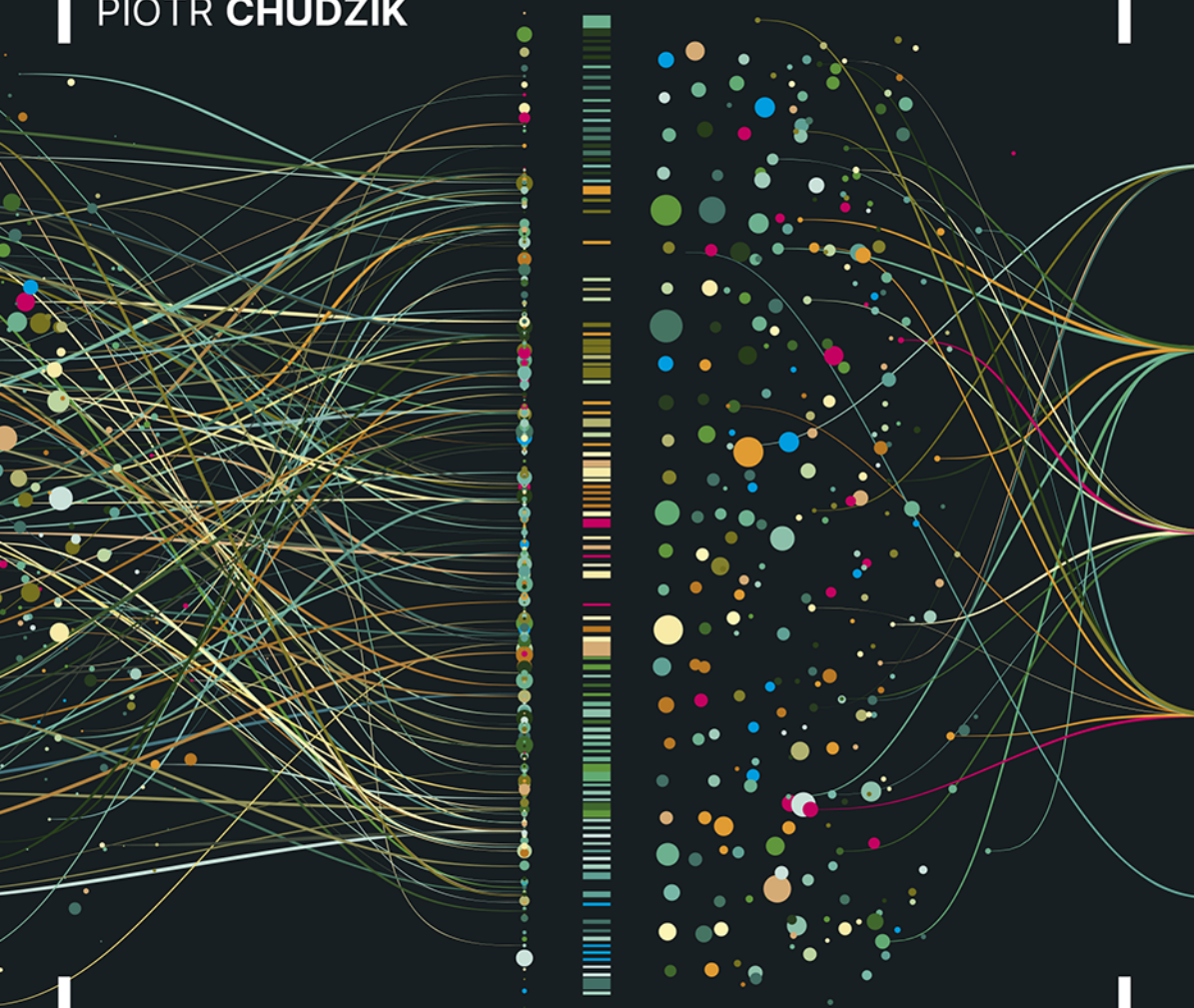
PIOTR **CHUDZIK**



Airflow

Monitorowanie przepływu danych

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą AdobeStock.com.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/airflo
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-289-1906-8

Printed in Poland.

