Xebia

Airflow Data pipelines

Ismael Cabral

About me, Ismael Cabral

- Background: Msc Sustainable tech / Data Science
- 6 years as Data Scientist / Machine Learning Engineer
- Now: Machine Learning Engineer at Xebia Data
- Currently working on the second edition of "Data Pipelines with Apache Airflow"

About **YOU**

- Background?
- What do you think Airflow does?
- Plans to apply Airflow?
- How many DAGs have you written?

Program

Day 1

- What is Airflow?
- Installation
- Your First DAG
- Scheduling
- Context & Templating
- Branching & TriggerRules
- Sensors

Day 2

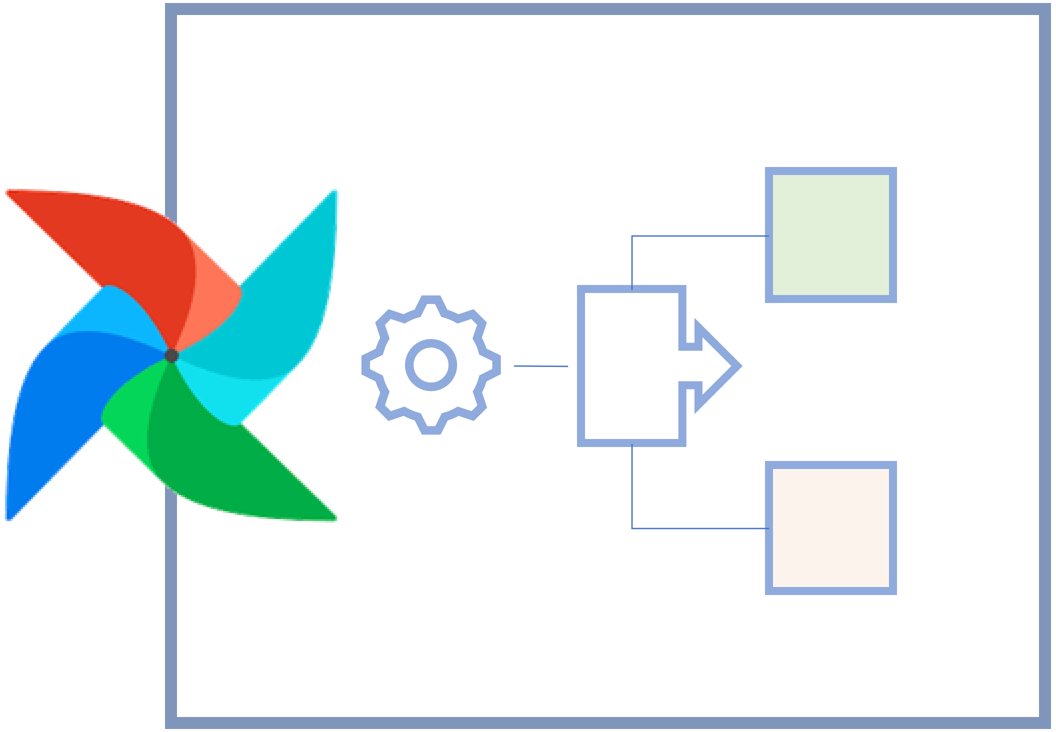
- Scheduling & Backfilling
- DAG configuration
- Variables, Xcoms & Connections
- Finish Capstone Project

Learning Goals

- Learn **fundamental concepts**: DAGs, Operators, Hooks
- Know your way around the **Airflow UI**
- Know a little about how Airflow **works internally**
- Be able to **debug** errors

Airflow

The big picture



What is Apache Airflow?

- **Open-source** platform for creating, scheduling and monitoring workflows
- Started at AirBnB in 2015
- Now used by 200+ companies (ING, LinkedIn, Paypal, HBO, …)
- Contributions from 600+ developers
- Licensed under the Apache License 2.0, for **free use and distribution.**

Xebia | Data

Scenario: We are working on a weather forecast data pipeline for a new app

Requirements:

- Run daily
- One geographical location

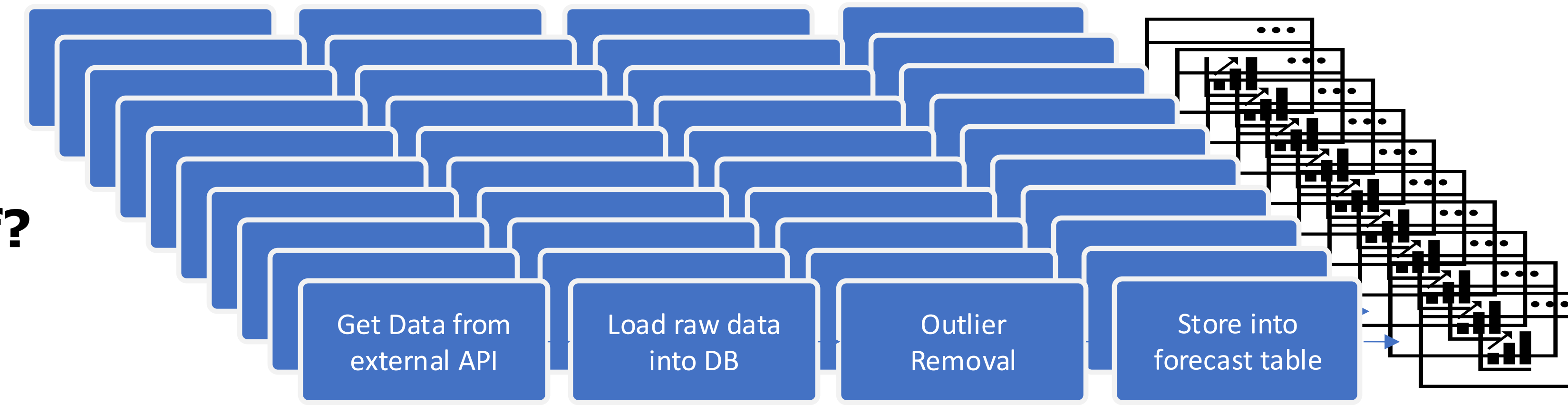
Many things can go wrong

Permission changes
Code-errors
Memory overflow
No data
Auth-errors
No connectivity
API-changes
Schema changes
Data format change
API not available
Rate-limits

Good news! The forecast app is scaling

Requirements:

- Run hourly
- 100 geographical locations



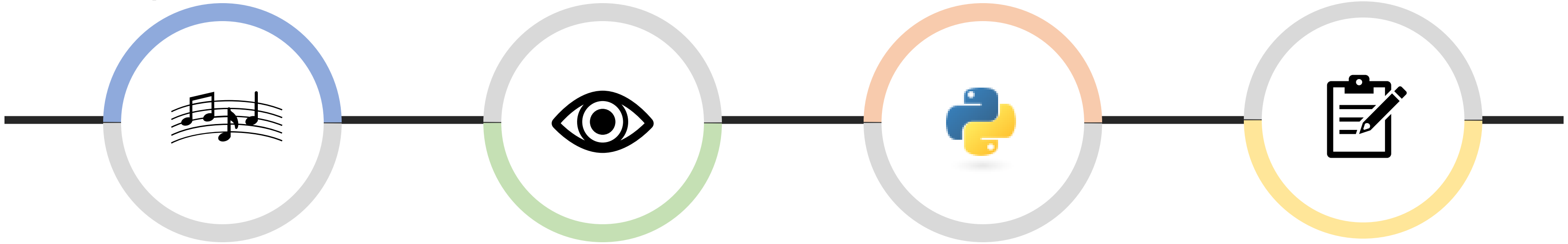
How do you keep track of?

- Schedule over different timezones
- The status of your pipeline
- Tasks failed
- Manage changes / failure
- Insightful Logging
- Overall Performance

Airflow is the orchestrator of your data pipelines

Airflow Scheduler coordinates that your tasks are "played" when they are needed

Airflow lets you write instructions in Python. So you have the right instruments to perform your tasks



Airflow UI gives an overview of what is going good and bad in the "playbook"

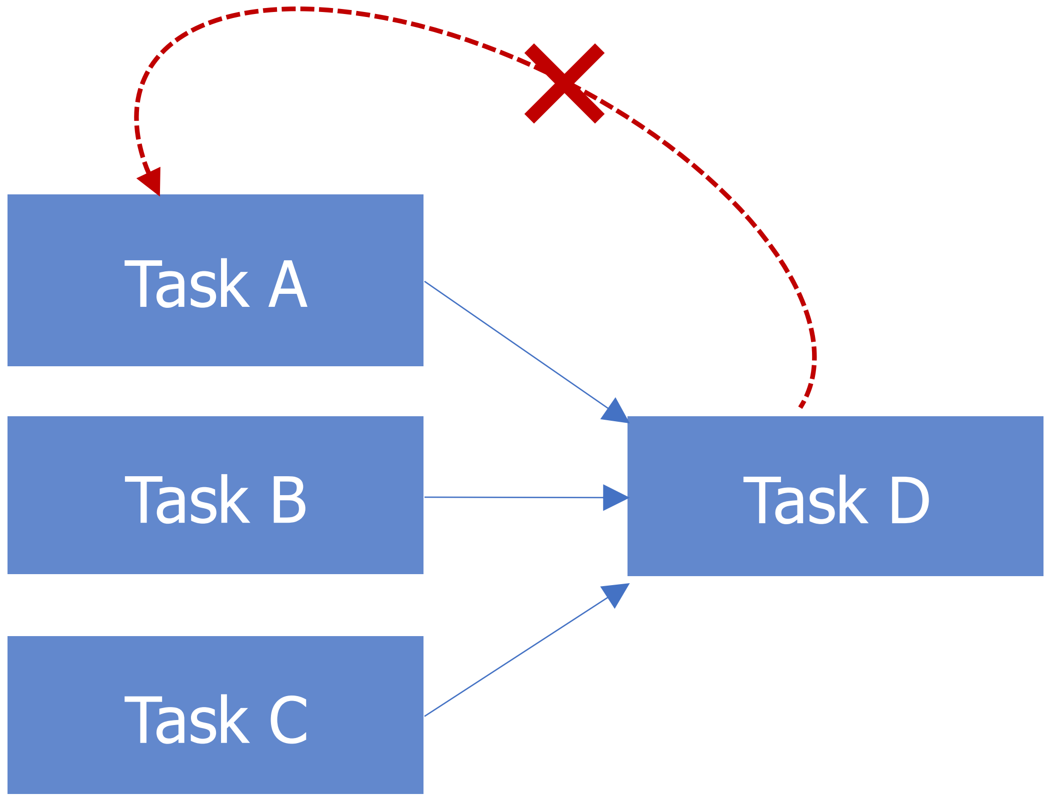
The organized **logs** stream brings clarity on the history of the runs

When NOT to use Airflow

- Airflow is not a (data) processing framework (such as Spark).
- Implementing highly dynamic / changing pipelines.
- Airflow focusses on orchestration and monitoring
- Not for streaming data solutions

Directed **A**cyclic **G**raph

AKA: Nodes with directed flow and no loops



Operators

They define and execute tasks within workflows

Action Operators

BashOperator: Executes a bash command or script.

PythonOperator: Runs a Python function as a task.

EmailOperator: Sends an email notification.

Transfer Operators

FileTransferOperator: Transfers files between different locations or systems.

SQLTransferOperator: Transfers data between databases using SQL queries.

S3ToRedshiftOperator: Loads data from Amazon S3 into Amazon Redshift.

Sensor Operators

HttpSensor: Polls an HTTP endpoint until it returns a successful response.

S3KeySensor: Waits for a specific key or file to be available in Amazon S3.

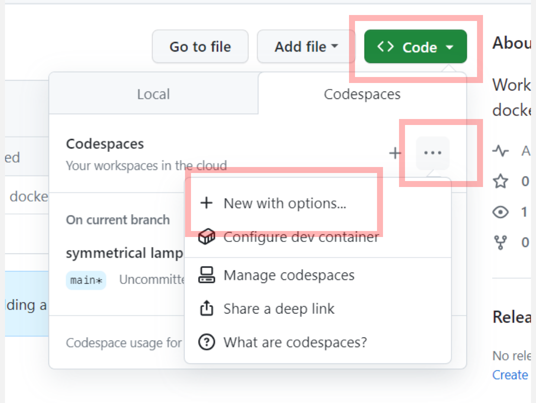
TimeSensor: Pauses the workflow until a specific time or time interval is reached.

