**Apache Airflow**

## **Section 1:**

## **1. What is Apache Airflow?**

**Simple Definition**:  
 Apache Airflow is an **open-source tool to schedule, monitor, and manage workflows (data pipelines)**.

* Think of it as a **task manager for data**.
* Instead of writing one long Python script that runs forever, you break the workflow into **tasks**, and Airflow takes care of:  
  + **When** they run (scheduling)
  + **In what order** (dependencies)
  + **Where** they run (on which machine/executor)
  + **If they fail** (retries, alerts)

### **Analogy**

Imagine running a **restaurant kitchen**:

* **Chef (Scheduler)** decides *when* each dish should start.
* **Cooks (Workers)** execute the actual cooking tasks.
* **Order Board (DAG)** shows the sequence: wash veggies → cut → cook → plate.
* **Manager (Web UI)** monitors which dish is in progress, completed, or failed.

Airflow is like the **kitchen manager for your data pipelines**.

### **Real-World Use Cases**

* Extract data from an API → Clean it → Load it into a database (**ETL pipelines**)
* Run machine learning training jobs on a schedule
* Generate daily reports automatically
* Move data from cloud storage to a warehouse (e.g., S3 → Snowflake → Power BI)

Basically, any **workflow with steps and dependencies** can be handled by Airflow.

## **2. Core Components**

Airflow has 4 main pieces working together:

1. **Scheduler**
   * Decides which tasks need to run and when.
   * Follows your DAG schedule.
2. **Executor**
   * Decides *how* to run tasks (local, in parallel, on Celery, Kubernetes, etc.).
3. **Workers**
   * The actual machines/processes that execute the tasks.
4. **Metadata Database**
   * Stores state of tasks (success, fail, retry, queued).
   * Usually PostgreSQL/MySQL (default is SQLite for testing).
5. **Webserver (UI)**
   * Lets you see DAGs, task status, logs, retries, etc.

### 

### 

### **Diagram (mental picture)**

[Webserver (UI)]

|

[Scheduler] ----> [Metadata DB]

|

[Executor decides who runs it]

|

[Workers execute tasks]

## **3. Core Concepts**

You’ll hear these again and again:

* **DAG (Directed Acyclic Graph)** → a collection of tasks with dependencies (like a flowchart).
* **Task** → a single step in a workflow (e.g., extract data from API).
* **Operator** → a template for tasks (e.g., PythonOperator, BashOperator, SQLOperator).
* **Task Instance** → one run of a task at a specific time.
* **Scheduling** → deciding when a DAG runs (daily, hourly, on events).

## **4. How does Airflow work (High-Level)?**

1. You **define your workflow** in Python (DAG file).
2. Airflow **reads your DAG** and schedules tasks.
3. Scheduler sends tasks to the **Executor**.
4. Executor assigns tasks to **Workers**.
5. Task status is stored in the **Metadata DB**.
6. You monitor progress in the **Web UI**.

## **Recap (Cheat Sheet)**

* **Airflow = Workflow Orchestrator** for data pipelines.
* **Core Components** = Scheduler, Executor, Workers, Metadata DB, Web UI.
* **Core Concepts** = DAGs (workflows), Tasks (steps), Operators (templates).
* **Airflow works by**: defining workflow in Python → Scheduler → Executor → Workers → DB → UI monitoring.

## **Section 2:**

## **Core Components (Deeper Dive)**

### **1️⃣ Scheduler**

* The **brain** of Airflow.
* Continuously monitors DAGs.
* Decides *what task to run, when, and where*.
* Example: If your DAG says "run daily at 9 AM", the **scheduler** checks the time and tells executor *“Hey, start Task A now”*.

### **2️⃣ Metadata Database**

* Stores the **state** of DAGs and tasks:  
  + success, failure, retry, queued, skipped.
* Without it, Airflow won’t remember past runs.
* Default = **SQLite** (not good for production).
* In real projects → **Postgres** or **MySQL**.

### **3️⃣ Executor**

* Decides **how tasks will run**:  
  + In sequence?
  + In parallel on multiple workers?
  + On Kubernetes?
* Executors we’ll learn later: **Sequential, Local, Celery, Kubernetes**.

### **4️⃣ Workers**

* The **hands** of Airflow.
* Execute the tasks assigned by the executor.
* Example: Running a Python script, fetching from API, running SQL query, etc.

### **5️⃣ Webserver (UI)**

* The **dashboard** for Airflow.
* Lets you:  
  + See DAGs
  + Trigger runs manually
  + Check logs
  + Retry failed tasks
* By default available at [**http://localhost:8081**](http://localhost:8080).

