**Challenge: COE\_DAAI\_DATA\_ENGINEER\_INTERVIEW\_CHALLENGE**

**Candidate:** André Luis Monteiro Pinto

**Company:** Bridge351

The desired process flow looks as follows:

1. A scheduled Airflow DAG executes a preparatory task,
2. Airflow triggers a processor in Apache NiFi,
3. NiFi executes an ETL process,
4. Airflow waits for NiFi to finish,
5. Airflow continues with some other task.

Python code is supplied wherever necessary for understanding the text, but you can view the entire codebase (including sub-functions not shown in the text) on [GitHub](https://github.com/andreluispinto/nifi_airflow).

**Thoughts on the infrastructure and separation of concerns**

While NiFi does have the option to schedule processors with CRON strings, it is usually a bad idea to schedule jobs within NiFi itself — unless you don’t have another choice. Using Airflow we can monitor and schedule all of our tasks, wherever they may be, in one single place with a simple and good-looking interface, access to logs and highly customizable scheduling pipelines which can interact with, in-/exclude or depend on each other.

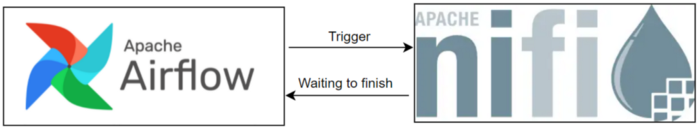
Similarly, while Airflow can also also execute ETL-tasks (for example coded in Python) that should ideally be implemented in NiFi, you *really* shouldn’t use Airflow to do so. On the one hand Airflow is built to be a monitoring and scheduling software, on the other hand, we would loose all of NiFi’s inherent advantages concerning data extraction, transformation and loads.

Using Airflow solely as scheduler and letting NiFi do the heavy lifting in our backend, we get the best of both worlds: Airflow’s ability to author, schedule and monitor workflows with NiFi’s scalable, powerful and highly configurable directed graphs to route and transform data. Furthermore, the colleagues will also thank you if you don’t create 100% CPU utilisation and an overloaded RAM by executing data-heavy tasks on the EC2 instance meant for scheduling tasks.

**General structure**

We have two main points of contact:

* **Startup/Trigger**: the DAG will trigger a starting point of NiFi’s ETL pipeline.
* **Waiting to finish**: the DAG will need to get a signal whenever NiFi has finished its part of our overall pipeline.



Connection of systems — Image created by Author.

We will start with configuring NiFi in the next chapter, so that we have everything set up once we write the Airflow DAG and need to specify processor IDs.

**NiFi configuration**

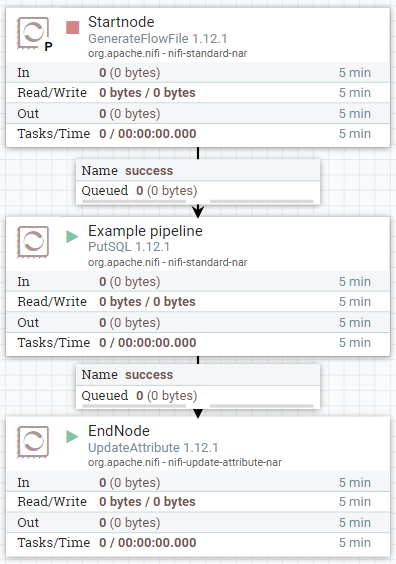
We need two obligatory processors:

* A GenerateFlowFile processor to act as a starting node of our pipeline which can be triggered from Airflow.
* An UpdateAttribute processor to act as the end node of our pipeline whose state can be queried by Airflow.

We implement our pipeline, consisting of any and however many processors we desire, between these two processors. To keep this example short, we use a single PutSQL processor as stand-in for the entire ETL-pipeline.

In the default mode the starting node should be switched off while all subsequent processors are running, ready to be triggered with a flowfile. Instead of a GenerateFlowFile processor, you could also use a GenerateTableFetch processor — or any other processor that creates flowfiles once switched on.

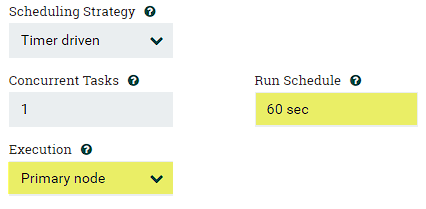
Pipeline in NiFi could look something like this:



Processor setup — Image created by Author.