Spis treści

Od autora

W dzisiejszych czasach generujemy coraz więcej danych, a co za tym idzie — coraz więcej informacji. Dane mogą mieć różne formaty: od prostych plików tekstowych po filmy i obrazy. Każdy z tych obiektów może zawierać wiele przydatnych informacji, które następnie mogą być wykorzystane do podejmowania właściwych decyzji. Poza formatem danych należy również pamiętać o sposobie przechowywania i przenoszenia danych. W tym celu wykorzystujemy wspomniane pliki przechowywane na dysku twardym, w chmurze lub nawet jako załączniki do wiadomości w poczcie elektronicznej. Wiele informacji przechowujemy też w specjalnych środowiskach (np. bazach danych), które również dzielą się zależnie od ich cech oraz przeznaczenia (np.: relacyjne bazy danych, bazy dokumentowe, bazy grafowe).

Aby móc wykorzystać pełen potencjał danych, musimy często wykonywać na nich odpowiednie procesy analityczno-agregujące, by uzyskać jak najwięcej informacji, które następnie pozwolą nam na wyciągnięcie wniosków i podjęcie decyzji. Tutaj może się pojawić problem: w jaki sposób pogodzić rozmaite formaty danych, odpowiednio je ze sobą połączyć, wykonać mapowanie i konwertowanie itd.? Istnieje wiele formatów, ale ostatecznie zależy nam na jednym, wspólnym formacie danych pozwalającym na sprawne i czytelne dla człowieka analizy. Dodatkowo narzędzi do wykonywania analizy i procesowania danych jest na rynku sporo, przykładowo: Apache Spark, Microsoft Power BI, platforma Databricks czy biblioteki Pandas i NumPy. To generuje kolejny problem: jak pogodzić nie tylko różne formaty danych, lecz także różne środowiska i narzędzia? Tutaj z pomocą przychodzą tzw. orkiestratory zadań, do których należy Apache Airflow Orkiestratory służą do zarządzania zadaniami w ramach danego problemu.

Jako osoba techniczna starałem się pisać tę książkę w sposób raczej „dokumentowy". Skupiałem się głównie na elementach praktycznych — znajdziesz tutaj przede wszystkim opis poszczególnych modułów narzędzia Apache Airflow, przeprowadzę Cię przez proces instalacji i przygotowania środowiska pracy oraz wyróżnię poszczególne elementy Airflow, abyś zakończył lekturę nie tylko z wiedzą na temat orkiestratora, ale również z dobrymi praktykami. Do książki zostały dołączone materiały, które zawierają więcej przykładów niż te, które znajdziesz na poniższych stronach, dlatego warto o nich pamiętać — dzięki temu możesz uniknąć niepotrzebnego przepisywania z grafik.

Nie przedłużając — życzę przyjemnej i owocnej nauki.

ROZDZIAŁ 6.

Hooki i pozostałe operatory

Hooki

Poza zmiennymi i innymi obiektami do przechowywania informacji istnieją również specjalne obiekty, które są zależne od dostawcy (*provider*) i narzędzia, z którego zamierzam korzystać. Mowa o hookach (ang. *hooks*), nazywanych przeze mnie uchwytami. Przykładowo: dla PostgreSQL-a uchwyt pozwoli na wykonywanie poleceń narzędzi „wewnętrznych", jak `bulk_dump`, a dla Apache Spark istnieje hook pozwalający na wykonanie polecenia `spark_submit`. Możesz zauważyć, że narzędzia dostępne za pomocą tego obiektu są często dostępne również wewnątrz operatorów i sensorów. Ich największą zaletą jest integracja w taki sam sposób jak pozostałe obiekty poznane w tej książce, czyli mają dostęp do informacji z zakładki *Variables*, połączeń z *Connections* czy szablonów Jinja. Dzięki temu możemy rozszerzyć funkcjonalność naszych zadań (szczególnie tych opartych na Pythonie), mając dostęp do wszystkich elementów środowiska Apache Airflow w bezpieczny sposób (np. nie musimy oddzielnie definiować połączeń do bazy danych).

Wykorzystajmy w naszym przykładzie `PostgresHook`, ponieważ mamy już zainstalowane zależności.

