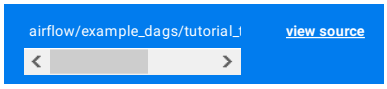
Tutorial on the Taskflow API

This tutorial builds on the regular Airflow Tutorial and focuses specifically on writing data pipelines using the Taskflow API paradigm which is introduced as part of Airflow 2.0 and contrasts this with DAGs written using the traditional paradigm.

The data pipeline chosen here is a simple ETL pattern with three separate tasks for Extract, Transform, and Load.

Example “Taskflow API” ETL Pipeline

Here is very simple ETL pipeline using the Taskflow API paradigm. A more detailed explanation is given below.



```
import json

from airflow.decorators import dag,
from airflow.utils.dates import days
# These args will get passed on to e
# You can override them on a per-tas
default_args = {
    'owner': 'airflow',
}
@dag(default_args=default_args, sche
def tutorial_taskflow_api_etl():
    """
    ### TaskFlow API Tutorial Docume
    This is a simple ETL data pipeli
    the TaskFlow API using three sim
    Documentation that goes along wi
    located
    [here](https://airflow.apache.or
    """
    @task()
    def extract():
        """
        #### Extract task
        A simple Extract task to get
        pipeline. In this case, gett
        hardcoded JSON string.
        """
        data_string = '{"1001": 301.

        order_data_dict = json.loads
        return order_data_dict
    @task(multiple_outputs=True)
    def transform(order_data_dict: d
        """
        #### Transform task
        A simple Transform task whic
        computes the total order val
        """
        total_order_value = 0

        for value in order_data_dict
            total_order_value += val

        return {"total_order_value":
    @task()
    def load(total_order_value: floa
        """
        #### Load task
        A simple Load task which tak
        instead of saving it to end
        """

        print("Total order value is:
    order_data = extract()
    order_summary = transform(order_
    load(order_summary["total_order_
tutorial_etl_dag = tutorial_taskflow
```

It's a DAG definition file

If this is the first DAG file you are looking at, please note that this Python script is interpreted by Airflow and is a configuration file for your data pipeline. For a complete introduction to DAG files, please look at the core Airflow tutorial which covers DAG structure and definitions extensively.

Instantiate a DAG

We are creating a DAG which is the collection of our tasks with dependencies between the tasks. This is a very simple definition, since we just want the DAG to be run when we set this up with Airflow, without any retries or complex scheduling. In this example, please notice that we are creating this DAG using the

`@dag` decorator as shown below, with the python function name acting as the DAG identifier.

airflow/example_dags/tutorial_t view source

```
@dag(default_args=default_args, sche
def tutorial_taskflow_api_etl():
    """
    ### TaskFlow API Tutorial Docume
    This is a simple ETL data pipeli
    the TaskFlow API using three sim
    Documentation that goes along wi
    located
    [here](https://airflow.apache.or
    """
```

Tasks

In this data pipeline, tasks are created based on Python functions using the `@task` decorator as shown below. The function name acts as a unique identifier for the task.

airflow/example_dags/tutorial_t view source

```
    @task()
    def extract():
        """
        #### Extract task
        A simple Extract task to get
        pipeline. In this case, getti
        hardcoded JSON string.
        """
        data_string = '{"1001": 301.

        order_data_dict = json.loads
        return order_data_dict
```

The returned value, which in this case is a dictionary, will be made available for use in later tasks.

The Transform and Load tasks are created in the same manner as the Extract task shown above.

Main flow of the DAG

Now that we have the Extract, Transform, and Load tasks defined based on the Python functions, we can move to the main part of the DAG.

airflow/example_dags/tutorial_t view source

```
    order_data = extract()
    order_summary = transform(order_
    load(order_summary["total_order_
```

That's it, we are done! We have invoked the Extract task, obtained the order data from there and sent it over to the Transform task for summarization, and then invoked the Load task with the summarized data. The dependencies between the tasks and the passing of data between these tasks which could be running on different workers on different nodes on the network is all handled by Airflow.

Now to actually enable this to be run as a DAG, we invoke the python function `tutorial_taskflow_api_etl` set up using the `@dag` decorator earlier, as shown below.

airflow/example_dags/tutorial_ view source

```
tutorial_etl_dag = tutorial_taskflow
```

But how?

For experienced Airflow DAG authors, this is startlingly simple! Let's contrast this with how this DAG had to be written before Airflow 2.0 below:

airflow/example_dags/tutorial_ view source

```
import json

# The DAG object; we'll need this to
from airflow import DAG

# Operators; we need this to operate
from airflow.operators.python import
from airflow.utils.dates import days
# These args will get passed on to e
# You can override them on a per-tas
default_args = {
    'owner': 'airflow',
}
with DAG(
    'tutorial_etl_dag',
    default_args=default_args,
    description='ETL DAG tutorial',
    schedule_interval=None,
    start_date=days_ago(2),
    tags=['example'],
) as dag:
    dag.doc_md = __doc__
    def extract(**kwargs):
        ti = kwargs['ti']
        data_string = '{"1001": 301
        ti.xcom_push('order_data', d
    def transform(**kwargs):
        ti = kwargs['ti']
        extract_data_string = ti.xco
        order_data = json.loads(extr

        total_order_value = 0
        for value in order_data.valu
            total_order_value += val

        total_value = {"total_order_
        total_value_json_string = js
        ti.xcom_push('total_order_va
    def load(**kwargs):
        ti = kwargs['ti']
        total_value_string = ti.xcom
        total_order_value = json.loa

        print(total_order_value)
    extract_task = PythonOperator(
        task_id='extract',
        python_callable=extract,
    )
    extract_task.doc_md = """\
#### Extract task
A simple Extract task to get data re
In this case, getting data is simula
This data is then put into xcom, so
"""

    transform_task = PythonOperator(
```

```
        task_id='transform',
        python_callable=transform,
    )
    transform_task.doc_md = """\
#### Transform task
A simple Transform task which takes
and computes the total order value.
This computed value is then put into
"""

    load_task = PythonOperator(
        task_id='load',
        python_callable=load,
    )
    load_task.doc_md = """\
#### Load task
A simple Load task which takes in th
from xcom and instead of saving it t
"""

    extract_task >> transform_task >
```

