

|  |
| --- |
| **Workflow example** |
|  |



|  |
| --- |
| **Open Airflow UI** |
| airflow webserver -p <port> # common default 8080 |

|  |  |
| --- | --- |
| **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> | |

|  |  |
| --- | --- |
| **DAG detail page** | |
| Shows the task duration, task tries, timings, etc. and lets you start/delete the DAG. | |
| **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). |

|  |  |
| --- | --- |
| **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 run. |
| Browse/ Audit logs | Shows history of events and commands. |
| Brows/DAG run | Shows details about DAGs that have run. |
| Browse/SLA Misses | log of all missed SLAs. |

|  |  |
| --- | --- |
| **Web UI Vs Command line** | |
| Equally powerful depending on needs. | |
| **Web UI** | **Command line** |
| Easier to use. | Easier to access. |

|  |  |
| --- | --- |
| **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. |  |

|  |  |
| --- | --- |
| **Test an Airflow task** | |
| **How** | **Example** |
| Shell command | Airflow tasks test <DAG\_id> <task\_id> [execution\_date] |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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** | |
| Crontab Explained in Linux [With Examples] | |
| **Example** | |
| 0 12 \* \* \* | Run daily at noon. |
| \* \* 25 2 \* | Run once per minute on February 25. |
| 0,15,30,45 \* \* \* \* | Run every 15 minutes. |



|  |  |  |  |
| --- | --- | --- | --- |
| **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** |
| Bash  Operator | 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" ) |
| Python  Operator | Executes a python function when task is called. | PythonOperator | *from airflow.operator.python import PythonOperator*  ….<with dag definition>:  example\_task3 = *PythonOperato*r(  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 |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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). | | |
| **types** | **Description** | | **Operator** | **Example** |
| File sensor | Checks for existence of a file at a certain location. | | File  Sensor | *from airflow.sensors.filesystem import FileSensor*  …<with dag definition>:  example\_task5 = *FileSensor*(  task\_id= "file check",  filepath= "examplepath.csv",  poke\_interval=300,) |
| ExternalTask  Sensor | Waits for a task in another DAG to complete. | | ExternalTask  Sensor | *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,) |
| Http  Sensor | Requests a web URL and check for content. | | Http  Sensor | *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,) |
| Sql  Sensor | Run a SQL query to check for content. | | Sql  Sensor | *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,) |



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

|  |  |
| --- | --- |
| **Executors** | |
| **Executor** | **Description** |
| Sequential  Executor | \*Default executor.  \*Runs a single task at a time.  \*Useful for debugging. |
| Local  Executor | \*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). |
| Kubernetes  Executor | \*Can run on multiple systems at same time.  \*More difficult to setup (requires a Kubernetes configuration). |



|  |
| --- |
| **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 |

|  |  |
| --- | --- |
| **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 (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> |

|  |  |
| --- | --- |
| **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 debugging information. |

A grey sticker of a gate

Description automatically generatedA grey sticker of a gate

Description automatically generatedA grey sticker of a gate

Description automatically generated

|  |  |
| --- | --- |
| **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. | |
| **Required** | **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. |  |  |



|  |  |
| --- | --- |
| **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 for the branch operator to have any effect if not defined in the stream the task will always run. | |
| **Operator** | **Example** |
| Branc  hPython  Operator | *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** | |
|  | |

|  |  |
| --- | --- |
| **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. |  |