Airflow Installation

Fork Repo: https://github.com/godatadriven/airflow_workspace

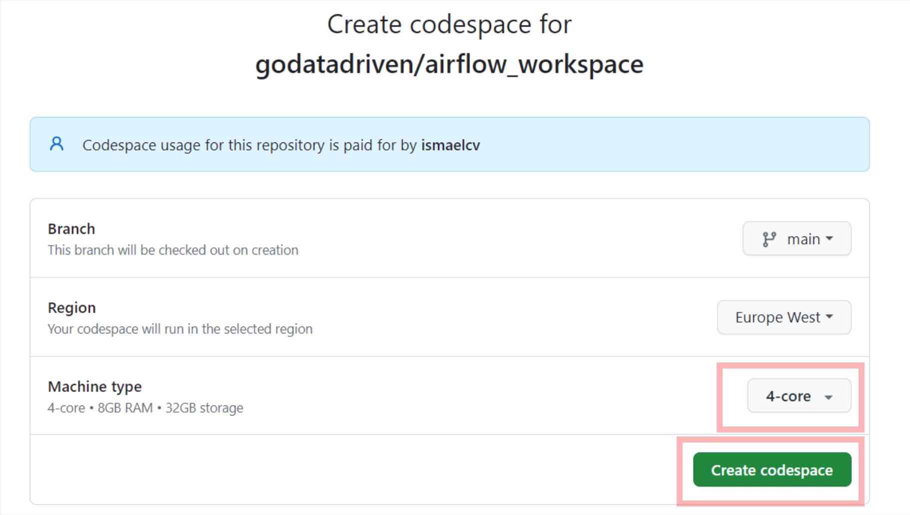
Select:

Code > ... > + New with options ...



Select:

4-core > Create codespace



VS Code Web environment.

- This will be your workspace for the rest of this training
- In the terminal (ctrl + `) check you have more than 4GB of allocated memory:

```
docker run --rm "debian:bullseye-slim" bash -c
'numfmt --to iec $(echo $(($(getconf _PHYS_PAGES)
* $(getconf PAGE_SIZE))))
```

- **(ONLY RUN ONCE)** You need to run database migrations and create the first user account. It is all defined in the docker compose file so just run:

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

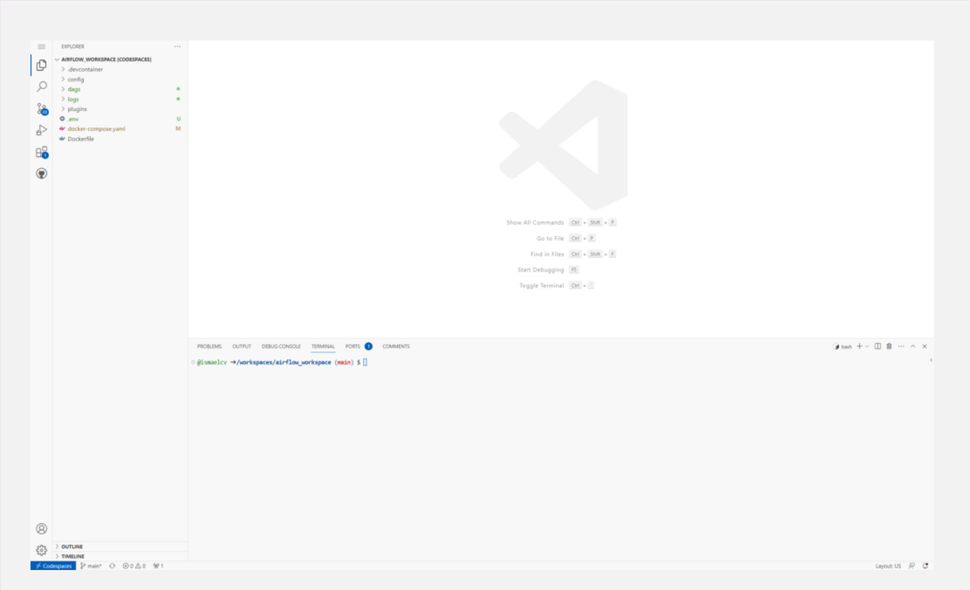
outputs

```
airflow-init_1 | Upgrades done
airflow-init_1 | Admin user airflow created
airflow-init_1 | 2.6.1 start_airflow-init_1
exited with code 0
```

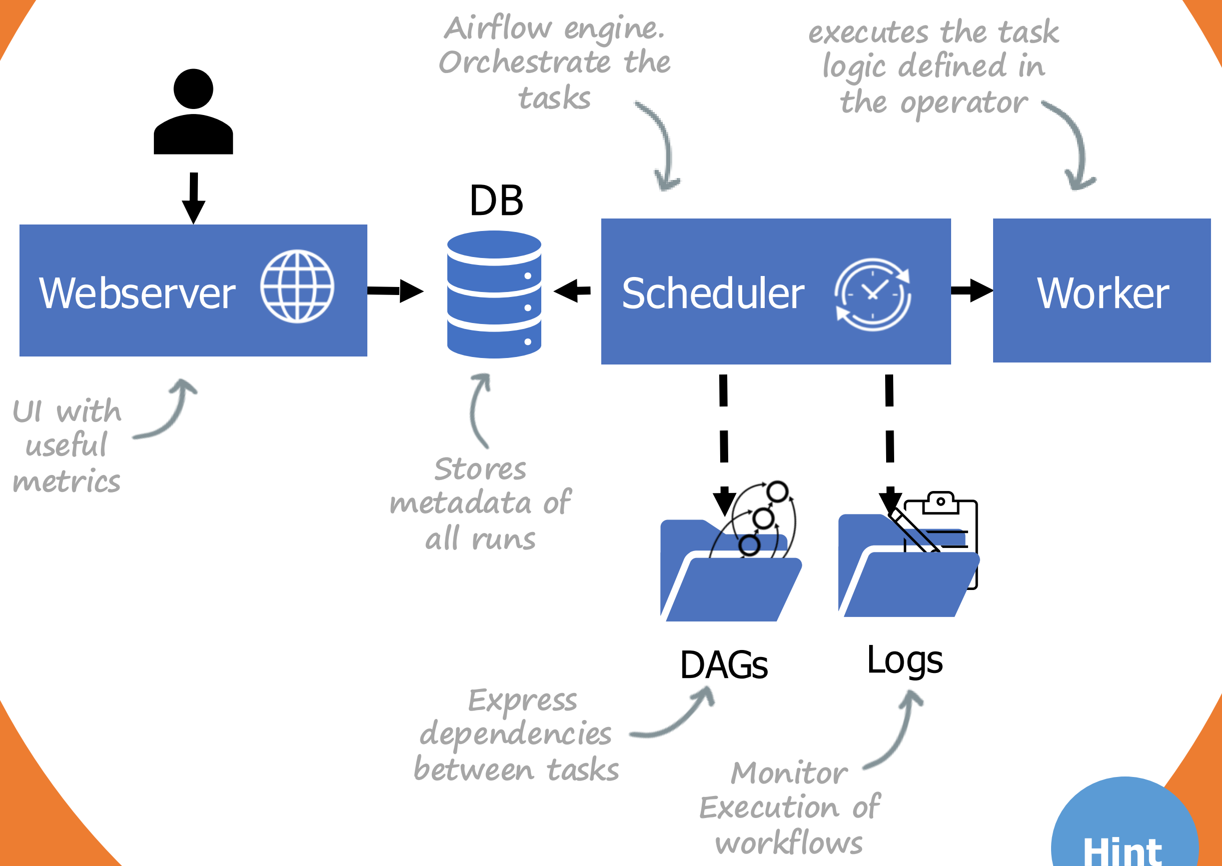
- Now you can start all services:

```
docker compose up
```

Airflow UI → http://0.0.0.0:8080



Xebia | Data



Core Components

Hint

Check all the Airflow components in the console with:

```
docker ps
```

Xebia | Data

Capstone Project:

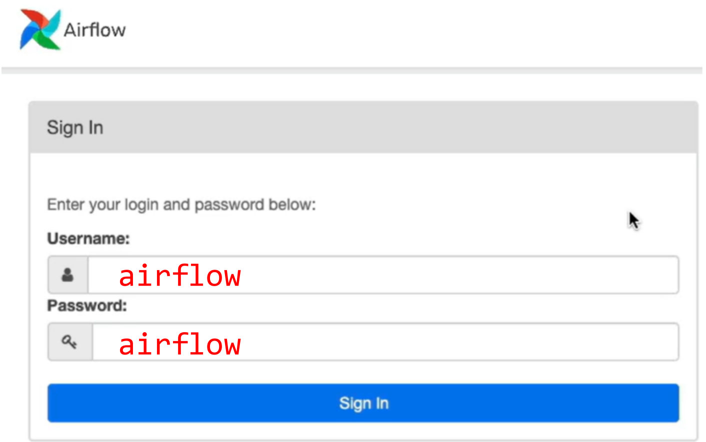
You are a rocket scientist for a day (or two)!

Xebia | Data

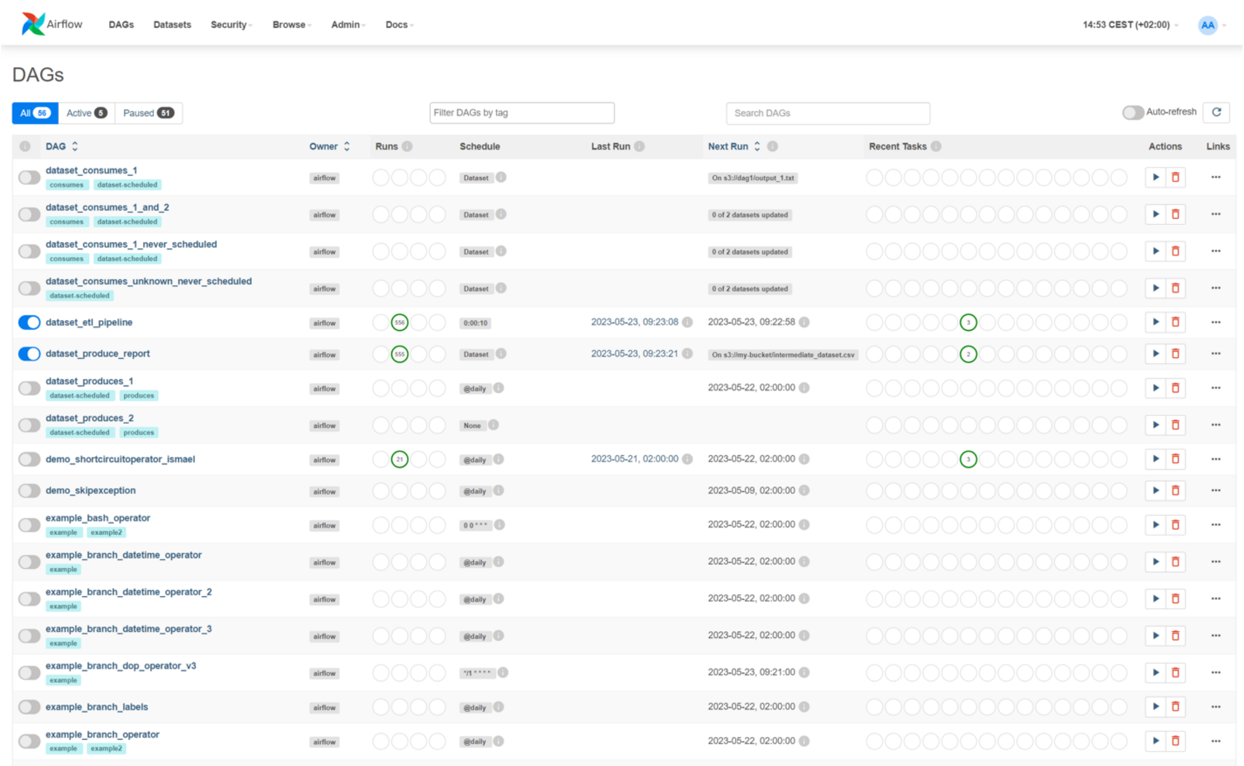
Airflow UI

Walkthrough

Localhost:8080/



DAG View

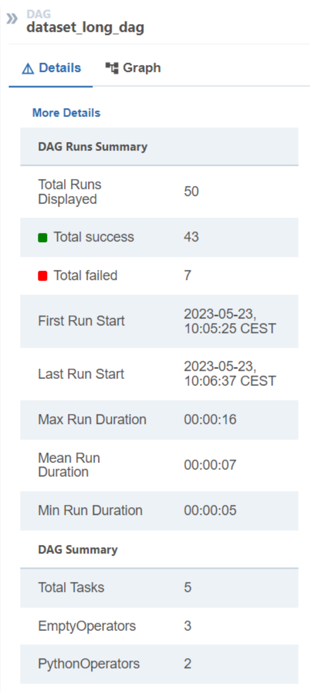


- List all the DAG's in the Airflow Instance
  - Tags
  - Schedule
  - Run information
  - Delete
  - Pause/ Unpause Dags