Wersja klasyczna — *postgres_hook.py*

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook

def export_data_to_csv(tbl_name):
    hook = PostgresHook(postgres_conn_id="example_db_postgres")
    hook.bulk_dump(tbl_name, f"/tmp/{tbl_name}.csv")

with DAG(dag_id="postgres_hook", start_date=None, schedule=None, tags=["r7", "with"]):
    t_export_data = PythonOperator(
        task_id="export_data",
        python_callable=export_data_to_csv,
        op_args=["customers"]
    )

t_export_data
```

Wersja TaskFlow — *postgres_hook_tf.py*

```
from airflow.decorators import dag, task
from airflow.providers.postgres.hooks.postgres import PostgresHook

@dag(dag_id="postgres_hook_tf", start_date=None, schedule=None,
     tags=["r7", "taskflow"])
```

```
def postgres_hook_tf():
    @task
    def export_data_to_csv(tbl_name):
        hook = PostgresHook(postgres_conn_id="example_db_postgres")
        hook.bulk_dump(tbl_name, f"/tmp/{tbl_name}.csv")

    t_export_data = export_data_to_csv("orders")
    t_export_data

postgres_hook_tf()
```

W naszym przykładzie skorzystamy z `PostgresHook` dostępnego w module `airflow.providers.postgres.hooks.postgres`. Następnie został on wykorzystany w naszej funkcji dla zadania opartego na Pythonie (`PythonOperator`). Dzięki niemu możemy bez problemu połączyć się z naszą bazą danych, używając połączenia `example_db_postgres` — nie musimy definiować i zabezpieczać tych danych w inny sposób. Zadaniem naszego operatora jest zapisanie danych z podanej tabeli za pomocą zmiennej `tbl_name` do pliku *.csv* w katalogu */tmp* znajdującym się na naszym workerze (tam, gdzie wykonujemy zadanie).

Kiedy zweryfikujemy logi, otrzymamy informacje o wykonaniu polecenia `COPY` i zapisu danych do odpowiedniego pliku.

`t_export_data` (wersja klasyczna)

```
---
[2025-04-27, 12:51:24 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 12:51:24 CEST] {postgres.py:188} INFO - Running copy expert: COPY
↪customers TO STDOUT, filename: /tmp/customers.csv
[2025-04-27, 12:51:24 CEST] {base.py:84} INFO - Retrieving connection 'example_db_
↪postgres'
[2025-04-27, 12:51:24 CEST] {python.py:240} INFO - Done. Returned value was: None
[2025-04-27, 12:51:24 CEST] {taskinstance.py:341} ▶ Post task execution logs
---
```

`t_export_data` (wersja TaskFlow)

```
---
[2025-04-27, 12:51:48 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 12:51:49 CEST] {postgres.py:188} INFO - Running copy expert: COPY orders
↪TO STDOUT, filename: /tmp/orders.csv
[2025-04-27, 12:51:49 CEST] {base.py:84} INFO - Retrieving connection 'example_db_
↪postgres'
[2025-04-27, 12:51:49 CEST] {python.py:240} INFO - Done. Returned value was: None
[2025-04-27, 12:51:49 CEST] {taskinstance.py:341} ▶ Post task execution logs
---
```

Możemy zweryfikować również katalog */tmp* na maszynie, na której działa nasz worker.

