Introduction to Airflow in Python

A beginner's guide to the basic concepts of Apache Airflow

Black Raven (James Ng)
09 Aug 2020 · 25 min read

This is a memo to share what I have learnt in Apache Airflow, capturing the learning objectives as well as my personal notes. The course is taught by Mike Metzger from DataCamp, and it includes 4 chapters:

Chapter 1. Intro to Airflow
Chapter 2. Implementing Airflow DAGs
Chapter 3. Maintaining and monitoring Airflow workflows
Chapter 4. Building production pipelines in Airflow

A data engineer's job includes writing scripts, adding complex CRON tasks, and trying various ways to meet an ever-changing set of requirements to deliver data on schedule. Airflow can do all these while adding scheduling, error handling, and reporting.
This course will guide you in the basic concepts of Airflow and help you implement data engineering workflows in production. You'll implement many different data engineering tasks in a predictable and repeatable fashion.

Photo by Jacek Dylag on Unsplash

Chapter 1. Intro to Airflow

Data engineering is taking any action involving data and turning it into a reliable, repeatable, and maintainable process.

Workflow is a set of steps to accomplish a given data engineering task, such as downloading files, copying data, filtering information, writing to a database, etc.

Airflow is a platform to
· program workflows including: creation, scheduling, and monitoring
· implement workflow as DAGs (Directed Acyclic Graphs)
· be accessed via code, command-line (CLI), or web user interface (UI)

Running a task in Airflow

You've just started looking at using Airflow within your company and would like to try to run a task within the Airflow platform. You remember that you can use the `airflow run` command to execute a specific task within a workflow.

Note that an error while using `airflow run` will return `airflow.exceptions.AirflowException:` on the last line of output.

An Airflow DAG is set up for you with a dag_id of `etl_pipeline`. The task_id is `download_file` and the start_date is `2020-01-08`. Which command would you enter in the console to run the desired task?

- `airflow run dag task 2020-01-08`
- `airflow run etl_pipeline task 2020-01-08`
- `airflow run etl_pipeline download_file 2020-01-08`

Answer: airflow run etl_pipeline download_file 2020–01–08
Syntax: `airflow run <dag_id> <task_id> <start_date>`

Examining Airflow commands

While researching how to use Airflow, you start to wonder about the `airflow` command in general. You realize that by simply running `airflow` you can get further information about various sub-commands that are available.

Which of the following is *NOT* an Airflow sub-command?

- `list_dags`
- `edit_dag`
- `test`
- `scheduler`

Answer: edit_dag
You can use the `airflow -h` command to obtain further information about any Airflow command.

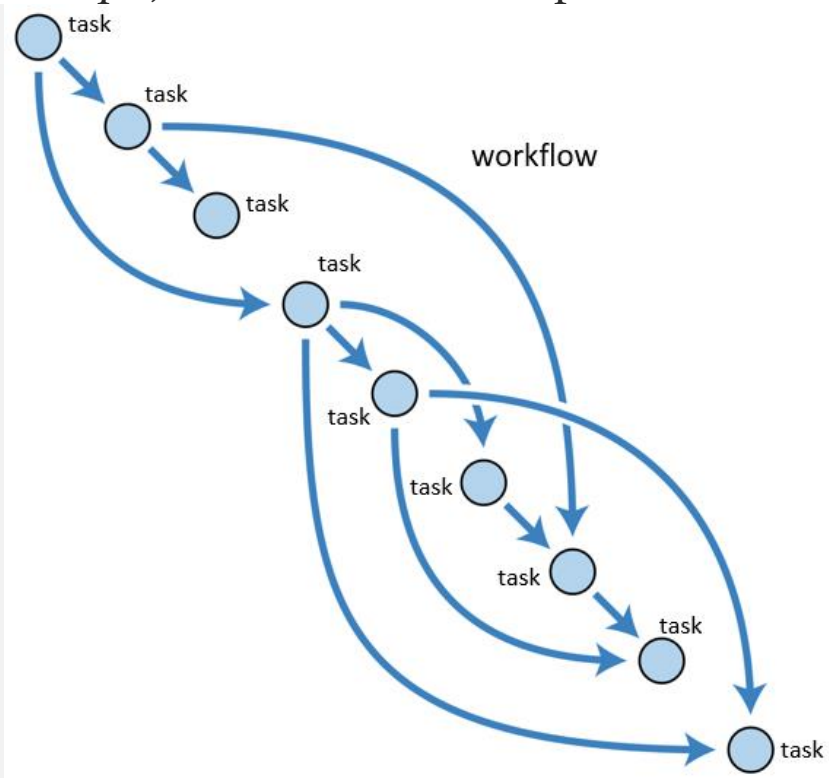
Airflow DAGs

DAG (Directed Acyclic Graph) is

· *Directed*, an inherent flow, dependencies between components

· *Acyclic*, does not loop/cycle/repeat

· *Graph*, the actual set of components



Directed Acyclic Graph (DAG), image by the author

Defining a simple DAG

You've spent some time reviewing the Airflow components and are interested in testing out your own workflows. To start you decide to define the default arguments and create a DAG object for your workflow.

```
# Import the DAG object
from airflow.models import DAG

# Define the default_args dictionary
default_args = {
    'owner': 'dsmith',
    'start_date': datetime(2020, 1, 14),
    'retries': 2
}

# Instantiate the DAG object
etl_dag = DAG('example_etl', default_args=default_args)
```

Syntax: `dag_variable = DAG('dag_name', default_args=default_args)`

Working with DAGs and the Airflow shell

While working with Airflow, sometimes it can be tricky to remember what DAGs are defined and what they do. You want to gain some further knowledge of the Airflow shell command so you'd like to see what options are available.

Multiple DAGs are already defined for you. How many DAGs are present in the Airflow system from the command-line?

```
repl:~$ airflow list_dags
[2020-08-04 16:25:56,974] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-08-04 16:25:57,376] {dagbag.py:90} INFO - Filling up the DagBag from /home/repl/workspace/dags


-------------------------------------------------------------------
DAGS
-------------------------------------------------------------------
example_dag
update_state

repl:~$
```

Answer: there are 2 DAGs

Troubleshooting DAG creation

Now that you've successfully worked with a couple workflows, you notice that sometimes there are issues making a workflow appear within Airflow. You'd like to be able to better troubleshoot the behavior of Airflow when there may be something wrong with the code.

Two DAGs are defined for you and Airflow is setup. Note that any changes you make within the editor are automatically saved.

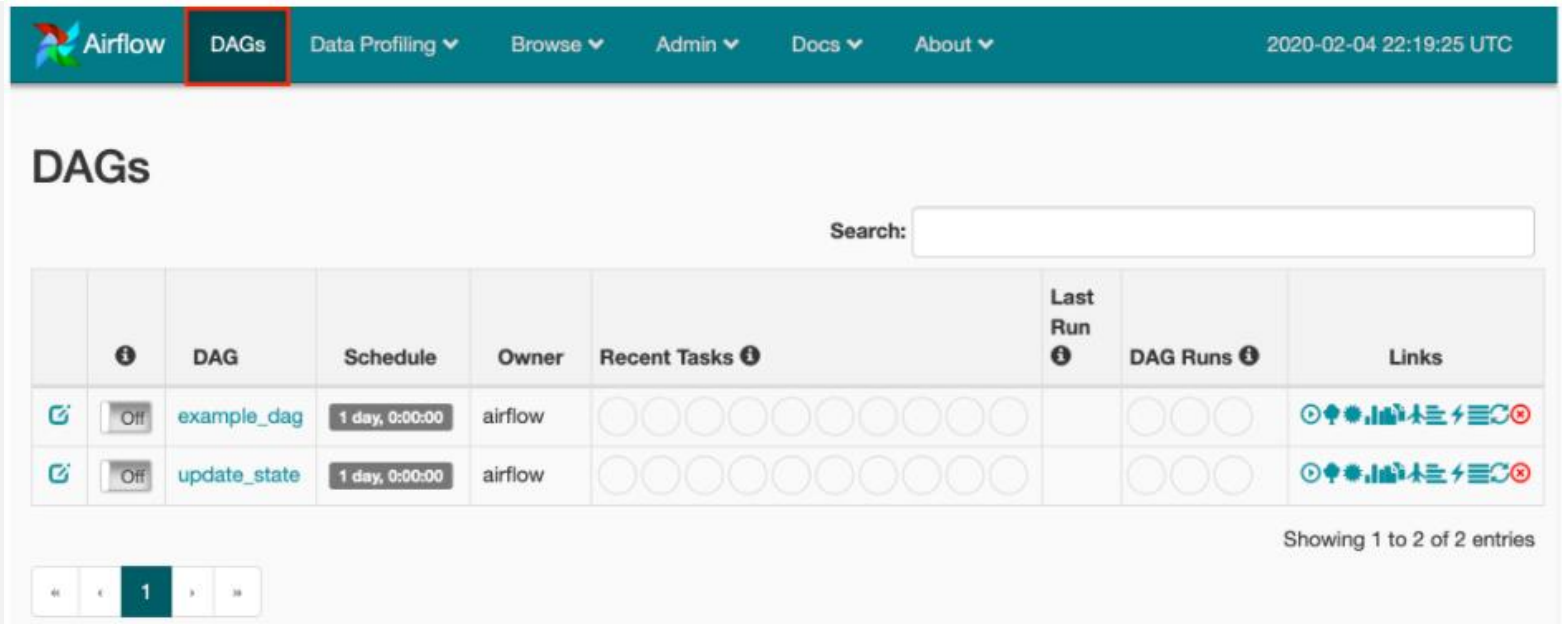
```
from airflow.models import DAG
default_args = {
    'owner': 'jdoe',
    'start_date': '2019-01-01'
}
dag = DAG( dag_id='etl_update', default_args=default_args )
```

