



# Apache Airflow

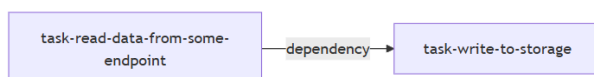
## Introduction

**Airflow** is a batch-oriented framework for creating data pipelines.

It uses **DAG** to create data processing networks or pipelines.

- DAG stands for -> Direct Acyclic Graph. It flows in one direction. You can't come back to the same point, i.e. acyclic.
- In many data processing environments, a series of computations are run on the data to prepare it for one or more ultimate destinations. This type of data processing flow is often referred to as a data pipeline. A DAG or data processing flow can have multiple paths, also called branching.

The simplest DAG could be like this.



where

- **read-data-from-some-endpoint** & **write-to-storage** - represent a task (unit of work)
- Arrow --> represents processing direction and dependencies to check on what basis the next action will be triggered.

## Ok, so why should we use Airflow?

- If you like **Everything As Code** and **everything** means everything, including your configurations. This helps to create any complex level pipeline to solve the problem.

- If you like open source because almost everything you can get as an inbuilt operator or executors.
- **Backfilling** features. It enables you to reprocess historical data.

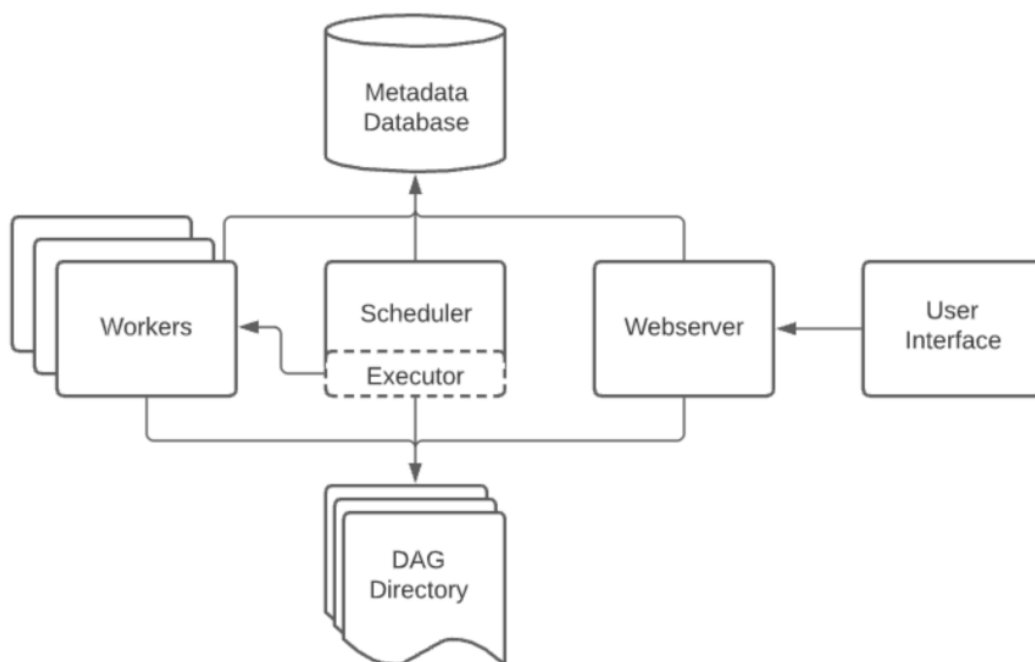
### And, why shouldn't you use Airflow?

- If you want to build a streaming data pipeline.

---

## Airflow Architecture

So, we have at least an idea that Airflow is created to build the data pipelines. Below we can see the different components of Airflow and their internal connections.



We can see the above components when we install Airflow, and implicitly Airflow installs them to facilitate the execution of pipelines. These components are

- **DAG directory**, to keep all DAG in place to be read by scheduler and executor.
- **Scheduler** parses DAGs, checks their schedule interval and starts scheduling DAGs tasks for execution by passing them to airflow workers.
- **Workers**, responsible for doing actual work. It picks up tasks and executes them.
- **Web server** presents a handy user interface to inspect, trigger, and debug the behaviour of DAGs and tasks.

