

4. laboratorijska vježba

- [Uvod u Apache Airflow](#)
 - [Kako Apache Airflow radi?](#)
- [Instalacija kroz Hello World primjer](#)
 - [Stvaranje DAG datoteke](#)
- [PostgreSQL](#)
 - [Instalacija](#)
 - [Kreiranje baze](#)
 - [Primjer](#)
 - [Zapisivanje podataka u bazu sa Sparkom](#)
- [Airflow + spark + postgresSQL i mySQL](#)
 - [Postavljanje okruženja](#)
- [Operatori](#)
 - [BashOperator](#)
 - [SparkSubmitOperator](#)
 - [PostgresOperator](#)
 - [MySqlOperator](#)
 - [Zadatak](#)
- [Bonus zadatak \(nije obavezno\)](#)
 - [TriggerDagRunOperator](#)
 - [Slanje maila](#)
 - [Kreiranje uloga i novih korisničkih računa](#)
 - [Zadatak 1](#)
 - [Zadatak 2](#)

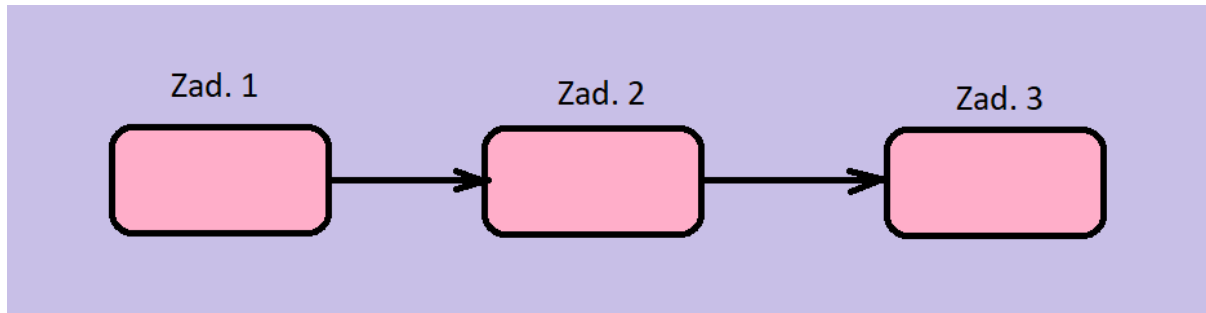
Uvod u Apache Airflow

Open source alati poput Apache Airflowa razvijeni su kako bi se olakšalo rukovanje golemom količinom podataka. Prije nego što objasnimo što je točno Apache Airflow, važno je razumjeti što su podatkovni cjevovodi (engl. *data pipelines*). Kako biste ih lakše dočarali, zamislite da nizu zadataka predajete skup podataka. Svaki zadatak izvršava nekakvu obradu tih podataka te se njima podatci kreću od izvora do određenog ciljnog sustava. Upravo taj niz zadataka obrade podataka između izvornog i ciljnog sustava su podatkovni cjevovodi, a njihova glavna uloga je automatizirati kretanje i transformaciju podataka. Apache Airflow je alat koji se koristi upravo za izgradnju, planiranje i praćenje podatkovnih cjevovoda.

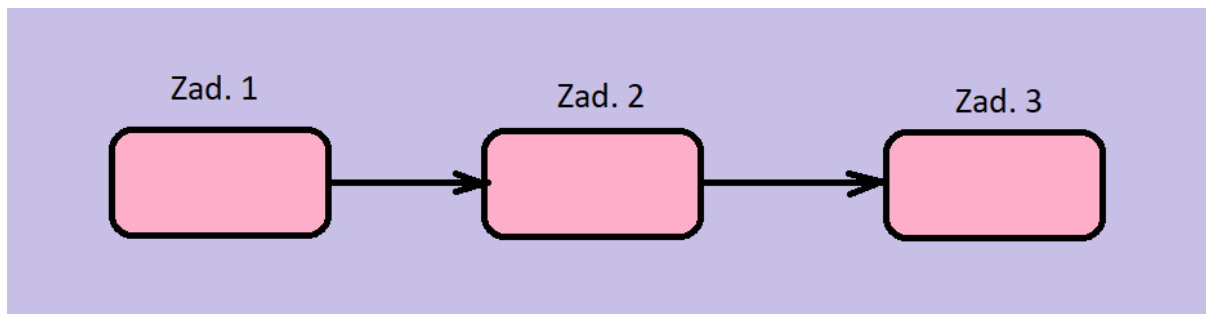
Kako Apache Airflow radi?

U Apache Airflowu se cjevovod prikazuje koristeći usmjerene acikličke grafove (engl. *Directed Acyclic Graph - DAG*). Svaki DAG je u matematičkom smislu graf i prema tome se

sastoji od čvorova koji su povezani. Sastoji se od barem jednog čvora, odnosno zadatka (engl. *task*), a zadatak se implementira koristeći neku vrstu operatora (više o operatorima u nadolazećim poglavljima). Veze između *taskova* predstavljaju međusobne ovisnosti između njih. Ovi grafovi su aciklički, što znači da nema ciklusa, odnosno petlji. Primjer ispravnog i neispravnog grafa pogledajte na sljedećim slikama:



Slika 1. Ispravan DAG



Slika 2. Prikaz petlje (neispravan DAG)

Instalacija kroz Hello World primjer

Za početak se pozicionirajte u novi direktorij. Unutar njega spremite [docker-compose](#) datoteku u vlastiti *docker-compose.yaml*. Uzmite trenutak da proučite YAML datoteku kako bi vam bilo jasnije što će se sve kasnije stvoriti (*airflow-scheduler*, *airflow-webserver*...). Unutar direktorija stvorite nove poddirektorije: *logs*, *dags* i *plugins*. Probajte pronaći koji se dio yaml datoteke koristi tim direktorijima.