```
(venv) /airflow-sandbox
➜ ls -l /tmp/ | grep customers
-rw-r--r-- 1 root root     645 Apr 27 10:51 customers.csv
(venv) /airflow-sandbox
➜ ls -l /tmp/ | grep orders
-rw-r--r-- 1 root root     754 Apr 27 10:51 orders.csv
```

Kiedy nasze środowisko składa się z więcej niż jednego workera, warto utworzyć wspólną przestrzeń dyskową (np. zamontować dysk sieciowy) między workerami, aby można było bez problemu wymieniać informacje pomiędzy etapami w DAG-u.

TriggerDagRunOperator

Jest to operator pozwalający na wykonanie innego operatora. Innymi słowy: to zadanie uruchamia inny DAG. Będzie to przydatne w sytuacji, kiedy działanie jednego DAG-a jest uzależnione od wyniku innego. Najczęściej DAG-i, które są uruchamiane za pomocą tego operatora, nie mają schedulera (`scheduler=None`). Operator zawiera zestaw unikatowych argumentów:

- `trigger_dag_id` — ID DAG-a, który chcemy uruchomić.
- `trigger_run_id` — ID dla naszego wykonania. Najczęściej pomijane i wykorzystywane automatycznie generowane ID.
- `conf` — konfiguracja DAG-a, czyli DAG Config.
- `execution_date` — data logiczna, z którą uruchomi się nasz DAG. W tym wypadku korzystamy z modułu `datetime` lub `pendulum`. Dodatkowo musimy podać informacje o lokalizacji/strefie czasowej.
- `reset_dag_run` — jeżeli istnieje DAG o podanym ID uruchomienia, to otrzymamy błąd `DagRunAlreadyExists` — jeżeli dla tego argumentu ustawimy wartość `True`, to dany run zostanie oczyszczony, aby ponownie go uruchomić.
- `wait_for_completion` — domyślnie `False`. Jeżeli ustawimy `True`, to „główny" DAG będzie czekał, aż DAG uruchomiony za pomocą tego operatora zakończy swoje działanie (zmieni status z `running` na inny).
- `poke_interval` — domyślnie `60`. Definiuje w sekundach, jak często weryfikujemy stan uruchomionego DAG-a. Ta wartość jest wykorzystywana tylko wtedy, kiedy ustawiliśmy `True` dla argumentu `wait_for_completion`.
- `allowed_states` — lista określająca, jakie statusy wynikowe akceptujemy z naszego DAG-a, aby zadanie ustawiono na `success`. Domyślnie jest to jeden element: `success`.
- `failed_states` — lista określająca, jakie statusy wynikowe akceptujemy z naszego DAG-a, aby zadanie ustawiono na `failed`. Domyślnie lista jest pusta.

Zweryfikujmy teraz przykład. Wspominałem wcześniej, że najczęściej stosujemy zasadę 1 plik = 1 DAG. Tutaj w celach edukacyjnych oraz dla ułatwienia analizy kodu umieściłem wszystko w jednym miejscu.

`TriggerDagRunOperator` nie uruchomi innego DAG-a, jeżeli jest on zatrzymany.

Wersja klasyczna — *trigger_dag_example.py*

```
from airflow import DAG
import pytz
```

```
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator
from airflow.operators.trigger_dagrun import TriggerDagRunOperator
from datetime import datetime

def echo_logical_date(ld):
    print(ld)

with DAG(dag_id="child_dag_python", start_date=None, schedule=None,
         tags=["r7", "with"]):
    t_python_echo = PythonOperator(
        task_id="python_echo",
        python_callable=echo_logical_date,
        op_args=["{{ ds }}"]
    )

with DAG(dag_id="child_dag_bash", start_date=None, schedule=None,
         tags=["r7", "with"]):
    t_bash_echo = BashOperator(
        task_id="bash_echo",
        bash_command="echo {{ params.info }} && sleep 120"
    )

with DAG(dag_id="trigger_other_dag", start_date=None, schedule=None,
         tags=["r7", "with"]):
    t_trigger_python = TriggerDagRunOperator(
        task_id="trigger_python",
        trigger_dag_id="child_dag_python",
        execution_date=datetime(2010, 1, 1, tzinfo=pytz.timezone('America/New_York'))
    )

    t_trigger_bash = TriggerDagRunOperator(
        task_id="trigger_bash",
        trigger_dag_id="child_dag_bash",
        conf={"info": "Hello from Trigger!"},
        wait_for_completion=True,
        poke_interval=15
    )

t_trigger_python
t_trigger_bash
```

Wersja TaskFlow — *trigger_dag_example_tf.py*

```
from airflow.decorators import dag, task
import pytz
from datetime import datetime
from airflow.operators.trigger_dagrun import TriggerDagRunOperator

@task
def echo_logical_date(ld):
    print(ld)

@dag(dag_id="child_dag_python_tf", start_date=None, schedule=None, tags=["r7",
↪"taskflow"])
def child_dag_python_tf():
    echo_logical_date("{{ ds }}")

@dag(dag_id="child_dag_bash_tf", start_date=None, schedule=None, tags=["r7",
↪"taskflow"])
```