- **Metadata Database**, used by the scheduler, executor, and webserver to store state so that all of them can communicate and take decisions. - follow this link to see, how to set and get metadata variables. Accessing Metadata Database

For now, this is enough architecture. Let's move to the next part.

---

## Installing Airflow

Let's start with the installation of the Apache Airflow. Now, if already have pip installed in your system, you can skip the first command. To install pip run the following command in the terminal.

```
sudo apt-get install python3-pip
```

Next airflow needs a home on your local system. By default **~/airflow** is the default location but you can change it as per your requirement.

```
export AIRFLOW_HOME=~/airflow
```

Now, install the apache airflow using the pip with the following command.

```
pip3 install apache-airflow
```

Airflow requires a database backend to run your workflows and to maintain them. Now, to initialize the database run the following command.

```
airflow initdb
```

We have already discussed that airflow has an amazing user interface. To start the webserver run the following command in the terminal. The default port is 8080 and if you are using that port for something else then you can change it.

```
airflow webserver -p 8080
```

Now, start the airflow scheduler using the following command in a different terminal. It will run all the time and monitor all your workflows and triggers them as you have assigned.

```
airflow scheduler
```

Now, create a folder name DAGs in the airflow directory where you will define your workflows or DAGs and open the web browser and go open: <http://localhost:8080/admin/> and you will see something like this:

Airflow							
DAGs							
2020-11-17 07:11:56 UTC							
DAGs							
Search:							
	DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
<input checked="" type="checkbox"/>	LAKSHAY	1 day, 0:30:00	airflow				
<input checked="" type="checkbox"/>	example_bash_operator	@@*	Airflow				
<input checked="" type="checkbox"/>	example_branch_dag_operator_v3	@*	Airflow				
<input checked="" type="checkbox"/>	example_branch_operator	@daily	Airflow				
<input checked="" type="checkbox"/>	example_complex	None	airflow				
<input checked="" type="checkbox"/>	example_external_task_marker_child	None	airflow				
<input checked="" type="checkbox"/>	example_external_task_marker_parent	None	airflow				

Preferences

General

Resources

ADVANCED

FILE SHARING

PROXIES

NETWORK

Docker Engine

Experimental Features

Kubernetes

Software Updates

Resources

Advanced

CPUs: 4

Memory: 4.25 GB

Swap: 1 GB

Disk image size: 80 GB (23.7 GB used)

Cancel

Apply & Restart

## Installation Steps (on Mac)

1. Create a file name as `airflow_runner.sh`. Copy the below commands in the script.

```
docker run --rm "debian:buster-slim" bash -c 'numfmt --to iec $(echo $(( $(getconf _PHYS_PAGES) * $(getconf PAGE_SIZE) )))'
curl -Lfo 'https://airflow.apache.org/docs/apache-airflow/2.2.4/docker-compose.yaml'
mkdir -p ./dags ./logs ./plugins
echo -e "AIRFLOW_UID=$(id -u)" > .env
```

2. Provide execute access to file. **chmod +x airflow\_runner.sh**

3. Run **source airflow\_runner.sh**
4. Once the above steps are completed successfully, run **docker-compose up airflow-init** to initialise the database.

After initialisation is complete, you should see a message like the below.

```
airflow-init_1    | Upgrades done
airflow-init_1    | Admin user airflow created
airflow-init_1    | 2.3.0
start_airflow-init_1 exited with code 0
```

Now, we are ready to go for the next step.

## Components of Apache Airflow

- **DAG:** It is the Directed Acyclic Graph – a collection of all the tasks that you want to run which is organized and shows the relationship between different tasks. It is defined in a python script.
- **Web Server:** It is the user interface built on the Flask. It allows us to monitor the status of the DAGs and trigger them.
- **Metadata Database:** Airflow stores the status of all the tasks in a database and do all read/write operations of a workflow from here.
- **Scheduler:** As the name suggests, this component is responsible for scheduling the execution of DAGs. It retrieves and updates the status of the task in the database.

## User Interface

- Now that you have installed the Airflow, let's have a quick overview of some of the components of the user interface.









### DAGS VIEW

- It is the default view of the user interface. This will list down all the DAGS present in your system. It will give you a summarized view of the DAGS

like how many times a particular DAG was run successfully, how many times it failed, the last execution time, and some other useful links.