## **Diagram with Flow**

[Webserver UI] ---> For Monitoring

|

[Scheduler] ---> Decides tasks to run

|

[Metadata DB] <--- Stores task status

|

[Executor] ---> Assigns execution strategy

|

[Workers] ---> Actually run tasks

## **Hands-On (UI Exploration)**

Open your Airflow UI (http://localhost:8081) and look for these:

* **DAGs Tab** → list of workflows.
* **Browse → Jobs / Task Instances** → see what scheduler is doing.
* **Admin → Connections** → where you define external systems (like MySQL, S3, APIs).

Try this:

1. In the UI, pick any **example DAG** (like example\_bash\_operator).
2. Toggle it **ON**.
3. Trigger a manual run (▶️ button).
4. Open the **Graph View** → see tasks flowing.
5. Open **Task Logs** → see what workers actually executed.

## **Recap (Cheat Sheet)**

* **Scheduler** = decides what runs.
* **Executor** = decides how it runs.
* **Workers** = actually run it.
* **Metadata DB** = remembers task history.
* **Webserver (UI)** = monitor, trigger, debug DAGs.

## **Section 3:**

## **Core Concepts in Airflow**

### **1️⃣ DAG (Directed Acyclic Graph)**

* A **workflow** in Airflow.
* Made up of tasks, arranged with dependencies.
* **Directed** = each task points to the next one.
* **Acyclic** = no loops allowed (can’t go back to a previous task).

Example DAG (real life):

Fetch Data → Clean Data → Load Data → Send Report

In Airflow, this is represented as a **DAG file** written in Python.

### 

### **2️⃣ Task**

* A single **unit of work** in your DAG.
* Example: “Fetch data from API” or “Run SQL query”.

### **3️⃣ Operator**

* A **template** to create tasks.
* Different operators = different kinds of tasks.

Some common ones:

* **PythonOperator** → run Python code.
* **BashOperator** → run shell commands.
* **SQLOperator** → run SQL.
* **EmailOperator** → send email.

Example:

from airflow.operators.bash import BashOperator

task1 = BashOperator(

task\_id="print\_date",

bash\_command="date",

dag=dag

)

This creates a **task** that prints today’s date.

### 

### 

### **4️⃣ Task Dependencies**

* Define **order of execution**.
* Use >> or <<.

Example:

task1 >> task2 # task1 runs first, then task2

### **5️⃣ Sensor**

* A special type of operator that **waits for something**.
* Example: wait until a file exists in S3 before continuing.

### **6️⃣ Hook**

* A way to **connect to external systems** (databases, APIs, cloud storage).
* Example: PostgresHook for PostgreSQL, S3Hook for AWS S3.

### **7️⃣ XComs (Cross-Communication)**

* A way for tasks to **share data** with each other.
* Example: Task A fetches an API response → Task B processes it.

## **Hands-On (Your First DAG)**

Create a simple DAG in dags/hello\_dag.py

from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime

# Define DAG

with DAG(

dag\_id="hello\_dag",

start\_date=datetime(2023, 1, 1),

schedule\_interval="@daily", # runs daily

catchup=False

) as dag:

# Task 1

task1 = BashOperator(

task\_id="print\_date",

bash\_command="date"

)

# Task 2

task2 = BashOperator(

task\_id="say\_hello",

bash\_command="echo 'Hello from Airflow!'"

)

# Define dependencies

task1 >> task2

### **What happens here?**

* DAG name = hello\_dag.
* Runs **once every day** (schedule: @daily).
* **Task 1** → prints the current date.
* **Task 2** → prints a message.
* Dependency = task1 must finish before task2 starts.

### 

### **Try This in UI**

1. Save the file in your **Airflow DAGs folder** (dags/hello\_dag.py).
2. In the UI, refresh → you’ll see hello\_dag.
3. Toggle it ON.
4. Trigger a run → check **Graph View** → then **logs**.

## **Recap (Cheat Sheet)**

* **DAG** = workflow.
* **Task** = one step in the workflow.
* **Operator** = template for tasks.
* **Sensor** = waits for condition.
* **Hook** = connection to external system.
* **XComs** = share data between tasks.

## **Section 4:**

### **1️⃣ More About Airflow**

* Airflow is **not** a data processing tool.  
  + It doesn’t transform/process data itself.
  + It **orchestrates** tasks (decides *when, where, and in what order*).
* Workflows are **Python code**, so they are flexible & version-controllable (Git).
* You can **monitor, retry, alert** on failures.
* Airflow is **extensible** → you can create custom operators, hooks, sensors.

Think of Airflow as a **director of a movie**.

* The director doesn’t act, sing, or dance.
* He just makes sure actors, singers, dancers do their parts **in the right sequence**.

### **2️⃣ Different Architectures**

Airflow can run in different setups depending on scale:

#### **a) Local Executor (small projects/testing)**

* Scheduler + Executor run on the **same machine**.
* Tasks run in parallel using local processes.
* Good for small pipelines.

