DeliveryTech

# Introductory workshop to data pipelines orchestration with Apache Airflow

Enrica Pasqua

# Who Am I?



| Italy | Computer Science | Pythonist | Berlin | Mkt Tech |
| --- | --- | --- | --- | --- |

# What we’ll bring home today

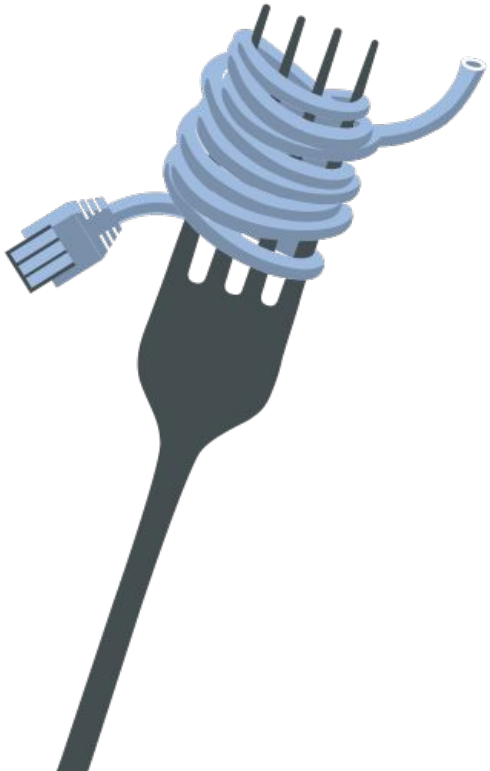


This workshop walks you through some of the fundamental Airflow concepts:

1st part → Introduction to Apache Airflow

2nd part → Play with Dockerized instance of Airflow

# Preparation



# Prerequisite - Install Docker



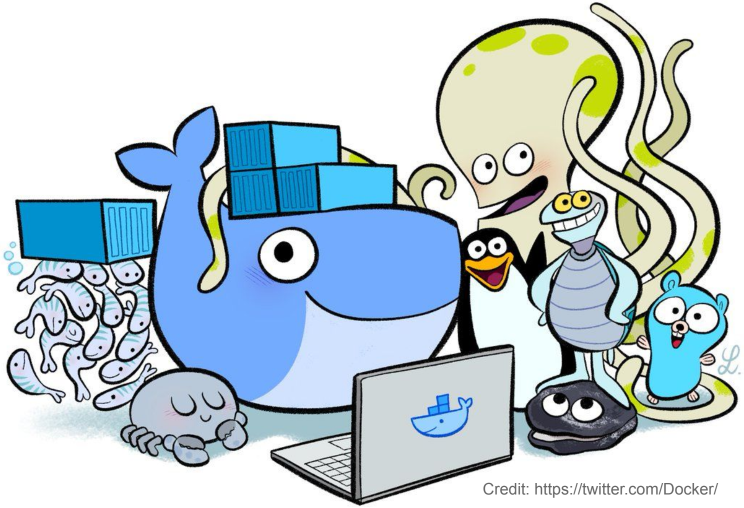
Install Docker and Docker Compose

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.

Containers make it possible to isolate applications into small, lightweight execution environments that share the operating system kernel.

→ No connection or questions?

Ask us and check the content in the **usb stick** :)



Credit: https://twitter.com/Docker/

# Run Airflow



Have a **dockerized Airflow instance** running on your machine. 2 ways:

**<u>mode-1:</u>** Clone the **GitHub** repository into an empty folder:

a) Download the repository:
   https → `git clone https://github.com/enricapq/docker-airflow-workshop.git`
   ssh → `git clone git@github.com:enricapq/docker-airflow-workshop.git`

b) Go inside the sub directory `docker-airflow` (that contains the dockerfile) and then execute:
   `docker build --rm -t enrica/docker-airflow .`

c) Execute (from the project root `docker-airflow-workshop`)
   `docker-compose -f docker-compose.yml up -d`

**<u>mode-2:</u>** Copy all the files from the folder `docker-airflow-workshop` (in the **usb stick**) into an empty folder on your disk. Go to this folder.

a) Load the image from the given **.tar** file executing:
   `docker image load -i docker-airflow.tar`

b) Execute: `docker-compose -f docker-compose.yml up -d`

Open the **browser** and go to: `http://localhost:8080` or `http://192.168.99.100:8080`

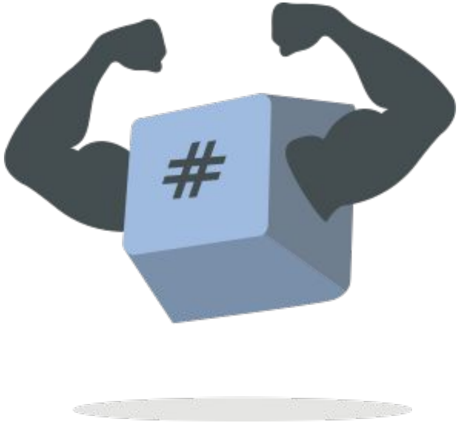
# Docker Cheat Sheet

**Delivery**Tech

| | |
|---|---|
| `docker build --rm -t enrica/docker-airflow .` | Create image from Dockerfile |
| **`docker build --rm -t enrica/docker-airflow .`** | Create image from Dockerfile (from the sub folder *docker-airflow* ) |
| `docker build -p 8080:8080 --rm -t enrica/docker-airflow .` | Build image and pass a port |
| `docker images` | See all images |
| **`docker-compose -f docker-compose.yml up -d`** | Run containers (launch it from the folder project *docker-airflow-workshop*) |
| `docker images` | See all containers |
| `docker rmi enrica/docker-airflow --force` | Remove single image |
| **`docker-compose -f docker-compose.yml down`** | Stop containers |
| `docker image save enrica/docker-airflow > docker-airflow.tar` | Save image locally |