Kako bi docker-compose ispravno radio dodajte u početni direktorij datoteku *.env* sa sljedećim vrijednostima (na Windows OS radi i bez toga):

```
gear .env
1  AIRFLOW_UID=1000
2  AIRFLOW_GID=0
```

Stvaranje DAG datoteke

U našem prvom primjeru želimo stvoriti jedan jednostavan zadatak (tip operatora će biti `PythonOperator`) koji ispisuje "Hello World". Također, možemo specificirati vrijeme izvršavanja zadatka.

Unutar *dags* direktorija kreirajte *hello_world_dag.py* datoteku. Na samom početku trebate unijeti odgovarajuće module. Kako bi ih koristili možda ćete ih trebati instalirati na vlastito računalo, npr. koristeći *pip install* naredbu (konkretno za Airflow bi glasilo *pip install apache-airflow*).

```
# Uvođenje potrebnih biblioteka
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

# Definicija DAG-a
with DAG(
    dag_id="hello_world_id",
    start_date=datetime(2023, 3, 14),
    schedule="@hourly",
    catchup=False,
) as dag:

    # Definicija funkcije koju će zadatak izvršavati
    def hello_world():
        print('Hello World')

    # Definicija zadatka
    task1 = PythonOperator(
        task_id="hello_world",
        python_callable=hello_world
    )
```

Pri definiciji DAG-a, dali smo mu identifikacijsku oznaku. Također, odredili smo datum kada će se zadatak početi izvršavati i koliko često. Primijetite da je `start_time` u prošlosti. Airflow iz toga zaključuje da DAG “kasni” te će ga naknadno pokrenuti onoliko puta koliko bi ga puta pokrenuo da je zaista počeo kada smo specificirali. Na primjer, ako je `start_time` postavljen na datum prije 5 dana, a `schedule` je postavljen na `@daily`, DAG će se pokrenuti 5 puta za redom. Ako ne želite da se DAG-ovi “nadoknađuju”, postavite parametar `catchup` na `False`.

Nakon DAG-a, definirajte zadatak. Zadatak se definira koristeći jedan od mnogih operatora. U ovom slučaju, to će biti `PythonOperator` koji pokreće neku python funkciju, u ovom slučaju `hello_world`. Operatori će biti detaljnije opisani u nadolazećim poglavljima.

Kada ste napisali prvi DAG, pokrenite *docker-compose*:

```
docker-compose up airflow-init
```

Trebali biste pronaći stvorene kontejnere unutar Docker Desktop aplikacije. Zatim, pokrenite kontejnere naredbom:

```
docker-compose up
```

Pričekajte da se naredba izvrši i provjerite jesu li se kontejneri pokrenuli. Kada ste sigurni da jesu idite na <http://localhost:8080> (za više informacija o korisničkom sučelju Airflowa, slijedite poveznicu: <https://docs.astronomer.io/learn/airflow-ui>) gdje se možete ulogirati s Airflow korisničkim imenom te Airflow lozinkom.

U listi DAG-ova pronađite svoj *hello_world_dag* te ga pokrenite. Pogledajte je li se zadatak ispravno izvršio (tamno zelena boja = success), što se ispisalo u logove i sl.

Ako sve ispravno radi, proučite i pokrenite već stvorene DAG-ove (na primjer: tutorial, tutorial_dag, example_bash_operator i example_python_operator) i pogledajte njihove implementacije na sljedećoj poveznici:

https://github.com/apache/airflow/tree/main/airflow/example_dags.

PostgreSQL

PostgreSQL je moćan objektno-relacijski sustav baze podataka otvorenog koda (engl. *open source*) koji koristi i proširuje SQL u kombinaciji s mnogim značajkama koje sigurno pohranjuju i skaliraju najsloženija radna opterećenja podataka (engl. *data workloads*). Za više informacija o PostgreSQL-u, posjetite <https://www.postgresql.org/about/>.

Instalacija

pgAdmin je alat koji omogućava interakciju i grafički pregled baze. Možete ga preuzeti na poveznici: <https://www.pgadmin.org/download/>. Kada ga instalirate i pokrenete prvi put, postavite glavnu zaporku. Svaki sljedeći put kada pokrenete pgAdmin, pitat će Vas za tu zaporku pa je nemojte zaboraviti.

Kreiranje baze

Bazu ćemo podići koristeći docker. U terminalu pokrenite naredbu:

```
docker run -p 5433:5432 -e POSTGRES_USER=<user> -e  
POSTGRES_PASSWORD=<password> -e POSTGRES_DB=<db_name> postgres
```

Postavite varijable okoline na željene vrijednosti. Otvorite pgAdmin i kreirajte poveznicu s bazom tako da desnim klikom odaberete neku od grupa servera (možete kreirati novu grupu, ali postojeća grupa je isto u redu) i pritisnete *Register > Server...* što Vas odvodi do prozora za kreiranje servera. Upišite ime servera u kartici *General*, a u kartici *Connection* u polje *Host name/address* upišite "localhost" te u polje *Password* proizvoljnu zaporku servera i završite odabirom *Save*.