#### **b) Celery Executor (medium-large projects)**

* Tasks are distributed across **multiple worker machines**.
* Uses **Celery + Message Broker (Redis/RabbitMQ)**.
* Good when you need **scalability**.

#### **c) Kubernetes Executor (cloud-native)**

* Each task runs in its **own Kubernetes pod**.
* Very scalable, cost-efficient (you only use resources when needed).
* Perfect for enterprise-scale pipelines.

### 

### **3️⃣ How Does Airflow Work (Step by Step)?**

1. You **write a DAG file** in Python (e.g., hello\_dag.py).
2. The **Scheduler** reads the DAG → checks schedule.
3. Scheduler places tasks into the **Metadata DB** (as "to be executed").
4. **Executor** checks how to run it (Local/Celery/K8s).
5. Executor sends task to **Workers**.
6. **Workers** run the task (Python script, SQL, Bash command, etc.).
7. Task state is updated in **Metadata DB**.
8. You monitor everything in the **Webserver (UI)**.

### **Diagram**

[DAG File in Python]

|

[Scheduler]

|

[Metadata Database]

|

[Executor]

|

-------------------

| | |

[Worker1] [Worker2] [Worker3]

## **Hands-On Check (Architecture in Docker Setup)**

Since you’re running Airflow with Docker Compose, here’s what’s happening:

* airflow-scheduler → Scheduler
* airflow-webserver → Web UI
* airflow-worker → Worker (Celery Executor)
* postgres → Metadata DB
* redis → Message broker (needed for Celery)

Run this to see containers:

docker ps

You’ll see something like:

* airflow-webserver
* airflow-scheduler
* airflow-worker
* redis
* postgres

That’s your **Celery Executor architecture** in action.

## **Recap (Cheat Sheet)**

* **Airflow = Orchestrator, not processor**.
* **Architectures**:  
  + Local Executor → small scale.
  + Celery Executor → multiple workers.
  + Kubernetes Executor → cloud scale.
* **Flow**: DAG file → Scheduler → Metadata DB → Executor → Workers → UI.

## **Section 5:**

## **1. What is Docker (Quick Recap)**

* Docker lets you run software in **containers** (lightweight, isolated environments).
* Instead of installing Airflow + Python + Postgres manually, Docker pulls prebuilt images and runs them together.
* Think of it like running **apps on your phone** — each app has everything it needs, so they don’t conflict.

## **2. Why Use Docker for Airflow?**

* Airflow has many moving parts (**webserver, scheduler, workers, database, broker**).
* Installing them manually is painful.
* Docker Compose lets you **define everything in one file (docker-compose.yaml)** and start all services with:

docker-compose up -d

## **3. The docker-compose.yaml File**

Here’s a simplified version (trainer-style):

version: '3'

services:

postgres: # Metadata DB

image: postgres:13

environment:

- POSTGRES\_USER=airflow

- POSTGRES\_PASSWORD=airflow

- POSTGRES\_DB=airflow

redis: # Message Broker (needed for Celery Executor)

image: redis:latest

airflow-webserver: # UI

image: apache/airflow:2.7.0

restart: always

depends\_on:

- postgres

- redis

ports:

- "8080:8080"

environment:

- AIRFLOW\_\_CORE\_\_EXECUTOR=CeleryExecutor

- AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONN=postgresql+psycopg2://airflow:airflow@postgres/airflow

- AIRFLOW\_\_CELERY\_\_BROKER\_URL=redis://redis:6379/0

- AIRFLOW\_\_CELERY\_\_RESULT\_BACKEND=db+postgresql://airflow:airflow@postgres/airflow

airflow-scheduler: # Scheduler

image: apache/airflow:2.7.0

restart: always

depends\_on:

- airflow-webserver

airflow-worker: # Worker (executes tasks)

image: apache/airflow:2.7.0

restart: always

depends\_on:

- airflow-scheduler

### **Mapping to Airflow Components**

* **postgres** → Metadata Database
* **redis** → Message Broker for Celery
* **airflow-webserver** → UI at http://localhost:8080
* **airflow-scheduler** → Decides what runs
* **airflow-worker** → Executes tasks

## **4. Installing & Running Airflow with Docker Compose**

1. **Download the official Docker Compose file**

curl -LfO 'https://airflow.apache.org/docs/apache-airflow/stable/docker-compose.yaml'

1. **Set Airflow home**

mkdir airflow-docker && cd airflow-docker

1. **Initialize Airflow DB**

docker-compose up airflow-init

1. **Start Airflow**

docker-compose up -d

1. **Check running services**

docker ps

1. Open UI → [http://localhost:8081](http://localhost:8080)
   * Default user: airflow / airflow

## **5. Recap (Cheat Sheet)**

* **Docker** = runs Airflow components in isolated containers.
* **docker-compose.yaml** = defines Airflow’s services (webserver, scheduler, workers, DB, broker).
* **Command Flow**:  
  + docker-compose up airflow-init → initialize DB
  + docker-compose up -d → start Airflow
* **UI** → available at port **8080**.