refresh_data_workflow.py

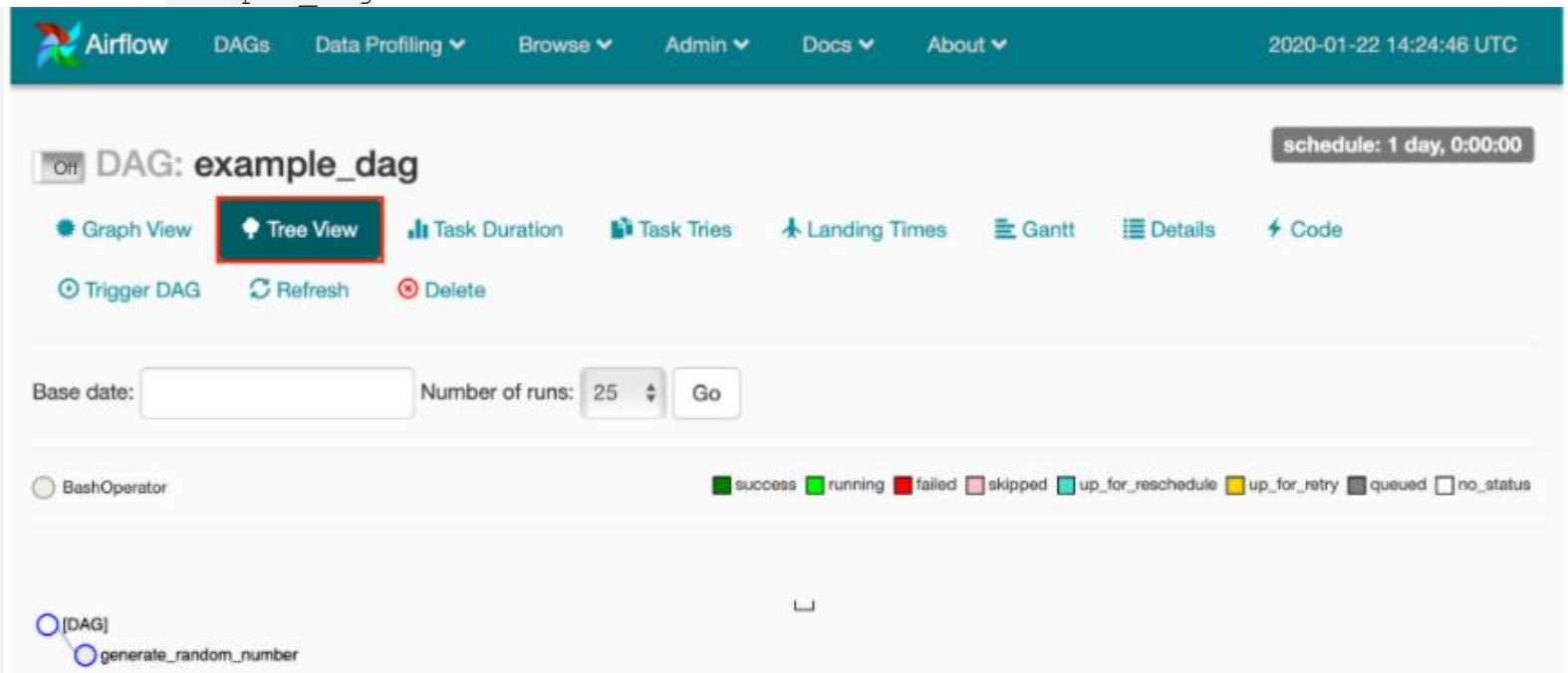
```
from airflow.models import DAG
default_args = {
    'owner': 'jdoe',
    'email': 'jdoe@datacamp.com',
    'start_date': '2019-01-01'
}
dag = DAG( dag_id='refresh_data', default_args=default_args )
```

Airflow web interface

To view DAGs



Click on `example_dag`



Starting the Airflow webserver

You've successfully created some DAGs within Airflow using the command-line tools, but notice that it can be a bit tricky to handle scheduling / troubleshooting / etc. After reading the documentation further, you realize that you'd like to access the Airflow web interface. For security reasons, you'd like to start the webserver on port `9090`.

Which `airflow` command would you use to start the webserver on port `9090`?

- `airflow webserver`
- `airflow start webserver 9090`
- `airflow webserver -9090`
- `airflow webserver -p 9090`

Answer: airflow webserver -p 9090
Sometimes the defaults for Airflow aren't exactly what you'd like to use. Using the built in tools to configure the setup to your specifications is a very common function of a data engineer.

Navigating the Airflow UI

To gain some familiarity with the Airflow UI, you decide to explore the various pages. You'd like to know what has happened on your Airflow instance thus far.

Which of the following `events` have not run on your Airflow instance?

- `cli_scheduler`
- `cli_webserver`
- `cli_worker`

Answer: cli_worker

Examining DAGs with the Airflow UI

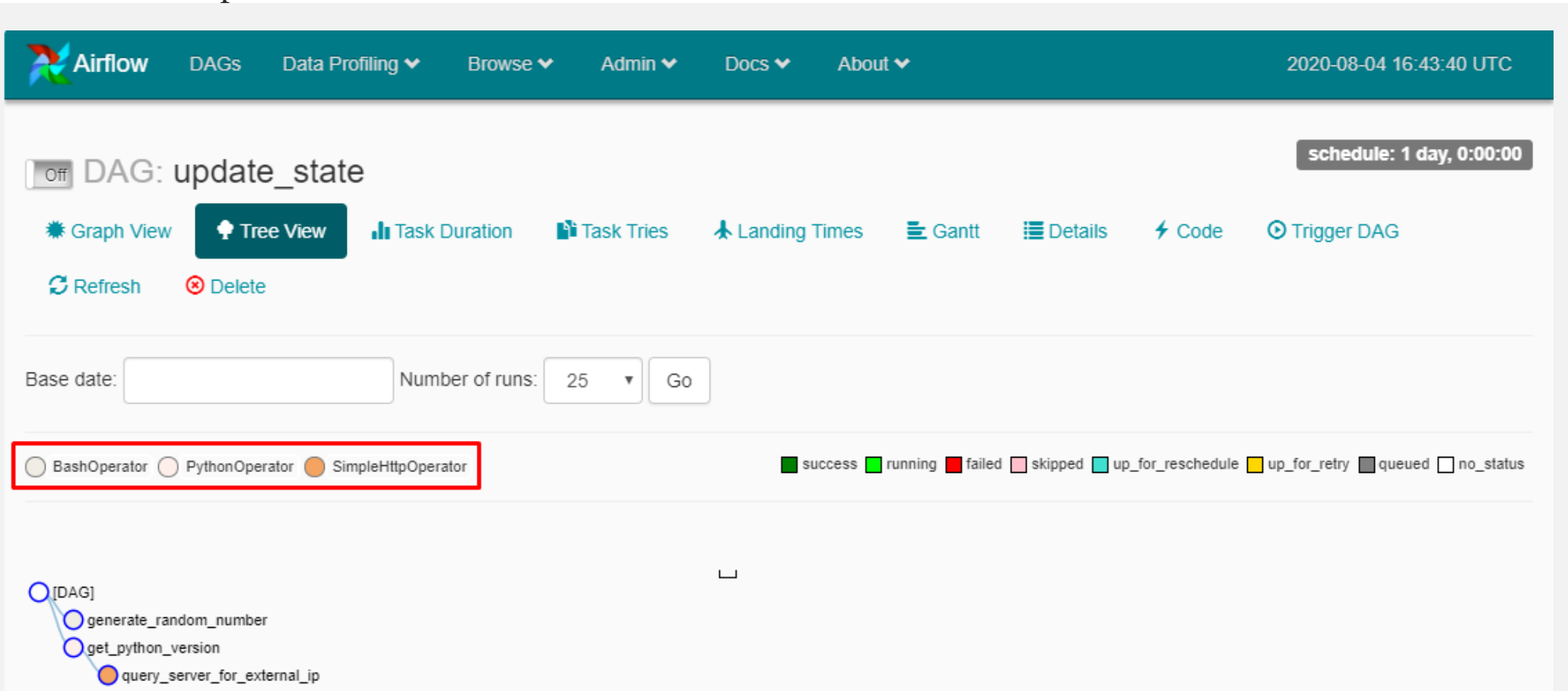
You've become familiar with the basics of an Airflow DAG and the basics of interacting with Airflow on the command-line. Your boss would like you to show others on your team how to examine any available DAGs. In this instance, she would like to know which operator is **NOT** in use with the DAG called `update_state`, as your team is trying to verify the components used in production workflows.