Uočite da smo preusmjerili port docker kontejnera s 5432 na 5433 jer se instalacijom pgAdmina port 5432 zauzme pa ćete morati koristiti drugi port.

Primjer

Unutar kreirane baze odaberite jednu od postojećih baza (postgres ili <db_name>) i desnim klikom otvorite *Query Tool*. U novootvorenom prozoru ćemo poslati upite koji kreiraju tablicu, pune je primjerima i na kraju ispisuju unesene primjere. Da kreirate tablicu, upišite sljedeći kod:

```
CREATE TABLE CryptoTable (  
    name VARCHAR(255) NOT NULL,  
    val DECIMAL(11, 5) NOT NULL  
);
```


Kreirat će se tablica “CryptoTable” čiji su stupci “name” (tekst od maksimalno 255 znakova) i “val” (decimalni broj od 11 znakova, od čega je 5 iza decimalne točke) te ne smiju biti NULL.

Kreirana tablica je prazna pa ćemo dodati nekoliko primjera:

```
Insert into CryptoTable(name, val)  
Values  
    ('BTC', 20000),  
    ('ETH', 1700),  
    ('DOGE', 0.0005);
```

Na kraju, pogledajmo što se nalazi u bazi jednostavnim SELECT upitom:

Rezultat upita je sljedeći:

	name character varying (255) 	val numeric (11,5) 
1	BTC	20000.00000
2	ETH	1700.00000
3	DOGE	0.00050

Baš kao što smo očekivali, podaci su se uspješno pohranili u tablicu.

Zapisivanje podataka u bazu sa Sparkom

Podatke obrađene u 3. laboratorijskoj vježbi smo spremili u datoteku, a sada ćemo ih, umjesto u datoteku, preusmjeriti u bazu.

Da bi pySpark mogao pisati u bazu, mora imati postgresSQL JDBC (Java Database Connectivity) pokretač (engl. *driver*). Preuzeti ga možete sa sljedeće poveznice: <https://jdbc.postgresql.org/download/> i dodajte ga u klasnu putanju sparka (SPARK_HOME/jars).

Sada ćete zapisati podatke u bazu. Format podataka prije zapisivanja u bazu je dataframe. Pozovite sljedeću naredbu i podatci će se zapisati u bazu:

```
df.write.format('jdbc')\
    .option('url', 'jdbc:postgresql://<adresa_baze>:5433/<DB_name>')\
    .option('driver', 'org.postgresql.Driver')\
    .option('dbtable', '<table_name>')\
    .option('user', '<DB_user>')\
    .option('password', '<DB_password>')\
    .mode('overwrite')\
    .save()
```

Ako provjerite tablicu u bazi, trebala bi biti ažurirana s vrijednostima iz *dataframe-a*.

Čitanje iz baze se obavlja na sličan način:

```
df = spark.read.format('jdbc')\
    .option('url', 'jdbc:postgresql://<adresa_baze>:5433/<DB_name>')\
    .option("driver", "org.postgresql.Driver")\
    .option('dbtable', '<table_name>')\
    .option('user', '<DB_user>')\
    .option('password', '<DB_password>')\
    .load()
```

Airflow + spark + postgresSQL i mySQL

Sada kada ste se upoznali s ovim tehnologijama, povezat ćemo ih u jednu aplikaciju. Ona će uz pomoć Airflowa zakazivati i orkestrirati koje će se operacije izvoditi. Operacije će se izvršavati na Spark klasteru, a njihov rezultat zapisivati u bazu podataka.

Postavljanje okruženja

Kao i u prijašnjim laboratorijskim vježbama, preporučeni programski jezik je Python, te će u njemu biti prikazan sljedeći primjer.

Kreirajte projekt u IDE-u Vašeg odabira i u njega spremite `docker-compose.yaml`, `Dockerfile` i `requirements.txt` datoteke koje ćete preuzeti na stranici predmeta (`docker-compose.yaml` je malo izmijenjen u odnosu na onaj koji ste koristili u gornjem primjeru, bitno je da preuzmete taj sa stranice predmeta).

`requirements.txt` je datoteka u kojoj su popisane sve ovisnosti Pythona o vanjskim bibliotekama. U ovom slučaju su to `pySpark`, `findspark` i operatori koji nisu u osnovnom Airflow paketu (`postgresSQL` i `mySQL` operatori itd.).

`Dockerfile` će izgraditi `docker` sliku u kojoj će se postaviti okruženje u `docker` kontejneru `Airflow-a`. Pri izgradnji slike se preuzimaju pokretači za bazu i spremaju u `SPARK_HOME/jars`, postavljaju se varijable okruženja i kopiraju datoteke u kojima su definirani Airflow *taskovi*.

`docker-compose.yaml` pokreće sve komponente `Airflow-a`.