Grid View

The state of your DAG runs

- Wide view of all the Runs
- If you see all green, you stop worrying
- If you see yellow / red you can go deeper to review what happened

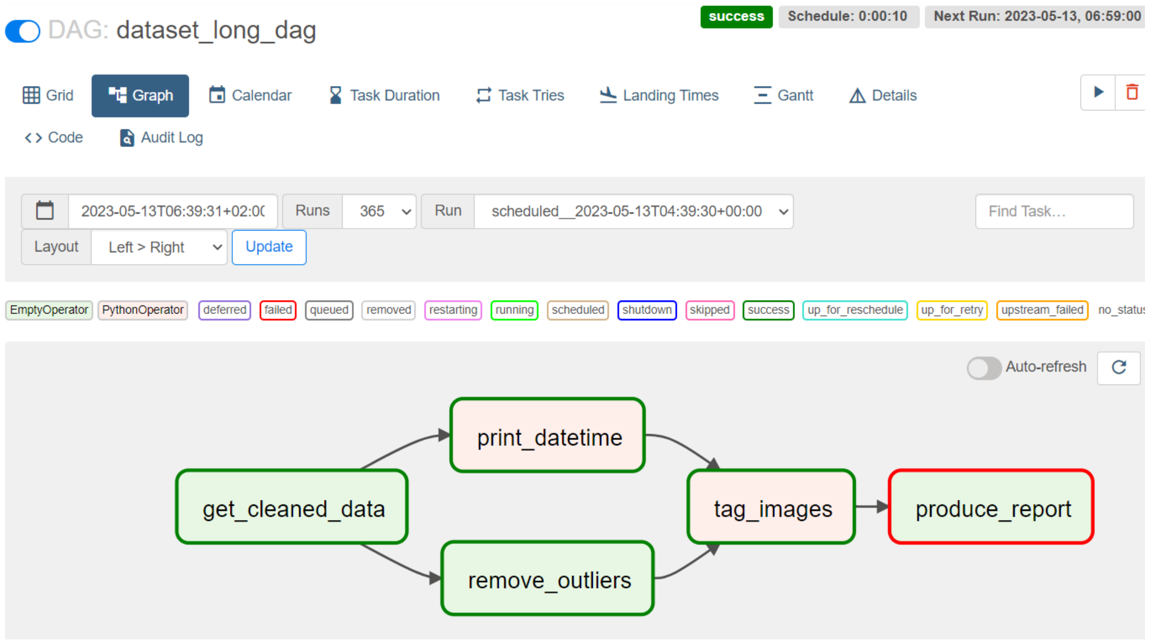


- Details the state of each task per run
- Total Success / Fail tasks
- Duration per run
- Task metadata (Name, id, etc)

Graph view

Logical Sequence of your DAG

- You can see how tasks depend on each other
- What kind of operators do we have by color filling
- Status of each task by color edge
- Verify task dependencies are correct

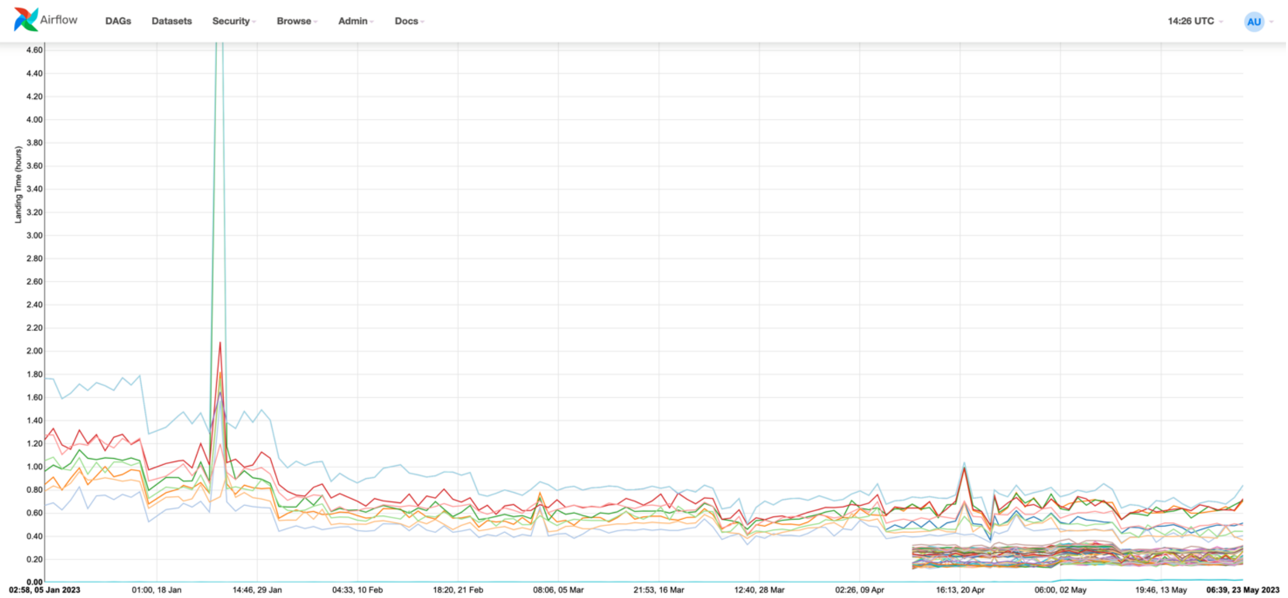


Xebia | Data

Landing times

Measures Schedule time vs Realized time

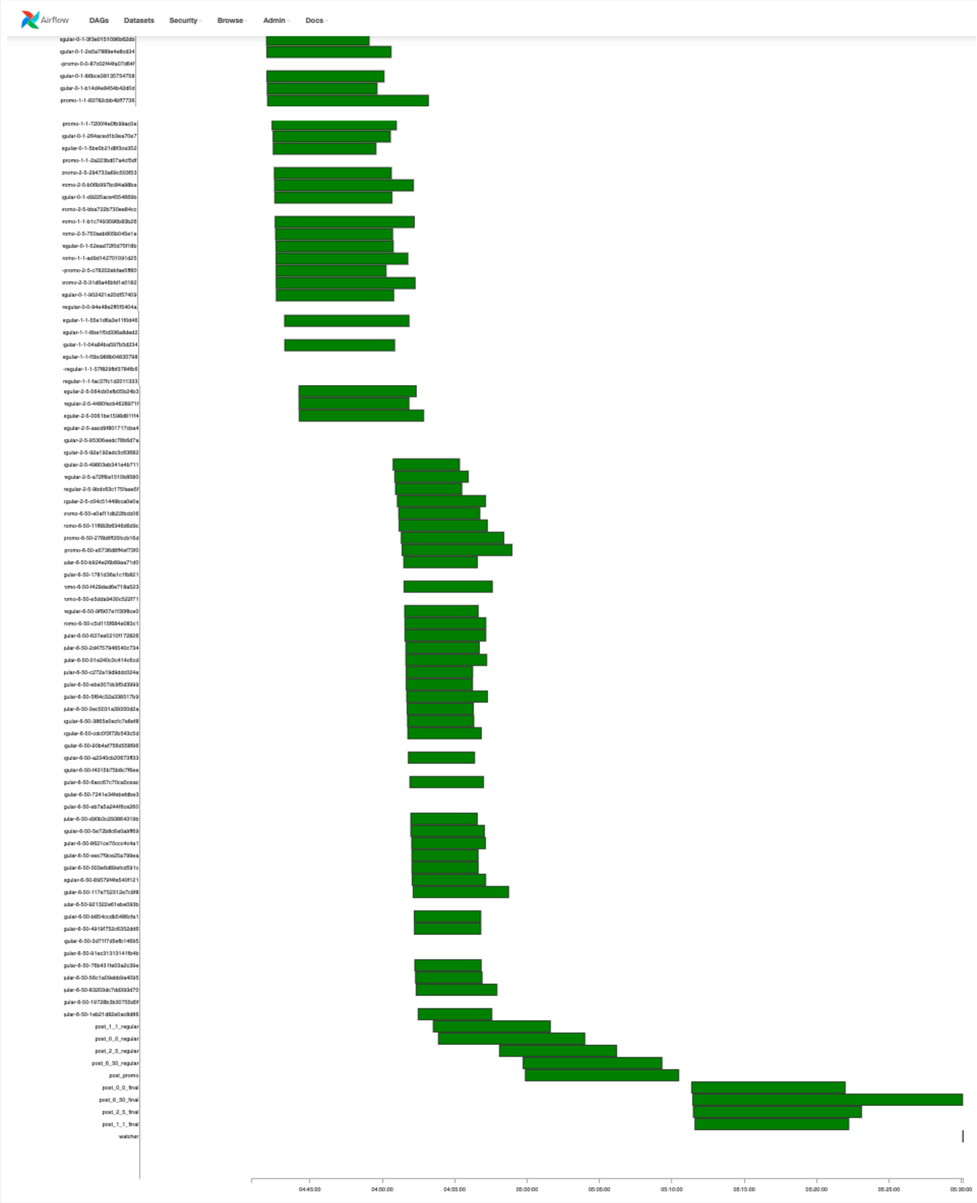
- Overall view of your system performance
- Identify periods in time when there was a task failure
- See how task duration is improving (or degrading) over time
- Identify when new tasks have been added