Remember that the Airflow UI allows various methods to view the state of DAGs. The `Tree View` lists the tasks and any ordering between them in a tree structure, with the ability to compress / expand the nodes. The `Graph View` shows any tasks and their dependencies in a graph structure, along with the ability to access further details about task runs. The `Code` view provides full access to the Python code that makes up the DAG.

Remember to select the operator **NOT** used in this DAG.

- `BashOperator`
- `PythonOperator`
- `JdbcOperator`
- `SimpleHttpOperator`

Answer: JdbcOperator



Chapter 2. Implementing Airflow DAGs

Learn the basics of implementing Airflow DAGs using operators, tasks, and scheduling.

Airflow operators

Operators represent a single task in a workflow, running independently on different tasks, and generally do not share information. E.g. DummyOperator, BashOperator.

```
DummyOperator(task_id='example', dag=dag)
```

```
BashOperator(
    task_id='bash_example',
    bash_command='echo "Example!"',
    dag=ml_dag)
```

```
BashOperator(
    task_id='bash_script_example',
    bash_command='runcleanup.sh',
    dag=ml_dag)
```

Defining a BashOperator task

The `BashOperator` allows you to specify any given Shell command or script and add it to an Airflow workflow.

As such, you’ve been running some scripts manually to clean data (using a script called `cleanup.sh`) prior to delivery to your colleagues in the Data Analytics group. As you get more of these tasks assigned, you've realized it's becoming difficult to keep up with running everything manually, much less dealing with errors or retries. You'd like to implement a simple script as an Airflow operator.

The Airflow DAG `analytics_dag` is already defined for you and has the appropriate configurations in place.

```
# Import the BashOperator
from airflow.operators.bash_operator import BashOperator

# Define the BashOperator
cleanup = BashOperator(
    task_id='cleanup_task',
    # Define the bash_command
    bash_command='cleanup.sh',
    # Add the task to the dag
    dag=analytics_dag
)
```

Multiple BashOperators

Airflow DAGs can contain many operators, each performing their defined tasks.
You’ve successfully implemented one of your scripts as an Airflow task and have decided to continue migrating your individual scripts to a full Airflow DAG. You now want to add more components to the workflow. In addition to the `cleanup.sh` used in the previous exercise you have two more scripts, `consolidate_data.sh` and `push_data.sh`. These further process your data and copy to its final location.

The DAG `analytics_dag` is available as before, and your `cleanup` task is still defined. The `BashOperator` is already imported.

```
# Define a second operator to run the `consolidate_data.sh` script
consolidate = BashOperator(
    task_id='consolidate_task',
    bash_command='consolidate_data.sh',
    dag=analytics_dag)

# Define a final operator to execute the `push_data.sh` script
push_data = BashOperator(
    task_id='pushdata_task',
    bash_command='push_data.sh',
    dag=analytics_dag)
```

Airflow tasks

Tasks are instances of operators, usually assigned to a variable in Python, and referred to by the **task_id** (not variable name) within the Airflow tools.

Tasks dependencies are referred to as upstream or downstream tasks. Upstream tasks need to be completed before downstream ones, defined using bitshift operators (>>) between 2 **task variables**.

```
# Define the tasks
task1 = BashOperator(task_id='first_task',
                     bash_command='echo 1',
                     dag=example_dag)

task2 = BashOperator(task_id='second_task',
                     bash_command='echo 2',
                     dag=example_dag)

# Set first_task to run before second_task
task1 >> task2   # or task2 << task1
```

Chained dependencies:

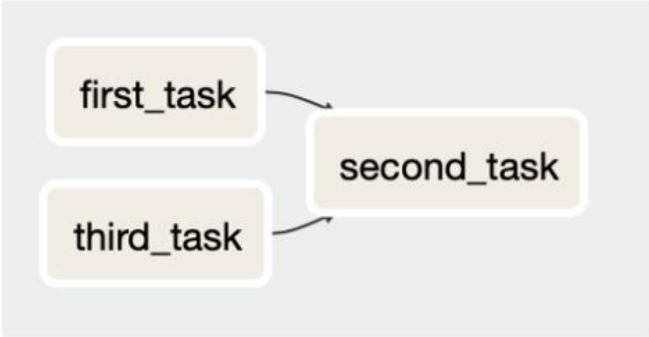
```
task1 >> task2 >> task3 >> task4
```

Mixed dependencies:

```
task1 >> task2 << task3
```

or:

```
task1 >> task2
task3 >> task2
```



Define order of BashOperators

Now that you've learned about the bitshift operators, it's time to modify your workflow to include a pull step and to include the task ordering. You have three currently defined components, `cleanup`, `consolidate`, and `push_data`.

The DAG `analytics_dag` is available as before and the `BashOperator` is already imported.

```
# Define a new pull_sales task
pull_sales = BashOperator(
    task_id='pullsales_task',
    bash_command='wget https://salestracking/latestinfo?json',
    dag=analytics_dag
)

# Set pull_sales to run prior to cleanup
pull_sales >> cleanup

# Configure consolidate to run after cleanup
cleanup >> consolidate

# Set push_data to run last
consolidate >> push_data
```

Determining the order of tasks

While looking through a colleague's workflow definition, you're trying to decipher exactly in which order the defined tasks run. The code in question shows the following:

```
pull_data << initialize_process
pull_data >> clean >> run_ml_pipeline
generate_reports << run_ml_pipeline
```

Troubleshooting DAG dependencies

You've created a DAG with intended dependencies based on your workflow but for some reason Airflow won't load / execute the DAG. Try using the terminal to:

- List the DAGs.
- Decipher the error message.
- Use `cat workspace/dags/codependent.py` to view the Python code.
- Determine which of the following lines should be removed from the Python code. You may want to consider the last line of the file.

```
repl:~$ airflow list_dags
[2020-08-04 17:19:09,844] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-08-04 17:19:10,229] {dagbag.py:90} INFO - Filling up the DagBag from /home/repl/workspace/dags
[2020-08-04 17:19:10,232] {dagbag.py:267} ERROR - Failed to bag_dag: /home/repl/workspace/dags/codependent.py
airflow.exceptions.AirflowDagCycleException: Cycle detected in DAG. Faulty task: third_task to first_task


-------------------------------------------------------------------
DAGS
-------------------------------------------------------------------


repl:~$ cat workspace/dags/codependent.py

from airflow.models import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

default_args = {
  'owner': 'dsmith',
  'start_date': datetime(2020, 2, 12),
  'retries': 1
}

codependency_dag = DAG('codependency', default_args=default_args)

task1 = BashOperator(task_id='first_task',
                     bash_command='echo 1',
                     dag=codependency_dag)

task2 = BashOperator(task_id='second_task',
                     bash_command='echo 2',
                     dag=codependency_dag)

task3 = BashOperator(task_id='third_task',
                     bash_command='echo 3',
                     dag=codependency_dag)

# task1 must run before task2 which must run before task3
task1 >> task2
task2 >> task3
task3 >> task1
```

- `task1 >> task2`
- `task2 >> task3`
- `task3 >> task1`

Answer: task3 >> task1
For this particular issue, a loop, or cycle, is present within the DAG. Note that technically removing the first dependency would resolve the issue as well, but the comments specifically reference the desired effect. Commenting the desired effect in this way can often help resolve bugs in Airflow DAG execution.

Additional operators

PythonOperator executes a Python function (Example 1) or callable function with keyword arguments op_kwargs dictionary (Example 2).

```
from airflow.operators.python_operator import PythonOperator
def printme():
    print("This goes in the logs!")
python_task = PythonOperator(
    task_id='simple_print',
    python_callable=printme,
    dag=example_dag
)
```

Example 1

```python
def sleep(length_of_time):
  time.sleep(length_of_time)


sleep_task = PythonOperator(
    task_id='sleep',
    python_callable=sleep,
    op_kwargs={'length_of_time': 5}
    dag=example_dag
)
```

Example 2

EmailOperator sends an email, with html content and attachments, after configured with email server details.

```python
from airflow.operators.email_operator import EmailOperator


email_task = EmailOperator(
    task_id='email_sales_report',
    to='sales_manager@example.com',
    subject='Automated Sales Report',
    html_content='Attached is the latest sales report',
    files='latest_sales.xlsx',
    dag=example_dag
)
```

Using the PythonOperator

You've implemented several Airflow tasks using the BashOperator but realize that a couple of specific tasks would be better implemented using Python. You'll implement a task to download and save a file to the system within Airflow.