## **Section 6:**

## **1. The Grid View**

* The **default view** when you click a DAG.
* Shows a **table of DAG runs (rows)** vs **tasks (columns)**.
* Cells are colored based on status:  
  + 🟢 Success
  + 🔴 Failed
  + 🟡 Running
  + ⚪ Skipped
* Useful for **monitoring multiple runs quickly**.

Example:

Run Date | task1 | task2 | task3

-----------------------------------

2025-08-18 | 🟢 | 🟢 | 🟢

2025-08-19 | 🟢 | 🔴 | ⚪

## **2. The Graph View**

* Shows tasks as a **flowchart** (nodes & arrows).
* Good for visualizing **dependencies** between tasks.
* You can click a node (task) to see logs.

Example DAG flow:

task1 → task2 → task3

## **3. The Landing Times View**

* Focuses on **task start times and durations** across runs.
* Helps spot performance bottlenecks.
* Example: if task2 always starts late, maybe task1 is slow.

## **4. The Calendar View**

* Shows DAG runs on a **calendar**.
* Useful to see if your DAGs are running on schedule (daily, weekly, etc.).

## **5. The Gantt View**

* Visual timeline of tasks for a specific DAG run.
* Shows how long each task took.
* Great for **performance debugging**.

Example:

task1 |■■■■■■■■

task2 |■■■■■■■■■■

task3 |■■

## **6. The Code View**

* Lets you see the **Python code** of the DAG directly in the UI.
* Useful for debugging if you forgot what the DAG does.
* You can’t edit code here, but you can inspect.

## **Hands-On Exercise**

Open your Airflow UI → pick your hello\_dag (or example DAG).

1. Trigger a run.
2. Explore each view:  
   * **Grid** → see task statuses.
   * **Graph** → see the flow.
   * **Calendar** → check execution date.
   * **Gantt** → check task duration.
   * **Code** → see DAG definition.

This is how most Data Engineers **monitor live workflows** daily.

## 

## **Recap (Cheat Sheet)**

* **Grid View** → task status by run.
* **Graph View** → dependency flow.
* **Landing Times** → task execution times.
* **Calendar** → scheduled DAG runs.
* **Gantt** → performance timelines.
* **Code** → DAG source code.

## **Section 7:**

# **Data Pipeline with Airflow**

### **1️⃣ What is a DAG?**

* A **Directed Acyclic Graph** = your **workflow**.
* DAGs in Airflow are written in **Python files**.
* Inside a DAG → you define **tasks** and their **dependencies**.

Example workflow (ETL):

Extract data → Transform data → Load into DB

### **2️⃣ What is an Operator?**

* An **operator** is like a **task template**.
* Each operator knows how to run a certain type of job.

Common ones:

* BashOperator → run shell commands
* PythonOperator → run Python code
* SQLOperator → run SQL queries
* EmailOperator → send emails
* Sensor (special) → wait for something

Example:

from airflow.operators.bash import BashOperator

task1 = BashOperator(

task\_id="print\_date",

bash\_command="date",

dag=dag

)

### **3️⃣ Providers**

* Providers = **plugins/packages** that let Airflow talk to external systems (databases, cloud services, APIs).
* Example: apache-airflow-providers-postgres lets you connect to PostgreSQL.
* They give you extra **operators, hooks, sensors** for that system.

### **4️⃣ Creating a Table (Example Use Case)**

Let’s say we want to store user data in Postgres. First, we’ll need to:

1. Create a **Postgres connection** in Airflow UI (Admin → Connections).
2. Use an **operator** (e.g., PostgresOperator) to create the table.

Example:

from airflow.providers.postgres.operators.postgres import PostgresOperator

create\_table = PostgresOperator(

task\_id="create\_user\_table",

postgres\_conn\_id="my\_postgres",

sql="""

CREATE TABLE IF NOT EXISTS users (

id SERIAL PRIMARY KEY,

name VARCHAR(50),

email VARCHAR(50)

);

"""

)

### **5️⃣ Creating a Connection (UI)**

* Go to **Airflow UI → Admin → Connections → Add Connection**.
* Example:  
  + Conn Id = my\_postgres
  + Conn Type = Postgres
  + Host = postgres (service name in docker-compose)
  + Schema = airflow
  + Login = airflow
  + Password = airflow
  + Port = 5432

This allows Airflow to talk to Postgres inside your Docker network.

