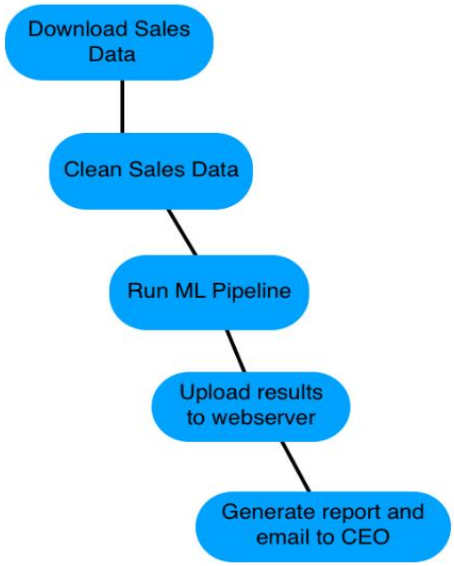
AIRFLOW SHEET – INTRODUCTION

Workflow example



DAG definition

DAGS (directed acyclic graph)	A set of tasks and dependencies that make up a workflow, written in Python.

Simple DAG examples

```
form airflow import DAG

etl_dag = DAG(  dag_id='etl_pipeline',
   default_arg={"start_date": "2024-01-08"}
```

```
from airflow import DAG
from datetime import datetime

arguments = {
   "owner": "username",    "email":""example.gmail.com",
   "start_date": datetime(2024,1,20)}

with DAG('dag_name', default_arg= arguments) as variable:

 ....<operators>
```

Test an Airflow task

How	Example
Shell command	airflow tasks test <DAG_id> <task_id> [execution_date]

Command line Vs Python

Command line	Python
Start Airflow process	Create a DAG
Manually run DAGs/ Tasks	Edit individual properties of a DAG
Get logging information from Airflow	

Airflow web UI

Page groups	Info
DAGs	shows the DAGs that are available, owner, last few runs, schedule, when last run happened, when next run is planned, which tasks have ran.
Browse/ Audit logs	shows history of events and commands.
Brows/DAG run	shows details about that DAGs that have run.
Browse/SLA Misses	log of all missed SLAs.

DAG detail page

Shows the task duration, task tries, timings, etc. and lets you start/delete the DAG

Sub page	Description
graph	shows live view of the DAG and its tasks in a flow.
code	shows the Python code of the DAG (read only).

Open Airflow UI

```
airflow webserver -p <port> # common default 8080
```

Web UI Vs Command line

Equally powerful depending on needs.

Web UI	Command line
Easier to use.	Easier to access.

AIRFLOW SHEET – INTRODUCTION

Arguments

Groups:		Commands:	
Argument	**Description**	**Argument**	**Description**
config	View configuration.	cheat-sheet	Display cheat sheet.
connections	Manage connections.	dag-processor	Start a standalone DAG processor instance.
dags	Manage DAGs.	info	Show information about current Airflow and environment.
db	Database operations.	kerberos	Start a kerberos ticket renewer.
jobs	Manage jobs.	plugins	Dump information about loaded plugins.
pools	Manage pools.	rotate-fernet-key	Rotate encrypted connection credentials and variables.
providers	Display providers.	scheduler	Start a scheduler instance.
roles	Manage roles.	standalone	Run an all-in-one copy of airflow.
tasks	Manage tasks.	sync-perm	Update permissions for existing roles and optionally DAGs.
users	Manage users.	triggerer	Start a triggerer instance.
variables	Manage variables.	version	Show the version.
		webserver (-p <port>)	Start an airflow webserver instance.

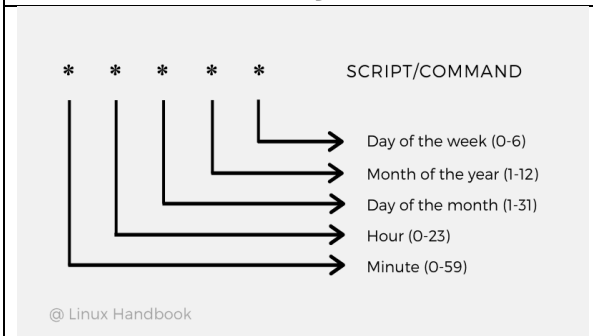
Optional arguments:

Argument	Description
-h, --help	Show help message

Schedule DAGs

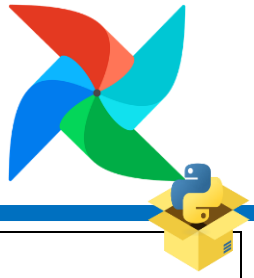
Attributes	Description
start_date	The date/time to initially schedule the DAG run.
end_date (optional)	When to stop running new DAG instances.
max_tries / retries (optional)	How many times to retry before failing the DAG run.
retry_delay (optional)	The delay between tries.
schedule_interval	How often to schedule the DAG (using the Unix cron format or using presets @hourly, @daily etc. 'None' meaning don't ever and @once meaning only once).