DAGs












Search:

	DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
 	LAKSHAY	1 day, 0:00:00	airflow	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>	<div></div>

## GRAPH VIEW

In the graph view, you can visualize each and every step of your workflow with their dependencies and their current status. You can check the current status with different color codes like:

	Task is successfully completed.
	Task is in progress.
	Task failed
	Task has been skipped
	Task failed once, executor is retrying
	Task is in the queue.
	No Status

 Graph View  Tree View  Task Duration  Task Times  Landing Times  Gantt  Details  Code  Trigger DAG  Refresh  Delete

LAKSHAY

Toggle wrap

```
1  """
2  Code that goes along with the Airflow located at:
3  http://airflow.readthedocs.org/en/latest/tutorial.html
4  """
5  from airflow import DAG
6  from airflow.operators.bash_operator import BashOperator
7  from datetime import datetime, timedelta
8
9
10 
```

## Define your Task and DAG

Airflow provides three ways to define your DAG

1. Classical
2. with context manager
3. Decorators

It doesn't matter which way you define your workflow but sticking to only one helps to debug and review the codebase. Mixing different definitions can perplex the code (though it is a personal choice)

```
# Classical

import pendulum
from airflow import DAG
from airflow.operators.dummy import DummyOperator

dag = DAG("classical_dag", start_date=pendulum.datetime(2022, 5, 15, tz="UTC"),
          schedule_interval="@daily", catchup=False)

op = DummyOperator(task_id="a-dummy-task", dag=dag)
```

```
# with context manager

with DAG(
    "context_manager_dag", start_date=pendulum.datetime(2022, 5, 15, tz="UTC"),
    schedule_interval="@daily", catchup=False
) as dag:
    op = DummyOperator(task_id="a-dummy-task")
```

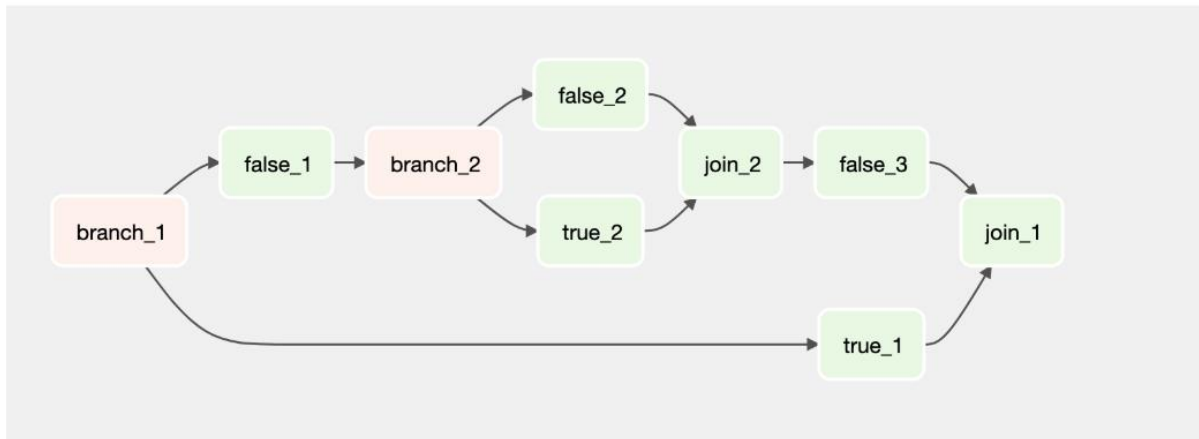
```
# Decorators

@dag(start_date=pendulum.datetime(2022, 5, 15, tz="UTC"),
     schedule_interval="@daily", catchup=False)
def generate_decorator_dag():
    op = DummyOperator(task_id="a-dummy-task")

dag = generate_decorator_dag()
```

## How to create a bit complex tasks flow?

Let's take this example.



light red - shows branch flow (two or more flows) i.e. **branch\_1**, **branch\_2**

light green - normal task for different purpose. i.e. **false\_1**, **false\_2**, **true\_2** etc.

Now, without worrying about code, let's create the task flow to represent the above structure.

## How bit shift operator (>> or <<) defines task dependency?

The `__rshift__` and `__lshift__` methods of the `BaseOperator` class implements the Python bit shift logical operator in the context of setting a task or a DAG downstream of another. See the implementation [here](#).