```
def child_dag_bash_tf():
    @task.bash
    def echo_bash():
        return f"echo {{ params.info }} && sleep 120"

@dag(dag_id="trigger_other_dag_tf", start_date=None, schedule=None, tags=["r7",
↪"taskflow"])
def trigger_other_dag_tf():
    t_trigger_python = TriggerDagRunOperator(
        task_id="trigger_python",
        trigger_dag_id="child_dag_python_tf",
        execution_date=datetime(2010, 1, 1, tzinfo=pytz.timezone('America/New_York'))
    )

    t_trigger_bash = TriggerDagRunOperator(
        task_id="trigger_bash",
        trigger_dag_id="child_dag_bash_tf",
        conf={"info": "Hello from Trigger!"},
        wait_for_completion=True,
        poke_interval=15
    )

    t_trigger_python
    t_trigger_bash

child_dag_python_tf()
child_dag_bash_tf()
trigger_other_dag_tf()
```

Jest tutaj sporo implementacji. W ramach tego przykładu mamy trzy DAG-i. DAG o ID `child_dag_python` ma za zadanie za pomocą operatora Pythona wyświetlić datę logiczną, która jest przekazywana przez główny DAG w operatorze `TriggerDagRunOperator` za pomocą argumentu `execution_date`. Ponieważ musimy podać lokalizację, za pomocą modułu `pytz` ustawiłem ją na `America/New York`. Drugi DAG ma ID `child_dag_bash` i w jego przypadku za pomocą `BashOperator` wyświetlamy informację pod zmienną `info` przekazaną za pomocą konfiguracji DAG-a, a następnie wykonujemy polecenie `sleep` przez 120 sekund. Nasz główny DAG o ID `trigger_other_dag` wykonuje wcześniej opisane DAG-i, gdzie dodatkowo czekamy, aż DAG zakończy swoje działanie, weryfikując jego status co 15 sekund.

Pozostaje nam zweryfikować logi.

`trigger_other_dag` — `python`

```
--
[2025-04-27, 13:59:03 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 13:59:03 CEST] {taskinstance.py:341} ▶ Post task execution logs
--
```

`trigger_other_dag` — `bash`

```
--
[2025-04-27, 13:59:03 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 13:59:03 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 13:59:18 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
```

```
[2025-04-27, 13:59:33 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 13:59:48 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 14:00:03 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 14:00:18 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 14:00:33 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 14:00:48 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 14:01:03 CEST] {trigger_dagrun.py:257} INFO - Waiting for child_dag_bash
↪on 2025-04-27 11:59:03.231794+00:00 to become allowed state ['success'] ...
[2025-04-27, 14:01:18 CEST] {trigger_dagrun.py:271} INFO - child_dag_bash finished with
↪allowed state success
[2025-04-27, 14:01:18 CEST] {taskinstance.py:341} ▶ Post task execution logs
--

child_dag_bash

--
[2025-04-27, 13:59:04 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 13:59:04 CEST] {subprocess.py:78} INFO - Tmp dir root location: /tmp
[2025-04-27, 13:59:04 CEST] {subprocess.py:88} INFO - Running command: ['/usr/bin/
↪bash', '-c', 'echo Hello from Trigger! && sleep 120']
[2025-04-27, 13:59:04 CEST] {subprocess.py:99} INFO - Output:
[2025-04-27, 13:59:04 CEST] {subprocess.py:106} INFO - Hello from Trigger!
[2025-04-27, 14:01:04 CEST] {subprocess.py:110} INFO - Command exited with return code 0
[2025-04-27, 14:01:04 CEST] {taskinstance.py:341} ▶ Post task execution logs
--