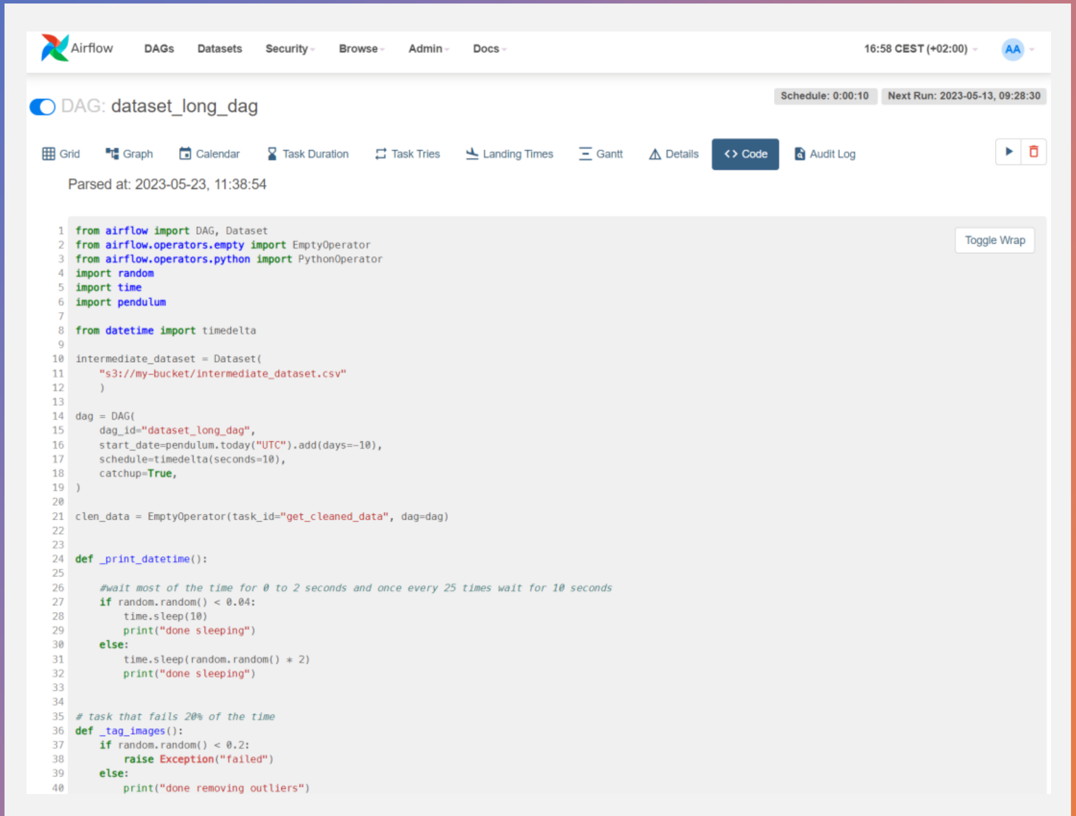
Xebia | Data

Gant view

Detailed task duration times



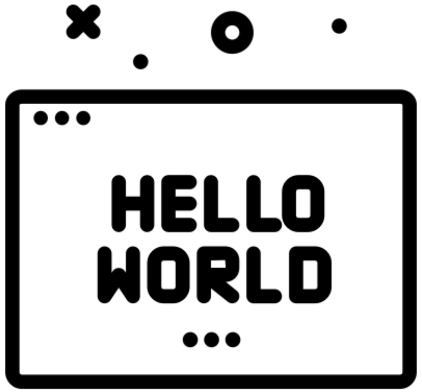
- Bar size tells you how long a task takes to complete
- Helps you to identify bottlenecks in your pipeline
- Overlapping bars, means that the tasks are running in parallel



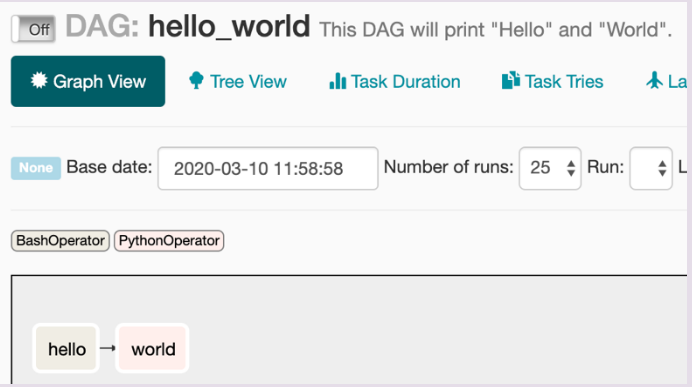
Code view

- Displays the code of the data pipeline
- Make sure that the latest code is running

Coding a DAG



- Create file called hello_world.py in dags/
- Write the hello_world.py code
- Wait 1-5 mins to pick up the new file
- By default new DAGs are paused



A Dag in Python

```python
from airflow.models import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
import pendulum

with DAG(
    dag_id="hello_world",
    start_date = pendulum.today('UTC').add(days=-14),
    description='This DAG will print "Hello“ & "World".',
    schedule="@daily",
):

        hello = BashOperator(
                task_id="hello",
                bash_command="echo 'hello’"
        )

        world = PythonOperator(
                task_id="world",
                python_callable=lambda: print("world")
        )

        hello >> world
```

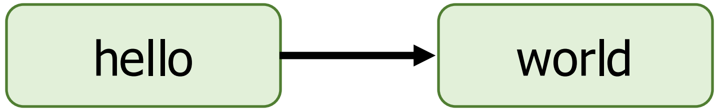
imports

Our **DAG** object

Tasks: hello and world
Operators: BashOperator and PythonOperator

Task dependencies

The DAG looks like this:



There are many ways to create a DAG

```python
from airflow.models import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
import pendulum

dag =  DAG(
   dag_id="hello_world",
   start_date = pendulum.today('UTC'),
   schedule="@daily",
)

hello = BashOperator(
   task_id="hello",
   bash_command="echo 'hello'",
   dag=dag
)

def _hello_world():
   print("world")

world = PythonOperator(
   task_id="world",
   python_callable = _hello_world,
   dag=dag
)

hello >> world
```

Basic

```python
from airflow.models import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
import pendulum

def _hello_world():
   print("world")

with DAG(
   dag_id="hello_world",
   start_date = pendulum.today('UTC'),
   schedule="@daily",
):

          hello = BashOperator(
                    task_id="hello",
                    bash_command="echo 'hello'"
          )

          world = PythonOperator(
                    task_id="world",
                    python_callable= _hello_world
          )

          hello >> world
```

Intermediate

```python
from airflow.models import DAG
from airflow.decorators import task
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
import pendulum

with DAG(
   dag_id="hello_world",
   start_date = pendulum.today('UTC'),
   schedule="@daily",
):

          hello = BashOperator(
                    task_id="hello",
                    bash_command="echo 'hello'"
          )

          @task
          def world():
             print("world")

          hello >> world()
```

Advanced

Be Aware!

Changes in `start_date`

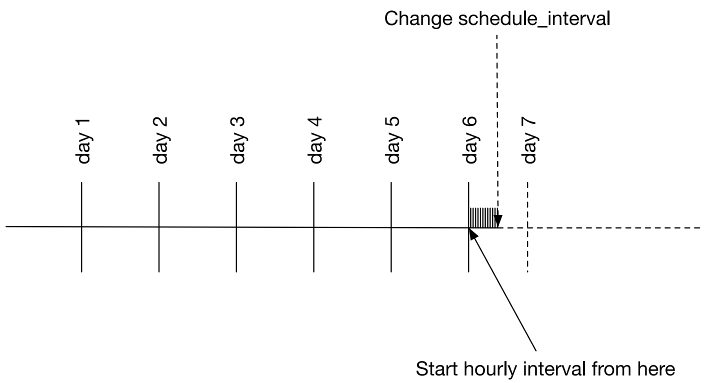
- Airflow does no cope with well with `start_date` changes. It holds on to the first `start_date` you give it.
- Ways to work around it:
  - Change the DAG name, this registers as a new DAG in Airflow
  - Manually backfill the missing DAG runs with "airflow backfill -s [start date] -e [end date] [DAG]‶

Changes in `schedule`

`schedule="@daily"`

↓

`schedule="@hourly"`

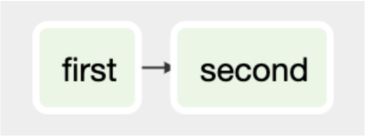


- Airflow also does not cope nicely with changing schedule
- It will take the new schedule, starting from the last DAG run
- With the new hourly interval, several DAG runs will be executed because the last run was at midnight

Task Dependencies

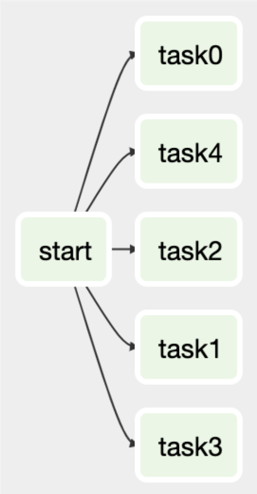
Common way to set dependencies

```
first >> second
     or:
second << first
```



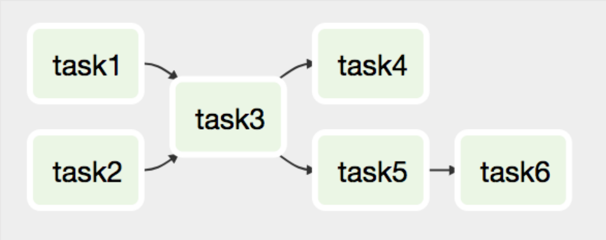
One to many

```
start = EmptyOperator(task_id="start", dag=dag)

tasks = [EmptyOperator(task_id=f"task{i}",
dag=dag) for i in range(5)]

start >> tasks
```



Chaining tasks

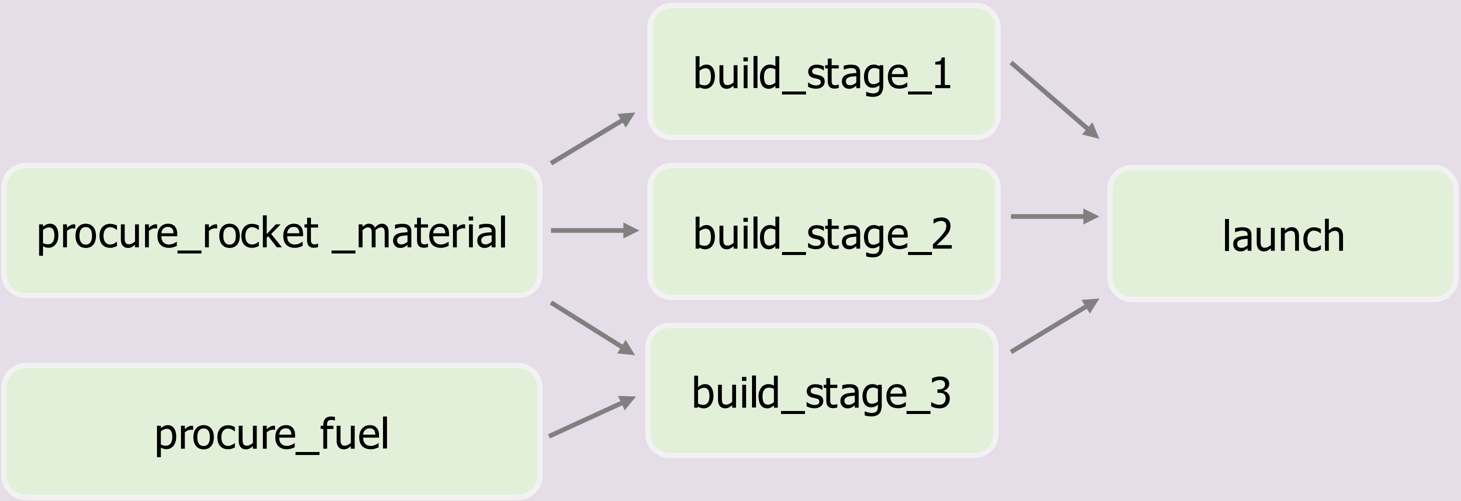
```
t1 = EmptyOperator(task_id="task1", dag=dag)
t2 = EmptyOperator(task_id="task2", dag=dag)
t3 = EmptyOperator(task_id="task3", dag=dag)
t4 = EmptyOperator(task_id="task4", dag=dag)
t5 = EmptyOperator(task_id="task5", dag=dag)
t6 = EmptyOperator(task_id="task6", dag=dag)

[t1, t2] >> t3 >> t4
t3 >> t5 >> t6
```



Exercise 1

Create a structure DAG for our launch?