DeliveryTech

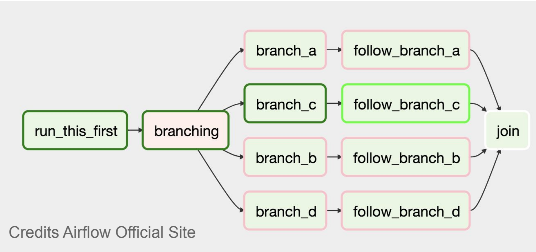
# Introduction to Apache Airflow



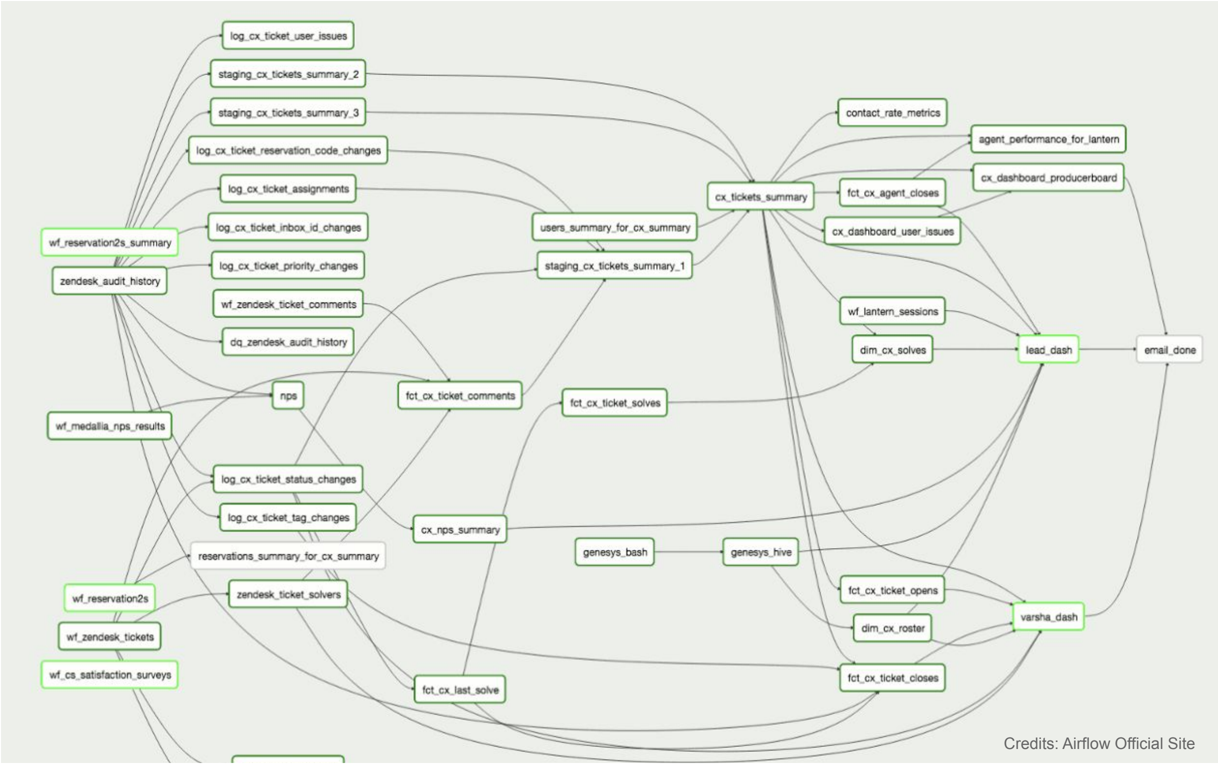
# What is Airflow

**Delivery**Tech

Apache Airflow is an open-source tool for orchestrating workflows



Credits Airflow Official Site



Credits: Airflow Official Site

# What is Airflow



- Originally created by Maxime Beauchemin at Airbnb in 2014
- In March 2016 incubated in Apache Foundation
- Currently it has more than 700 contributors on Github with ~6000 commits
- Written in Python

# What Airflow can do



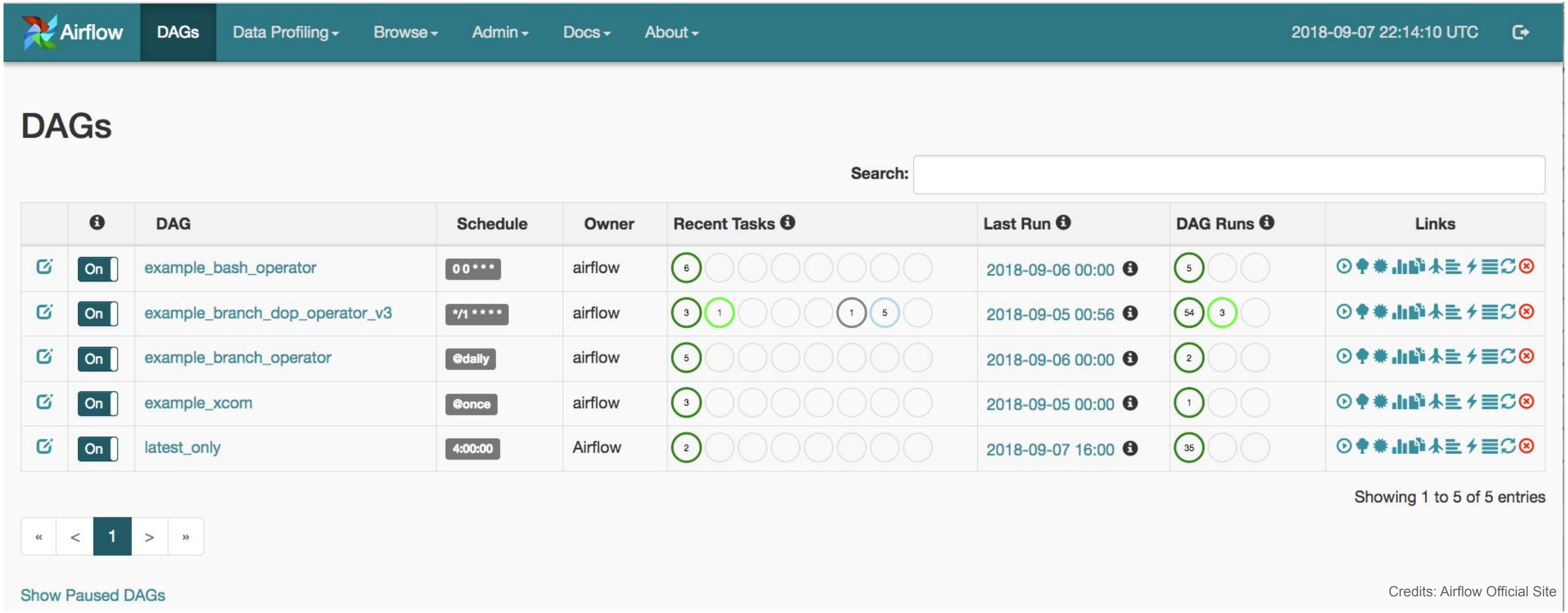
- **Orchestrate sequences of tasks**: when and in which order

  → through a structured way for defining sequences and tasks dependencies

- Handle **failure** well and apply **error handling policy** (e.g. **retries**, notify per email)
- Handle complicated **dependencies** and only runs what it has to
- Real time **Logging**
- Manage the **resources** necessary to run tasks
- Store **variables** and external **connection** configurations (to be referenced by tasks)