The `requests` library is imported for you, and the DAG `process_sales_dag` is already defined.

```
# Define the method
def pull_file(URL, savepath):
    r = requests.get(URL)
    with open(savepath, 'wb') as f:
        f.write(r.content)
    # Use the print method for logging
    print(f'File pulled from {URL} and saved to {savepath}')

# Import the PythonOperator class
from airflow.operators.python_operator import PythonOperator

# Create the task
pull_file_task = PythonOperator(
    task_id='pull_file',
    # Add the callable
    python_callable=pull_file,
    # Define the arguments
    op_kwargs={'URL':'http://dataserver/sales.json', 'savepath':'latestsales.json'},
    dag=process_sales_dag
```

You can use `.format()` or other variable substitution methods as desired, especially if working with a Python version earlier than 3.6.

More PythonOperators

To continue implementing your workflow, you need to add another step to parse and save the changes of the downloaded file. The DAG `process_sales_dag` is defined and has the `pull_file` task already added. In this case, the Python function is already defined for you, `parse_file(inputfile, outputfile)`.

Note that often when implementing Airflow tasks, you won't necessarily understand the individual steps given to you. As long as you understand how to wrap the steps within Airflow's structure, you'll be able to implement a desired workflow.

```
# Add another Python task
parse_file_task = PythonOperator(
    task_id='parse_file',
    # Set the function to call
    python_callable=parse_file,
    # Add the arguments
    op_kwargs={'inputfile':'latestsales.json', 'outputfile':'parsedfile.json'},
    # Add the DAG
    dag=process_sales_dag
)
```

EmailOperator and dependencies

Now that you've successfully defined the PythonOperators for your workflow, your manager would like to receive a copy of the parsed JSON file via email when the workflow completes. The previous tasks are still defined and the DAG `process_sales_dag` is configured.

```
# Import the Operator
from airflow.operators.email_operator import EmailOperator

# Define the task
email_manager_task = EmailOperator(
    task_id='email_manager',
    to='manager@datacamp.com',
    subject='Latest sales JSON',
    html_content='Attached is the latest sales JSON file as requested.',
```

```
    files='parsedfile.json',
    dag=process_sales_dag
)

# Set the order of tasks
pull_file_task >> parse_file_task >> email_manager_task
```

Airflow scheduling

Browse -> DAG Runs

· are specific instance of a workflow at a point in time

· can be run manually or via schedule_interval

· each workflow states: running, failed, success

cron syntax

```
# ┌─────────────── minute (0 - 59)
# │ ┌─────────────── hour (0 - 23)
# │ │ ┌─────────────── day of the month (1 - 31)
# │ │ │ ┌─────────────── month (1 - 12)
# │ │ │ │ ┌─────────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                       7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * command to execute
```

- Is pulled from the Unix cron format
- Consists of 5 fields separated by a space
- An asterisk `*` represents running for every interval (ie, every minute, every day, etc)
- Can be comma separated values in fields for a list of values

```
0 12 * * *              # Run daily at noon
```

```
* * 25 2 *              # Run once per minute on February 25
```

```
0,15,30,45 * * * *      # Run every 15 minutes
```

Schedule a DAG via Python

You’ve learned quite a bit about creating DAGs, but now you would like to schedule a specific DAG on a specific day of the week at a certain time. You’d like the code include this information in case a colleague needs to reinstall the DAG to a different server.

The Airflow `DAG` object and the appropriate `datetime` methods have been imported for you.

```
# Update the scheduling arguments as defined
default_args = {
  'owner': 'Engineering',
  'start_date': datetime(2019, 11, 1),
  'email': ['airflowresults@datacamp.com'],
  'email_on_failure': False,
  'email_on_retry': False,
  'retries': 3,
  'retry_delay': timedelta(minutes=20)
}

# Use the cron syntax for every Wednesday at 12:30pm
dag = DAG('update_dataflows', default_args=default_args, schedule_interval='30 12 * * 3')
```

Deciphering Airflow schedules

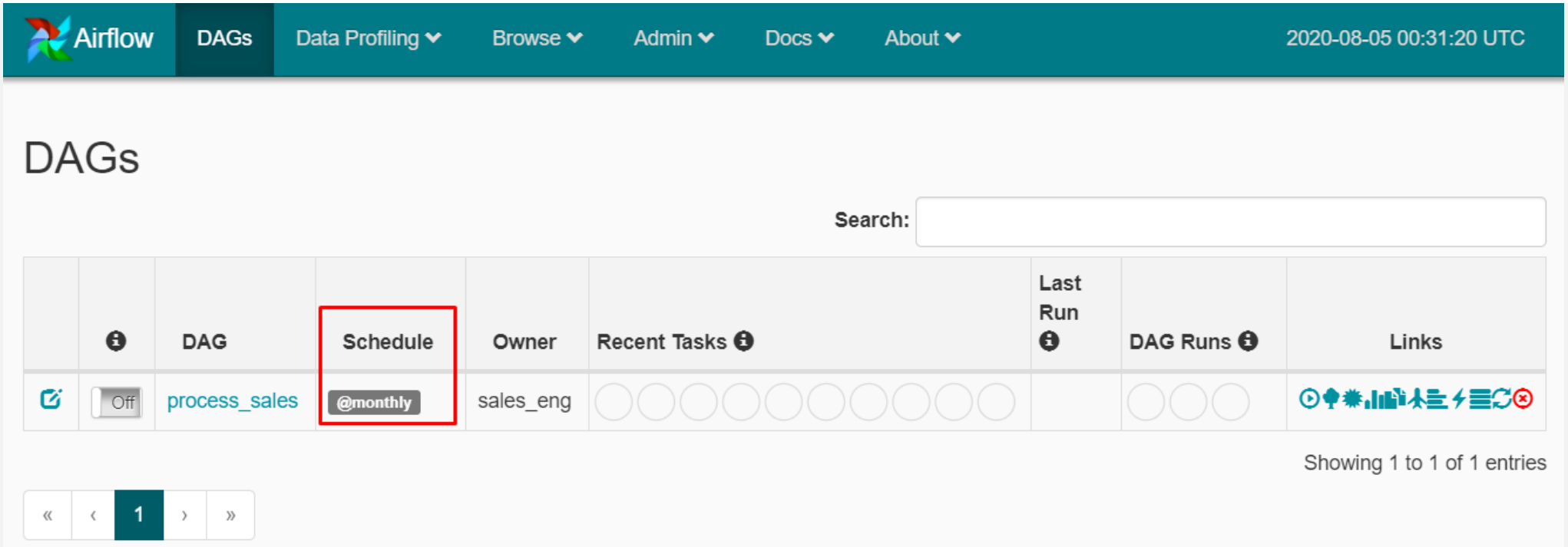
Given the various options for Airflow's `schedule_interval`, you'd like to verify that you understand exactly how intervals relate to each other, whether it's a cron format, `timedelta` object, or a preset.

Order the schedule intervals from least to greatest amount of time.

- * * * * *
- timedelta(minutes=5)
- @hourly
- * 0,12 * * *
- timedelta(days=1)
- @weekly

Troubleshooting DAG runs

You've scheduled a DAG called `process_sales` which is set to run on the first day of the month and email your manager a copy of the report generated in the workflow. The `start_date` for the DAG is set to February 15, 2020. Unfortunately it's now March 2nd and your manager did not receive the report and would like to know what happened.



Use the information you've learned about Airflow scheduling to determine what the issue is.

- The `schedule_interval` has not yet passed since the `start_date`.
- The `email_manager_task` is not downstream of the other tasks.
- The DAG run has an error.
- The `op_kwargs` are incorrect for the `EmailOperator`.

Answer: The `schedule_interval` has not yet passed since the `start_date`.

Chapter 3. Maintaining and monitoring Airflow workflows

Learn more about Airflow components such as sensors and executors while monitoring and troubleshooting Airflow workflows.

Airflow sensors

Sensors are operators that wait for a certain condition to be true, e.g. creation of a file, database record upload, response from a web. Sensors are assigned to tasks, and the frequency to check for the condition to be true can be defined.