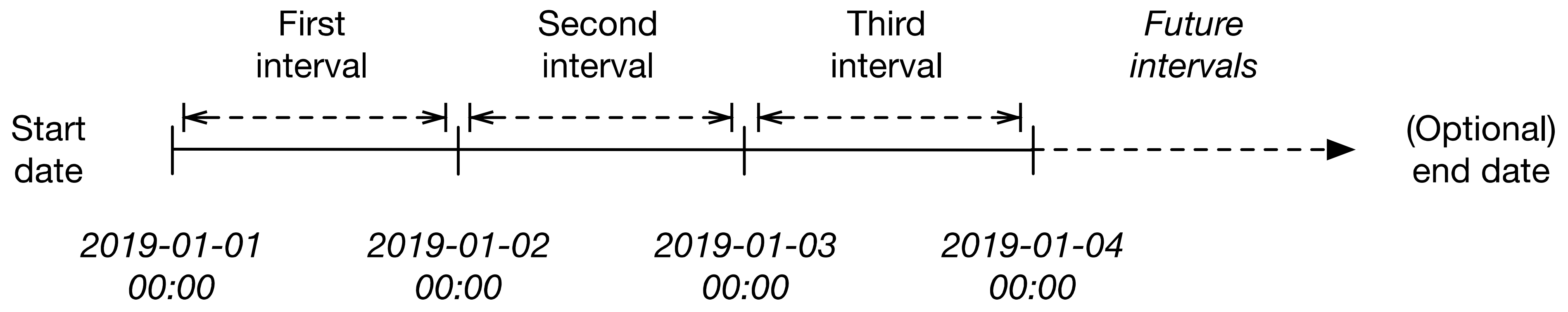
- You can use the `EmptyOperator`
- `from` **airflow.operators.empty import EmptyOperator**



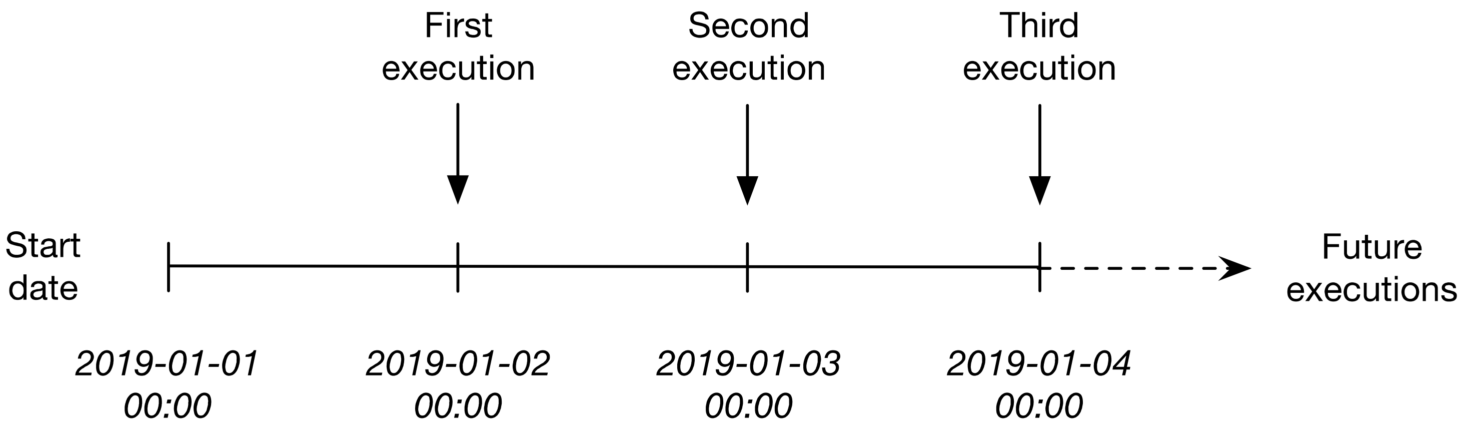
Time Scheduling

Schedule intervals

```
dag = DAG(
    dag_id="demo",
    start_date=datetime.datetime(2019, 1, 1),
    schedule="@daily",
)
```



Schedule execution



Airflow starts execution at the *END* of an interval!

Schedule aliases

Alias	Meaning	Equivalent cron
None	No schedule, only for manual triggering	
`@once`	Run only once	
`@hourly`	00:00 of each hour	`0 * * * *`
`@daily`	00:00:00 of each day	`0 0 * * *`
`@weekly`	00:00:00 every Sunday	`0 0 * * 0`
`@monthly`	00:00:00 of the first day of each month	`0 0 1 * *`
`@yearly`	00:00:00 on every January 1st	`0 0 1 1 *`

Xebia | Data

Cron vs timedelta

- We can set schedule intervals with cron, datetime.timedelta() and dateutil.relativedelta.relativedelta():

Cron	Equivalent timedelta/relativedelta
`0 * * * *`	`datetime.timedelta(hours=1)`
`0 0 * * *`	`datetime.timedelta(days=1)`
`0 0 * * 0`	`datetime.timedelta(weeks=1)`
`0 0 1 * *`	`dateutil.relativedelta(months=1)`
`0 0 1 1 *`	`dateutil.relativedelta(years=1)`

Timedelta does not know >=months.

Are you good in **cron**?

If you are not, you can always make use of crontab.guru

Exercise 2

From the previous exercise:

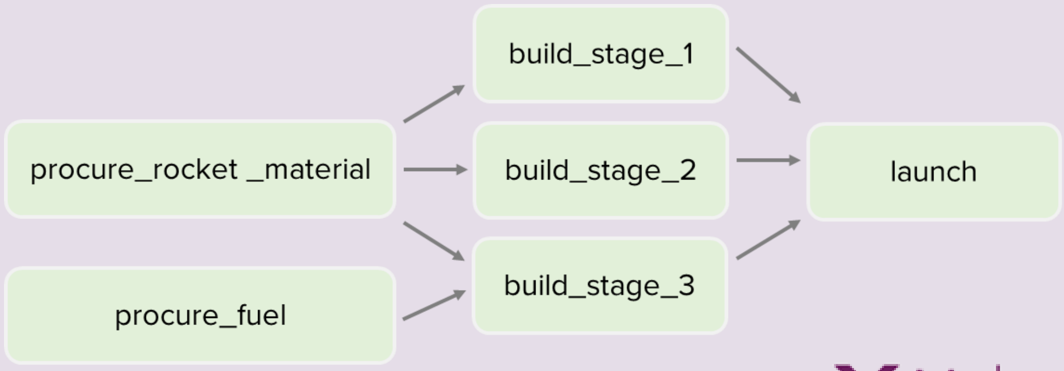
- *Try creating the following schedule intervals:*

  a) At 13:45 every Mon/Wed/Fri

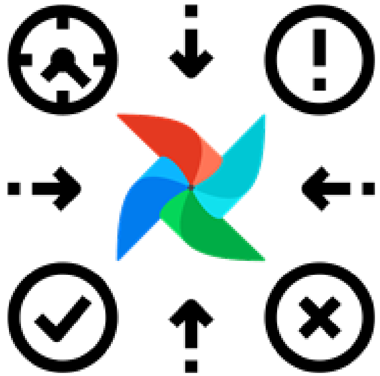
  b) Every 3 days

- Starting 90 days ago

- When to use **`cron`** or **`timedelta`**?



Airflow Context



The Airflow "context"

- Airflow provides information about the execution in the Airflow task "context".
- This includes:
  - The (previous/next) execution_date of the DAG run
  - String formatted execution dates
  - The DAG object
  - Additional variables passed into the context (e.g. templates_dict)

What’s in the context?

```python
def print_context(**context):
    pprint(context)
```

```
{'END_DATE': '2018-01-01',
 'conf': <module 'airflow.configuration' from '/opt/conda/lib/python3.6/site-packages/airflow/configuration.py'>,
 'dag': <DAG: templated_task_dag>,
 'dag_run': None,
 'ds': '2018-01-01',
 'ds_nodash': '20180101',
 'end_date': '2018-01-01',
 'execution_date': <Pendulum [2018-01-01T00:00:00+00:00]>,
 'inlets': [],
 'latest_date': '2018-01-01',
 'macros': <module 'airflow.macros' from '/opt/conda/lib/python3.6/site-packages/airflow/macros/__init__.py'>,
 'next_ds': '2018-01-02',
 'next_execution_date': datetime.datetime(2018, 1, 2, 0, 0, tzinfo=<TimezoneInfo [UTC, GMT, +00:00:00, STD]>),
 'outlets': [],
 'params': {},
 'prev_ds': '2017-12-31',
 'prev_execution_date': datetime.datetime(2017, 12, 31, 0, 0, tzinfo=<TimezoneInfo [UTC, GMT, +00:00:00, STD]>),
 'run_id': None,
 'tables': None,
 'task': <Task(PythonOperator): demo_templating>,
 'task_instance': <TaskInstance: templated_task_dag.demo_templating 2018-01-01T00:00:00+00:00 [None]>,
 'task_instance_key_str': 'templated_task_dag__demo_templating__20180101',
 'templates_dict': None,
 'test_mode': True,
 'ti': <TaskInstance: templated_task_dag.demo_templating 2018-01-01T00:00:00+00:00 [None]>,
 'tomorrow_ds': '2018-01-02',
 'tomorrow_ds_nodash': '20180102',
 'ts': '2018-01-01T00:00:00+00:00',
 'ts_nodash': '20180101T000000+0000',
 'var': {'json': None, 'value': None},
 'yesterday_ds': '2017-12-31',
 'yesterday_ds_nodash': '20171231’
}
```

Xebia | Data

PythonOperator vs all other operators

- In all operators, arguments are templated strings
- The PythonOperator takes code, which is therefore templated differently

```
print_exec_date = BashOperator(
    task_id="demo_templating",
    bash_command="echo {{ execution_date }}"
)
```

```
def _print_exec_date():
    print("{{ execution_date }}")
```

This does *not* work

```
def _print_exec_date(**context):
    print(context["execution_date"])
```

This does work

```
print_exec_date = PythonOperator(
    task_id="demo_templating",
    python_callable=_print_exec_date,
)
```

python_callable vs templates_dict

- Python callable takes code
- The values in templates_dict are templated strings

```
def _print_exec_date(**context):
    print(context[“templates_dict"][”execution_date”])


print_exec_date = PythonOperator(
    task_id="demo_templating",
    python_callable=_print_exec_date,
    provide_context=True,
     templates_dict={
        “execution_date”: “{{ execution_date }}”
    },
)
```

Exercise 3

Let's familiarize with the context

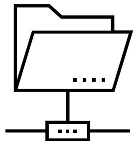
- *Echo the following messages with the bash operator:*
  - *"[task] is running in the [dag] pipeline"*
- *Print the following messages with the python operator:*
  - *"This script was executed at [date]"*
  - *"Three days after execution is [date]"*
  - *"This script run date is [date]"*

Sensors

Sensors

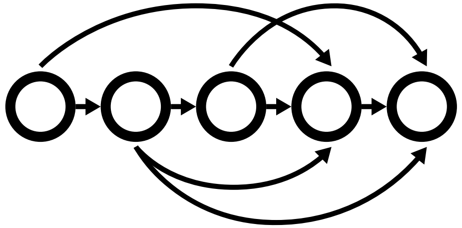
- Poll for a certain condition to be True

External data uploaded somewhere between 13:00 and 17:00



FTP

Airflow sensor
13:01 check data? No
13:02 check data? No
13:03 check data? No
13:04 check data? No
13:05 check data? No
13:06 check data? Yes



```python
from airflow.contrib.sensors.ftp_sensor import FTPSensor

wait_for_data = FTPSensor(
    task_id="wait_for_data",
    path="foobar.json",
    ftp_conn_id="bob_ftp",
)
```

Sensors

Implement your own condition PythonSensor

```python
from datetime import datetime

from airflow.sensors.python import PythonSensor


def _time_for_coffee():
    """I drink coffee between 6 and 12"""
    if 6 < datetime.now().hour < 12:
        return True
    else:
        return False


time_for_coffee = PythonSensor(
    task_id="time_for_coffee",
    python_callable=_time_for_coffee,
    mode="reschedule",
)
```

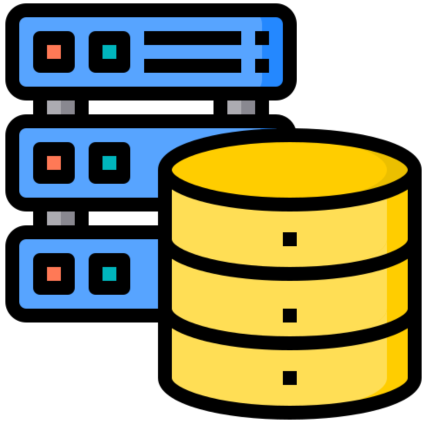
Sensors
The sensor deadlock

- Always set `mode="reschedule"`
- Airflow queues tasks in slots
- The (default!) "poke" mode holds the slot while waiting for the next poke
- With (default) parallelism 32, and 32 sensors, your system has no more free slots to do actual work
- Therefore set mode="reschedule". This releases the slot and claims a new one for every Sensor poke.