Cron syntax



Example	
0 12 * * *	Run daily at noon.
* * 25 2 *	Run once per minute on February 25.
0,15,30,45 * * * *	Run every 15 minutes.

AIRFLOW SHEET – INTRODUCTION



Operators 1 Basic

Operators represent a single task in Airflow (e.g. run a command/ python script).
Generally, they do not share information between each other (but it is possible)
Airflow can contain many various operators to perform different tasks.
one DAG can have multiple operators whose execution can be ordered using upstream/downstream operators.

Operator types	Description	Operator	Example
BashOperator	Executes given Bash command or script, when task is called.	BashOperator	*from airflow.operators.bash import BashOperator* ....<dag definition>: example_task1 = *BashOperator*(task_id="bash_example", bash_command= 'echo "example" ', dag = <dag> #Required when using airflow v1. *)*
			from airflow.operators.bash import BashOperator ....< with dag definition>: example_task2= *BashOperator*(task_id="bash_script_example". bash_command= "runcleanup.sh")
PythonOperator	Executes a python function when task is called.	PythonOperator	*from airflow.operator.python import PythonOperator* ....<with dag definition>: example_task3 = *PythonOperator*(task_id="python_example", python_callable= <function alias>, op_kwargs={<variable>: <value>})
Email Operator	Sends an email when task is called.	EmailOperator	from airflow.operators.email import EmailOperator ...<with dag definition>: example_task4= EmailOperator(task_id="email example", to= "example@example.com" subject="example subject" html_content="example text" files= "example_attachement.xlsx"
Upstream	Task1 should run before task2.	>>	example_task1>> example_task2
			# Can have multiple in a row. example_task1 >> example_task2 >> example_task3
Downstream	Task1 should run after task2.	<<	exampletask1 << example_task2
			# Can mix upstream and downstream too. example_task1<< example_task2 >> example_task3

AIRFLOW SHEET – INTRODUCTION

Operators 2 Sensors

A sensor waits for a certain condition to be true.

Additional default arguments

Argument	Description
mode	How to check for the condition mode = "poke" = the default, indicates to run repeatedly till true) mode = "reschedule"= indicates to give up task slot and try again later.
poke_interval	Is used in "poke" mode and indicates how long to wait before checking again in seconds (min 1 minute).
timeout	Is how long to wait before returning false (make sure its shorter than poke_interval).

Operator types	Description	Operator	Example
File sensor	Checks for existence of a file at a certain location.	FileSensor	*from airflow.sensors.filesystem import FileSensor* ...<with dag definition>: example_task5 = *FileSensor*(task_id= "file check", filepath= "examplepath.csv", poke_interval=300,)
ExternalTaskSensor	Waits for a task in another DAG to complete.	ExternalTaskSensor	*from airflow.sensors.external_task_sensor import ExternalTaskSensor* ...<with dag definition>: example_task6= *ExternalTaskSensor*(task_id="wait for example task", external_dag_id= "external_dag_example", external_task_id = "external_task_id_example", allowed_states=["success"], poke_interval= 300,)
HttpSensor	Requests a web URL and check for content.	HttpSensor	*from airflow.providers.http.sensors.http import HttpSensor* ...<with dag definition>: example_task7 = *HttpSensor*(task_id="http sensor task " http_con_idd="http_default" endpoint="https:webadres.nl" method="GET" response_check=lambda response: response.json()['status'] == 'success', poke_interval=300,)
SqlSensor	Run a SQL query to check for content.	SqlSensor	*from airflow.providers.sqlite.sensors.sqlite import SqlSensor* ...<with dag definition>: example_task8= *SqlSensor*(task_id='sql_check', sql=sql, conn_id=conn_id, poke_interval=300,)

AIRFLOW SHEET – INTRODUCTION

Executors

Executor	Description
SequentialExecutor	*Default executor. *Runs a single task at a time. *Useful for debugging.
LocalExecutor	*Runs on a single system. *Treats tasks as processes and thus can run multiple tasks parallel (defined by user as unlimited (all available resources will be used) or a set amount of simultaneous tasks).
KubernetesExecutor	*Can run on multiple systems at same time. *More difficult to setup (requires a Kubernetes configuration).

Defining Airflow SLA

SLA is the amount of time a task or DAG should require to run. SLA Miss is a task that didn't fulfil this expectation when that happens Airflow will log this (and send an email if configured).