- Derived from `airflow.sensors.base_sensor_operator`
- Sensor arguments:
- `mode` - How to check for the condition
  - `mode='poke'` - The default, run repeatedly
  - `mode='reschedule'` - Give up task slot and try again later
- `poke_interval` - How often to wait between checks
- `timeout` - How long to wait before failing task
- Also includes normal operator attributes

File sensor checks for the existence of a file at a certain location (directory).

```
from airflow.contrib.sensors.file_sensor import FileSensor

file_sensor_task = FileSensor(task_id='file_sense',
                              filepath='salesdata.csv',
                              poke_interval=300,
                              dag=sales_report_dag)

init_sales_cleanup >> file_sensor_task >> generate_report
```

poke_interval = 300 seconds

- `ExternalTaskSensor` - wait for a task in another DAG to complete
- `HttpSensor` - Request a web URL and check for content
- `SqlSensor` - Runs a SQL query to check for content
- Many others in `airflow.sensors` and `airflow.contrib.sensors`

Other sensors available in Airflow

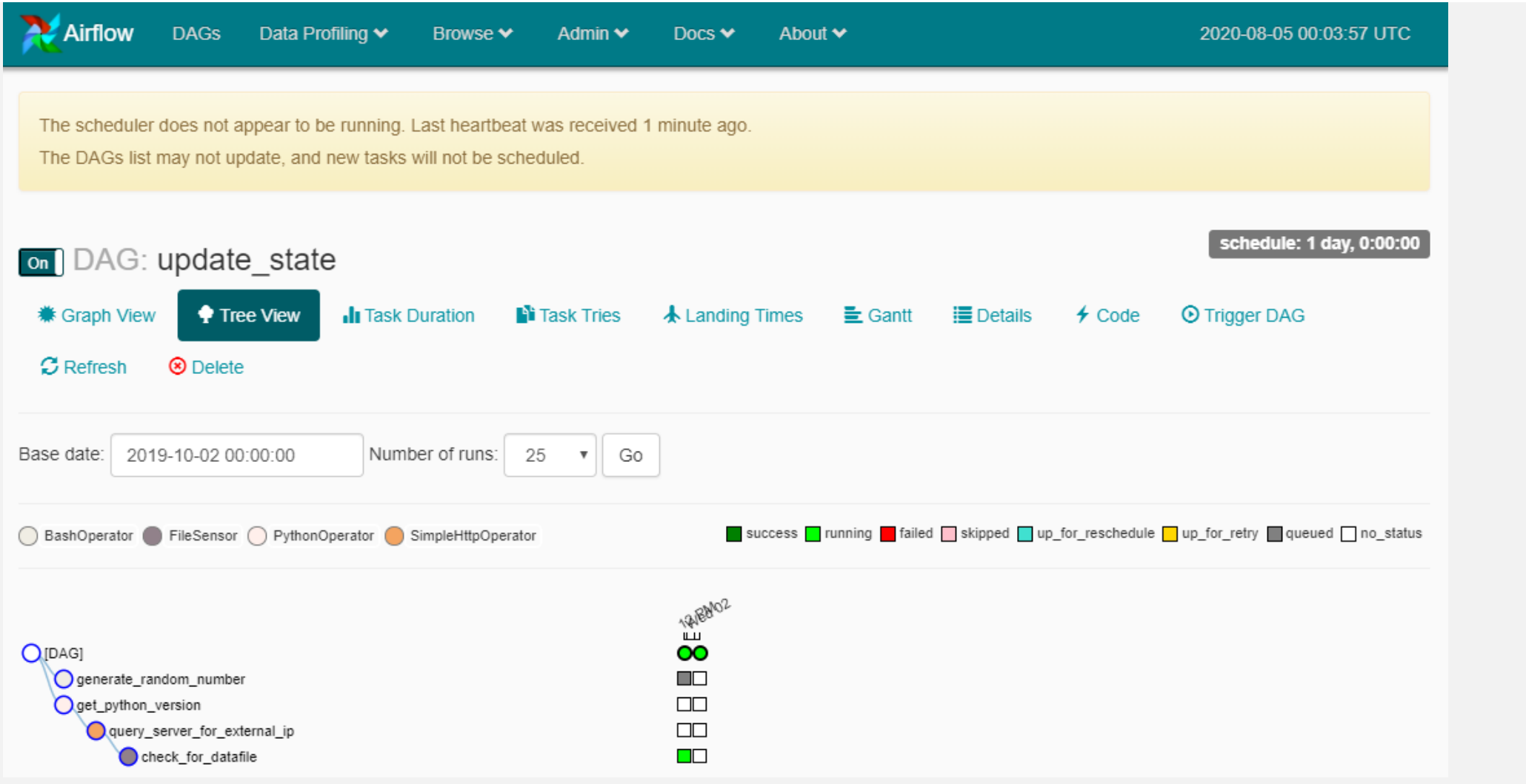
Sensors vs operators

As you've just learned about sensors, you want to verify you understand what they have in common with normal operators and where they differ.

Sensors	Both	Operators
Has a `poke_interval` attribute.	Are assigned to DAGs.	`BashOperator`
`FileSensor`	Have a `task_id`.	Only runs once per DAG run.
Derives from `BaseSensorOperator`.		

Sensory deprivation

You've recently taken over for another Airflow developer and are trying to learn about the various workflows defined within the system. You come across a DAG that you can't seem to make run properly using any of the normal tools. Try exploring the DAG for any information about what it might be looking for before continuing.



- The DAG is waiting for the file `salesdata_ready.csv` to be present.
- The DAG expects a response from the `SimpleHttpOperator` before starting.
- `part1` needs a dependency added.

Answer: The DAG is waiting for the file `salesdata_ready.csv` to be present.

Airflow executors

An executor is the component that runs the task in a workflow, for example, SequentialExecutor, LocalExecutor, CeleryExecutor.

Sequential Executor is the default Airflow execution engine that runs one task at a time. It is useful for debugging, but not recommended for production due to the limitation of task resources.

LocalExecutor runs on a single system, treats each task as a process, and is able to start as many concurrent tasks as permitted by the system resources (ie, CPU cores, memory, etc). It is a good choice for a single production Airflow system and can utilise all the resources of a given host system.

CeleryExecutor uses a Celery backend as task manager. It is a general queuing system written in Python that allows multiple systems (parallelism) to communicate as a basic cluster. Using a CeleryExecutor, multiple Airflow systems can be configured as workers for a given set of workflows/tasks. You can add extra systems at any time to better balance workflows, but it is more difficult to set up and configure.

Determining the executor

While developing your DAGs in Airflow, you realize you're not certain the configuration of the system. Using the commands you've learned, determine which of the following statements is true.

```
repl:~$ airflow list_dags
[2020-08-05 00:28:46,653] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-08-05 00:28:47,031] {dagbag.py:90} INFO - Filling up the DagBag from /home/repl/workspace/dags


-------------------------------------------------------------------
DAGS
-------------------------------------------------------------------
update_state

repl:~$
```

- This system can run 12 tasks at the same time.
- This system can run one task at a time.
- This system can run as many tasks as needed at a time.

Answer: This system can run one task at a time.

Executor implications

You're learning quite a bit about running Airflow DAGs and are gaining some confidence at developing new workflows. That said, your manager has mentioned that on some days, the workflows are taking a lot longer to finish and asks you to investigate. She also mentions that the `salesdata_ready.csv` file is taking longer to generate these days and the time of day it is completed is variable.

This exercise requires information from the previous two lessons — remember the implications of the available arguments and modify the workflow accordingly.

```
from airflow.models import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.contrib.sensors.file_sensor import FileSensor
from datetime import datetime

report_dag = DAG(
    dag_id = 'execute_report',
    schedule_interval = "0 0 * * *"
)

precheck = FileSensor(
    task_id='check_for_datafile',
    filepath='salesdata_ready.csv',
    start_date=datetime(2020,2,20),
    mode='reschedule',
    dag=report_dag
)

generate_report_task = BashOperator(
    task_id='generate_report',
    bash_command='generate_report.sh',
    start_date=datetime(2020,2,20),
    dag=report_dag
)

precheck >> generate_report_task
```

```
repl:~/workspace$ airflow list_dags
[2020-08-05 00:34:22,734] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-08-05 00:34:23,066] {dagbag.py:90} INFO - Filling up the DagBag from /home/repl/workspace/dags


-------------------------------------------------------------------
DAGS
-------------------------------------------------------------------
execute_report
```

By modifying the sensor properties (from mode='poke' to mode='reschedule'), Airflow is given a chance **to run another task while waiting** for the `salesdata_ready.csv` file. This required recognizing the connection between an executor and the number and type of tasks in a workflow. Alternatively, you could also modify the executor type to something with a parallelism greater than 1 to allow the tasks to complete.