### **6️⃣ What is a Sensor?**

* A **sensor** waits for something before continuing.
* Example: Wait until a file exists in S3.

from airflow.sensors.filesystem import FileSensor

wait\_for\_file = FileSensor(

task\_id="wait\_for\_input\_file",

filepath="/opt/airflow/data/input.csv",

poke\_interval=30, # check every 30s

timeout=600 # stop waiting after 10 min

)

### **7️⃣ Hooks**

* **Hooks = connectors** to external systems.
* Example: PostgresHook, S3Hook, HttpHook.
* Operators often use hooks under the hood.

### **8️⃣ Example Mini Pipeline (Extract → Process → Store)**

Create a DAG user\_pipeline.py in dags/

from airflow import DAG

from airflow.operators.python import PythonOperator

from datetime import datetime

# Dummy data

def extract\_users(\*\*kwargs):

users = [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]

# Push to XCom

return users

def process\_users(\*\*kwargs):

ti = kwargs['ti']

users = ti.xcom\_pull(task\_ids='extract\_users') # Pull result from previous task

processed = [u["name"].upper() for u in users]

print("Processed:", processed)

return processed

def store\_users(\*\*kwargs):

ti = kwargs['ti']

users = ti.xcom\_pull(task\_ids='process\_users')

print("Storing users:", users) # Normally write to DB

with DAG(

dag\_id="user\_pipeline",

start\_date=datetime(2023,1,1),

schedule\_interval="@daily",

catchup=False

) as dag:

t1 = PythonOperator(

task\_id="extract\_users",

python\_callable=extract\_users

)

t2 = PythonOperator(

task\_id="process\_users",

python\_callable=process\_users

)

t3 = PythonOperator(

task\_id="store\_users",

python\_callable=store\_users

)

# Set dependencies

t1 >> t2 >> t3

### **What this DAG does**

1. **Extract users** → returns sample list of dicts.
2. **Process users** → converts names to uppercase.
3. **Store users** → prints them (in real case, insert into DB).

## **Recap (Cheat Sheet)**

* **DAG** = workflow.
* **Operator** = task template.
* **Sensor** = waits for condition.
* **Hook** = external connector.
* **Providers** = plugins for external systems.
* Mini pipeline example = Extract → Process → Store.

**Section 8:**

# **DAG Scheduling in Apache Airflow**

### **1️⃣ What is DAG Scheduling?**

* Airflow lets you automatically trigger DAGs at a defined frequency.
* Controlled by the schedule\_interval (in Airflow <2.6) or timetable (in >=2.6).
* Defines when the DAG should run.

### **2️⃣ Types of Scheduling**

#### **a) Manual Trigger**

* You click “Trigger DAG” in the UI.
* Good for testing.

#### **b) Time-based Scheduling**

* Most common.

Examples:

* + @daily → once every day
  + @hourly → every hour
  + @weekly → once a week
  + @monthly → once a month
  + None → DAG won’t run automatically

Example:

with DAG(

dag\_id="daily\_job",

schedule\_interval="@daily",

start\_date=datetime(2023,1,1),

catchup=False

) as dag:

…

#### **c) Cron Expressions**

* For advanced custom schedules.
* Format: minute hour day month day\_of\_week

Examples:

* "0 9 \* \* \*" → run every day at 9 AM
* "0 \*/2 \* \* \*" → run every 2 hours
* "30 6 \* \* 1-5" → run at 6:30 AM Mon–Fri

Example:

with DAG(

dag\_id="cron\_example",

schedule\_interval="0 9 \* \* \*",

start\_date=datetime(2023,1,1),

catchup=False

) as dag:

...

#### **d) Dataset Scheduling (Airflow 2.4+)**

* DAGs can trigger based on data availability, not just time.
* Example:  
  + DAG A produces a dataset.
  + DAG B runs when that dataset is updated.

Example:

from airflow.datasets import Dataset

my\_dataset = Dataset("/path/to/my/data.csv")

with DAG(

dag\_id="producer",

schedule\_interval="@daily",

start\_date=datetime(2023,1,1),

catchup=False

) as dag:

...

with DAG(

dag\_id="consumer",

schedule=[my\_dataset], # depends on dataset

start\_date=datetime(2023,1,1),

catchup=False

) as dag:

…

#### **e) Event/External Trigger**

* DAGs can also be triggered programmatically (via API/CLI).

Example CLI:

airflow dags trigger user\_pipeline

### **3️⃣ Catchup Parameter**

* Airflow replays past DAG runs between start\_date and today if catchup=True.
* Default = True.
* Usually set catchup=False for learning/testing.