**StartNode**

To be able to trigger our starting node from Airflow and to create **exactly one** flowfile (we don’t want to trigger the pipeline multiple times) we have to configure the processor as follows:



Processor configs — Image created by Author.

By setting Executionto the **primary node**, we ensure only one node executes the processor. By setting Run Scheduleto **60 seconds** we give the Airflow DAG enough time to stop the processor before a second flow file is created.

**EndNode**

In the UpdateAttributeprocessor

* set Store Stateto Store state locally and
* add a custom property named last\_tms with the value ${now()}.



Processor configs — Image created by Author.

Whenever a flowfile passes through the processor, the code now() will be executed and its result — a timestamp — will be stored in the property last\_tms of the processor’s state.

We can check this by running our pipeline manually (switch the GenerateFlowFile processor on, wait for a flowfile to be created and switch it off again) and then access the NiFi-API in the browser under our.cluster.adress.com:9443/nifi-api/processors/{id}/state. The following JSON is an example showing you the current state of our processor:

Now we have everything we need in NiFi — a startnode processor, an endnode processor and whatever we may want to pack in between the two: data extraction, transformation and/or load tasks.

**Airflow configuration**

The Airflow DAG will consist of four tasks: the two in the middle are interacting with Apache NiFi’s API. The first and the last are stand-ins for other operations we might want to schedule / execute from Airflow to prepare or finalize tasks.

https://miro.medium.com/max/479/1*EaneVG3Bjrr5ix0U__D-vg.png

Airflow task dependencies — Image created by Author.

To interact with the NiFi API you can either write the own code and API calls (the entire API is documented in depth [here](https://nifi.apache.org/docs/nifi-docs/rest-api/index.htm)), or you can make use of packages like [nipyapi](https://pypi.org/project/nipya" \t "_blank). For the sake of brevity and to stay independent of any package’s implementation, I wrote my own API calls for this text.

**Initial task**

This is merely a stand-in for some other tasks we might want to do as preparation — in essence this task can also be left out.

**Startup task**

The startup tasks consists of three steps

* Set the GenerateFlowFile NiFi processor to RUNNING.
* Wait 15 Seconds (to give the processor time to create a flowfile).
* Set the GenerateFlowFile NiFi processor to STOPPED.

You can change a processor’s state by retrieving the current state with a GETrequest from /nifi-api/processors/{id}, locally setting the state in a custom JSON and putting with a PUT request to /nifi-api/processors/{id}/run-status

In order to focus on the essential structure of the logic at hand I excluded sub-functions, such as get\_token() and update\_processor\_status() . The task’s python code is as follows:

**Waiting task**

For Airflow to notice when NiFi has finished the ETL operations, we need to continually query nifi-api/processors/{id}/state and parse the resulting JSON for the value of last\_tmsuntil a change in the state appears. We do this in a while-loop by checking the API every 60 seconds:

The parse\_state() as well as all other sub-functions can be found in the public [GitHub repository](https://github.com/andreluispinto/nifi_airflow).

**Continuation task**

The last task as a stand-in for whatever code you want to execute after the NiFi pipeline.

**Airflow DAG**

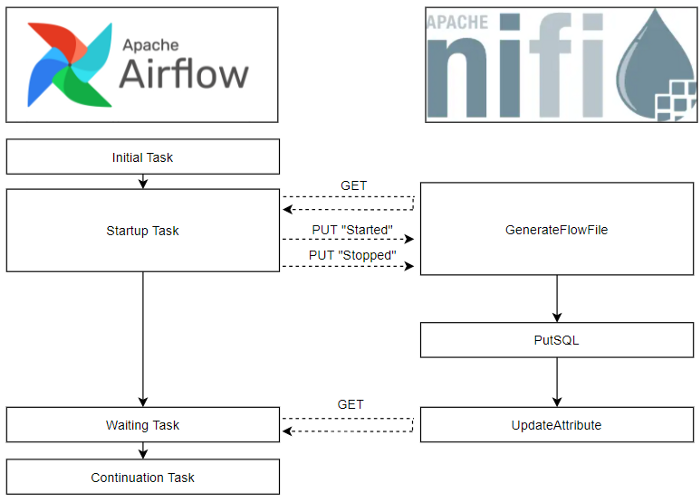
The DAG consists of the above functions and we merely need to configure their depencencies. Since our tasks are supposed to run one after the other, the DAG is straight as an arrow.