# Web UI - DAGs View

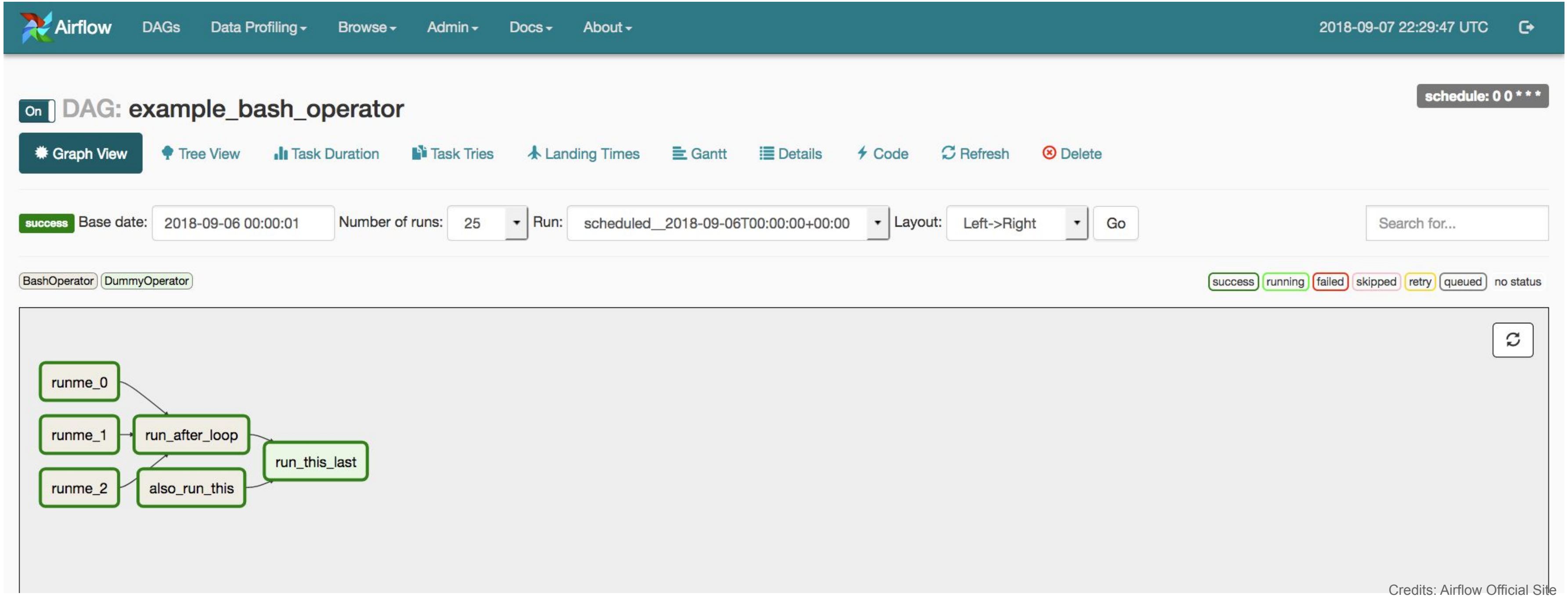





Credits: Airflow Official Site

# Web UI - Graph View

DeliveryTech



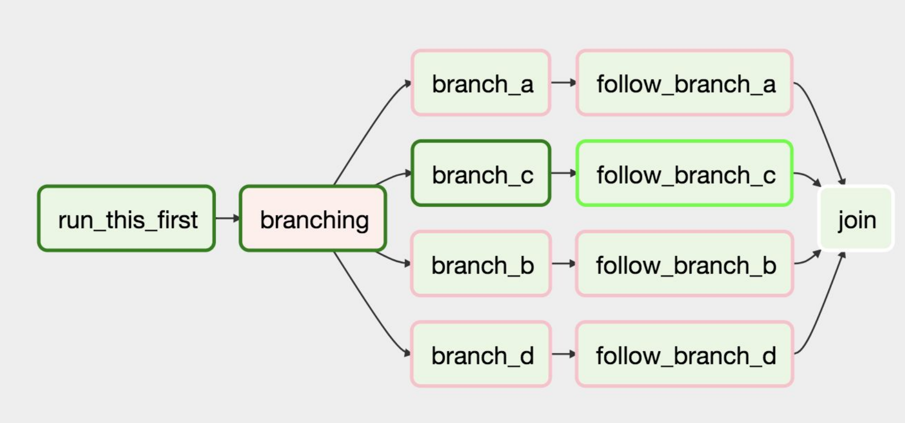

Credits: Airflow Official Site

A **workflow** is a **sequence of tasks** organized in a way that reflects their relationships and dependencies.

An individual workflow in Airflow is **represented** as a **DAG**.

A DAG is a **Directed Acyclic Graph**

- Graph: the tasks, the nodes, of a workflow make up a Graph
- Directed: tasks are ordered
- Acyclic: no loops



# The DAG



```
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
```

**# Library**

```
def print_hello():
    return 'Hello world :)'
```

```
dag = DAG('hello_world',
          description='Hello world DAG',
          schedule_interval='0 17 * * *',
          start_date=datetime(2019, 2, 10)
          )
```

**# Define the DAG**

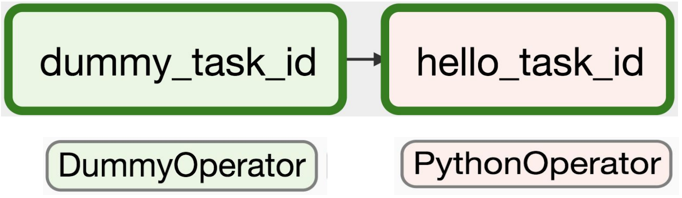
```
dummy_task = DummyOperator(task_id='dummy_task_id',
                           retries=5,
                           dag=dag)

python_task = PythonOperator(task_id='hello_task_id',
                             python_callable=print_hello,
                             dag=dag)
```

**# Define the tasks**

```
dummy_task >> python_task
```

**# Define the tasks dependencies**

**DAG describe HOW to run the workflow**




**Operators determine WHAT gets done, are TEMPLATES for actions**

# Task - status




**NO STATUS** → not active

**QUEUED** → queued for being executed

**RUNNING** → executing the task

**SUCCESS** → task completed with success

**RETRY** → failed, but retrying hoping in a success

**FAILED** → retry or not, failed

**SKIPPED** → previous task established that it’s not needed

**RESCHEDULED** → rescheduling due to concurrency limits reached

# Trigger rules



**Normal workflow**: **trigger tasks** when all their **directly upstream tasks** have **succeeded**

All **Operators** have a **`trigger_rule`** argument which defines the **rule** by which the **generated task** get triggered:

ALL_SUCCESS → (default) all parents have succeeded

ALL_FAILED → all parents are in a failed or upstream_failed state

ALL_DONE → all parents are done with their execution

ONE_SUCCESS → fires as soon as at least one parent succeeds (not wait for all parents to be done)

ONE_FAILED → fires as soon as at least one parent has failed (not wait for all parents to be done)

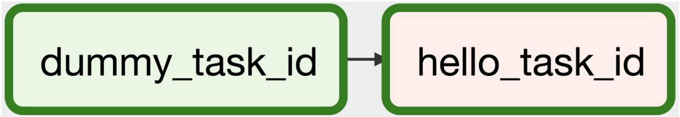
NONE_FAILED → all parents have not failed, i.e. all parents have succeeded or been skipped

DUMMY → dependencies are just for show

```
task4 = DummyOperator(task_id='task4', dag=dag,
                      trigger_rule=TriggerRule.ALL_DONE)
```

**Operators**: the way you define the **action** of a **single task**

```
dummy_task = DummyOperator(task_id='dummy_task_id',
                           retries=5,
                           dag=dag)

python_task = PythonOperator(task_id='hello_task_id',
                             python_callable=print_hello,
                             dag=dag)
```




3 groups of Operators:

- **Action** → performs an action or tells another system to perform an action

  e.g. BashOperator, PythonOperator

- **Transfer** → move data from a system to another

  e.g. RedshiftToS3Transfer, BigQueryToCloudStorageOperator

- **Sensor** → will keep running until a certain criterion is met (through polling)

  e.g. S3KeySensor: wait for a key in a S3 bucket

  e.g. SFTPSensor: wait for a file or directory to be present on SFTP

… through Operators it's possible to perform common actions

like executing bash commands, run SQL query on a DB, transfer data between systems, listen for changes on a server, send emails etc.

easily instantiating the proper Operator class.