### **4️⃣ Example Scheduling DAG**

from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime

with DAG(

dag\_id="scheduled\_dag",

schedule\_interval="0 9 \* \* \*", # every day at 9 AM

start\_date=datetime(2023,1,1),

catchup=False

) as dag:

task = BashOperator(

task\_id="print\_time",

bash\_command="date"

)

## **Recap (Cheat Sheet)**

* schedule\_interval → controls timing.
* Options:  
  + @daily, @hourly, @weekly, etc.
  + Cron syntax ("0 9 \* \* \*").
  + Dataset-based triggers.
  + Manual/API triggers.
* catchup=False → prevents backlog runs.

**Section 9:**

# **Data Pipeline with Airflow**

### **1. What is a DAG?**

* DAG = Directed Acyclic Graph
* In Airflow, a DAG is your data pipeline.
* It’s a collection of tasks with dependencies, showing the order of execution.
* “Directed” → tasks flow in one direction.
* “Acyclic” → no cycles/loops (task A → B → A is not allowed).

Hands-on (basic DAG example):

from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime

# Define a DAG

with DAG(

dag\_id="first\_dag",

start\_date=datetime(2025, 8, 19),

schedule\_interval="@daily", # run daily

catchup=False

) as dag:

# Define tasks

task1 = BashOperator(

task\_id="print\_date",

bash\_command="date"

)

task2 = BashOperator(

task\_id="say\_hello",

bash\_command="echo 'Hello from Airflow!'"

)

# Set dependencies

task1 >> task2

This DAG runs every day: first prints the date, then prints “Hello from Airflow”.

**2. What is an Operator?**

### **Simple Explanation**

* In Airflow, a **task = Operator**.
* Operators are **building blocks** of a DAG.
* Each operator does one job (like run a script, query a DB, call an API).
* You can think of **DAG = pipeline**, **Operators = steps inside pipeline**.

Types of Operators:

1. **Action Operators** → do something (run bash, Python code, SQL, etc.)
2. **Transfer Operators** → move data between systems (e.g., from MySQL to S3).
3. **Sensor Operators** → wait for something to happen (like a file to arrive).

### **Hands-on Example (different operators)**

from airflow import DAG

from airflow.operators.bash import BashOperator

from airflow.operators.python import PythonOperator

from datetime import datetime

# Python function for PythonOperator

def greet():

print("Hello! This is a Python Operator.")

with DAG(

dag\_id="operators\_example",

start\_date=datetime(2025, 8, 19),

schedule\_interval=None,

catchup=False

) as dag:

# Bash Operator

bash\_task = BashOperator(

task\_id="bash\_task",

bash\_command="echo 'This is a Bash Operator'"

)

# Python Operator

python\_task = PythonOperator(

task\_id="python\_task",

python\_callable=greet

)

bash\_task >> python\_task

Here:

* First task (BashOperator) prints a message using Bash.
* Second task (PythonOperator) runs a Python function.

# **3. Providers in Airflow**

### **Simple Explanation**

* **Providers = Plug-ins (packages) in Airflow** that allow it to connect to external systems.
* Example: If you want to use Airflow with **AWS, GCP, MySQL, Postgres, Slack**, etc., you need the respective provider package.
* Without providers, Airflow only knows the basics — providers add extra integrations.

Providers contain:

1. **Operators** → to run tasks on that system (like S3CreateBucketOperator).
2. **Hooks** → to connect to external systems (like S3Hook).
3. **Sensors** → to wait for events in that system (like S3KeySensor).
4. **Connections** → templates to easily connect (like AWS credentials).

### 

### **Example**

If you want to connect to **Postgres**:

pip install apache-airflow-providers-postgres

Then in your DAG:

from airflow.providers.postgres.operators.postgres import PostgresOperator

# Example task to create a table in Postgres

create\_table = PostgresOperator(

task\_id="create\_table",

postgres\_conn\_id="my\_postgres\_conn",

sql="""

CREATE TABLE IF NOT EXISTS users (

id SERIAL PRIMARY KEY,

name VARCHAR(50),

age INT

);

"""

)

Here:

* apache-airflow-providers-postgres → provider package.
* PostgresOperator → operator from that provider.
* my\_postgres\_conn → connection you’ll define in Airflow UI.

### **Real-life Analogy**

Think of Airflow as your **mobile phone**.

* Default phone apps = Airflow core (DAGs, scheduler, etc.).
* Providers = apps you install (WhatsApp, Uber, Instagram).
* You install only the ones you need.

# **3.4 Create a Table**

### **Simple Explanation**

* In Airflow, when we say **“Create a Table”**, it usually means creating a table in a **database** (Postgres, MySQL, etc.) using an **Operator**.
* We don’t create tables inside Airflow itself — instead, Airflow **sends SQL commands** to the database through a connection.

So the steps are:

1. Install the provider (e.g., Postgres provider).
2. Create a **connection** in Airflow UI (to your DB).
3. Use an **Operator** (like PostgresOperator) to send SQL.