Depending in the purpose you will want to edit the DAG-parameters, but the following is a good start:

**Overview and closing thoughts**

The graphic below presents the procedural flow of our scripts as well as the interactions between the systems. After some preparatory tasks in Airflow, our NiFi pipeline is triggered, Airflow waits until the pipeline is finished and then continues with some other task.

While there are other ways of connecting Airflow & NiFi, the one presented here is an easy way without much overhead. Once set up, it is easy to expand to other use cases, integrate data pipelines without changing the overall setup and add / remove additional parts in Airflow or NiFi without interfering with the connection between the two parts.



Interconnection and process flow — Image created by Author.

I will present a structure and go through the logic of how to insert a NiFi ETL-pipeline into the scheduled flow of an Airflow DAG.

We can now also replace the dummy PutSQL processor in our NiFi pipeline for whatever processors we need — we just need to make sure that a flowfile reaches the UpdateAttribute processor when we are ready to let Airflow continue its work. Similarly, we can add content to the initial task or the continuation task in Airflow.

from airflow import DAG

from airflow.operators.python\_operator import PythonOperator

from airflow.utils.dates import days\_ago

from src.nifi.get\_token import get\_token

from src.nifi.update\_processor\_status import update\_processor\_status

from src.nifi.get\_processor\_state import get\_processor\_state

from src.utils.parse\_state import parse\_state

from src.utils.pause import pause

def prepare():

"""Where something happens before the NiFi pipeline is triggered."""

pass

def startup():

# Initialize the following variables according to the setup / needs:

url\_nifi\_api = "https://the.cluster.address.com:9443/nifi-api/"

processor\_id = (

"" # e.g. hardcoded / pass them via the `provide\_context` functionality

)

access\_payload = {

"username": "",

"password": "",

} # e.g. retrieve via Airflow's `BaseHook` functionality

token = get\_token(url\_nifi\_api, access\_payload)

response = update\_processor\_status(processor\_id, "RUNNING", token, url\_nifi\_api)

print(response)

pause(15) # wait for 15 seconds to give NiFi time to create a flow file

response = update\_processor\_status(processor\_id, "STOPPED", token, url\_nifi\_api)

print(response)

def wait\_for\_update():

# Initialize the following variables according to the setup / needs:

url\_nifi\_api = "https://the.cluster.address.com:9443/nifi-api/"

processor\_id = "" # e.g. pass them via the DAG's `provide\_context` functionality

access\_payload = "" # e.g. retrieve the via Airflow's `BaseHook` functionality

timestamp\_property = "last\_tms" # the processor's attribute name

token = get\_token(url\_nifi\_api, access\_payload)

# Get current timestamp

processor\_state = get\_processor\_state(url\_nifi\_api, processor\_id, token=token)

value\_start = parse\_state(processor\_state, timestamp\_property)

# query and wait until an update happens or we time out.

while True:

processor\_state = get\_processor\_state(url\_nifi\_api, processor\_id, token=token)

value\_current = parse\_state(processor\_state, timestamp\_property)

if value\_start == value\_current:

print("Waiting...")

pause(60)

else:

print(f"Update found: {value\_current}")

break

def finalize():

pass

with DAG(

dag\_id="my\_dag\_name",

schedule\_interval=None,

start\_date=days\_ago(2),

catchup=False,

) as dag:

preparation = PythonOperator(

task\_id="preparation",

python\_callable=prepare,

)

startup\_task = PythonOperator(

task\_id="startup\_task",

python\_callable=startup,

)

waiting\_task = PythonOperator(

task\_id="waiting\_task",

python\_callable=wait\_for\_update,

)

finalization = PythonOperator(

task\_id="finalization",

python\_callable=finalize,

)

preparation >> startup\_task >> waiting\_task >> finalization

Find out more about Airflow and Nifi here:

* [Apache Airflow](https://airflow.apache.org/)
* [Apache NiFi](https://nifi.apache.org/)

**SQL**

Write the SQL statements that allow you to answer the following questions

• Total number of rows;

SELECT COUNT(\*) FROM table\_name;