child_dag_bash

--
[2025-04-27, 13:59:04 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 13:59:04 CEST] {logging_mixin.py:190} INFO - 2010-01-01
[2025-04-27, 13:59:04 CEST] {python.py:240} INFO - Done. Returned value was: None
[2025-04-27, 13:59:04 CEST] {taskinstance.py:341} ▶ Post task execution logs
--
```

Możemy zauważyć, że w przypadku `TriggerDagRunOperator` nie zobaczymy logów. Tylko w przypadku oczekiwania na zakończenie działania dostaniemy powiadomienia o weryfikacji statusu uruchomionego DAG-a.

ShortCircuitOperator

Ten operator to kuzyn dwóch innych operatorów: `PythonOperator` i `BranchPythonOperator`. Za jego pomocą również możemy sterować przepływem pracy (*workflow*), ale w jego sytuacji decydujemy, czy zadania po nim mają się wykonać. Innymi słowy: jeżeli operator zwróci `False`, to wszystkie zadania od niego zależne zostaną ustawione na `skipped` zamiast `failed`. Jeżeli zaś chodzi o argumenty, to ten operator posiada ten sam zestaw co jego kuzyni.

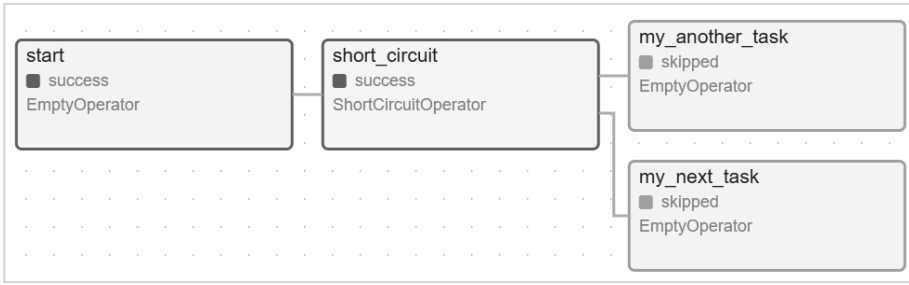
Wersja klasyczna — *short_circuit_example.py*

```
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from airflow.operators.python import ShortCircuitOperator

def continue_condition():
    from random import randint
    return randint(0,10) % 2 == 0

with DAG(dag_id='short_circuit_example', start_date=None, schedule=None, tags=["r7",
↪"with"]):
    t_start = EmptyOperator(task_id='start')

    t_short_circuit = ShortCircuitOperator(
        task_id='short_circuit',
        python_callable=continue_condition
    )
    t_task1 = EmptyOperator(task_id='my_next_task')
    t_task2 = EmptyOperator(task_id='my_another_task')

t_start >> t_short_circuit >> [t_task1, t_task2]
```

Wersja TaskFlow — *short_circuit_example_tf.py*

```
from airflow.decorators import dag, task

@dag(dag_id='short_circuit_example_tf', start_date=None, schedule=None, tags=["r7",
↪"taskflow"])
def short_circuit_example_tf():
    @task.short_circuit
    def continue_condition():
        from random import randint
        return randint(0,10) % 2 == 0

    @task
    def empty_task():
        print()

    t_start = empty_task()
    t_short_circuit = continue_condition()
    t_task1 = empty_task.override(task_id="my_next_task")()
    t_task2 = empty_task.override(task_id="my_another_task")()

    t_start >> t_short_circuit >> [t_task1, t_task2]

short_circuit_example_tf()
```

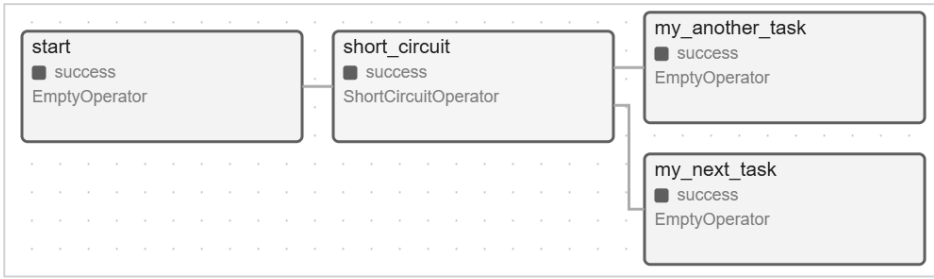
W ramach naszego przykładu zdefiniowałem trzy zadania oparte na `EmptyOperator`: `start`, `my_next_task` i `my_another_task` oraz jedno zadanie oparte na `ShortCircuitOperator`: `short_circuit`. Jeżeli zadanie `short_circuit` zwróci `False`, to kolejne zadania nie zostaną wykonane i przejdą w status `skipped`, co możemy zweryfikować w logach. Wynik działania obu scenariuszy możemy zobaczyć na rysunkach 6.1 i 6.2.



RYSUNEK 6.1. ShortCircuitOperator — wynik False