→ **Operators** are **Python classes** instantiated when defining a task.

→ Abstract the complexity behind an action (e.g. get connection, copy file, rename file...)

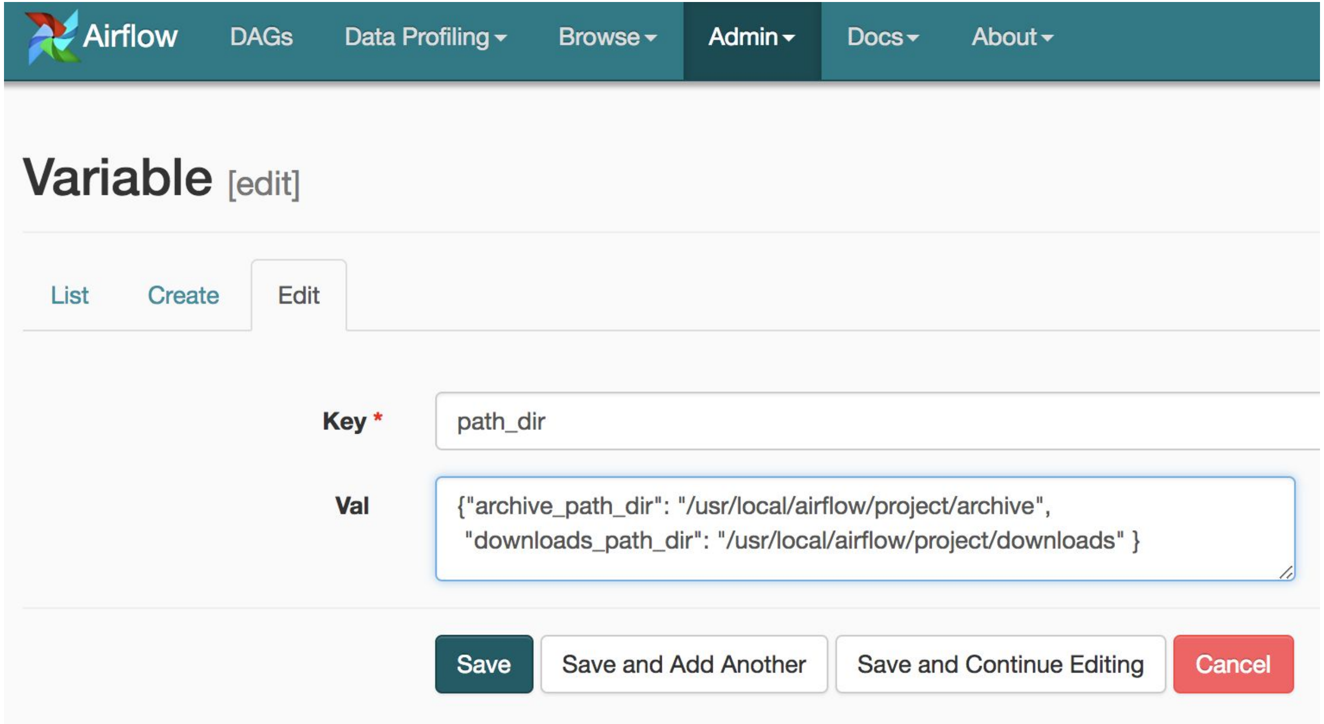
→ Developers choose the Operator that match the task they want to perform

```python
class SqliteOperator(BaseOperator):
    """
    Executes sql code in a specific Sqlite database

    :param sqlite_conn_id: reference to a specific sqlite database
    :type sqlite_conn_id: string
    :param sql: the sql code to be executed. (templated)
    :type sql: string or string pointing to a template file. File must have
        a '.sql' extensions.
    """

    template_fields = ('sql',)
    template_ext = ('.sql',)
    ui_color = '#cdaaed'

    @apply_defaults
    def __init__(
            self, sql, sqlite_conn_id='sqlite_default', parameters=None,
            *args, **kwargs):
        super(SqliteOperator, self).__init__(*args, **kwargs)
        self.sqlite_conn_id = sqlite_conn_id
        self.sql = sql
        self.parameters = parameters or []

    def execute(self, context):
        self.log.info('Executing: %s', self.sql)
        hook = SqliteHook(sqlite_conn_id=self.sqlite_conn_id)
        hook.run(self.sql, parameters=self.parameters)
```

# Variables

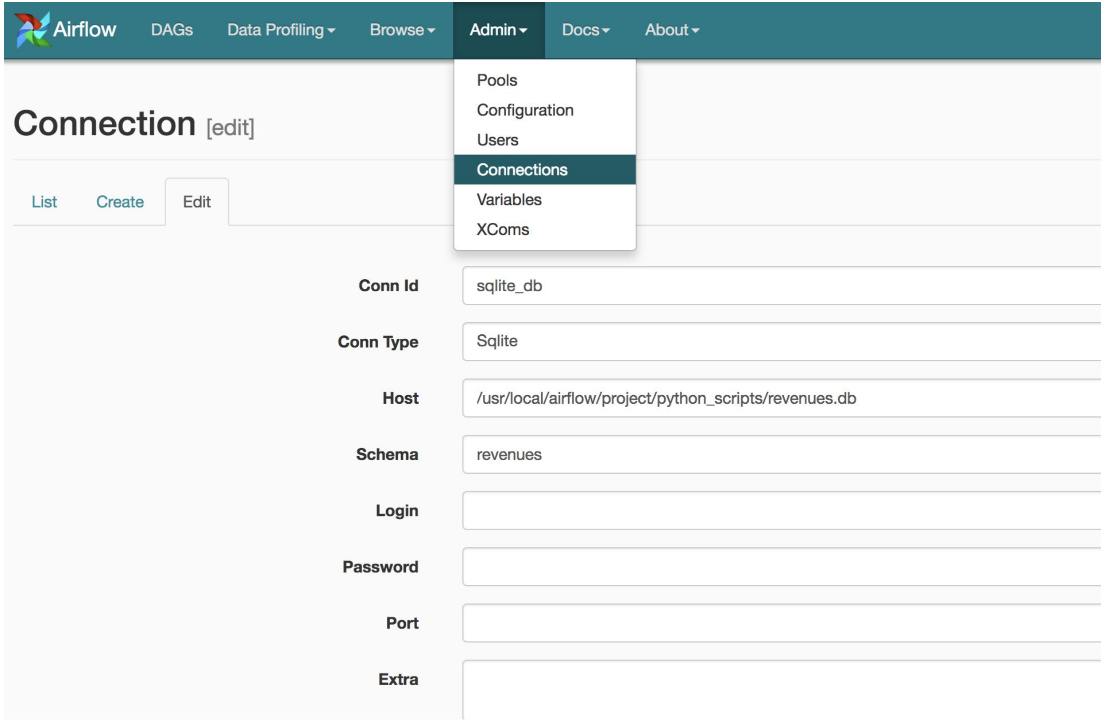


**Variables** → define **key/value pairs** in the Metadata DB (value can be **nested JSON** as well)



**Connections →** Information about how to connect to services provided by external systems

**Hooks** → interface to external System (e.g. SQLiteHook)



# Airflow Architecture

**Metadata DB** → stores all job information + vars, connections, xcom …

**Web Server** (Flask app for UI)

**Scheduler**

→ schedules jobs according to the dependencies defined in the DAG

→ puts tasks in the queue

**Worker** → execute tasks

All these components can be in the same computer or in distributed mode.