U ovom primjeru ćete, osim u PostgreSQL bazu pisati i u MySQL bazu. Nju ćete isto kao i PostgreSQL pokrenuti u docker kontejneru koristeći upute s poveznice: https://hub.docker.com/_/mysql. Driver za MySQL bazu će se također automatski preuzeti u Dockerfileu.

Sada izgradite docker sliku i tagirajte je kao `extending_airflow`. Kada se izgradi, pokrenite Airflow s naredbom `docker compose up`. Na kraju dodajte PostgreSQL i MySQL docker kontejnere u istu mrežu kao i ostale komponente Airflow-a kako bi kasnije mogli komunicirati.

Operatori

U sljedećim poglavljima obrađuje se nekoliko tipova operatora koji će kasnije biti potrebni za zadatak.

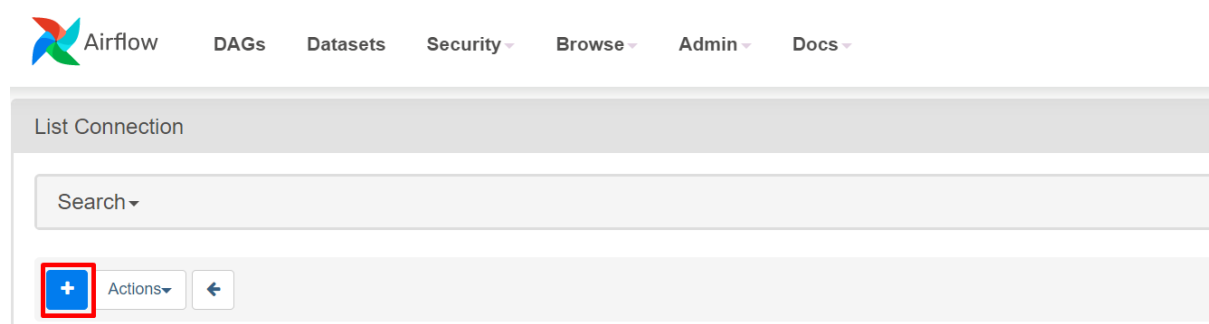
BashOperator

Ako želite da Airflow pokreće bash skripte, koristit ćete `BashOperator`. Moguće je specificirati putanju do skripte, ali ako se radi o nekoj jednostavnijoj naredbi, moguće ju je definirati preko parametra `bash_command`. Primjer jedne takve naredbe je sljedeći:

```
bash_operator = BashOperator(
    task_id="example",
    bash_command="echo 'Hello World'",
)
```

SparkSubmitOperator

Airflow će pokretati spark poslove uz pomoć `SparkSubmitOperator` operatora. Potrebno mu je predati putanju do datoteke u kojoj je specificiran spark posao i `conn_id`. `Conn_id` predstavlja id veze na spark cluster. Ovu vezu kreirate tako da otvorite Airflow nadzornu ploču na adresi `localhost:8080` te, kada se ulogirate, odaberete *Admin > Connections*. Kreirajte novu vezu pritiskom na plavi gumb.



Postavite tip veze (*Connection Type*) na Spark, a ostale vrijednosti onako kako je na sljedećoj slici:

Edit Connection	
Connection Id *	spark_local
Connection Type *	Spark <small>Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package.</small>
Description	
Host	local[*]
Port	
Extra	{"queue": "root.default", "master": "local[*]", "spark-binary": "spark-submit", "namespace": "default"}

Više o Spark vezi pronađite na sljedećoj poveznici: <https://airflow.apache.org/docs/apache-airflow-providers-apache-spark/stable/connections/spark.html>.

Extra polje: {"queue": "root.default", "master": "local[*]", "spark-binary": "spark-submit", "namespace": "default"}.

Kada ste napravili vezu, napišite spark zadatak. Primjer jedne definicije `SparkSubmitOperator` zadatka (engl. *task*) je sljedeća:

```
spark_submit_operator = SparkSubmitOperator(
    application='/path/to/file/script.py',
    conn_id='spark_local',
    task_id='example',
    dag=dag_spark
)
```

`application` parametar definira putanju do datoteke koja definira spark posao. Kao i u prethodnoj laboratorijskoj vježbi, za definiciju spark posla koristit ćete *PySpark*.



`application` putanja NIJE putanja do datoteke na vašem računalu, već do one koja se nalazi unutar kontejnera (/opt/airflow/raw_data/application.py).

Postgres Operator

Koristeći `PostgresOperator` inicijalizirat ćete postgres bazu podataka. Prije nego napišete Airflow zadatak, stvorite vezu na postgres bazu. Postupak je sličan kao kod kreiranja veze na spark cluster, ali s malenim promjenama.

Connection successfully tested

Edit Connection

Connection Id *	postgres_local
Connection Type *	Postgres <small>Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package.</small>
Description	
Host	host.docker.internal
Schema	test
Login	test
Password	****
Port	5433
Extra	

Save Test ↩

Unesite korisničko ime i zaporku baze (one varijable okruženja koje ste postavili prilikom pokretanja docker slike baze) te IP adresu i port na kojem baza radi. Kao što vidite na slici, postavite parametar *Host* (IP adresa) na `host.docker.internal`. Ova adresa je adresa localhost, ali unutar dockera, a ne računala domaćina (*host*).



Postavljanje IP adrese na ovaj način je prikladan samo za testiranje (a time i za ovu laboratorijsku vježbu) te ne radi u produkcijskom okruženju.