Description	Example
As an argument on the task.	from airflow.operators.bash import BashOperator from datetime import timedelta ....< with dag definition>: example_task9= *BashOperator*(task_id="bash_script_example". bash_command= "examp.sh", *sla*=timedelta(seconds=30))
As an argument on the DAG.	from airflow import DAG from datetime import timedelta default_arguments = { "owner": "name", "*sla*": timedelta(minutes=20)} with *DAG*('dag_name', default_arg= default_arguments) as variable: ....<operators>

Determine used executor

Via	Command
cmd airflow.cfg	cat airflow/airflow.cfg | grep "executor = "
cmd	airflow info # then look for executor |

Debugging common bugs

Bug	Solution
Scheduler appears not to be running.	airflow scheduler # Turns on the scheduler.
DAG does not show up in dags list.	cat airflow/airflow.cfg | grep "dags_folder =" # The python file containing the dags has to be in this indicated path.
Syntax errors.	airflow dags list-import-errors # Outputs some debugging information.

Datetime library

datetime.datetime()

Takes the argument [days, seconds, minutes, hours, weeks] and represents a specific point in time.

Example

datetime(2024,6,5,15,30) #2024-06-05 15:30:00
datetime.now() # 2024-06-05 08:45:50.364082

datetime.timedelta()

Takes the argument [days, seconds, minutes, hours, weeks] and represents the duration of something,

example

from datetime import timedelta

timedelta(days=4, hours=10) # 4 day, 10:00:00

AIRFLOW SHEET – INTRODUCTION

Templates

Templates allow substituting information during the execution of a DAG run.
They are created using the Jinja language.

Require	Description
Templated command/ function.	A nested query, {{ variable}} being used for data that needs to be substituted, which you provide in a dictionary in the params variable of the operator.

Example (templated BashOperator)

```
from airflow.operators.bash import BashOperator

templated_command """
echo "Reading {{params.filename}}"
"""

example_task10=
BashOperator(task="template_task",
  bash_command= templated_command,
  params={"filename": "file1.txt"}
```

Templates 2 for loop

The for loop makes it possible to run a list of e.g. files within a single operator task.

Additional require	Description
Opening of the for loop.	Done similar as in python but than in between {% <for a in b> %}.
Closing the for loop.	Unlike python the loop must be closed with {% endfor%}.

Example (templated BashOperator)

```
from airflow.operators.bash import BashOperator

templated_command """
{% for filename in params.filenames %}
   echo "Reading {{filename}}"
{% endfor }
"""

example_task11=
BashOperator(task="for_loop_task",
  bash_command= templated_command,
  params={"filename": ["file1.txt", file2.txt]}
```

Preset variables

Base		Macros	
Variable	**Description**	**Variable**	**Description**
{{ds}}	Returns a string containing the current date. #YYYY-MM-DD	{{macros.datetime}}	Equivalent to Python datetime.
{{ds_nodash}}	Returns a string containing the current date #YYYYMMDD.	{{macros.timedelta}}	Equivalent to Python timedelta.
{{prev_ds}}	Returns a string containing the previous execution date. #YYYY-MM-DD	{{macros.uuid}}	Equivalent to Python uuid (creating unique identifiers).
{{prev_ds_nodash}}	Returns a string containing the previous execution date. #YYYYMMDD	{{macros.ds_add (<date>, <int>}}	Increases the date by the int number of days.
{{dag}}	Access the DAG object within the code.		
{{conf}}	Access the Airflow configuration with the code.		

AIRFLOW SHEET – INTRODUCTION

Check availability of templated values

Not all variables of the operators support template values

How to check:	Example
1:Import libraries.	from airflow.operators.bash import BashOperator help(BashOperator)
2: Help(<airflow object>.	
3: Look for template_fields, see here which variables support templated values.	Data and other attributes defined here: template_ext = ('.sh', '.bash') template_fields = ('bash_command', 'env') ui_color = '#f0ede4'

Operators 3 Branching

The branch operator branches(splits) the tasks within a DAG following a set condition within the branch operator. It is still mandatory too set the order of the operators using upstream/downstream in order for the branch operator to have any effect if not defined in the stream the task will always run.

Operator	Example

BranchPythonOperator	*from airflow.operator.python import BranchPythonOperator*
	
	def branch_example(**kwargs):
	  if int(kwargs["ds_nodash"]) % 2==0:
	  return "even_day_task" #task_id of the first even day task
	  else:
	  return "odd_day_task" #task_id of the first odd day task
	
	example_task11 = *BranchPythonOperator*(
	  task_id = "branch task",
	  provide_context=True, #provides the python function access too macros and runtime variables
	  python_callable=branch_example)
	
	example_task >> branch_task>> even_day_task>> even_day_task2
	branch_task>> odd_day_task >> odd_day_task2

Result