When distributed, Airflow utilizes an external tool (Celery) to dispatch tasks.

# Execution Modes

DeliveryTech

**Sequential**:

- Default mode - Minimum setup - works with SQLite DB
- Processes 1 task at time, no parallelism
- should only be used for testing/debugging

**Local**:

- Pseudo-distributed Mode: local workers pick up and run jobs locally via multiprocessing
- Ok for a moderate number of scheduled jobs
- Adopt DB server (e.g. MySQL or PostgreSQL) to support executors

**Celery**:

- Distributed mode (task level)
- Highly scalable in terms of number of workers
- Use Celery as mechanism to distribute work (with message broker Redis, RabbitMQ, …)
- Can be monitored (with Flower)

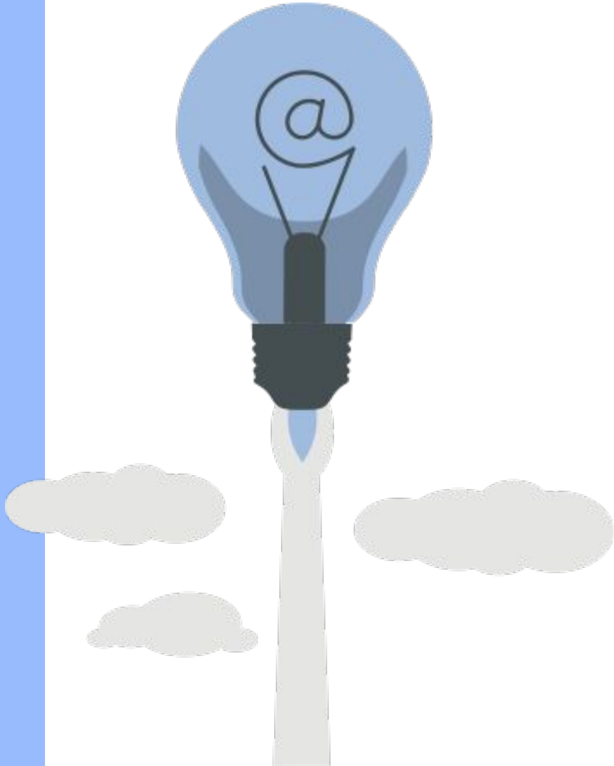
# Still not convinced?



- Tasks can **pass parameters** (**XCom** shared space) to other tasks downstream
- Built-in **authentication** with **encrypted** passwords
- Can **(re)run** tasks
- **Schedules** are **defined in code** (versioning), not in a separate tool and database

DeliveryTech

# Let’s play with Airflow



# Exercise n.1 - Hello Friends

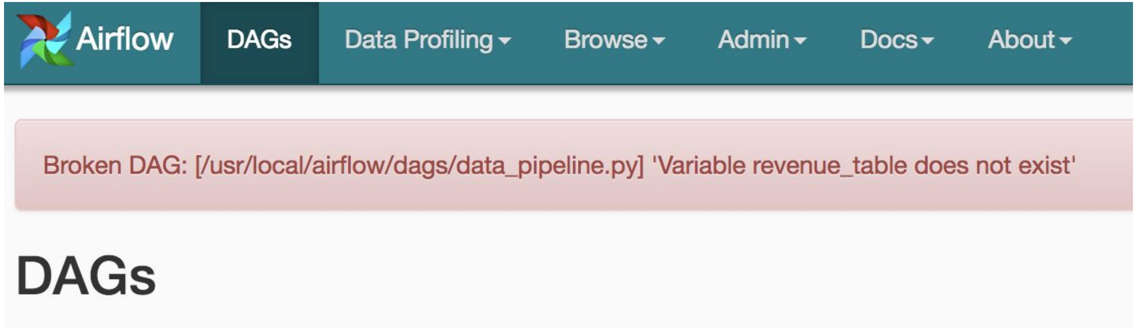
In the **`dags` folder** open with an **editor** the **`hello_friends.py`** file.

Complete, and add to the DAG, the task `bash_task` using the `BashOperator` (already imported) for printing “hello” for the list of name in `my_friends`.

```
my_friends = ["Ana", "Bahadir", "Daniela", "Gabriel", "Hamed", "Ivan", "Jose", "Luis",
              "Lukasz", "Nico", "Sri", "Thiago", "Tomas", "Yue"]

templated_cmd = """
                {% for friend in params.friends %}
                    echo Hello {{friend}}!
                {% endfor %}
                """

bash_task = BashOperator(task_id='give_me_an_id',
                         bash_command=None,
                         params={'friends': []},
                         dag=None)


dummy_task >> python_task
```

DAG: **data_pipeline**.py

You can’t see the DAG **data_pipeline** in the DAGs page and yo have an error message at the top of the DAGs page

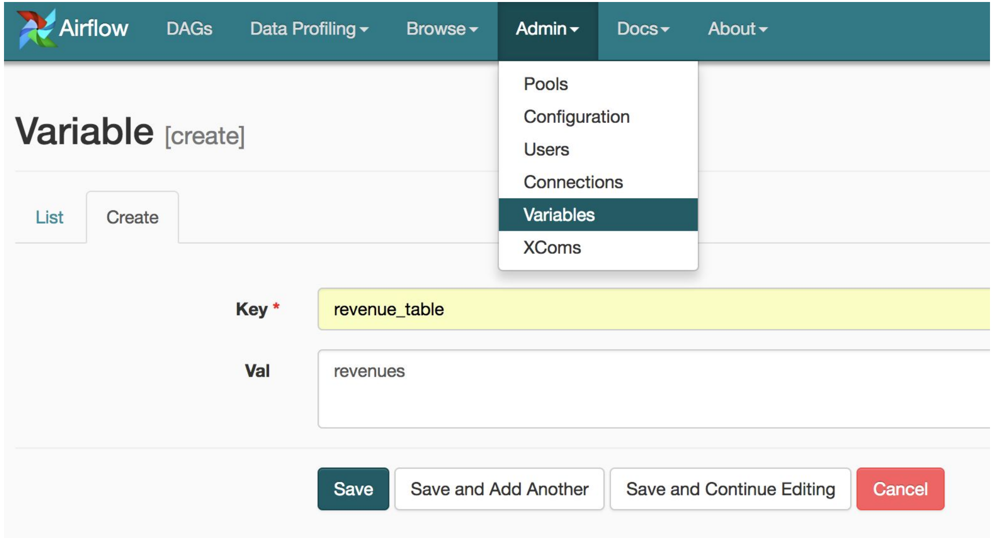



This is due to the missing variable `revenue_table`.