Kada napravite vezu, napišite zadatak. Primjer jednog *taska* je sljedeći:

```
init_postgres = PostgresOperator(  
    task_id="init_postgres",  
    sql="""  
        create table IF NOT EXISTS postgresdb(  
            name VARCHAR(50) NOT NULL,  
            time BIGINT,  
            value FLOAT  
        );  
        """,  
    postgres_conn_id='postgres_local',  
    dag=dag_spark  
)
```

Ovim zadatkom inicijalizirat će se tablica “postgresdb” čiji je prvi stupac niz od maksimalno 50 znakova, a drugi je cijeli broj. Oba stupca moraju biti postavljena.

MySqlOperator

Osim postgres baze podataka, koristit ćemo i mySQL bazu. Razlika između ove dvije vrste baze podataka je u tome da je mySQL relacijska baza podataka, a Postgres je objektno-relacijska baza podataka. Također, Postgres nudi širi spektar vrsta podataka i omogućava nasljeđivanje svojstava objekata, zbog čega je i složeniji.

Postavljanje okruženja za mySQL bazu napravite kao i za postgres bazu, pokrenite mySQL bazu preko dockera, dodajte kontejner u mrežu, kreirajte vezu i inicijalizirajte bazu uz pomoć Airflow operatora.

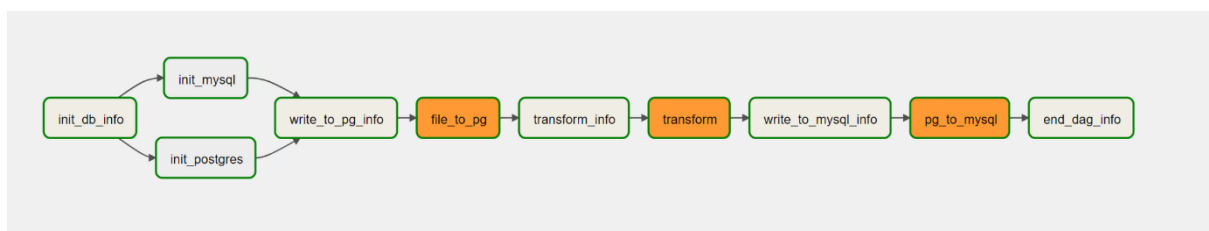
Zadatak

Napravite DAG koji se sastoji od sljedećih *taskova*:

1. Inicijaliziraj postgres i mySQL bazu podataka
 - a. ova dva zadatka neka se izvode u paraleli, tako da sljedeći zadatak ne može početi ako oba nisu gotova
 - b. prije nego što kreirate baze pročitajte zadatak do kraja kako biste znali koje stupce stvoriti u kojoj bazi
2. Prepiši podatke iz CSV datoteke u postgres bazu podataka
3. Obradi podatke i zapiši natrag u postgres bazu u novu tablicu
 - a. Pretvorite stupac koji sadrži unix vrijeme u milisekundama u timestamp oblika YYYY-MM-DD HH-mm-ss
 - b. Dodajte stupac koji će predstavljati razliku sadašnje i prethodne cijene za tu kriptovalutu u tablici
 - c. Izbacite nepotrebne stupce
4. Prepiši podatke iz postgres baze u mySQL bazu - koristite `com.mysql.jdbc.Driver` za mySQL driver (za testiranje spark poslova bez Airflow-a na računalu domaćinu, preuzmite *driver* sa sljedeće poveznice: <https://dev.mysql.com/downloads/connector/j/>; za testiranje spark posla kao Airflow *taska* se taj *driver* preuzme u `Dockerfileu`).

Sve *taskove* pišite u zasebne `.py` datoteke, a te datoteke spremite u direktorij `raw_data` u korijenu projekta. Taj direktorij je montiran (engl. *mounted*) od docker kontejnera i sve promjene koje se dogode na računalu domaćinu će se reflektirati na kontejner. Kao što se vidi u `docker-compose.yaml` datoteci, putanja do te datoteke je `/opt/airflow/raw_data`.

Za 2., 3. i 4. zadatak koristit ćete `SparkSubmitOperator`, a između tih koraka ćete `BashOperatorom` logirati faze (npr. “Učitao podatke u postgres bazu”, “Prepisao podatke iz postgres baze u mySQL” itd.) Konačni DAG bi trebao izgledati slično sljedećem:



Različiti tipovi zadataka su različito obojeni prema vrsti operatora preko kojega su kreirani.

Bonus zadatak (nije obavezno)

U sljedećem poglavlju ćete prethodno napravljeni DAG rastaviti na više njih, iz jednog DAG-a pokrenuti drugi i poslati obavijest o stanju DAG-a na mail.