So, **bit shift** has been used as syntactic sugar for `set_upstream (<<)` and `set_downstream (>>)` tasks.

For example `task1 >> task2` is same as `task2 << task1` is same as `task1.set_downstream(task2)` is same as `task1.set_upstream(task2)`

This operator plays important role to build relationships among the tasks.

## Effective Task Design in Airflow



To ensure tasks run efficiently and reliably in Airflow, keep these 3 core principles in mind:

### 1. Atomicity

- Each task should do just one thing — keep it small and focused.
- If your task handles too much, break it into smaller tasks.

### 2. Idempotency

- A task should give the same result every time it's run with the same input.
- For data tasks: Check if results already exist or safely overwrite them.
- For database tasks: Use upsert to avoid duplication or errors during reruns.

### 3. Backfilling (Historical Data Processing)

- Controlled using the catchup parameter in the DAG.
- catchup=True (default): runs from past scheduled intervals.
- catchup=False: starts from the current date and skips the past.

## Runtime Variables

All operators load context a pre-loaded variable to supply the most used variables during the DAG run. Python examples can be shown here

```
from urllib import request

import airflow
from airflow import DAG
from airflow.operators.python import PythonOperator

dag = DAG(
    dag_id="showcontext",
    start_date=airflow.utils.dates.days_ago(1),
    schedule_interval="@hourly",
)

def _show_context(**context):
    """
    context here contains these preloaded items
    to pass in dag during runtime.

    Airflow's context dictionary can be found in the
    get_template_context method, in Airflow's models.py.
    """
    {
        'dag': task.dag,
        'ds': ds,
        'ds_nodash': ds_nodash,
        'ts': ts,
        'ts_nodash': ts_nodash,
        'yesterday_ds': yesterday_ds,
        'yesterday_ds_nodash': yesterday_ds_nodash,
        'tomorrow_ds': tomorrow_ds,
        'tomorrow_ds_nodash': tomorrow_ds_nodash,
        'END_DATE': ds,
        'end_date': ds,
        'dag_run': dag_run,
        'run_id': run_id,
        'execution_date': self.execution_date,
        'prev_execution_date': prev_execution_date,
        'next_execution_date': next_execution_date,
        'latest_date': ds,
        'macros': macros,
        'params': params,
        'tables': tables,
        'task': task,
        'task_instance': self,
        'ti': self,
        'task_instance_key_str': ti_key_str,
        'conf': configuration,
        'test_mode': self.test_mode,
    }
    """
    start = context["execution_date"]
    end = context["next_execution_date"]
    print(f"Start: {start}, end: {end}")

show_context = PythonOperator(
    task_id="show_context",
    python_callable=_show_context,
    dag=dag
```

The above variables are pre-loaded under **context** and can be used anywhere in the operator. *Dynamic reference* happens in the **Jinja templating** way.

e.g. `{{ds}}`, `{{next_ds}}`, `{{dag_run}}`

## Templating fields and scripts

Two attributes in the BaseOperator define what can we put in for templating.

**template\_fields**: Holds the list of variables which are templateable

**template\_ext**: Contains a list of file extensions that can be read and templated at runtime

See this example for those two fields' declaration

```
class BashOperator(BaseOperator):
    template_fields = ('bash_command', 'env') # defines which fields are templateable
    template_ext = ('.sh', '.bash') # defines which file extensions are templateable

    def __init__(
        self,
        *,
        bash_command,
        env: None,
        output_encoding: 'utf-8',
        **kwargs,
    ):
        super().__init__(**kwargs)
        self.bash_command = bash_command # templateable (can also give path to .sh or .bash script)
        self.env = env # templateable
        self.output_encoding = output_encoding # not templateable
```

## Example of DAG which uses Airflow context for templating

Let's take an example to showcase the power of templating

```
from datetime import datetime

BashOperator(
    task_id="print_now",
    bash_command="echo It is currently {{ macros.datetime.now() }}",
)
```

## Note

Check [here](#) for the micro list.

But now you might be thinking about where we got **PythonOperator**, **DAG** etc. We will see Airflow's critical **modules** to understand it.