# Exercise n.2 - Data Pipeline - Missing Variable

Fix it adding a the missing variable `revenue_table`

From the menu Admin → Variables → Create




Create a Key and associate it a value like in the above image. Then Save.

# Exercise n.2 - Data Pipeline - Missing Variable



Now you added the variable revenue_table, but you got another error:

```
Broken DAG: [/usr/local/airflow/dags/data_pipeline.py] 'Variable path_dir does not exist'
```

From the menu Admin → Variables → Create a new variable

| | |
|---|---|
| Key | path_dir |
| Val | {"archive_path_dir": "/usr/local/airflow/project/archive", "downloads_path_dir": "/usr/local/airflow/project/downloads"} |

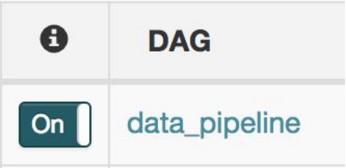
Now the DAG finally appears:

| | | DAG | Schedule | Owner | Recent Tasks | Last Run | DAG Runs | |
|---|---|---|---|---|---|---|---|---|
| | Off | data_pipeline | 1 day, 0:00:00 | Airflow | | | | |

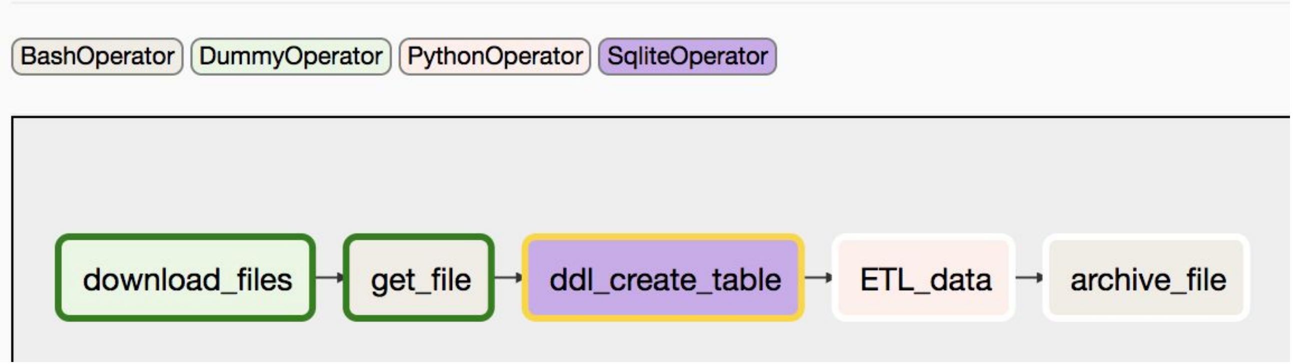
# Exercise n.2 - Data Pipeline - Switch ON



Switch it ON



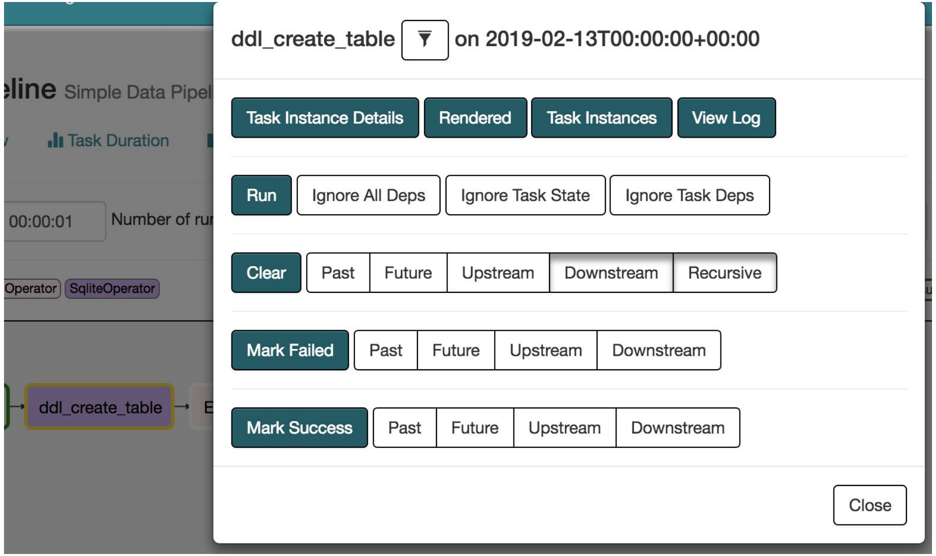

And automatically it will run …



# Exercise n.2 - Data Pipeline - Check task in retry

**DeliveryTech**

Check why the ddl_create_table is on retry → View Log



# Exercise n.2 - Data Pipeline - Missing Connection



The Log says:

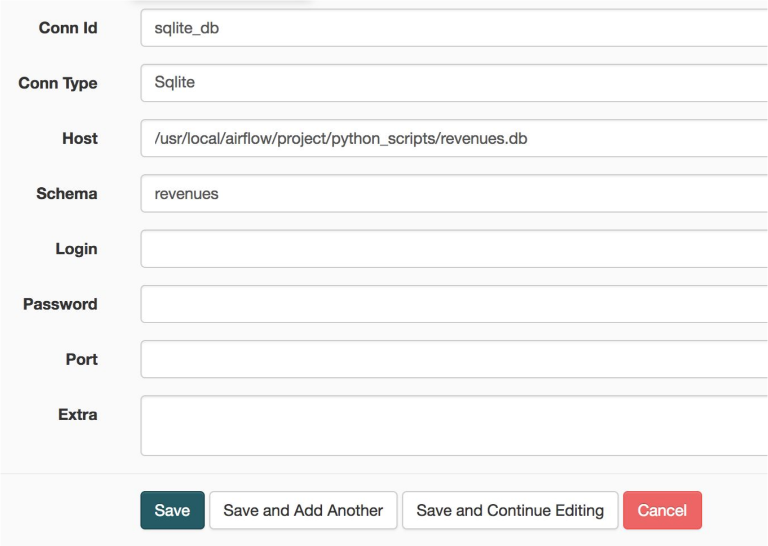
```
{{sqlite_operator.py:50}} INFO - Executing:
                    CREATE TABLE IF NOT EXISTS revenues(
                    year INTEGER,
                    quarter     INTEGER,
                    retailer_country TEXT,
                    retailer_type TEXT,
                    order_method_type TEXT,
                    revenue INTEGER)
              ;
{{models.py:1788}} ERROR - The conn_id `sqlite_db` isn't defined
```

Add the `sqlite_db` connection.