TriggerDagRunOperator

Koristeći `TriggerDagRunOperator` možemo iz jednog DAG-a pokrenuti drugi DAG s identifikacijskim kodom koji specificiramo kao parametar klase. Primjer jednog takvog operatora je u nastavku:

```
trigger_dag = TriggerDagRunOperator(  
    task_id="trigger_task",  
    trigger_dag_id="target_dag",  
    wait_for_completion=True,  
    poke_interval=5,  
    dag=dag  
)
```

`trigger_dag` pokrene DAG s *id*-jem `target_dag` i nastavlja izvršavati svoje zadatke neovisno o tome je li pokrenuti DAG završio ili ne. U slučajevima gdje na ovaj način pokrenemo DAG koji preuzima podatke, a ti isti podaci se čitaju kasnije u originalnom DAG-u, može doći do pogreške ako se podaci pokušaju pročitati prije nego što su preuzeti. Zato postoji parametar `wait_for_completion` koji zaustavlja originalni DAG sve dok pozvani DAG ne završi s radom. Također, parametar `poke_interval` označava koliko se često provjerava je li pozvani DAG završio (*default* vrijednost je 60 (sekundi)).

Uvođenjem `TriggerDagRunOperatora` dobivamo mogućnost pokretanja DAG-a iz drugog DAG-a, a posljedica toga je da možemo rastaviti DAG na više manjih prema logici posla koji rade (npr. inicijalizacija baze u jedan DAG, zapisivanje u drugi itd.). Neki od razloga zašto bi se pojedini dijelovi DAG-a prebacili u zaseban DAG su:

- radi bolje organizacije (ponekad je smisleno razdvojiti DAG-ove u logičke komponente)
- recikliranje DAG-ova (neki DAG-ovi se mogu koristiti na više različitih mjesta; ako je DAG prespecifičan, ne može ga se reciklirati)
- pojednostavljenje ovisnosti (engl. *dependency*) (veliki DAG-ovi često ovise o mnogim bibliotekama; manji DAG-ovi koriste manje funkcija, a samim time ovise o manje vanjskih biblioteka)



Grupiranje zadataka moguće je izvesti i na druge načine, mogu se koristiti podDAG-ovi (SubDAG; koji su zastarjeli (*deprecated*)) ili još bolje grupe zadataka (`TaskGroup`) (koji su zamijenili podDAG-ove), ali u ovoj vježbi se neće obrađivati

Slanje maila

Ponekad je bitno dobiti obavijest da je DAG ili *task* (ne)uspješno izveden, a ponekad želite poslati obavijest koja informira o fazi izvođenja DAG-a ili obrađene podatke koje je DAG do sada obradio. Da biste dobili informaciju o statusu DAG-a na mail, postavite parametar DAG-a `on_failure_callback` na *callback* funkciju koja će se pozvati kada se dogodi greška u izvođenju DAG-a. Ako želite dobiti obavijest o neuspješnom izvođenju *taska*, postavite parametar *taska* `email_on_failure` na `True` te postavite mail na koji želite dobiti obavijest parametrom `email`. Ako pak želite poslati mail u nekoj fazi izvođenja DAG-a, koristite `EmailOperator`.

U ovom primjeru bit će pokazano postavljanje okruženja za slanje obavijesti na Gmail, ali u teoriji možete koristiti bilo koji (ali je postavljanje kompleksnije). Prije postavljanja samog Airflow okruženja, prvo morate postaviti svoj Gmail. U postavkama Vašeg računa, u kartici *Sigurnost*, odobrite potvrdu u dva koraka. Kada ste to napravili, odaberite *Potvrda u dva koraka*. Potvrdite svoj identitet upisivanjem zaporke te na sljedećem ekranu na dnu odaberite Zaporke aplikacije. Pod *Odabir uređaja* odaberite *Drugo (prilagođen naziv)* i upišite proizvoljno ime ključa, tako da znate da je to ključ za Airflow. Kopirajte ključ i zalijepite ga u datoteku.

Nakon postavljanja računa, postavite Airflow da može slati mailove. U datoteci `airflow.cfg`, koja se zadano nalazi u `/opt/airflow/`, u dijelu `[smtp]` postavite vrijednosti na sljedeće:

```
[smtp]
```

```
smtp_host = smtp.gmail.com
smtp_starttls = True
smtp_ssl = False
smtp_user = <mail@mail.com>
smtp_password = <kopirani_ključ>
smtp_port = 25
smtp_timeout = 30
smtp_retry_limit = 5
```



Datoteka `airflow.cfg` se učitava pri pokretanju Airflowa te njeno mijenjanje dok su Airflow kontejneri pokrenuti nema učinka. Jedan od načina da postavite ove vrijednosti je da kopirate datoteku na vlastito računalo i u `docker-compose.yaml` konfiguraciji nadodate vrijednost u `x-airflow-common.volumes`:
`lokalna/putanja/do/airflow.cfg:/opt/airflow/airflow.cfg`

Sada morate nadodati novu konekciju kao što ste to radili za operatore baza i spark. ID postavite na `smtp_default`, tip veze neka bude *Email*, domaćina na `smtp.gmail.com` i na kraju korisničko ime i lozinku na Vaš email i ključ koji ste maloprije kopirali.

Kada ste završili postavljanje, odaberite jedan od DAG-ova i na njemu isprobajte jedan ili pak sve načine slanja mailova koji su gore opisani.