### **Example: Create a Users Table in Postgres**

from airflow import DAG

from airflow.providers.postgres.operators.postgres import PostgresOperator

from datetime import datetime

# Define the DAG

with DAG(

dag\_id="create\_users\_table\_dag",

start\_date=datetime(2025, 1, 1),

schedule\_interval=None, # manual run

catchup=False,

) as dag:

create\_table = PostgresOperator(

task\_id="create\_users\_table",

postgres\_conn\_id="my\_postgres\_conn", # connection set in Airflow UI

sql="""

CREATE TABLE IF NOT EXISTS users (

id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

signup\_date DATE

);

"""

)

Here:

* PostgresOperator runs SQL in Postgres.
* postgres\_conn\_id="my\_postgres\_conn" → refers to a connection you must set up in Airflow UI.
* The SQL creates a table named users.

### **Real-life Analogy**

Think of Airflow like a **remote control**.

* The **database** is your TV.
* You press a button (operator) → command (SQL) goes to TV (database) → channel changes (table created).

# **3.5 Create a Connection**

### **Simple Explanation**

* **Airflow connections** are like stored credentials to external systems (databases, APIs, cloud storage, etc.).
* Instead of hardcoding username/password/host in code, you create a connection in Airflow UI and just reference it by conn\_id.

So when your DAG says postgres\_conn\_id="my\_postgres\_conn", Airflow looks into its connection store to know:

* host = localhost
* port = 5432
* username = airflow
* password = airflow
* database = mydb

### **How to Create a Connection in Airflow UI**

1. Go to **Airflow UI** → **Admin** → **Connections**.
2. Click **+** (Add a new connection).
3. Fill details:  
   * **Conn Id**: my\_postgres\_conn
   * **Conn Type**: Postgres
   * **Host**: localhost
   * **Schema**: mydb
   * **Login**: airflow
   * **Password**: airflow
   * **Port**: 5432
4. Save ✅

Now, any DAG can use this connection by referring to postgres\_conn\_id="my\_postgres\_conn".

### **Alternative (via CLI)**

You can also create a connection using Airflow CLI:

airflow connections add 'my\_postgres\_conn' \

--conn-type 'postgres' \

--conn-host 'localhost' \

--conn-login 'airflow' \

--conn-password 'airflow' \

--conn-schema 'mydb' \

--conn-port '5432’

### **Real-life Analogy**

Think of **connection** like a **saved Wi-Fi network** on your phone.

* You don’t enter password every time.
* You just choose the Wi-Fi name → phone already knows the password.
* Similarly, DAGs just reference conn\_id, and Airflow knows the details.

# **3.6 What is a Sensor?**

### **Simple Explanation**

* A **Sensor** in Airflow is a special kind of **Operator** that waits for something to happen before moving forward.
* It **pauses execution** until a certain condition is met.

Example:

* Wait for a file to arrive in S3
* Wait for a table to exist in a database
* Wait for an API endpoint to return data

So, while a **normal operator** “does something”, a **sensor** “waits for something”.

### **Types of Sensors**

1. **FileSensor** → Waits until a file exists.

from airflow.sensors.filesystem import FileSensor

wait\_for\_file = FileSensor(

task\_id="wait\_for\_file",

filepath="/opt/airflow/data/input.csv",

poke\_interval=30, # check every 30 seconds

timeout=600, # fail after 10 minutes

mode="poke" # poke = active checking, reschedule = efficient

)

1. **S3KeySensor** → Waits for a file/object in S3.
2. **ExternalTaskSensor** → Waits for a task in another DAG to complete.

### **Sensor Parameters**

* **poke\_interval** → How often to check (default 60s).
* **timeout** → Maximum wait time.
* **mode**:  
  + poke → keeps checking continuously (uses resources).
  + reschedule → releases worker slot until next check (better for long waits).

### **Real-life Analogy**

Think of a **Sensor** like **waiting at a railway crossing**:

* The gate is closed until the train passes.
* Once the condition (train passing) is met → the gate opens and you move ahead.

# **3.7 Create the Connection user\_api**

### **Why do we need connections in Airflow?**

* Airflow tasks often interact with **external systems** (APIs, databases, cloud storage).
* To avoid hardcoding credentials in DAGs, Airflow uses the **Connections** feature.
* Connections are **stored in Airflow’s metadata DB** and managed through the UI, CLI, or environment variables.

### **Creating a Connection in Airflow UI**

1. Go to **Airflow UI → Admin → Connections**
2. Click **“+”** to add a new connection.
3. Fill in details:  
   * **Conn Id** → user\_api
   * **Conn Type** → HTTP
   * **Host** → https://jsonplaceholder.typicode.com (a free fake API we can use for practice)
   * **Extra** → leave blank

This way, any DAG can use user\_api without rewriting API details.

