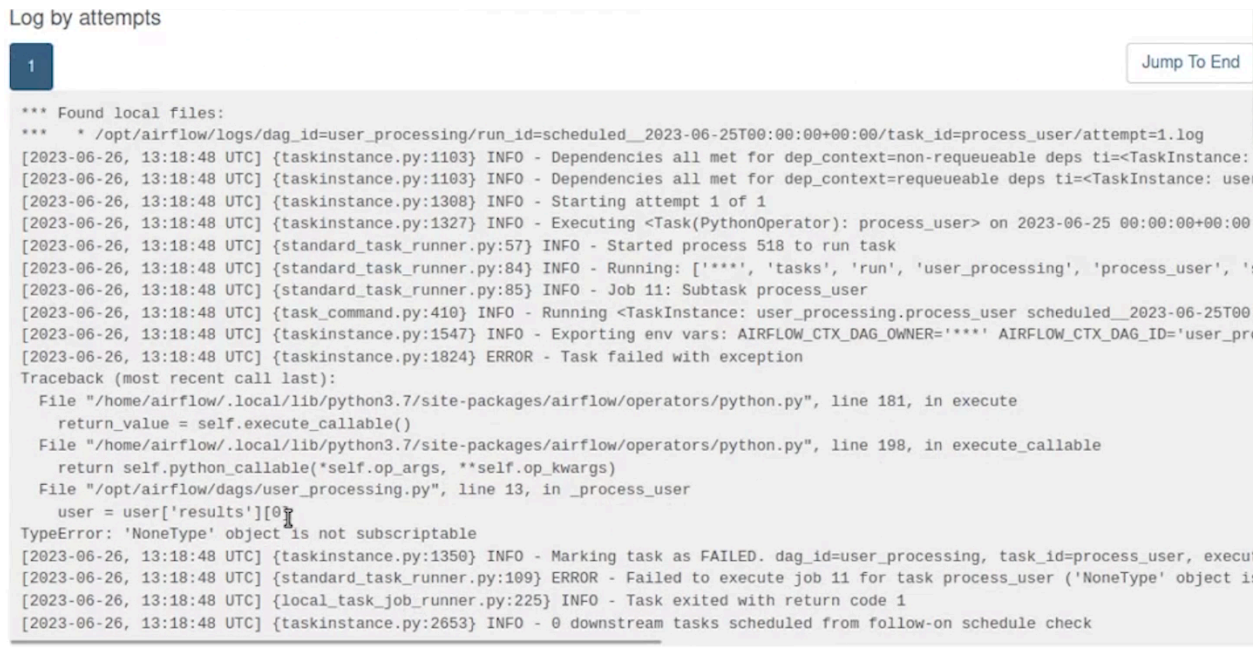
```
    1 usage
11  def _process_user(ti):
12      user = ti.xcom_pull(task_ids="extract_user")
13      user = user['results'][0]
14      processed_user = json_normalize({ 'firstname': user['name']['first'],
15                                        'lastname': user['name']['last'],
16                                        'country': user['location']['country'],
17                                        'username': user['login']['username'],
18                                        'password': user['login']['password'],
19                                        'email': user['email']})
20      processed_user.to_csv('/tmp/processed_user.csv', index=None, header=False)
20      processed_user.to_csv('/tmp/processed_user.csv', index=None, header=False)
21
22
23  with DAG('user_processing', start_date=datetime(2023,1,1), schedule_interval='@daily', catchup=False) as dag:
24
25
26      create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres',
27                                      sql='''CREATE TABLE IF NOT EXISTS users(firstname TEXT NOT NULL, lastname TEXT NOT NULL,
28
29      is_api_available = HttpSensor(task_id='is_api_available', http_conn_id='user_api', endpoint='api/')
30
31      extract_user = SimpleHttpOperator(task_id='extract_user', http_conn_id='user_api', endpoint='api/', method='GET',
32                                        response_filter=lambda response: json.loads(response.text))
33
34      process_user = PythonOperator(task_id='process_user', python_callable=_process_user, )
_process_user()  'email'
```

Появился оператор Python. Прямоугольники разного цвета из-за разного типа операторов. «Рамочки» также отличаются в зависимости от статуса. На данный момент они все белые, так как DAG не был запущен:

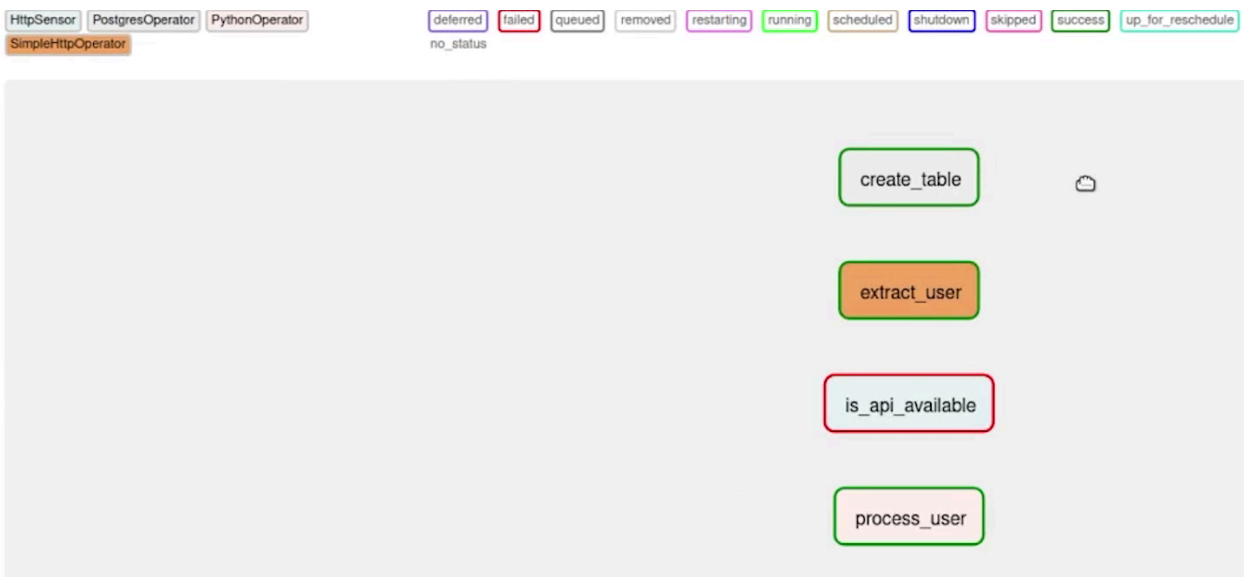


Запускаем DAG и видим ошибку:

Нам нужно делать процессинг юзера, но поскольку нет связности между тасками, в логах видим, что данных нет. Мы видим, что таски выполняются не по порядку, они не успели подгрузиться прежде, чем начал выполняться task process users:



Если мы попробуем запустить еще раз, то process users сработает, так как в папке tmp появился CSV-файл:



Как только появился CSV-файл, process users будет происходить. Однако не факт, что это тот process users, который мы достали в этом DAG Run. Он мог сохраниться из предыдущего из-за отсутствия связности тасков.

Таким образом не очень корректно запускать таски, чуть позднее мы добавим связность. Пока мы добавляем store user (сохранение их в таблицу):

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

    create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres',
                                    sql='''CREATE TABLE IF NOT EXISTS users(firstname TEXT NOT NULL, lastname TEXT NOT NULL,

    is_api_available = HttpSensor(task_id='is_api_available', http_conn_id='user_api', endpoint='api/')

    extract_user = SimpleHttpOperator(task_id='extract_user', http_conn_id='user_api', endpoint='api/', method='GET',
                                      response_filter=lambda response: json.loads(response.text))

    process_user = PythonOperator(task_id='process_user', python_callable=_process_user, )

    store_user = PythonOperator(task_id='store_user', python_callable=_store_user)
```

Это еще один оператор Python и еще одна функция. Внутри мы используем postgres hook, который соединяет с postgres через тот же connection и выполняет запрос, используя встроенную функцию «Copy», вставляя данные в postgres:

```python
1 usage
def _store_user():
    hook = PostgresHook(postgres_conn_id='postgres',)
    hook.copy_expert(sql="COPY users FROM stdin WITH DELIMITER as ',' ", filename='/tmp/processed_user.csv')
```

Stdin – считанный файл с CSV.

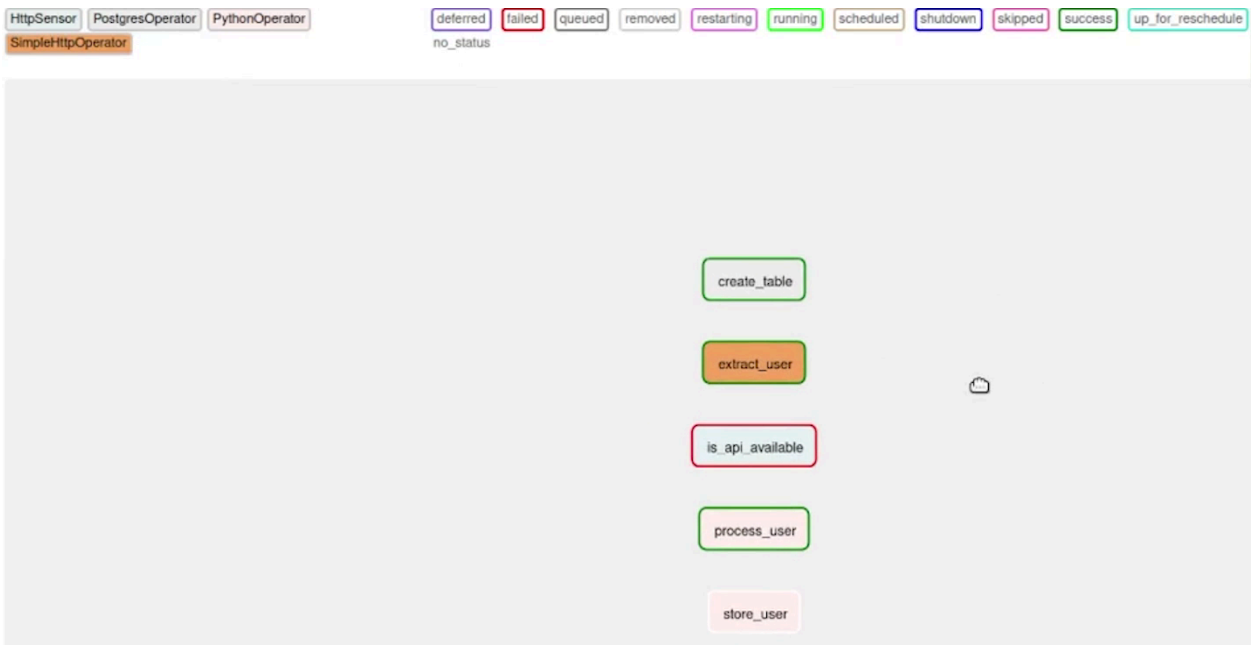
Так выглядит таблица «Users», и мы загрузили в нее данные:

user_processing.py

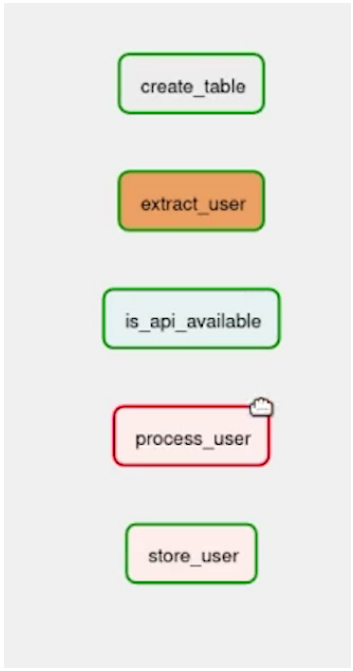
```python
    user = user['results'][0]
    processed_user = json_normalize({'firstname': user['name']['first'],
                                     'lastname': user['name']['last'],
                                     'country': user['location']['country'],
                                     'username': user['login']['username'],
                                     'password': user['login']['password'],
                                     'email': user['email']})
    processed_user.to_csv('/tmp/processed_user.csv', index=None, header=False)

1 usage
def _store_user():
    hook = PostgresHook(postgres_conn_id='postgres',)
```

```python
    hook = PostgresHook(postgres_conn_id='postgres',)
    hook.copy_expert(sql="COPY users FROM stdin WITH DELIMITER as ',' ", filename='/tmp/processed_user.csv')


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


    create_table = PostgresOperator(task_id='create_table', postgres_conn_id='postgres',
                                    sql='''CREATE TABLE IF NOT EXISTS users(firstname TEXT NOT NULL, lastname TEXT NOT NULL,

    is_api_available = HttpSensor(task_id='is_api_available', http_conn_id='user_api', endpoint='api/')

    extract_user = SimpleHttpOperator(task_id='extract_user', http_conn_id='user_api', endpoint='api/', method='GET',
                                      response_filter=lambda response: json.loads(response.text))

    process_user = PythonOperator(task_id='process_user', python_callable=_process_user, )

    store_user = PythonOperator(task_id='store_user', python_callable=_store_user)
```

Обновляем и видим, что появилась еще одна задача загрузки юзера в таблицу:



Если теперь все запустить, предварительно очистив все таски, то в первый раз происходит процессинг юзеров без данных, а process users сильно завязан на рандоме порядка выполнения:



Мы видим, что store user выполнился, то есть никакие данные не загрузились. Если Stdin-поток пустой, то это тоже считается успешным выполнением.

Мы построили пайплайн, но без связности. У нас есть отдельные операторы, которые выполняют свои задачи. На следующем уроке мы добавим связность.

Как вам урок?

Изучил, далее