This topic is covered in detail in this [blog](#).

```
lakshay@thinkpad-e490:~$ cricket scores
```

Match	Pakistan Super League, Final: Karachi Kings v Lahore Qalandars at Karachi, Nov 17, 2020
Status	Karachi Kings require another 86 runs with 9 wickets and 13.2 overs remaining
Summary	Lahore Qalandars - 134/7 Karachi Kings - 49/1 Karachi Kings 49/1 (6.4 ov, AD Hales 11*, Babar Azam 22*, Dilbar Hussain 0/13)
Match	Women's Big Bash League, 45th Match: Brisbane Heat Women v Perth Scorchers Women at Sydney, Nov 18, 2020
Status	Match scheduled to begin at 09:30 local time (22:30 GMT)
Summary	
Match	Women's Big Bash League, 46th Match: Adelaide Strikers Women v Melbourne Renegades Women at Sydney, Nov 18, 2020
Status	Match scheduled to begin at 14:30 local time (03:30 GMT)

## Importing the Libraries

Now, we will create the same workflow using Apache Airflow. The code will be completely in python to define a DAG. Let's start with importing the libraries that we need. We will use only the BashOperator only as our workflow requires the Bash operations to run only.

## Defining a DAG in Apache Airflow

Apache Airflow allows you to schedule and orchestrate workflows as **Directed Acyclic Graphs (DAGs)**. Here's how to set up a simple DAG from scratch using practical elements like task dependencies, scheduling, and logging.

---

## Defining DAG Arguments (default\_args)

Before creating a DAG, you define a dictionary of arguments that apply to all tasks within the DAG. These help manage retries, ownership, scheduling behavior, and failure alerts.

```

default_args = {
    'owner': 'lakshay',                # Who owns the DAG
    'depends_on_past': False,          # Should tasks depend on previous runs?
    'start_date': days_ago(2),        # When should the DAG start?
    'email': ['airflow@example.com'],  # Who to notify on failure
    'email_on_failure': False,        # Disable email on task failure
    'email_on_retry': False,          # Disable email on retries
    'retries': 1,                     # Number of retries before failing
    'retry_delay': timedelta(minutes=5), # Wait time between retries
}

```

You can override any of these per task if needed.

## Creating the DAG

Now that the arguments are defined, you create the actual DAG object.

```

dag = DAG(
    'live_cricket_scores',            # Unique ID for the DAG
    default_args=default_args,
    description='First example to get Live Cricket Scores',
    schedule_interval=timedelta(days=1), # How often should the DAG run
)

```

This DAG will run once daily and is responsible for retrieving live cricket scores.

## Defining the Tasks

In this example, we have **two Bash-based tasks**:

- **Task 1 (print):** Just prints a message.
- **Task 2 (get\_cricket\_scores):** Runs a command to fetch live scores (assuming the command exists on your machine).

Use the BashOperator to run terminal commands.

```
# Task 1: Print a message
t1 = BashOperator(
    task_id='print',
    bash_command='echo Getting Live Cricket Scores!!!',
    dag=dag,
)

# Task 2: Fetch cricket scores
t2 = BashOperator(
    task_id='get_cricket_scores',
    bash_command='cricket scores',
    dag=dag,
)
```

## Defining Task Dependencies

Connect the tasks using the `>>` operator:

**t1 >> t2**

This means: **run t1 first, then t2.**

Behind the scenes, `>>` is just a cleaner way of writing:

**t1.set\_downstream(t2)**

**# or**

**t2.set\_upstream(t1)**

---

## Viewing the DAG in the Airflow UI

After you've saved your DAG file in the Airflow **dags/** folder:

1. Go to the **Airflow Web UI**.
2. Refresh the page. Your new DAG (**live\_cricket\_scores**) will appear.
3. Toggle it **on**.
4. Click the **Play/Trigger** button to run it.
5. Go to **Graph View** to see the workflow visually. Each task appears as a node.

- Click on a task (e.g., **get\_cricket\_scores**) → **View Log** to see real-time output.