### **Example: Using the user\_api connection in a DAG**

from airflow import DAG

from airflow.providers.http.operators.http import SimpleHttpOperator

from airflow.utils.dates import days\_ago

with DAG(

dag\_id="user\_api\_example",

start\_date=days\_ago(1),

schedule\_interval=None,

catchup=False

) as dag:

get\_users = SimpleHttpOperator(

task\_id="get\_users",

http\_conn\_id="user\_api", # uses the connection we created

endpoint="users", # API endpoint

method="GET",

log\_response=True

)

This will call:  
https://jsonplaceholder.typicode.com/users

### **Real-life Analogy**

Think of an Airflow **Connection** like saving a **Wi-Fi password on your laptop**.

* Once saved, you don’t need to type it every time.
* Similarly, Airflow DAGs just say “use user\_api”, and all details (URL, credentials) are fetched automatically.

## **3.8 Extract Users**

Once the connection is created, the next step is to **pull data from the external API**.

We use **SimpleHttpOperator** for this. It allows us to make HTTP requests inside a DAG.

Example:

from airflow.providers.http.operators.http import SimpleHttpOperator

extract\_users = SimpleHttpOperator(

task\_id="extract\_users",

http\_conn\_id="user\_api", # Connection created in UI

endpoint="users", # API endpoint

method="GET", # HTTP Method

log\_response=True

)

**Explanation:**

* http\_conn\_id="user\_api" → tells Airflow to use the connection we set up earlier.
* endpoint="users" → means we are calling https://jsonplaceholder.typicode.com/users.
* method="GET" → fetches the data.
* log\_response=True → prints response in Airflow logs for debugging.

## **3.9 Process Users**

After extraction, the raw data needs **processing** (cleaning, selecting fields, transforming).

We use a **PythonOperator** for transformation logic.

Example:

from airflow.operators.python import PythonOperator

import json

def process\_users\_func(ti):

response = ti.xcom\_pull(task\_ids="extract\_users") # Pull API response

users = json.loads(response) # Convert JSON string to Python list

processed = [

{"id": u["id"], "name": u["name"], "email": u["email"]}

for u in users

]

return processed

process\_users = PythonOperator(

task\_id="process\_users",

python\_callable=process\_users\_func

)

**Explanation:**

* xcom\_pull() → used to fetch data returned by another task (extract\_users).
* json.loads(response) → converts raw JSON into Python list/dict.
* processed → only keeps **id, name, email** fields (ignores unnecessary fields).
* return processed → sends processed data to the next task via XCom.

## **3.10 Before Running Process Users**

We must **set task dependencies** so that process\_users runs **after** extract\_users.

Example:

extract\_users >> process\_users

This ensures Airflow **waits for extraction before processing**.

## **3.11 What is a Hook?**

* A **Hook** is a low-level interface that provides connectivity to an external system (API, DB, cloud).
* Operators use Hooks internally.
* Example:  
  + HttpHook → connect to APIs.
  + PostgresHook → connect to Postgres DB.
  + S3Hook → connect to AWS S3.

Example usage:

from airflow.hooks.base import BaseHook

conn = BaseHook.get\_connection("user\_api")

print(conn.host) # prints: https://jsonplaceholder.typicode.com

**Explanation:**

* We can directly access **connection details** inside DAG code.
* Hooks are useful if we want to write **custom Operators** or make direct API calls.

## **3.12 Store Users**

Final step: store processed users into a **database table**.  
 We can use **PostgresOperator** to insert data.

Example:

from airflow.providers.postgres.operators.postgres import PostgresOperator

store\_users = PostgresOperator(

task\_id="store\_users",

postgres\_conn\_id="postgres\_default",

sql="""

INSERT INTO users (id, name, email)

VALUES (

{{ ti.xcom\_pull(task\_ids='process\_users')[0]['id'] }},

'{{ ti.xcom\_pull(task\_ids='process\_users')[0]['name'] }}',

'{{ ti.xcom\_pull(task\_ids='process\_users')[0]['email'] }}'

);

"""

)

**Explanation:**

* postgres\_conn\_id="postgres\_default" → Airflow’s default Postgres connection.
* sql → inserts values fetched from process\_users output (using Jinja templates + XCom).
* This saves each user record into the database.

## **3.13 DAG Scheduling**

Scheduling decides **when and how often** a DAG runs.

Options:

* **Presets**:  
  + @once → runs only once.
  + @daily → runs once per day.
  + @hourly → runs once every hour.
* **Cron expression**:  
  + "0 9 \* \* \*" → run every day at 9 AM.
  + "\*/15 \* \* \* \*" → run every 15 minutes.
* **None** → DAG runs only when triggered manually.

Example:

with DAG(

dag\_id="user\_pipeline",

start\_date=days\_ago(1),

schedule\_interval="@daily",

catchup=False

) as dag:

extract\_users >> process\_users >> store\_users