Debugging and troubleshooting in Airflow

Typical issues (refer to video here):

· DAG won't run on schedule — scheduler is not running (fix this issue by running `airflow scheduler` from the command-line), not enough free slots for executor to run tasks (change to parallel executor, add system resources, or change DAG schedule to lower peak period)

· DAG won't load — DAG not in web UI, DAG not shown in `airflow list_dags` (verify DAG file is in correct folder, determine the DAGs folder via `airflow.cfg`)

· Syntax errors — code failed to compile (debug in VSCode IDE or Jupyter notebook)

DAGs in the bag

You've taken over managing an Airflow cluster that you did not setup and are trying to learn a bit more about the system configuration. Which of the following is true?

```
repl:~$ airflow list_dags
[2020-08-05 00:58:18,455] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-08-05 00:58:18,794] {dagbag.py:90} INFO - Filling up the DagBag from /home/repl/workspace/dags


-------------------------------------------------------------------
DAGS
-------------------------------------------------------------------
update_state

repl:~$
```

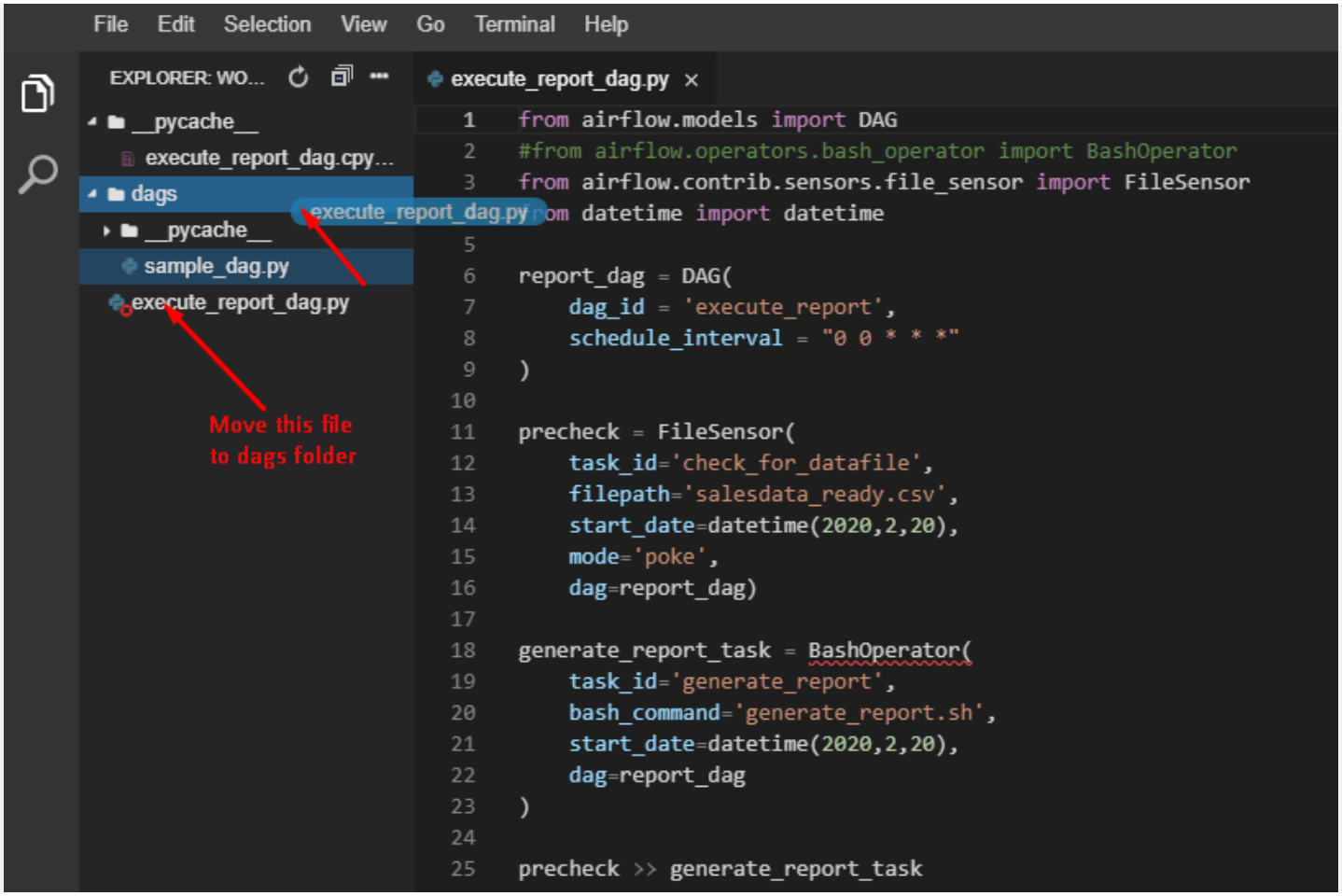
- The DAG is scheduled for hourly processing.
- The Airflow user does not have proper permissions.
- The `dags_folder` is set to `/home/repl/workspace/dags`.

Answer: The `dags_folder` is set to `/home/repl/workspace/dags`.

Missing DAG

Your manager calls you before you're about to leave for the evening and wants to know why a new DAG workflow she's created isn't showing up in the system. She needs this DAG called `execute_report` to appear in the system so she can properly schedule it for some tests before she leaves on a trip.

Airflow is configured using the `~/airflow/airflow.cfg` file. This is a multi-layered issue for why the DAG would not load.



Remember that sometimes having no apparent error does not necessarily mean everything is working as expected. It is common for there to be more than one simultaneous problem with loading workflows, even if the issues appear simple at first. Always try to consider the problems that could appear, and that there might be more than one as it will simplify your usage of Airflow.

SLAs and reporting in Airflow

Service Level Agreement (SLA) is the amount of time a task or a DAG should require to run. If the task/DAG does not meet the expected timing, it is called SLA Miss, with logs stored in the web UI (Browse → SLA Misses).

2 ways to define SLAs:

- Using the `'sla'` argument on the task

```
task1 = BashOperator(task_id='sla_task',
                     bash_command='runcode.sh',
                     sla=timedelta(seconds=30),
                     dag=dag)
```

- On the `default_args` dictionary

```
default_args={
 'sla': timedelta(minutes=20)
 'start_date': datetime(2020,2,20)
}
dag = DAG('sla_dag', default_args=default_args)
```

timedelta object

- In the `datetime` library
- Accessed via `from datetime import timedelta`
- Takes arguments of days, seconds, minutes, hours, and weeks

```
timedelta(seconds=30)
timedelta(weeks=2)
timedelta(days=4, hours=10, minutes=20, seconds=30)
```

Defining an SLA

You've successfully implemented several Airflow workflows into production, but you don't currently have any method of determining if a workflow takes too long to run. After consulting with your manager and your team, you decide to implement an SLA at the DAG level on a test workflow.

```
# Import the timedelta object
from datetime import timedelta

# Create the dictionary entry
default_args = {
  'start_date': datetime(2020, 2, 20),
  'sla': timedelta(minutes=30)
}

# Add to the DAG
test_dag = DAG('test_workflow', default_args=default_args, schedule_interval='@None')
```

Note that this type of SLA applies for the entire workflow, not just an individual task.

Defining a task SLA

After completing the SLA on the entire workflow, you realize you really only need the SLA timing on a specific task instead of the full workflow.

```
# Import the timedelta object
from datetime import timedeltatest_dag = DAG('test_workflow', start_date=datetime(2020,2,20),
schedule_interval='@None')

# Create the task with the SLA
task1 = BashOperator(task_id='first_task',
                     sla=timedelta(hours=3),
                     bash_command='initialize_data.sh',
                     dag=test_dag)
```

You can add specific SLAs to individual tasks as needed. Try adding various SLA settings to your workflows to determine how your systems are behaving overall.

Generate and email a report

Airflow provides the ability to automate almost any style of workflow. You would like to receive a report from Airflow when tasks complete without requiring constant monitoring of the UI or log files. You decide to use the email functionality within Airflow to provide this message.

All the typical Airflow components have been imported for you, and a DAG is already defined as `dag`.

```
# Define the email task
email_report = EmailOperator(
        task_id='email_report',
        to='airflow@datacamp.com',
        subject='Airflow Monthly Report',
        html_content="""Attached is your monthly workflow report - please refer to it for more detail""",
        files=['monthly_report.pdf'],
        dag=report_dag
)

# Set the email task to run after the report is generated
email_report << generate_report
```

Airflow will now email you with an attached report file after the `generate_report` task completes. You can use Airflow's functionality to send updates via many methods in addition to email. Make sure to look through the documentation for other ideas on monitoring your workflows.

Adding status emails

You've worked through most of the Airflow configuration for setting up your workflows, but you realize you're not getting any notifications when DAG runs complete or fail. You'd like to setup email alerting for the success and failure cases, but you want to send it to two addresses.