From the menu Admin → Connections → Create

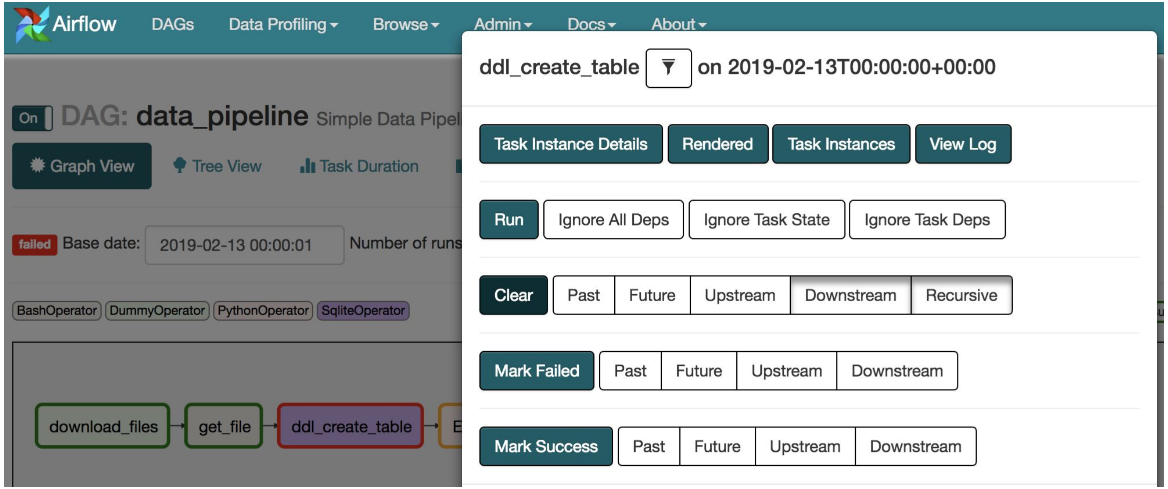
A new Sqlite connection with Host:

`/usr/local/airflow/project/python_scripts/revenues.db`



The task ddl_create_table is in failed status.

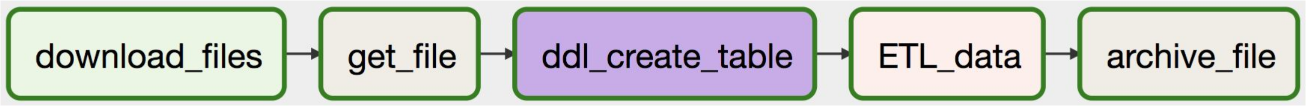
To Re-Run manually click on Clear.

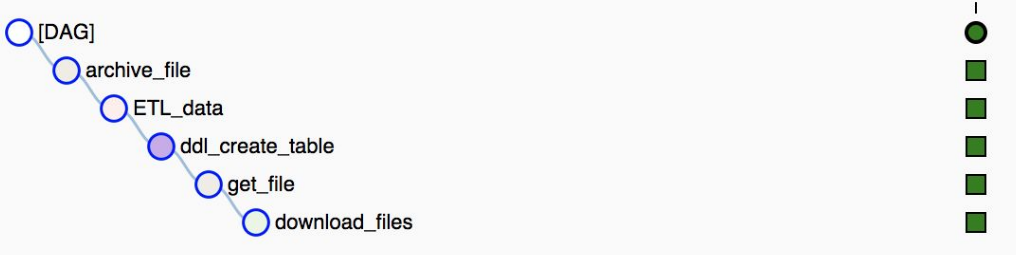



The Scheduler will insert it again in the queue and will run it again.

# Exercise n.2 - Data Pipeline - Success!

All the tasks finished successfully! :-)



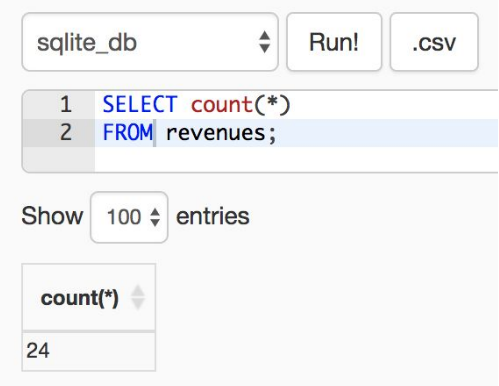



# Exercise n.2 - Data Pipeline - Verify insert

From the log of ETL_data task:

```
{{base_hook.py:83}} INFO - Using connection to: id: sqlite_db. Host: /usr/local/airflow/project/python_scripts/revenues.db,
{{data_transformer.py:25}} INFO - Initialized ETLoader for file 2017_Q2.csv
{{data_transformer.py:39}} INFO - The file 2017_Q2.csv initially contains 119 rows
{{data_transformer.py:44}} INFO - The csv 2017_Q2.csv has been loaded in a Pandas Dataframe
{{data_transformer.py:59}} INFO - The data have been transformed and grouped in 24 rows.
{{data_transformer.py:60}} INFO - Ended data transformation.
{{data_transformer.py:73}} INFO - Data inserted in revenues table.
{{data_transformer.py:78}} INFO - In total there are 24 records in revenues table.
```

Go to Data Profiling → Ad Hoc Query

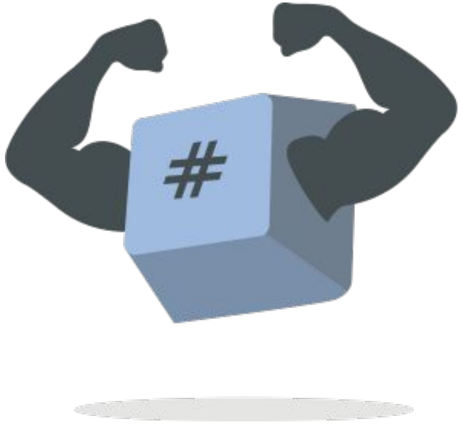


DeliveryTech

# Thank you!

Delivery Hero

# Backup slides



# Assign to a worker a queue



```
airflow worker -q ml_queue
```

```
run_train = BashOperator(
             task_id='task_ml',
             bash_command=<command>,
             queue='ml_queue',
             dag=dag)
```

# Sensors



Sensors are derived from `BaseSensorOperator`

They inherit:

- `timeout`
- `poke_interval`

on top of the `BaseOperator` attributes

Sensor operators keep executing at a time interval and succeed when a criteria is met and fail if and when they time out

```python
def poke(self, context):
    """
    Function that the sensors defined while deriving this class should
    override.
    """
    raise AirflowException('Override me.')


def execute(self, context):
    started_at = timezone.utcnow()
    if self.reschedule:
        # If reschedule, use first start date of current try
        task_reschedules = TaskReschedule.find_for_task_instance(context['ti'])
        if task_reschedules:
            started_at = task_reschedules[0].start_date
    while not self.poke(context):
        if (timezone.utcnow() - started_at).total_seconds() > self.timeout:
            # If sensor is in soft fail mode but will be retried then
            # give it a chance and fail with timeout.
            # This gives the ability to set up non-blocking AND soft-fail sensors.
            if self.soft_fail and not context['ti'].is_eligible_to_retry():
                self._do_skip_downstream_tasks(context)
                raise AirflowSkipException('Snap. Time is OUT.')
            else:
                raise AirflowSensorTimeout('Snap. Time is OUT.')
        if self.reschedule:
            reschedule_date = timezone.utcnow() + timedelta(
                seconds=self.poke_interval)
            raise AirflowRescheduleException(reschedule_date)
        else:
            sleep(self.poke_interval)
    self.log.info("Success criteria met. Exiting.")
```