Kreiranje uloga i novih korisničkih računa

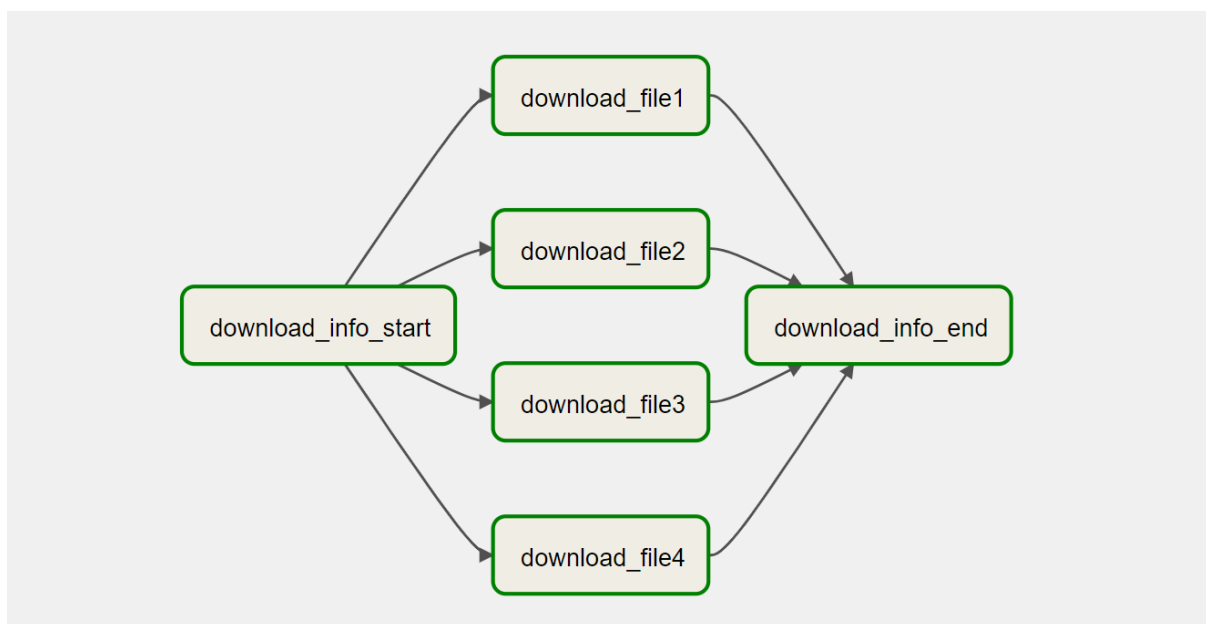
Da biste koristili Airflow, morate biti ulogirani s korisničkim računom. Airflow dolazi s jednim korisničkim računom koji već koristite, a to je račun s korisničkim imenom `airflow`.

Da biste pogledali detalje o tom računu, ali i o svim ostalima (kojih za sad nema), otvorite *Security > List Users*. Ovdje možete vidjeti koja je uloga pridodana korisniku. Da biste vidjeli koje uloge postoje, otvorite *Security > List Roles*. Za postojeće uloge možete vidjeti koje sve aktivnosti mogu raditi, a možete kreirati i potpuno nove i prilagođene uloge. Kreirajte novi račun s *Viewer* ulogom, ulogirajte se (prebacite se u *incognito* način rada na pretraživaču) i proučite razlike (pogotovo u *DAGs* dijelu).

Zadatak 1

U drugoj laboratorijskoj vježbi ste dohvaćali podatke s *Coincap* API-ja i obrađivali ih u trećoj laboratorijskoj vježbi. Za dohvaćanje podataka ste koristili Kafka *producer* i *consumer* koje ste sami napisali. U ovom zadatku ćete ponovno dohvaćati podatke u **zasebnom DAG-u** (koji ćete pokretati iz glavnog DAG-a koristeći `TriggerDagRunOperator`), ali na način na koji to sami izaberete. Postoji mnogo opcija, ali 3 najjednostavnije su:

1. Koristeći `BashOperator` dohvatiti podatke s API-ja (metoda `wget`, bez Kafke). S obzirom na to da dohvaćate podatke od zadnjih 100 dana, trebat će Vam barem 4 poziva (jer u jednom pozivu možete imati max. 30 dana), a njih implementirajte kao pojedinačne zadatke. Tada će taj DAG izgledati slično sljedećem:



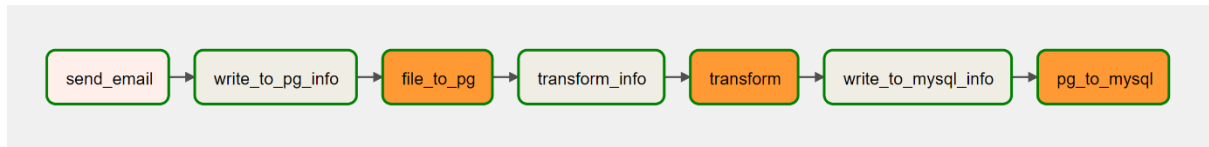
DAG *trigger_download*

Datoteke *download_fileN* možete izvršavati u paraleli.