• Number of distinct sensors present on the database;

**SELECT COUNT (DISTINCT value) FROM table\_name;**

• Number of rows for the sensor PPL340;

**SELECT COUNT(\*) FROM table\_name WHERE name="PPL340";**

• The number of rows by year for the sensor PPL340;

**SELECT year, COUNT(\*) FROM table\_name GROUP BY year WHERE name="PPL340";**

• Average number of readings by year for the sensor PPL340;

**SELECT AVG(value) FROM table\_name GROUP BY year WHERE name=”PPL340”;**

• For PPL340, Identify the years in which the number of readings is less than the average;

**SELECT**

**value**

**FROM table\_name**

**ON name= “PPL340”**

**GROUP BY**

**year**

**HAVING**

**value < AVG(value)**

**SPARK**

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result, the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

All of the examples on this page use sample data included in the Spark distribution and can be run in the spark-shell, pyspark shell, or sparkR shell.

## SQL

One use of Spark SQL is to execute SQL queries. Spark SQL can also be used to read data from an existing Hive installation. For more on how to configure this feature, please refer to the [Hive Tables](https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html) section. When running SQL from within another programming language the results will be returned as a [Dataset/DataFrame](https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes). You can also interact with the SQL interface using the [command-line](https://spark.apache.org/docs/latest/sql-distributed-sql-engine.html#running-the-spark-sql-cli) or over [JDBC/ODBC](https://spark.apache.org/docs/latest/sql-distributed-sql-engine.html#running-the-thrift-jdbcodbc-server).

## Datasets and DataFrames

A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL’s optimized execution engine. A Dataset can be [constructed](https://spark.apache.org/docs/latest/sql-getting-started.html#creating-datasets) from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). The Dataset API is available in [Scala](https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Dataset.html) and [Java](https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html). Python does not have the support for the Dataset API. But due to Python’s dynamic nature, many of the benefits of the Dataset API are already available (i.e. you can access the field of a row by name naturally row.columnName). The case for R is similar.

A DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of [sources](https://spark.apache.org/docs/latest/sql-data-sources.html) such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, [Python](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame), and [R](https://spark.apache.org/docs/latest/api/R/index.html). In Scala and Java, a DataFrame is represented by a Dataset of Rows. In [the Scala API](https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Dataset.html), DataFrame is simply a type alias of Dataset[Row]. While, in [Java API](https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html), users need to use Dataset<Row> to represent a DataFrame.

**BASH**

# Using the general form:

find /root/Maildir/ -mindepth 1 -type f -mtime +180 | xargs rm

**Data Building Tool – dbt**

dbt is a transformation workflow that helps you get more work done while producing higher quality results. You can use dbt to modularize and centralize your analytics code, while also providing your data team with guardrails typically found in software engineering workflows. Collaborate on data models, version them, and test and document your queries before safely deploying them to production, with monitoring and visibility.

dbt compiles and runs your analytics code against your data platform, enabling you and your team to collaborate on a single source of truth for metrics, insights, and business definitions. This single source of truth, combined with the ability to define tests for your data, reduces errors when logic changes, and alerts you when issues arise.

Read more about why we want to enable analysts to work more like software engineers in [The dbt Viewpoint](https://docs.getdbt.com/community/resources/viewpoint).

## dbt optimizes workflow[​](https://docs.getdbt.com/docs/introduction#dbt-optimizes-your-workflow)s

* Avoid writing boilerplate [DML](https://docs.getdbt.com/terms/dml) and [DDL](https://docs.getdbt.com/terms/ddl) by managing transactions, dropping tables, and managing schema changes. Write business logic with just a SQL select statement, or a Python DataFrame, that returns the dataset you need, and dbt takes care of [materialization](https://docs.getdbt.com/terms/materialization).
* Build up reusable, or modular, data models that can be referenced in subsequent work instead of starting at the raw data with every analysis.
* Dramatically reduce the time your queries take to run: Leverage metadata to find long-running models that you want to optimize and use [incremental models](https://docs.getdbt.com/docs/build/incremental-models) which dbt makes easy to configure and use.
* Write [DRY](https://docs.getdbt.com/terms/dry)er code by leveraging [macros](https://docs.getdbt.com/docs/build/jinja-macros), [hooks](https://docs.getdbt.com/docs/build/hooks-operations), and [package management](https://docs.getdbt.com/docs/build/packages).