# CLI commands

DeliveryTech

| | |
|---|---|
| airflow list_dags | print the list of active DAGs |
| airflow list_tasks <dag_id> | print the list of tasks of dag_id |
| airflow list_tasks <dag_id> --tree | print the hierarchy of tasks in dag_id |
| airflow initdb | initialize Metadata DB |
| airflow test <dag_id> <task_id> <date> | test task_id of dag_id |
| airflow run <dag_id> <task_id> <date> | run task_id of dag_id |
| airflow backfill <dag_id> -s <start_date> -e <end_date> | reload / backfill dag_id |
| airflow clear <dag_id> -s <start_date> -e <end_date> -t <task_regex> | clear the state of the tasks in dag_id |
| airflow backfill <dag_id> -s <start_date> -e <end_date> -m true | mark dag runs of dag_id as success |

# Hooks



```
class S3Hook(AwsHook):
    """
    Interact with AWS S3, using the boto3 library.
    """

    def get_conn(self):
        return self.get_client_type('s3')


    def create_bucket(self, bucket_name, region_name=None):
        """
        Creates an Amazon S3 bucket.


    def list_keys(self, bucket_name, prefix='', delimiter='',
                  page_size=None, max_items=None):
        """
        Lists keys in a bucket under prefix and not containing delimiter


    def get_key(self, key, bucket_name=None):
        """
        Returns a boto3.s3.Object
```

# Airflow limitations

DeliveryTech

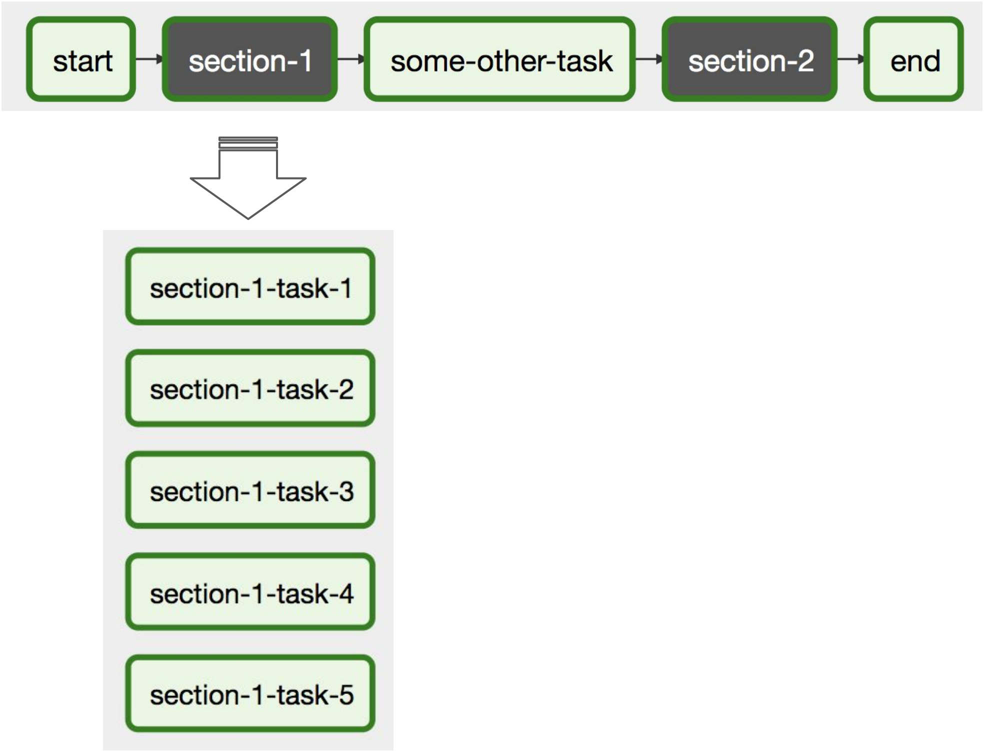
- **Workflows** are expected to be mostly **static** or slowly changing
- **Not stream** processing (processing after the fact)

  (e.g. Monday's data are processed on Tuesday)
- Once successful, re-run again only manually
- Time spent in searching an Operator that performs what the task should do

# SubDAGs



→ encapsulate repeating functionality




```python
section_1 = SubDagOperator(
    task_id='section-1',
    subdag=subdag(DAG_NAME, 'section-1', args),
    default_args=args,
    dag=dag,
)
```

```python
def subdag(parent_dag_name, child_dag_name, args):
    dag_subdag = DAG(
        dag_id='%s.%s' % (parent_dag_name, child_dag_name),
        default_args=args,
        schedule_interval="@daily",
    )

    for i in range(5):
        DummyOperator(
            task_id='%s-task-%s' % (child_dag_name, i + 1),
            default_args=args,
            dag=dag_subdag,
        )

    return dag_subdag
```

# Parallelism & Concurrency

DeliveryTech

In the config file airflow.cfg set

| | |
|---|---|
| `parallelism = 32` | max number of tasks instances that should run in parallel |
| `dag_concurrency = 16` | max number of tasks allowed to run concurrently by the scheduler |

# Airflow as CI

DeliveryTech

| Feature Type | Feature Area | GitLab CI | Airflow |
|---|---|---|---|
| Central Scheduler | Scheduling | Simple via Repo | Airflow Central Scheduler |
| Control Flow | Dependency Management | Sequential Staging | Directed Acyclic Graph |
| Metadata Database | Monitoring & Interaction | Yes - Pipeline specific - Not Linked with Scheduler, Issue 67 | Yes - Web UI which can control Scheduler and Queue |
| Parallelism | Scaleability | Yes - w/in Stages | Yes |
| Multiple Pipelines | Scaleability | No | Yes |
| Pass Data Between Jobs/Tasks | Dependency Management | Artifacts | XCom |
| Variables | Flexibility | Yes | Yes |
| Branching | Flexibility | Git Branches Only | Arbitrary Branches |
| Triggers | Flexibility/Dependency Management | Cron, Commits, API, Simple between stages | Complex w/ Dependencies (no commit based) |
| Dynamic Repeat Tasks | Flexibility | No | SubDAGs |
| Dynamic Environments | Flexibility/Scaleability | Review Apps | No |
| Pipeline Specification | Flexibility | Static | Programmatic |
| Retry | Resiliency | Simple | More Variables |
| Open Source | Documentation | Yes | Yes |

Credit: https://gitlab.com/gitlab-data/analytics/issues/69

- **Dynamic**: pipelines are configuration as code (Python), allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically.

- **Extensible**: Easily define your own operators, executors and extend the library so that it fits the level of abstraction that suits your environment.

- **Elegant**: Airflow pipelines are lean and explicit. Parameterizing your scripts is built into the core of Airflow using the powerful Jinja templating engine.

- **Scalable**: Airflow has a **modular architecture** and uses a message queue to orchestrate an arbitrary number of workers. Airflow is ready to scale to infinity.