```
--
[2025-04-27, 14:44:13 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 14:44:13 CEST] {baseoperator.py:423} WARNING - ShortCircuitOperator.
↪execute cannot be called outside TaskInstance!
[2025-04-27, 14:44:13 CEST] {python.py:240} INFO - Done. Returned value was: False
[2025-04-27, 14:44:13 CEST] {python.py:309} INFO - Condition result is False
[2025-04-27, 14:44:13 CEST] {python.py:336} INFO - Skipping downstream tasks
[2025-04-27, 14:44:13 CEST] {python.py:344} INFO - Done.
[2025-04-27, 14:44:13 CEST] {taskinstance.py:341} ▶ Post task execution logs
--
```

W sytuacji, kiedy odpowiedź będzie `True`, wszystkie zależne od niego zadania zostaną uruchomione.



RYSUNEK 6.2. ShortCircuitOperator — wynik True

```
--
[2025-04-27, 14:44:13 CEST] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-04-27, 14:44:13 CEST] {baseoperator.py:423} WARNING - ShortCircuitOperator.
↪execute cannot be called outside TaskInstance!
[2025-04-27, 14:44:13 CEST] {python.py:240} INFO - Done. Returned value was: True
[2025-04-27, 14:44:13 CEST] {python.py:309} INFO - Condition result is True
[2025-04-27, 14:44:13 CEST] {python.py:312} INFO - Proceeding with downstream tasks...
[2025-04-27, 14:44:13 CEST] {taskinstance.py:341} ▶ Post task execution logs
--
```

Pewnie się zastanawiasz, jak zachowałyby się kolejne zadania po `my_next_task` i/lub `my_another_task`. Znasz odpowiedź: wszystko jest zależne od ich `trigger_rule`. Domyślnie również zostaną ustawione w status `skipped`.

Wszystkie dane pod pełną kontrolą

W czasach, gdy wiele naszych działań przeniosło się do przestrzeni cyfrowej, tworzymy i agregujemy ogromne ilości danych. Przechowujemy je na dyskach urządzeń, mobilnych nośnikach pamięci, w chmurach, a nawet w formie załączników poczty elektronicznej. Aby uzyskać z nich jak najwięcej informacji, musimy wykonywać odpowiednie procesy analityczno-agregujące, które następnie pozwolą nam na wyciągnięcie właściwych wniosków, a potem podjęcie odpowiednich decyzji. W tym miejscu często pojawia się problem: w jaki sposób pogodzić różne formaty danych, odpowiednio je ze sobą połączyć, wykonać mapowanie i konwertowanie?

Wtedy do gry wkraczają tak zwane orkiestratory zadań, a należy do nich między innymi Apache Airflow. Jest to jedno z najpopularniejszych narzędzi służących do tworzenia, organizowania i monitorowania przepływów pracy, a także uruchamiania łańcuchów zadań na podstawie danych pochodzących z rozmaitych źródeł i występujących w różnych formatach.

Apache Airflow — darmowej usłudze dostępnej dla każdego, kto zna język Python — poświęcona jest ta książka:

- **Znajdziesz w niej opis poszczególnych modułów narzędzia Apache Airflow**
- **Korzystając z zawartych w niej wskazówek, przeprowadzisz proces instalacji i przygotujesz środowisko pracy**
- **Przyjrzysz się poszczególnym elementom Apache Airflow**
- **Poznasz dobre praktyki związane z pracą w orkiestratorze danych**

Piotr Chudzik — absolwent Politechniki Łódzkiej, zawodowo specjalizuje się w technologiach związanych z pracą z danymi: od modelowania po przetwarzanie i wizualizację danych. Pracuje jako data engineer i trener wielu technologii z zakresu *data* i DevOps dla firm. Zawsze otwarty na nowe doświadczenia i wiedzę, którą w przyszłości mógłby się podzielić z innymi. Interesuje się grami komputerowymi, światem nowych technologii i fantastyką. Jest fanem serii o wiedźminie i uniwersum *Warhammera*.

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-1906-8

9 788328 919068

Cena: 59,00 zł