## dbt provides more reliable analysis[​](https://docs.getdbt.com/docs/introduction#dbt-provides-more-reliable-analysis)

* No longer copy and paste SQL, which can lead to errors when logic changes. Instead, build reusable data models that get pulled into subsequent models and analysis. Change a model once and that change will propagate to all its dependencies.
* Publish the canonical version of a particular data model, encapsulating all complex business logic. All analysis on top of this model will incorporate the same business logic without needing to reimplement it.
* Use mature source control processes like branching, pull requests, and code reviews.
* Write data quality tests quickly and easily on the underlying data. Many analytic errors are caused by edge cases in the data: testing helps analysts find and handle those edge cases.

## dbt products[​](https://docs.getdbt.com/docs/introduction#dbt-products)

You can access dbt using dbt Core or dbt Cloud. dbt Cloud is built around dbt Core, but it also provides:

* Web-based UI so it’s more accessible
* Hosted environment so it’s faster to get up and running
* Differentiated features, such as metadata, in-app job scheduler, observability, integrations with other tools, integrated development environment (IDE), and more.

You can learn about dbt on [www.getdbt.com](https://www.getdbt.com/pricing/).

### dbt Cloud[​](https://docs.getdbt.com/docs/introduction#dbt-cloud)

dbt Cloud is the fastest and most reliable way to deploy dbt. Develop, test, schedule, and investigate data models all in one web-based UI. Read more about [Getting started with dbt Cloud](https://docs.getdbt.com/docs/get-started/getting-started/set-up-dbt-cloud) and [dbt Cloud features](https://docs.getdbt.com/docs/get-started/dbt-cloud-features).

### dbt Core[​](https://docs.getdbt.com/docs/introduction#dbt-core)

dbt Core is an open-source tool that enables data teams to transform data using analytics engineering best practices. You can install and use dbt Core on the command line. Read more about [Getting started with dbt Core](https://docs.getdbt.com/docs/get-started/getting-started-dbt-core).

## The power of dbt[​](https://docs.getdbt.com/docs/introduction" \l "the-power-of-dbt" \o "Direct link to heading)

As a dbt user, your main focus will be on writing models (i.e. select queries) that reflect core business logic – there’s no need to write boilerplate code to create tables and views, or to define the order of execution of your models. Instead, dbt handles turning these models into objects in your warehouse for you.

| **Feature** | **Description** |
| --- | --- |
| Handle boilerplate code to materialize queries as relations | For each model you create, you can easily configure a materialization. A materialization represents a build strategy for your select query – the code behind a materialization is robust, boilerplate SQL that wraps your select query in a statement to create a new, or update an existing, relation. Read more about [Materializations](https://docs.getdbt.com/docs/build/materializations). |
| Use a code compiler | SQL files can contain Jinja, a lightweight templating language. Using Jinja in SQL provides a way to use control structures in your queries. For example, if statements and for loops. It also enables repeated SQL to be shared through macros. Read more about [Macros](https://docs.getdbt.com/docs/build/jinja-macros). |
| Determine the order of model execution | Often, when transforming data, it makes sense to do so in a staged approach. dbt provides a mechanism to implement transformations in stages through the [ref function](https://docs.getdbt.com/reference/dbt-jinja-functions/ref). Rather than selecting from existing tables and views in your warehouse, you can select from another model. |
| Document your dbt project | dbt provides a mechanism to write, version-control, and share documentation for your dbt models. You can write descriptions (in plain text or markdown) for each model and field. In dbt Cloud, you can auto-generate the documentation when your dbt project runs. Read more about the [Documentation](https://docs.getdbt.com/docs/collaborate/documentation). |
| Test your models | Tests provide a way to improve the integrity of the SQL in each model by making assertions about the results generated by a model. Read more about writing tests for your models [Testing](https://docs.getdbt.com/docs/build/tests) |
| Manage packages | dbt ships with a package manager, which allows analysts to use and publish both public and private repositories of dbt code which can then be referenced by others. Read more about [Package Management](https://docs.getdbt.com/docs/build/packages). |
| Load seed files | Often in analytics, raw values need to be mapped to a more readable value (for example, converting a country-code to a country name) or enriched with static or infrequently changing data. These data sources, known as seed files, can be saved as a CSV file in your project and loaded into your data warehouse using the seed command. Read more about [Seeds](https://docs.getdbt.com/docs/build/seeds). |
| Snapshot data | Often, records in a data source are mutable, in that they change over time. This can be difficult to handle in analytics if you want to reconstruct historic values. dbt provides a mechanism to snapshot raw data for a point in time, through use of [snapshots](https://docs.getdbt.com/docs/build/snapshots). |