File Edit Selection View Go Terminal Help

EXPLORER: WO...

- dags
  - __pycache__
  - execute_report_dag.py

execute_report_dag.py

```python
from airflow.models import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.contrib.sensors.file_sensor import FileSensor
from datetime import datetime

default_args={
    'email': ['airflowalerts@datacamp.com', 'airflowadmin@datacamp.com'],
    'email_on_failure': True,
    'email_on_success': True
}
report_dag = DAG(
    dag_id = 'execute_report',
    schedule_interval = "0 0 * * *",
    default_args=default_args
)

precheck = FileSensor(
    task_id='check_for_datafile',
    filepath='salesdata_ready.csv',
    start_date=datetime(2020,2,20),
    mode='reschedule',
    dag=report_dag)

generate_report_task = BashOperator(
    task_id='generate_report',
    bash_command='generate_report.sh',
    start_date=datetime(2020,2,20),
    dag=report_dag
)

precheck >> generate_report_task
```

The workflow is successfully configured to send you email alerts when the DAG completes successfully or fails. Use these options in production to monitor the state of your workflows to help avoid surprises.

Chapter 4. Building production pipelines in Airflow

Use what you've learned to build a production quality workflow in Airflow.

Working with templates

Templates allow substitution of information during a DAG run, and provide added flexibility when defining tasks.

```
t1 = BashOperator(
       task_id='first_task',
       bash_command='echo "Reading file1.txt"',
       dag=dag)

t2 = BashOperator(
       task_id='second_task',
       bash_command='echo "Reading file2.txt"',
       dag=dag)
```

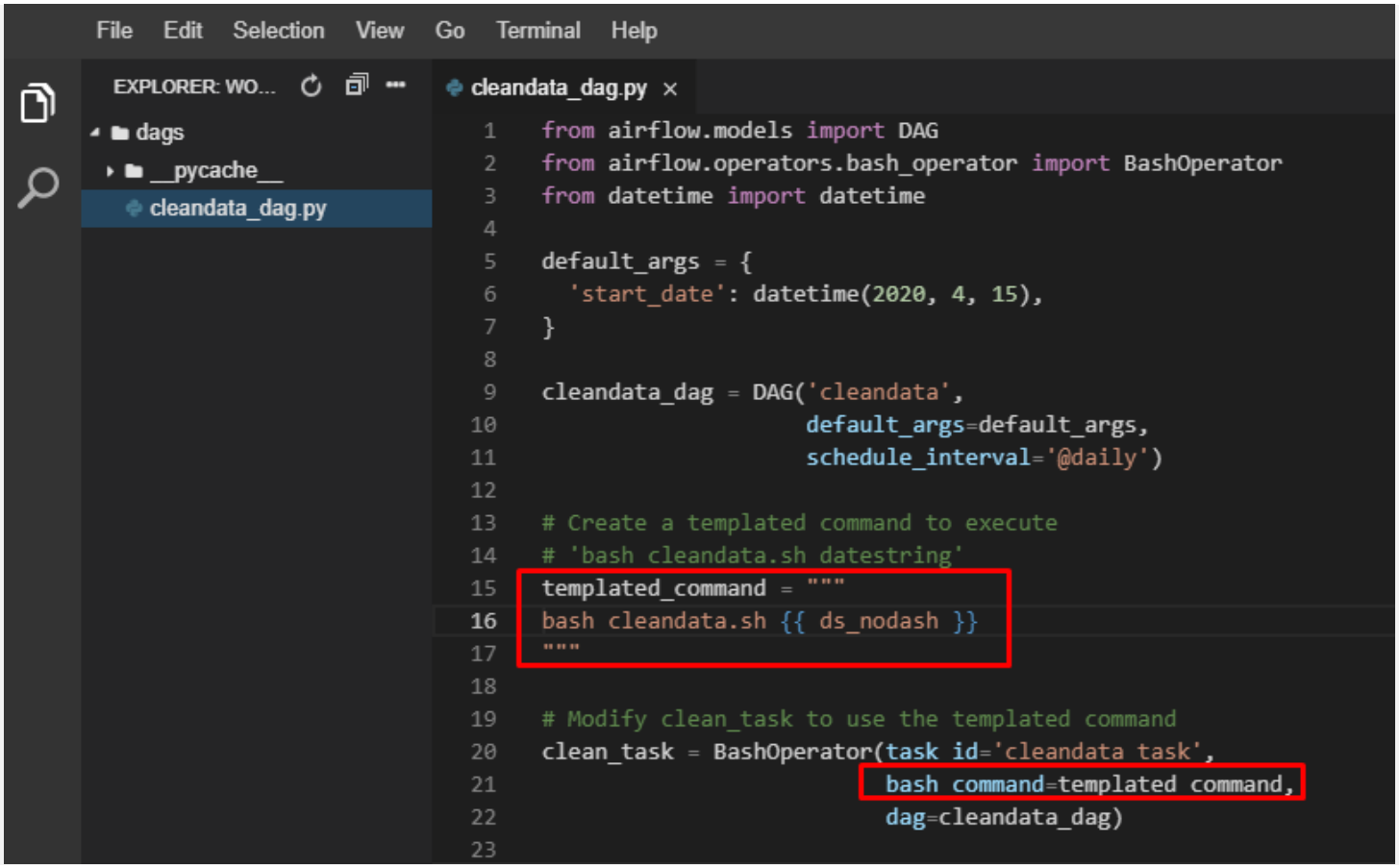
For example, the repetitive code above can be replaced with templated BashOperator.

```
templated_command="""
  echo "Reading {{ params.filename }}"
"""

t1 = BashOperator(task_id='template_task',
       bash_command=templated_command,
       params={'filename': 'file1.txt'}
       dag=example_dag)

t2 = BashOperator(task_id='template_task',
       bash_command=templated_command,
       params={'filename': 'file2.txt'}
       dag=example_dag)
```

Creating a templated BashOperator

You’ve successfully created a BashOperator that cleans a given data file by executing a script called `cleandata.sh`. This works, but unfortunately requires the script to be run only for the current day. Some of your data sources are occasionally behind by a couple of days and need to be run manually.

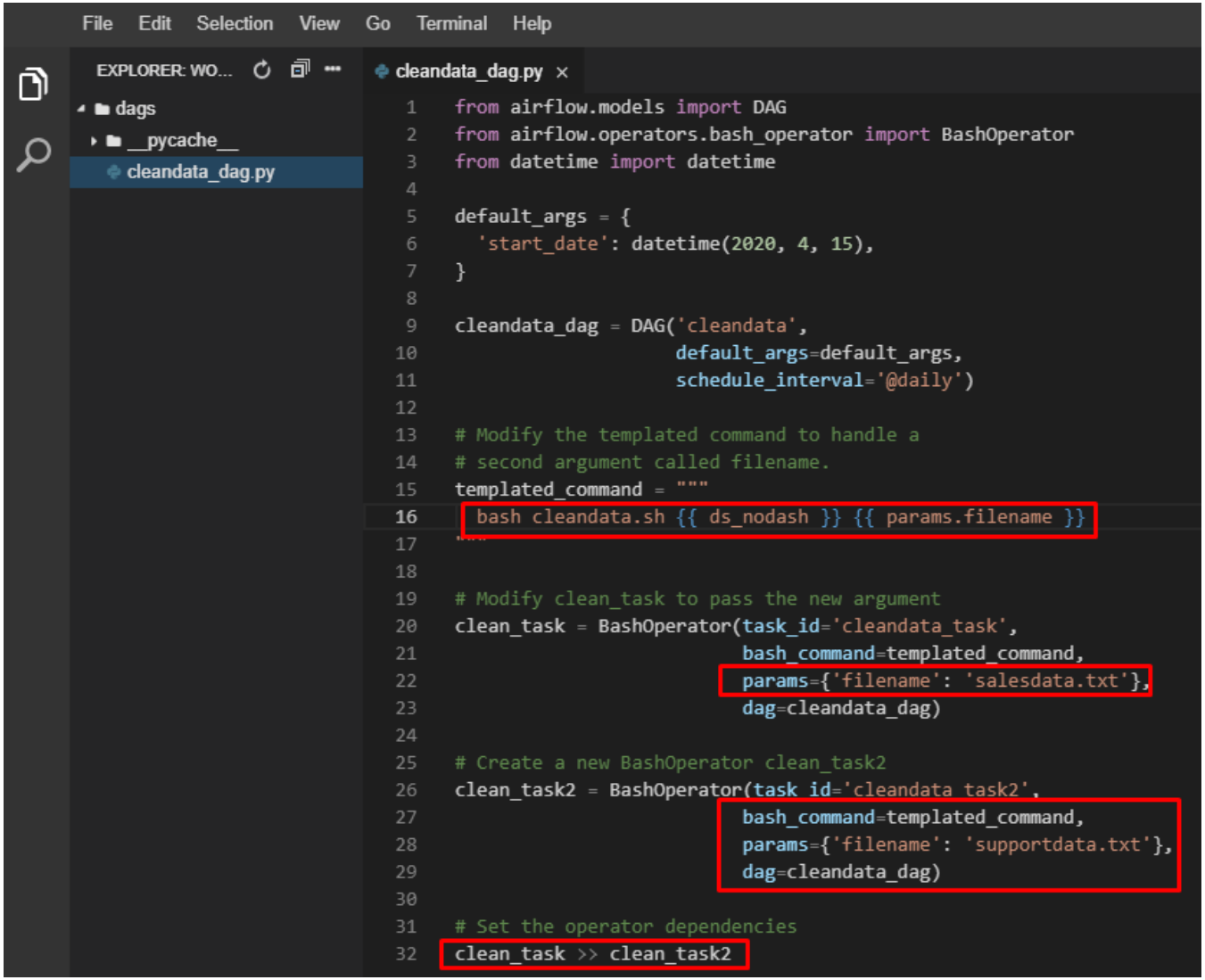
You successfully modify the `cleandata.sh` script to take one argument - the date in YYYYMMDD format. Your testing works at the command-line, but you now need to implement this into your Airflow DAG. For now, use the term `{{ ds_nodash }}` in your template - you'll see exactly what this is means later on.