Airflow

Metastore

XComs, Variables &
Connections



Storing data in the metastore

- Most tables are internal to Airflow
- Users can store information in:
  - Connections
  - Variables
  - XComs

- Tables
  - alembic_version
  - chart
  - connection
  - dag
  - dag_pickle
  - dag_run
  - dag_tag
  - import_error
  - job
  - known_event
  - known_event_type
  - kube_resource_version
  - kube_worker_uuid
  - log
  - serialized_dag
  - sla_miss
  - slot_pool
  - task_fail
  - task_instance
  - task_reschedule
  - users
  - variable
  - xcom

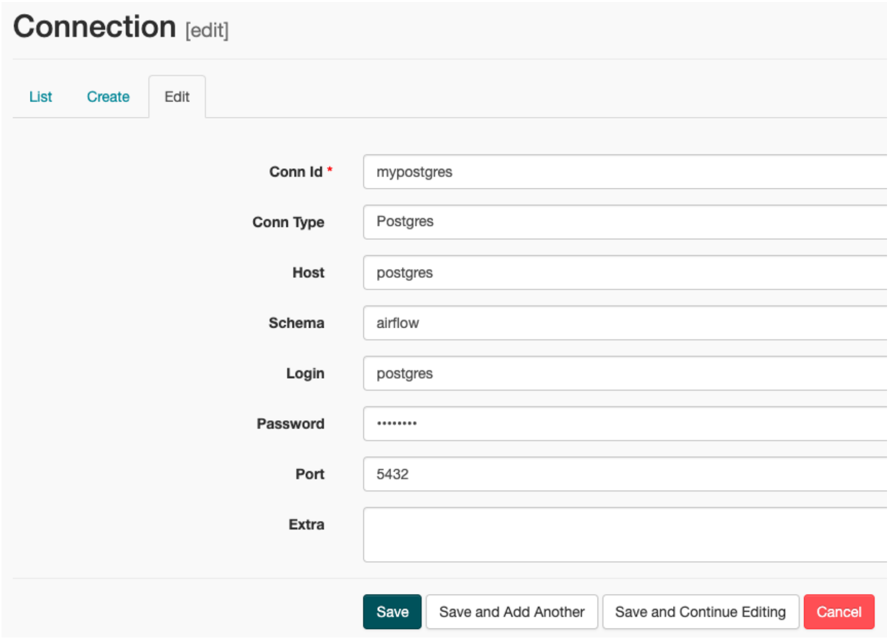
IF you want to sneak around:

```
docker ps
docker exec -it <db-postgres name> /bin/bash
psql -Uairflow
SELECT * FROM information_schema.tables;
SELECT * FROM xcom;
```

Xebia | Data

Connections

- Connection credentials can be stored in the Airflow database
- Configures a Fernet key for encryption in the configuration
  - https://airflow.apache.org/docs/stable/howto/secure-connections.html

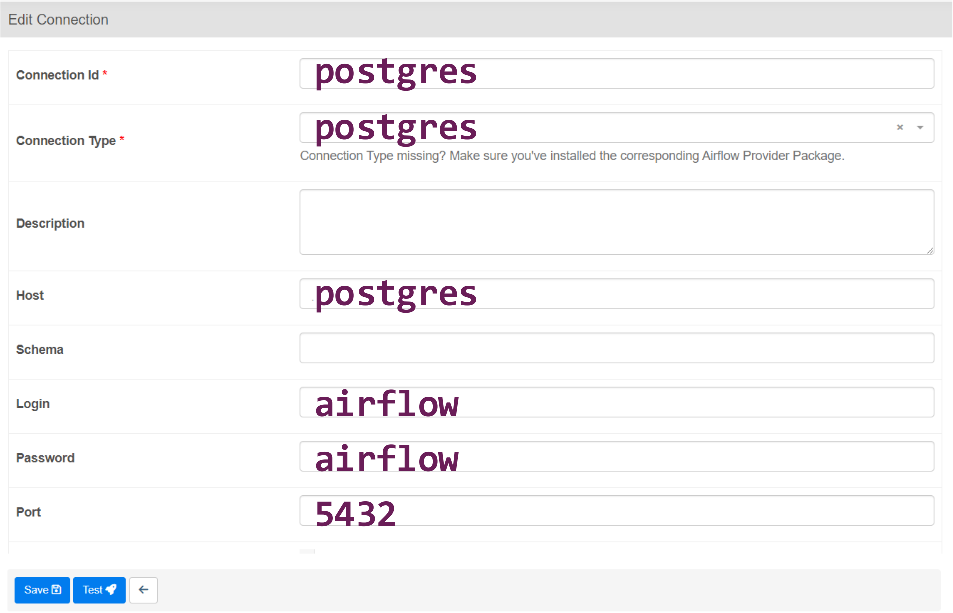


Xebia | Data

Let’s Define two connections

Will be useful for later

Airflow Postgres DB



This connection will allow us to connect to the Airflow Internal Postgres DB

The Space Devs API

Will allow us to make calls to the Rocket launches API

Hooks

- Used to interact with (external) systems/service
  - Usually created behind the scenes by the respective operators
  - Abstract away logic of interacting with systems
  - Handles authentication, caching, pagination, etc.

- Some examples
  - **S3Hook** – Upload/download files to/from S3
  - **SSHHook** – Upload/download files over SFTP
  - **SparkSubmitHook** – Send jobs to Spark cluster

Hooks - Example

```
from airflow.hooks.postgres_hook

hook = PostgresHook(
    postgres_conn_id="land_registry”
)

hook.get_records(
    "SELECT * FROM land_registry_price_paid_uk "
    "LIMIT 10”
)
```

Variables

- Can be used as “global” variables

- Stored as key-value pairs

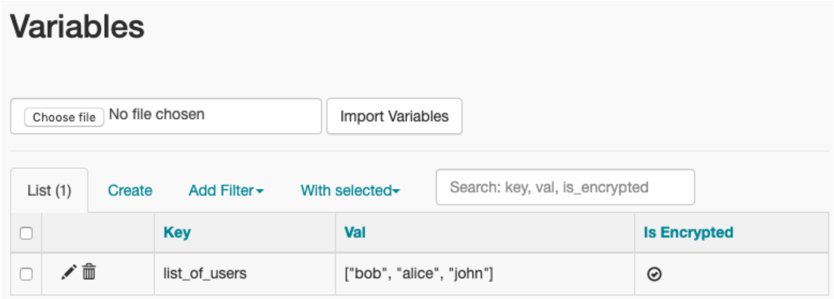
```
from airflow.models import Variable
Variable.get("myvar", deserialize_json=True, default_var=dict())
Variable.set("myvar", value, serialize_json=True)
```

- Also accessible in templating:

```
"{{ var.json.myvar }}"
"{{ var.value.myvar }}"
```

Variables - Example

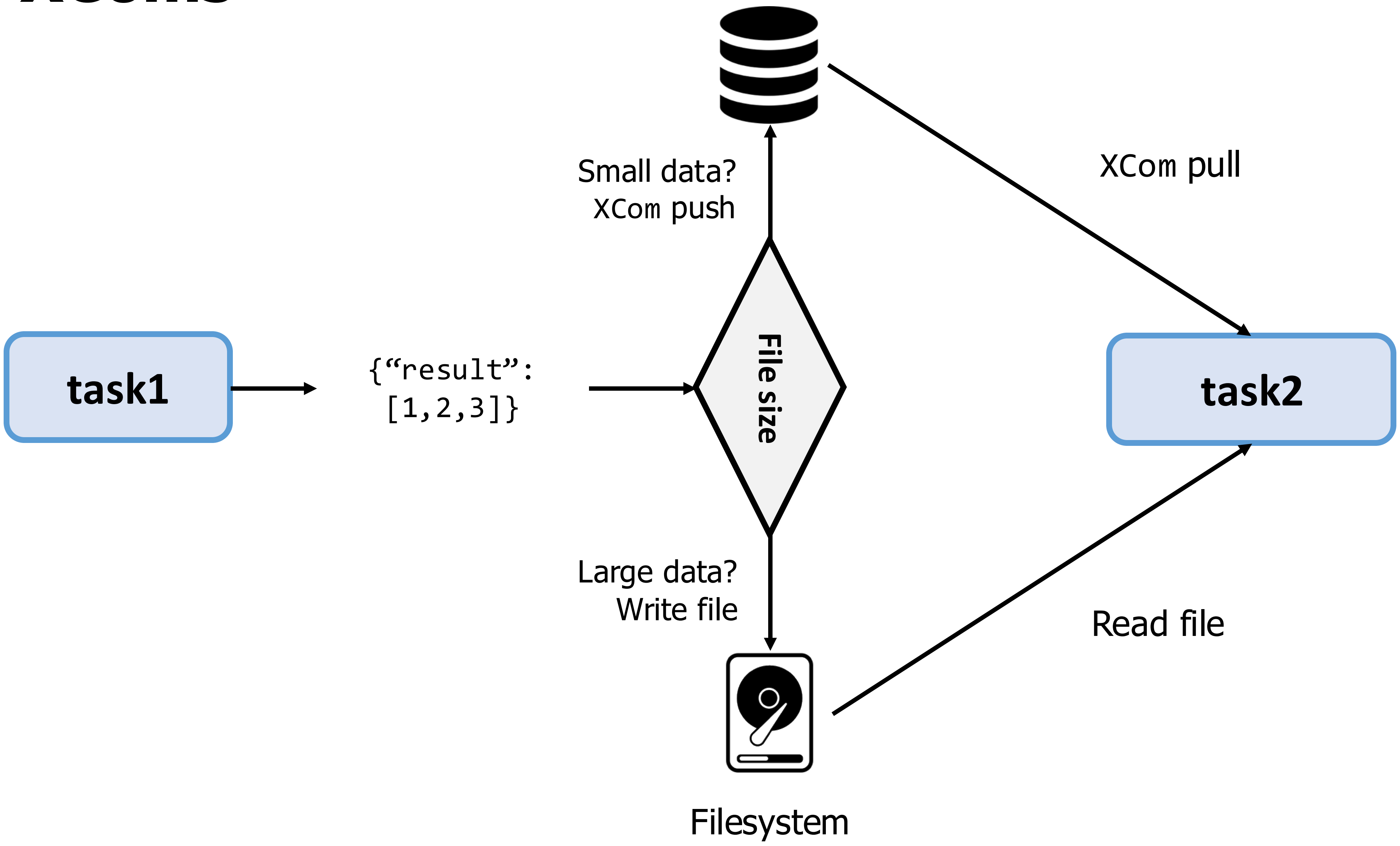
```python
import airflow.utils.dates
from airflow.models import DAG, Variable
from airflow.operators.python_operator import PythonOperator

dag = DAG(
    dag_id="example_variable",
    start_date=airflow.utils.dates.days_ago(3),
    schedule=None,
)


def _email_users():
    users = Variable.get("list_of_users", deserialize_json=True)
    for user in users:
        ...


email_users = PythonOperator(task_id="email_users",
python_callable=_email_users, dag=dag)
```



XComs (aka Cross-Communication)

- Useful when you want to share data between tasks
- XComs are meant for sharing (small) pieces of information
  - No max size enforced right now
  - Possibly coming in Airflow 2.1
  - SQLite: BLOB type (max 2GB)
  - PostgreSQL: BYTEA type (max 1GB)
  - MySQL: BLOB type (max 64KB)
- XComs for task-to-task communication, Variables for “global”
- Note: does not work between instances of the same task!
  (e.g. when using reschedulable sensors)

XComs



Small data?
XCom push
XCom pull
task1
{"result":
[1,2,3]}
File size
task2
Large data?
Write file
Read file
Filesystem

Xcoms - Example

```
import random

import airflow.utils.dates
from airflow.models import DAG
from airflow.operators.python import PythonOperator

def _push(task_instance, **_):
    teammembers = ["Bob", "John", "Alice"]
    result = random.choice(teammembers)
    task_instance.xcom_push(key="person_to_email", value=result)
    return result


def _pull(task_instance, **_):
    result_by_key = task_instance.xcom_pull(task_ids="push", key="person_to_email")
    result_by_return_value = task_instance.xcom_pull(task_ids="push")
    print(f"Email {result_by_return_value}")
    print(f"Email {result_by_key}")


with DAG(dag_id="example_xcom", start_date=airflow.utils.dates.days_ago(3), schedule="@daily"):

     push = PythonOperator(task_id="push", python_callable=_push, provide_context=True)
     pull = PythonOperator(task_id="pull", python_callable=_pull, provide_context=True)
     push >> pull
```

XCom via explicit push

XCom via return value

Xebia | Data

Xcoms – Default Behavior

- The BaseOperator holds an argument do_xcom_push (default True)
- If do_xcom_push and a value is returned, it is automatically pushed into the XCom table
- If no key is supplied during push/pull, a default key with value “return_key” is set

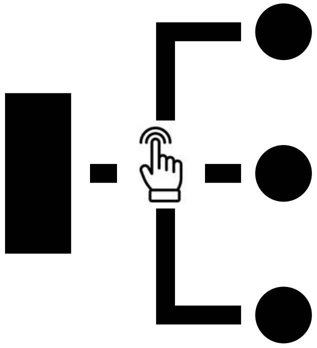
Capstone Project!