All of the processing shown above is being done in the new Airflow 2.0 dag as well, but it is all abstracted from the DAG developer.

Let's examine this in detail by looking at the Transform task in isolation since it is in the middle of the data pipeline. In Airflow 1.x, this task is defined as shown below:

airflow/example_dags/tutorial_ view source

```
    def transform(**kwargs):
        ti = kwargs['ti']
        extract_data_string = ti.xco
        order_data = json.loads(extr

        total_order_value = 0
        for value in order_data.valu
            total_order_value += val

        total_value = {"total_order_
        total_value_json_string = js
        ti.xcom_push('total_order_va
```

As we see here, the data being processed in the Transform function is passed to it using Xcom variables. In turn, the summarized data from the Transform function is also placed into another Xcom variable which will then be used by the Load task.

Contrasting that with Taskflow API in Airflow 2.0 as shown below.

airflow/example_dags/tutorial_ view source

```
    @task(multiple_outputs=True)
    def transform(order_data_dict: d
        """
        #### Transform task
        A simple Transform task whic
        computes the total order val
        """
        total_order_value = 0

        for value in order_data_dict
            total_order_value += val

        return {"total_order_value":
```

All of the Xcom usage for data passing between these tasks is abstracted away from the DAG author in Airflow 2.0. However, Xcom variables are used behind the scenes and can be viewed using the Airflow UI as necessary for debugging or DAG monitoring.

Similarly, task dependencies are automatically generated within TaskFlows based on the functional invocation of tasks. In Airflow 1.x, tasks had to be explicitly created and dependencies specified as shown below.

airflow/example_dags/tutorial_ view source

```
    extract_task = PythonOperator(
        task_id='extract',
        python_callable=extract,
    )
    extract_task.doc_md = """\
#### Extract task
A simple Extract task to get data re
In this case, getting data is simula
This data is then put into xcom, so
"""

    transform_task = PythonOperator(
        task_id='transform',
        python_callable=transform,
    )
    transform_task.doc_md = """\
#### Transform task
A simple Transform task which takes
and computes the total order value.
This computed value is then put into
"""

    load_task = PythonOperator(
        task_id='load',
        python_callable=load,
    )
    load_task.doc_md = """\
#### Load task
A simple Load task which takes in th
from xcom and instead of saving it t
"""

    extract_task >> transform_task >
```

In contrast, with the Taskflow API in Airflow 2.0, the invocation itself automatically generates the dependencies as shown below.

airflow/example_dags/tutorial_ view source

```
    order_data = extract()
    order_summary = transform(order_
    load(order_summary["total_order_
```

Multiple outputs inference

Tasks can also infer multiple outputs by using dict python typing.

```
@task
def identity_dict(x: int, y: int) -> Dict[str, int]:
    return {"x": x, "y": y}
```

By using the typing `Dict` for the function return type, the `multiple_outputs` parameter is automatically set to true.

Note, If you manually set the `multiple_outputs` parameter the inference is disabled and the parameter value is used.

Adding dependencies to decorated tasks from regular tasks

The above tutorial shows how to create dependencies between python-based tasks. However, it is quite possible while writing a DAG to have some pre-existing tasks such as **`BashOperator`** or **`FileSensor`** based tasks which need to be run first before a python-based task is run.

Building this dependency is shown in the code below:

```
@task()
    def extract_from_file():
    """
    #### Extract from file task
    A simple Extract task to get data ready for the rest of the data
    pipeline, by reading the data from a file into a pandas dataframe
    """
    order_data_file = '/tmp/order_data.csv'
    order_data_df = pd.read_csv(order_data_file)


file_task = FileSensor(task_id='check_file', filepath='/tmp/order_data.csv')
order_data = extract_from_file()

file_task >> order_data
```

In the above code block, a new python-based task is defined as `extract_from_file` which reads the data from a known file location. In the main DAG, a new `FileSensor` task is defined to check for this file. Please note that this is a Sensor task which waits for the file. Finally, a dependency between this Sensor task and the python-based task is specified.

What's Next?

You have seen how simple it is to write DAGs using the Taskflow API paradigm within Airflow 2.0. Please do read the Concepts page for detailed explanation of Airflow concepts such as DAGs, Tasks, Operators, etc, and the Python task decorator in particular.

More details about the Taskflow API, can be found as part of the Airflow Improvement Proposal AIP-31: "Taskflow API" for clearer/simpler DAG definition and specifically within the Concepts guide at Concepts - Taskflow API.

Previous Next

Was this entry helpful?