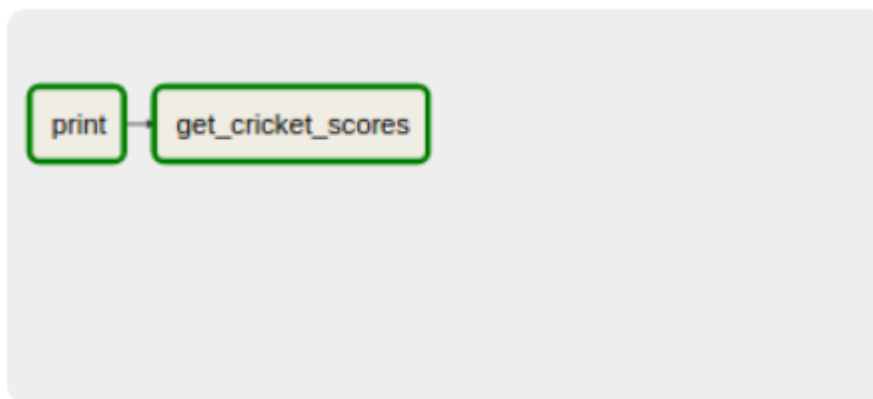
If all goes well, the task boxes will turn **green** to indicate success.

## Update the DAGS in Web UI

Now, refresh the user interface and you will see your DAG in the list. Turn on the toggle on the left of each of the DAG and then trigger the DAG.

		latest_only_with_trigger	4:00:00	airflow			
		live_cricket_scores	1 day, 9:00:00	lakshay			
		test_utils	None	airflow			
		tutorial	1 day, 9:00:00	airflow			

Click on the DAG and open the graph view and you will see something like this. Each of the steps in the workflow will be in a separate box and its border will turn dark green once it is completed successfully.



Click on the node “**get\_cricket scores**” to get more details about this step. You will see something like this.

get\_cricket\_scores on 2020-11-17T17:47:17.245061+00:00

Task Instance Details

Rendered

Task Instances

View Log

Download Log (by attempts):  
1

Run

Ignore All Dps

Ignore Task State

Ignore Task Dps

Clear

Past

Future

Upstream

Downstream

Recursive

Failed

Mark Failed

Past

Future

Upstream

Downstream

Mark Success

Past

Future

Upstream

Downstream

Close

Now, click on View Log to see the output of your code.

```
[2020-11-17 23:19:52,115] {bash_operator.py:115} INFO - Running command: cricket scores
[2020-11-17 23:19:52,121] {bash_operator.py:122} INFO - Output:
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Match | Pakistan Super League, Final: Karachi Kings v Lahore Qalandars at Karachi, Nov 17, 2020 |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Status | Karachi Kings require another 59 runs with 8 wickets and 55 balls remaining |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Summary | Lahore Qalandars - 134/7 |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | | Karachi Kings - 76/2 |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | | Karachi Kings 76/2 (10.5 ov, CAK Walton 15*, Babar Azam 33*, D Wiese 0/7) |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | | |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Match | Women's Big Bash League, 45th Match: Brisbane Heat Women v Perth Scorchers Women at Sydney, Nov 18, 2020 |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Status | Match scheduled to begin at 09:30 local time (22:30 GMT) |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Summary | |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | | |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Match | Women's Big Bash League, 46th Match: Adelaide Strikers Women v Melbourne Renegades Women at Sydney, Nov 18, 2020 |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Status | Match scheduled to begin at 14:30 local time (03:30 GMT) |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - | Summary | |
[2020-11-17 23:19:57,142] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,143] {bash_operator.py:126} INFO - | | |
[2020-11-17 23:19:57,143] {bash_operator.py:126} INFO - +-----+
[2020-11-17 23:19:57,143] {bash_operator.py:126} INFO - | Match | Women's Big Bash League, 47th Match: Melbourne Stars Women v Hobart Hurricanes Women at Sydney, Nov 18, 2020 |
[2020-11-17 23:19:57,143] {bash_operator.py:126} INFO - +-----+
```

## Operators

Operators help run your function or any executable program.

### Type of Operators

Primarily there are three types of Operators.

#### (i) Operators

Helps to trigger certain action. Few of them are

- **PythonOperator** - To wrap a python callables/functions inside it.
- **BashOperator** - To call your bash script or command. Within BashOperator we can also call any executable program.
- **DummyOperator** - to show a dummy task
- **DockerOperator** - To write and execute docker images.
- **EmailOperator** - To send an email (using SMTP configuration)

*and there many more operators do exists.* See the full operators list in the official documentation.