We start launching rockets

As a Rocket Scientist your manager asked you to have a good overview of the days a rocket launch has been made. And get insights about the launches.

You can get the full instructions in the Capstone Project file in the sharepoint folder.

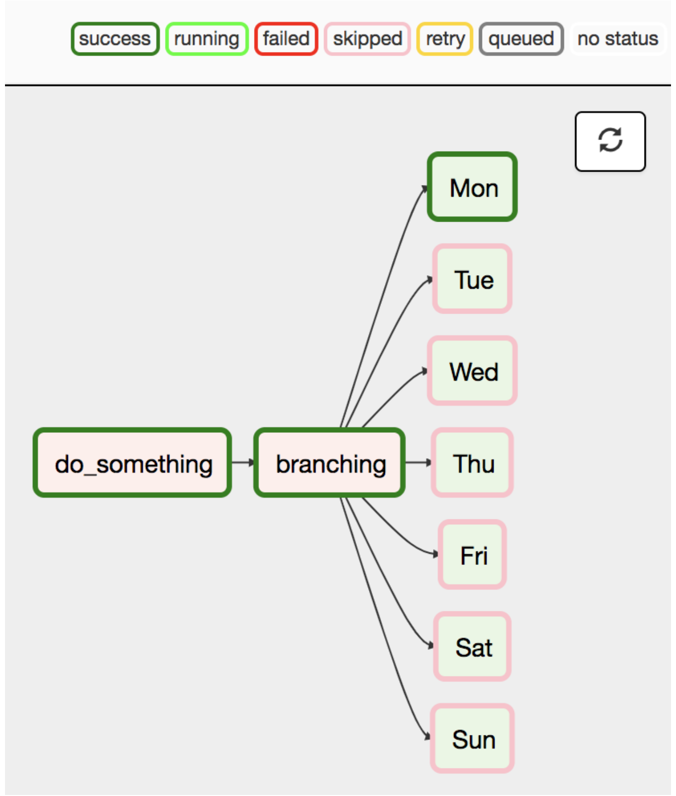
Branching & Trigger rules



PythonOperator vs all other operators

- Sometimes you'd like to execute some tasks based on a specific condition

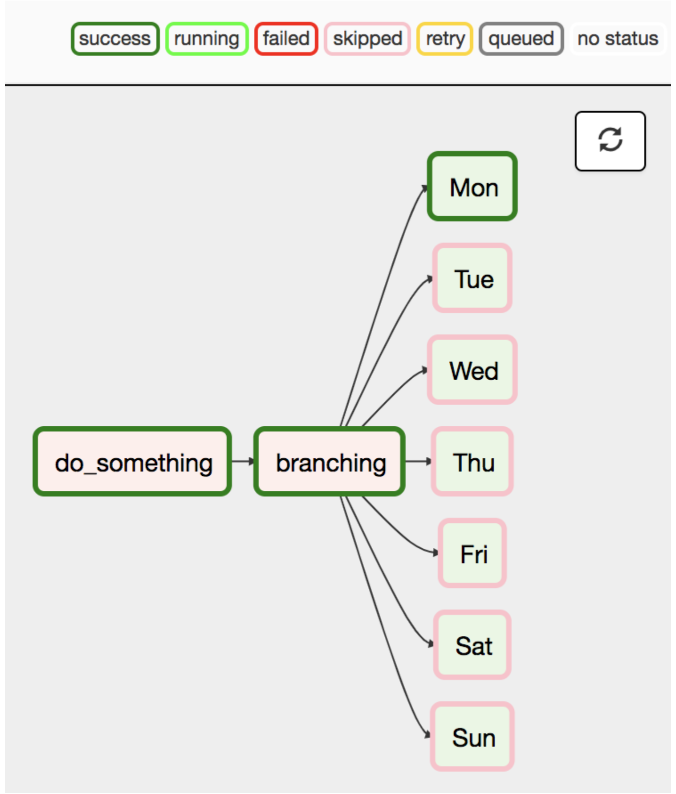
- Examples:
  - Running tasks on certain days of the week
  - Running a different set of tasks after a specific date (e.g. to account for a schema change)



The PythonBranchOperator

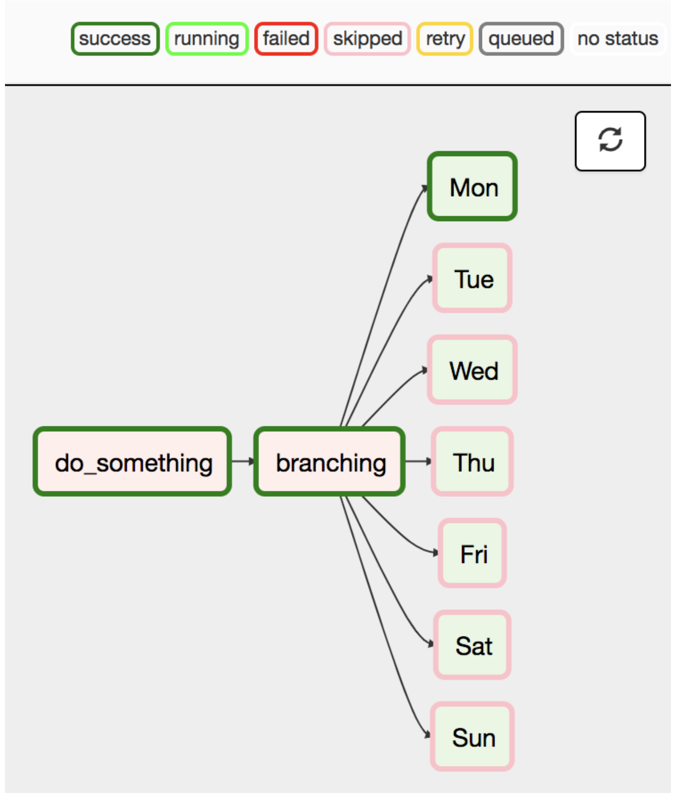
- Branches on a certain condition
  - Accepts a callable, which when called should return the name of downstream task(s) to run
  - Other tasks are automatically skipped

- Note that tasks should be a (direct) downstream dependency of the branch task



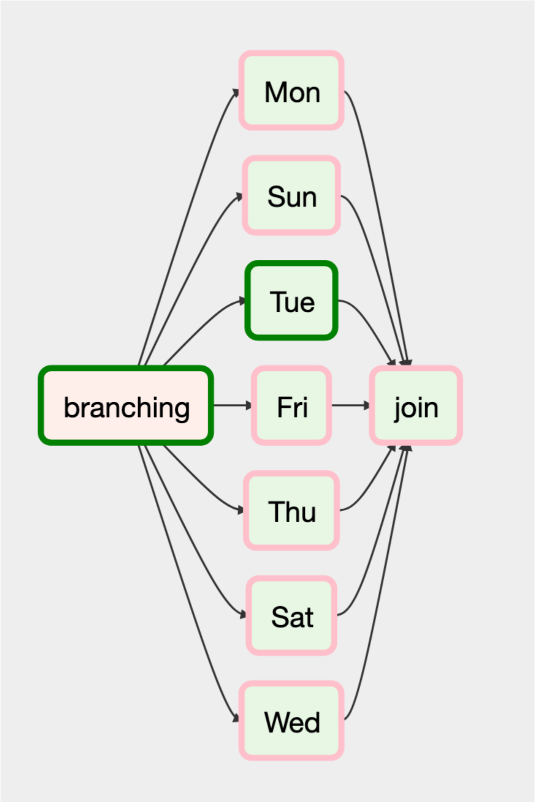
The PythonBranchOperator

```
def _get_weekday(execution_date, **context):
    return execution_date.strftime("%a")  # “Mon”


branching = BranchPythonOperator(
    task_id="branching",
    python_callable=_get_weekday,
)

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
for day in days:
    branching >> DummyOperator(task_id=day)
```



Continuing after a branch

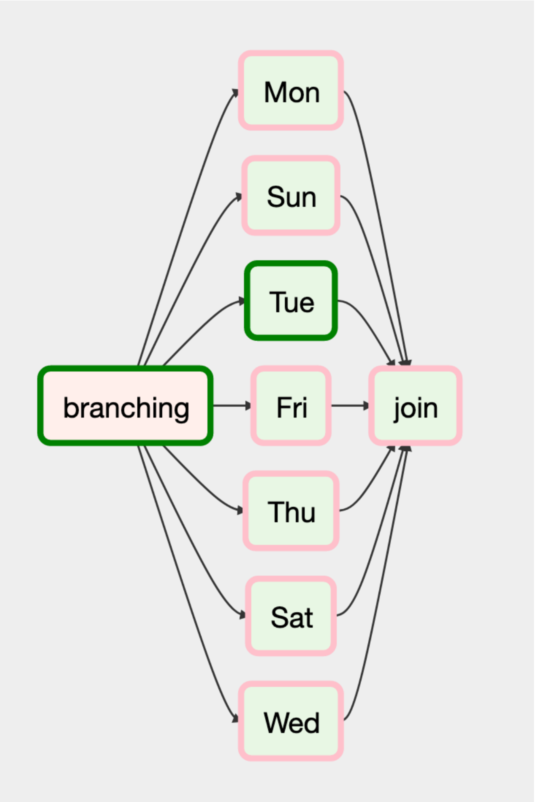
- Typically, execution after a branch is continued after a (dummy) join task
- This allows downstream tasks to be ‘unaware’ of the branch
- However, naively adding a join does not work. Any idea why?



Trigger rules

- Trigger rules determine when a task is executed
  - Defined using the trigger_rule Operator argument
  - Default rule is all_success (all upstream tasks must have completed successfully)

`TriggerRule.ALL_SUCCESS`	(Default) all parents have succeeded.
`TriggerRule.ALL_FAILED`	All parents are in a failed or upstream_failed state.
`TriggerRule.ALL_DONE`	All parents are done with their execution.
`TriggerRule.ONE_FAILED`	Fires as soon as at least one parent has failed, it does not wait for all parents to be done.
`TriggerRule.ONE_SUCCESS`	Fires as soon as at least one parent succeeds, it does not wait for all parents to be done.
`TriggerRule.DUMMY`	Dependencies are just for show, trigger at will.