2. Koristeći `SparkSubmitOperator` i Kafku koje ste koristili u prethodnim vježbama. S *topica* ćete čitati poruke na isti način kao i u prethodnoj laboratorijskoj vježbi.
3. Koristeći operatore iz paketa `airflow-provider-kafka` (npr. `ProduceToTopicOperator` i `ConsumeFromTopicOperator`). Ako se odlučite za ovaj način, morat ćete ažurirati `requirements.txt` s `airflow-provider-kafka`

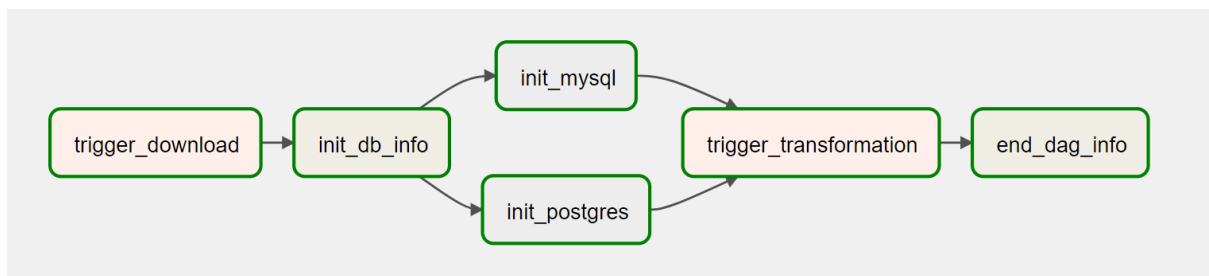
paketom, ponovno izgraditi docker sliku te izbrisati i ponovno pokrenuti Airflow kontejnere.

Kada napišete DAG za dohvaćanje podataka, napišite još jedan DAG u kojem ćete obrađivati podatke i spremati ih u baze (odnosno, DAG u kojem ćete koristiti sve `SparkSubmitOperator`). Obrada podataka neka bude ista kao i u prethodnoj vježbi, pronalazak dnevnih maksimuma u zadnjih ~100 dana, sortiranje i rangiranje i zaokruživanje na jednu decimalu. Imajte na umu da su Vam podaci raspoređeni u 4 json datoteke te ih prije obrade trebate sjediniti u jednu csv datoteku. DAG za obradu podataka može izgledati slično grafu u nastavku:



DAG *trigger_transformation*; u prvom koraku se šalje mail na adresu da je obrada podataka započela

Na kraju kada ste razdvojili DAG-ove, originalni DAG koji pokrećete će biti sličan grafu u nastavku:



Kada kreirate sve DAG-ove, kreirajte i novi korisnički račun koji će imati mogućnosti slične kao i uloga `Viewer`, ali proširene tako da može pokretati samo onaj DAG koji pokreće druga dva. Kreirajte novu ulogu s potrebnim pravima i novom računu predajte `Viewer` i novonastalu ulogu. *DAGs* prozor bi trebao izgledati ovako:

DAGs

[illegible]

Kao što se vidi iz slike, DAG-ovi koji se pokreću iz *main_DAG*-a ne mogu se pokrenuti od korisnika. Tako ste spriječili neželjeno pokretanje DAG-ova koji ovise o drugim DAG-ovima te je jedini način da se oni pokrenu u *main DAG*-u ili s Admin korisnikom.

Zadatak 2

Do sada se u ovoj laboratorijskoj vježbi niste jako dobro upoznali sa zakazivanjem (engl. *scheduling*). Airflow omogućava da se odredi vrijeme pokretanja pojedinih DAG-ova i tako automatizira neke poslove koji se moraju raditi periodično (npr. svako jutro u 8 sati, svakog 1.

u mjesecu itd.). Periodičnost izvođenja DAG-a određuje se pri definiciji DAG-a kroz parametar `schedule_interval`. Vrijednost ovog parametra može biti jedna od zadanih (npr. `@hourly`, `@daily`, `@weekly`...), ali može se i preciznije odrediti koristeći *cron* izraze. Više informacija o *cron* izrazima potražite na sljedećoj poveznici: <https://crontab.guru/>.

Postavite i parametar `start_date` kojim određujete datum kada će se DAG početi izvršavati. Ako je taj dan u prošlosti, Airflow će “nadoknaditi” sve propuštene termine izvršavanja, što znači da će se, ovisno koliko daleko u prošlost stavite `start_date`, jako puno DAG-ova početi izvoditi u paraleli i potencijalno srušiti Vaše kontejnere. Da biste spriječili ovakvo ponašanje, postavite zastavicu `catchup` na `False`.

U sljedećem zadatku ćete napraviti nekoliko DAG-ova:

1. Preuzmi i zapiši u bazu
 - a. Svake minute pomoću *CoinCap* API-ja dohvatite podatke o kriptovaluti po izboru i zapišite te podatke u datoteku
 - b. Učitajte preuzetu json datoteku te izbacite stupce *id*, *supply*, *maxSupply* i *explorer*
 - c. Promijenite stupac “timestamp” u “date” i “time”, gdje je “date” datum, a “time” vrijeme u danu te izbacite stupac “timestamp”
 - d. Zaokružite sve decimalne vrijednosti na tri decimale
 - e. Zapisane podatke prepisite u Postgres bazu
2. Pročitaj iz baze i ponovno zapiši
 - a. Svakih sat vremena pročitajte sve zapise nastale u prethodnom satu te redak s najvećom vrijednosti u stupcu “priceUsd” prepisite u novu tablicu
 - b. Dodajte stupac *changeMax1Hr* koji predstavlja za koliko se promijenila maksimalna vrijednost valute u odnosu na maksimalnu vrijednost prethodnih sat vremena (Primjer: u 10. satu je max. vrijednost bila 100, u 11. satu je max. vrijednost 90 => *changeMax1Hr* će biti -10)