## (ii) Sensors

A particular type of operator whose purpose is to wait for an event to start the execution. For instance,

- **ExternalTaskSensor** waits on another task (in a different DAG) to complete execution.
- **S3KeySensor** S3 Key sensors are used to wait for a specific file or directory to be available on an S3 bucket.
- **NamedHivePartitionSensor** - Waits for a set of partitions to appear in Hive.

## (iii) Transfers

Moves data from one location to another. e.g.

- **MySqlToHiveTransfer** Moves data from MySql to Hive.
- **S3ToRedshiftTransfer** load files from s3 to Redshift.

---

## Executors

Executors help to run the task instance (task instances are functions wrapped under operators).

---

### Types of Executors

#### Local Executors

- **Debug Executor:** A debug tool used from IDE. It is a single-process executor that queues tasks and executes them.
- **Sequential Executor:** Default executor that runs within the scheduler. It executes one task instance at a time, making it unsuitable for production.
- **Local Executor:** Also runs within the scheduler and can execute multiple task instances at a time. Still, it's not ideal for production as it doesn't scale well.

#### Remote Executors

- **Celery Executor:** Runs tasks on dedicated machines (workers). It uses a distributed task queue to distribute load across different workers to parallelize work. It scales horizontally, making it fault-tolerant and a strong candidate for production.



- **Kubernetes Executor:** Runs tasks in dedicated PODs (workers) using Kubernetes APIs to manage them. It scales efficiently and is an ideal production choice.
  - **LocalKubernetes Executor:** A hybrid of local and Kubernetes execution.
  - **CeleryKubernetes Executor:** Runs both CeleryExecutor and KubernetesExecutor simultaneously. The executor is chosen based on the task's queue. Useful in specific cases.
  - **Dask Executor:** Supports running Dask clusters on a single machine or across remote networks.
- 

## Hooks

Hooks provide a high-level interface to establish connections with databases or other external services.

---

## What if a required module is missing?

If the appropriate operator, executor, sensor, or hook isn't available, you can create your own. Airflow provides base classes to help with customization.

### Example: Custom Classes

```
from airflow.models import BaseOperator
from airflow.sensors.base import BaseSensorOperator
from airflow.hooks.base_hook import BaseHook
from airflow.utils.decorators import apply_defaults

class MyCustomOperator(BaseOperator):

    @apply_defaults
    def __init__(self, **kwargs):
        super(MyCustomOperator, self).__init__(**kwargs)
        # Initialization logic here

    def execute(self, context):
        # Task logic here
        pass

class MyCustomSensor(BaseSensorOperator):

    @apply_defaults
    def __init__(self, **kwargs):
        super(MyCustomSensor, self).__init__(**kwargs)
        # Initialization logic here

    def poke(self, context):
        # Poke logic here
        pass

class MyCustomHook(BaseHook):

    @apply_defaults
    def __init__(self, **kwargs):
        super(MyCustomHook, self).__init__(**kwargs)
        # Initialization logic here

    def get_connection(self):
        # Connection logic here
        pass
```

## Conclusion: Why Use Apache Airflow

Apache Airflow is a powerful, flexible, and scalable platform for building and managing complex data pipelines. It follows the "configuration as code" philosophy, allowing teams to version control workflows and dynamically generate tasks.

### ◆ What Makes Airflow Effective

- **Modular Design:** Easily extensible via custom operators, hooks, sensors, and executors.
- **Rich UI:** Monitor DAGs visually, debug failures, and review logs via the web interface.
- **Scalable Architecture:** Move from local testing to production-ready deployments with Celery or Kubernetes executors.
- **Backfilling & Scheduling:** Automatically reprocess historical data and schedule workflows with cron-like expressions.
- **Templating Support:** Leverage Jinja templating and runtime variables for dynamic, context-aware pipelines.

### ◆ **When to Use Airflow**

- Batch-oriented processing tasks
- Complex ETL pipelines
- Time-based job scheduling
- Dependency-driven workflows

### ◆ **When *Not* to Use Airflow**

- Real-time or low-latency streaming pipelines
- Extremely high-throughput micro-task execution