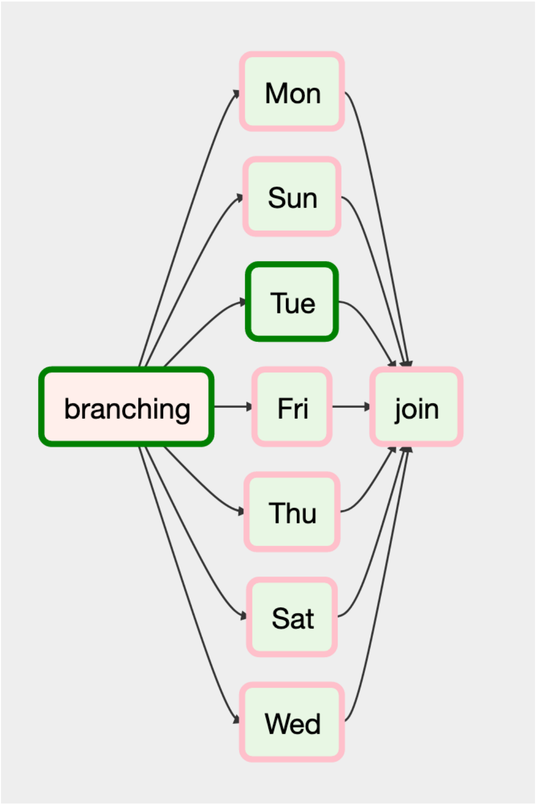


Fixing the join

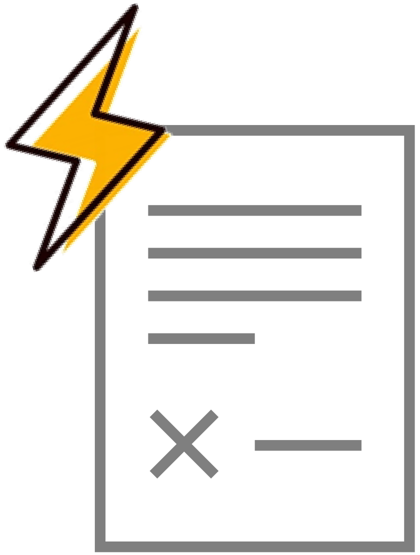
- Changing the trigger rule is enough:

```
join = EmptyOperator(
    task_id="join",
    trigger_rule="none_failed"
)
```

- Ensures join will run if none of the upstream tasks fail



Backfilling



Backfilling

- Backfilling is the concept of (re-)running Airflow tasks back in time.

- For example:
  - You created a daily job and want to run it 1 year back.
  - A task failed due to a missing file. You placed the file manually and now you want to re-run the task.
  - You changed code in a DAG and want to re-run specific tasks with the changed code.

Backfilling

- As mentioned before: Airflow doesn’t like to re-run task before the `start_date`.

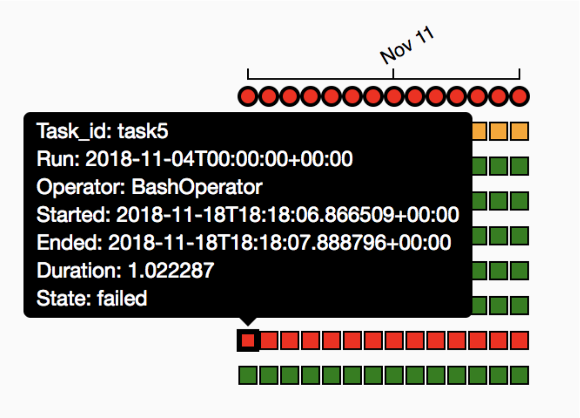
- Backfilling is possible via:
  1. Run **`docker compose ps`** and find the name of the scheduler
  2. Get into the scheduler via:

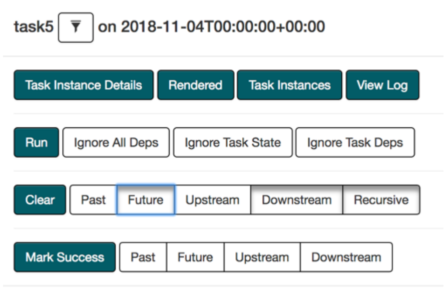
     ```
     docker exec -it <scheduler name> /bin/bash
     ```

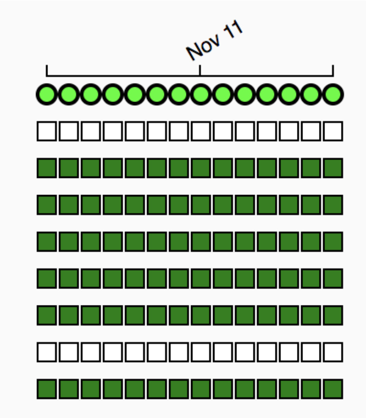
  3. Run:

     ```
     airflow dags backfill -s 2023-04-01 -e 2023-04-30 <dag_name>
     ```

Clear Tasks







- You can clear one or many task depending on selection criteria.
  - Only the ones that failed
  - Task in the Future
  - Upstream Downstream
  - All
- Airflow will rerun those tasks

Designing tasks for backfilling

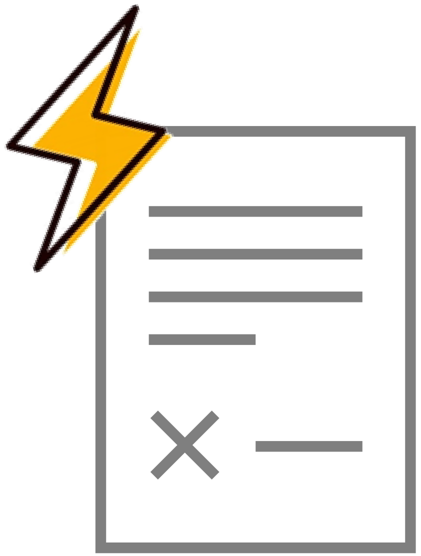
- Tasks should be atomic and idempotent

- Atomic
  - Tasks either succeed fully or not at all (no partial result)

- Idempotent
  - Re-running a task gives the same result
  - (Assuming other circumstances have not changed)

Dataset

Data-aware DAG triggers



Some DAG's are really looooooooong

Can we do something better?

- Maybe we can break them in smaller chunks

But then how do you trigger one after another?

Dag 1

Dag 2

Dag 3

Dag 4

How do make your DAGs data-aware?

Let's focus on a use case.

Team A: Fetch for External sources

Team B: Produces a Report

Meet the Dataset…

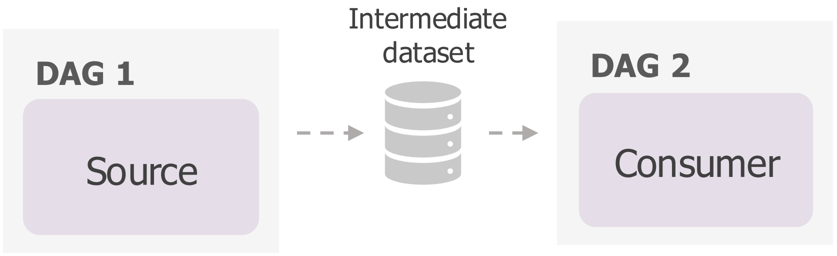
source.py

```python
from airflow import DAG, Dataset
from airflow.operators.empty import EmptyOperator

from datetime import timedelta
import pendulum

intermediate_dataset = Dataset(
    "s3://my-bucket/intermediate_dataset.csv"  URI
    )

dag = DAG(
    dag_id="dataset_etl_pipeline",
    start_date=pendulum.today("UTC").add(days=-10),
    schedule=timedelta(seconds=10), # every 5 minutes
    catchup=False,
)

fetch = EmptyOperator(task_id="fetch", dag=dag)
remove_outliers = EmptyOperator(task_id="remove_outliers", dag=dag)
update_db = EmptyOperator(
    task_id="update_db",
    dag=dag,
    outlets=[intermediate_dataset]
    )

fetch >> remove_outliers >> update_db
```

Consumer.py

```python
from airflow import DAG, Dataset
from airflow.operators.empty import EmptyOperator

import pendulum

intermediate_dataset = Dataset(
    "s3://my-bucket/intermediate_dataset.csv"
    )

dag = DAG(
    dag_id="dataset_produce_report",
    start_date=pendulum.today("UTC").add(days=-10),
    schedule=[intermediate_dataset],
    catchup=False,
)

get_cleaned_data = EmptyOperator(task_id="get_cleaned_data", dag=dag)
produce_report = EmptyOperator(task_id="produce_report", dag=dag)

get_cleaned_data >> produce_report
```

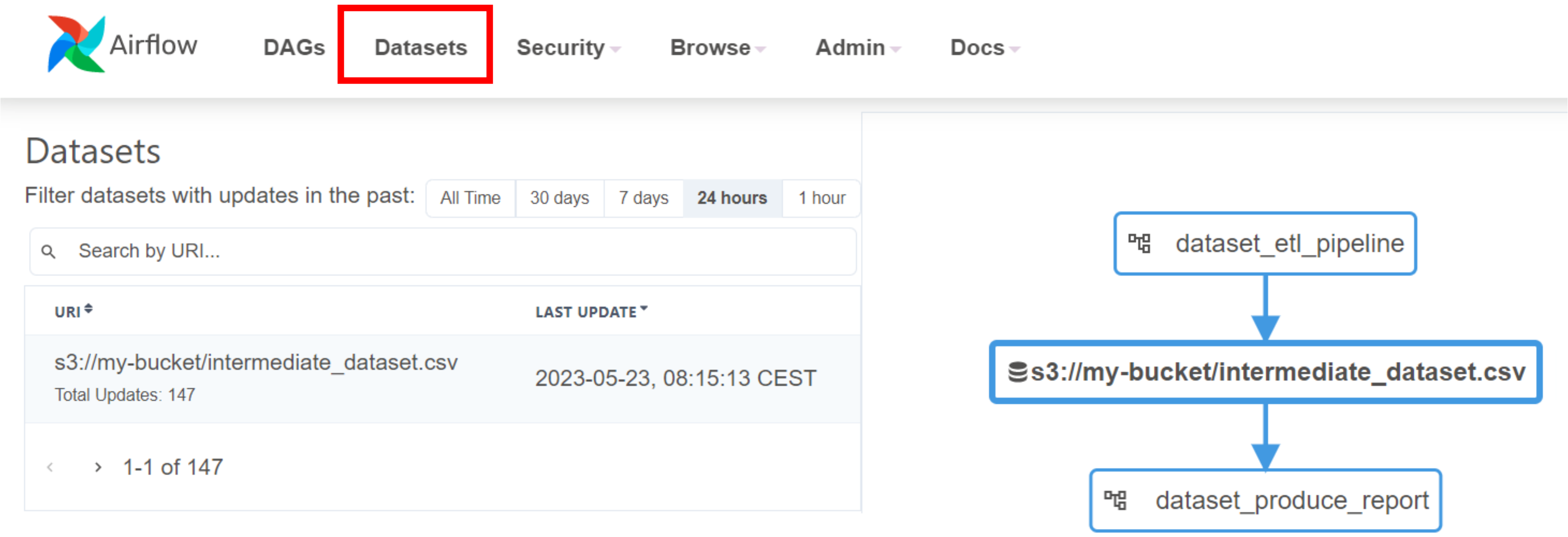
You can list multiple files. Will be triggered when all files are updated



New feature in Airflow ≥ 2.4

schedule replaces schedule_interval providing more flexibility on DAG trigger

A Dataset change becomes a trigger!



Dataset quirks…

- Airflow doesn't verify the data has been changed. It only verifies if the source DAG has executed correctly.
- Dataset URI acts as a Link / tag to create triggers.
- If two tasks update the same dataset, the Consumer DAG triggers after the first task completes.
- Schedules cannot be combined (datasets and cron expressions).
- Airflow only monitors datasets within DAGs, not external modifications, or DAGs on External Instances.

But wait a minute, that DAG didnt modify any file

Exercise: The Dataset

- Modify two of your previous DAG's to be a consumer and a producer.
- The producer DAG should trigger every 10 seconds
- The consumer DAG should trigger each time the producer DAG finish a run.
- Tip: Don't worry too much about actually storing/changing any data

Let us know what you think!

https://xebia-academy.my.salesforce-sites.com/apex/selectsurvey