# Run your dbt projects

You can run your dbt projects with [dbt Cloud](https://docs.getdbt.com/docs/get-started/dbt-cloud-features) and [dbt Core](https://github.com/dbt-labs/dbt-core" \t "_blank). dbt Cloud is a hosted application where you can develop directly from a web browser. dbt Core is an open source project where you can develop from the command line.

Among other features, dbt Cloud provides a development environment to help you build, test, run, and [version control](https://docs.getdbt.com/docs/collaborate/git-version-control) your project faster. It also includes an easier way to share your [dbt project's documentation](https://docs.getdbt.com/docs/collaborate/build-and-view-your-docs) with your team. These development tasks are directly built into dbt Cloud for an integrated development environment (IDE). Refer to [Develop in the Cloud](https://docs.getdbt.com/docs/get-started/develop-in-the-cloud) for more details.

With dbt Core, you can run your dbt projects from the command line. The command line interface (CLI) is available from your computer's terminal application such as Terminal and iTerm. When using the command line, you can run commands and do other work from the current working directory on your computer. Before running the dbt project from the command line, make sure you are working in your dbt project directory. Learning terminal commands such as cd (change directory), ls (list directory contents), and pwd (present working directory) can help you navigate the directory structure on your system.

When running your project from dbt Core or dbt Cloud, the commands you commonly use are:

* [dbt run](https://docs.getdbt.com/reference/commands/run) — Runs the models you defined in your project
* [dbt build](https://docs.getdbt.com/reference/commands/build) — Builds and tests your selected resources such as models, seeds, snapshots, and tests
* [dbt test](https://docs.getdbt.com/reference/commands/test) — Executes the tests you defined for your project

For information on all dbt commands and their arguments (flags), see the [dbt command reference](https://docs.getdbt.com/reference/dbt-commands). If you want to list all dbt commands from the command line, run dbt --help. To list a dbt command’s specific arguments, run dbt COMMAND\_NAME --help .

**Python (Database extraction)**

#### **01.Import the pyodbc module and create a connection to the database.**

#### **02.Execute an INSERT statement to test the connection to the database.**

#### **03.Retrieve a result set from a query, iterate over it and print out all records.**

import pyodbc

cnxn = pyodbc.connect('DRIVER={Devart ODBC Driver for SQLite};Direct=True;Database=mydatabase')

cursor = cnxn.cursor()

cursor.execute("INSERT INTO EMP (EMPNO, ENAME, JOB, MGR) VALUES (535, 'Scott', 'Manager', 545)")

cursor.execute("SELECT \* FROM EMP")

row = cursor.fetchone()

while row:

print (row)

row = cursor.fetchone()

import mysql.connector

try:

connection = mysql.connector.connect(host='localhost',

database='electronics',

user='pynative',

password='pynative@#29')

sql\_select\_Query = • “SELECT \* FROM table WHERE timestamp >= '2020-01-01 00:00:00' AND timestamp < '2020-01-01 01:00:00'“.

cursor = connection.cursor()

cursor.execute(sql\_select\_Query)

# get all records

records = cursor.fetchall()

print("Total number of rows in table: ", cursor.rowcount)

print("\nPrinting each row")

for row in records:

print("timestamp = ", row[0], )

print("sensor\_name = ", row[1])

print("value = ", row[2], "\n")

except mysql.connector.Error as e:

print("Error reading data from MySQL table", e)

finally:

if connection.is\_connected():

connection.close()

cursor.close()

print("MySQL connection is closed")

ODBC (Open Database Connectivity) — is an Application Programming Interface (API) designed to access data storage. Many databases are supplied with an ODBC drivers, so you can use any of them with the pyodbc interface to access databases such as SQL Server, Oracle, PostgreSQL as well or cloud applications such as Streak, Zoho CRM, BigCommerce, etc. from your Python application.