The DAG has been modified to use a templated command instead of hardcoding your workflow objects. This will come in very handy when creating production workflows. Note that for now, we didn't need to define a params argument in the BashOperator — this is ok as Airflow handles passing some data into templates automatically for us.

Templates with multiple arguments

You wish to build upon your previous DAG and modify the code to support two arguments — the date in YYYYMMDD format, and a file name passed to the `cleandata.sh` script.



Making use of multiple operators that vary by the parameters is a great use of templated commands in Airflow!

More templates

```
templated_command="""
{% for filename in params.filenames %}
  echo "Reading {{ filename }}"
{% endfor %}
"""

t1 = BashOperator(task_id='template_task',
       bash_command=templated_command,
       params={'filenames': ['file1.txt', 'file2.txt']}
       dag=example_dag)
```

```
Reading file1.txt
Reading file2.txt
```

Airflow built-in **runtime variables** provides information about DAG runs, tasks, and even the system configuration.

```
Execution Date: {{ ds }}                              # YYYY-MM-DD
Execution Date, no dashes: {{ ds_nodash }}            # YYYYMMDD

Previous Execution date: {{ prev_ds }}                # YYYY-MM-DD
Prev Execution date, no dashes: {{ prev_ds_nodash }}  # YYYYMMDD

DAG object: {{ dag }}

Airflow config object: {{ conf }}
```

Macros variable is a reference to the Airflow macros package which provides various useful objects/methods for Airflow templates.

- `{{ macros.datetime }}` : The `datetime.datetime` object
- `{{ macros.timedelta }}` : The `timedelta` object
- `{{ macros.uuid }}` : Python's `uuid` object
- `{{ macros.ds_add('2020-04-15', 5) }}` : Modify days from a date, this example returns 2020-04-20

Using lists with templates

Once again, you decide to make some modifications to the design of your `cleandata` workflow. This time, you realize that you need to run the command `cleandata.sh` with the date argument and the file argument as before, except now you have a list of 30 files. You do *not* want to create 30 tasks, so your job is to modify the code to support running the argument for 30 or more files.

The Python list of files is already created for you, simply called `filelist`.

```python
from airflow.models import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

filelist = [f'file{x}.txt' for x in range(30)]

default_args = {
  'start_date': datetime(2020, 4, 15),
}

cleandata_dag = DAG('cleandata',
                    default_args=default_args,
                    schedule_interval='@daily')

# Modify the template to handle multiple files in a
# single run.
templated_command = """
  <% for filename in params.filenames %>
  bash cleandata.sh {{ ds_nodash }} {{ filename }};
  <% endfor %>
"""

# Modify clean_task to use the templated command
clean_task = BashOperator(task_id='cleandata_task',
                          bash_command=templated_command,
                          params={'filenames': filelist},
                          dag=cleandata_dag)
```

You've successfully implemented a Jinja template to iterate over the files in a list and execute a bash command for each file. This type of flexibility and power provides a lot of options to best configure a workflow using Airflow.

Understanding parameter options

You've used a few different methods to add templates to your workflows. Considering the differences between options, why would you want to create individual tasks (ie, BashOperators) with specific parameters vs a list of files?

For example, why would you choose

```
t1 = BashOperator(task_id='task1', bash_command=templated_command,
        params={'filename': 'file1.txt'}, dag=dag)
t2 = BashOperator(task_id='task2', bash_command=templated_command,
        params={'filename': 'file2.txt'}, dag=dag)
t3 = BashOperator(task_id='task3', bash_command=templated_command,
        params={'filename': 'file3.txt'}, dag=dag)
```

over using a loop form such as

```
t1=BashOperator(task_id='task1',
     bash_command=templated_command,
     params={'filenames': ['file1.txt', 'file2.txt', 'file3.txt']},
     dag=dag)
```

- Using a loop form is slower.
- Using specific tasks allows better monitoring of task state and possible parallel execution.
- The params object can only handle lists of a few items.

Answer: Using specific tasks allows better monitoring of **task state** and possible **parallel** execution. When using a single task, all entries would succeed or fail as a single task. Separate operators allow for better monitoring and scheduling of these tasks.

Sending templated emails

While reading through the Airflow documentation, you realize that various operations can use templated fields to provide added flexibility. You come across the docs for the EmailOperator and see that the content can be set to a template. You want to make use of this functionality to provide more detailed information regarding the output of a DAG run.

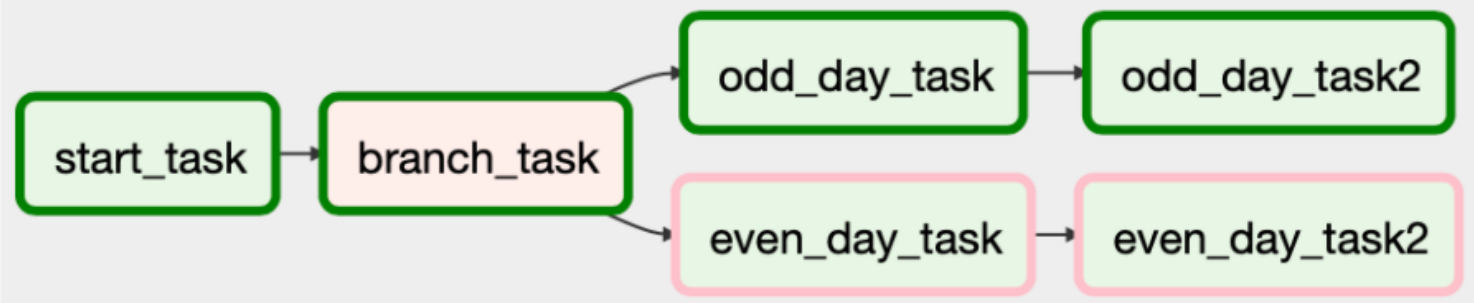
```python
from airflow.models import DAG
from airflow.operators.email_operator import EmailOperator
from datetime import datetime

# Create the string representing the html email content
html_email_str = """
Date: {{ ds }}
Username: {{ params.username }}
"""

email_dag = DAG('template_email_test',
                default_args={'start_date': datetime(2020, 4, 15)},
                schedule_interval='@weekly')

email_task = EmailOperator(task_id='email_task',
                           to='testuser@datacamp.com',
                           subject="{{ macros.uuid.uuid4() }}",
                           html_content=html_email_str,
                           params={'username': 'testemailuser'},
                           dag=email_dag)
```

As mentioned, there are many operators that can accept templated fields. When browsing the documentation, if a field is referred to as *templated*, it can use these techniques.

Branching

Branching provides conditional logic (tasks can be selectively executed or skipped), using **BranchPythonOperator**, which takes a python_callable to return the next task_id (or list of ids) to follow.

```
def branch_test(**kwargs):
  if int(kwargs['ds_nodash']) % 2 == 0:
    return 'even_day_task'
  else:
    return 'odd_day_task'

branch_task = BranchPythonOperator(task_id='branch_task',dag=dag,
       provide_context=True,
       python_callable=branch_test)

start_task >> branch_task >> even_day_task >> even_day_task2
branch_task >> odd_day_task >> odd_day_task2
```



Workflow on an odd day (not even day)

Define a BranchPythonOperator

After learning about the power of conditional logic within Airflow, you wish to test out the BranchPythonOperator. You'd like to run a different code path if the current execution date represents a new year (ie, 2020 vs 2019).

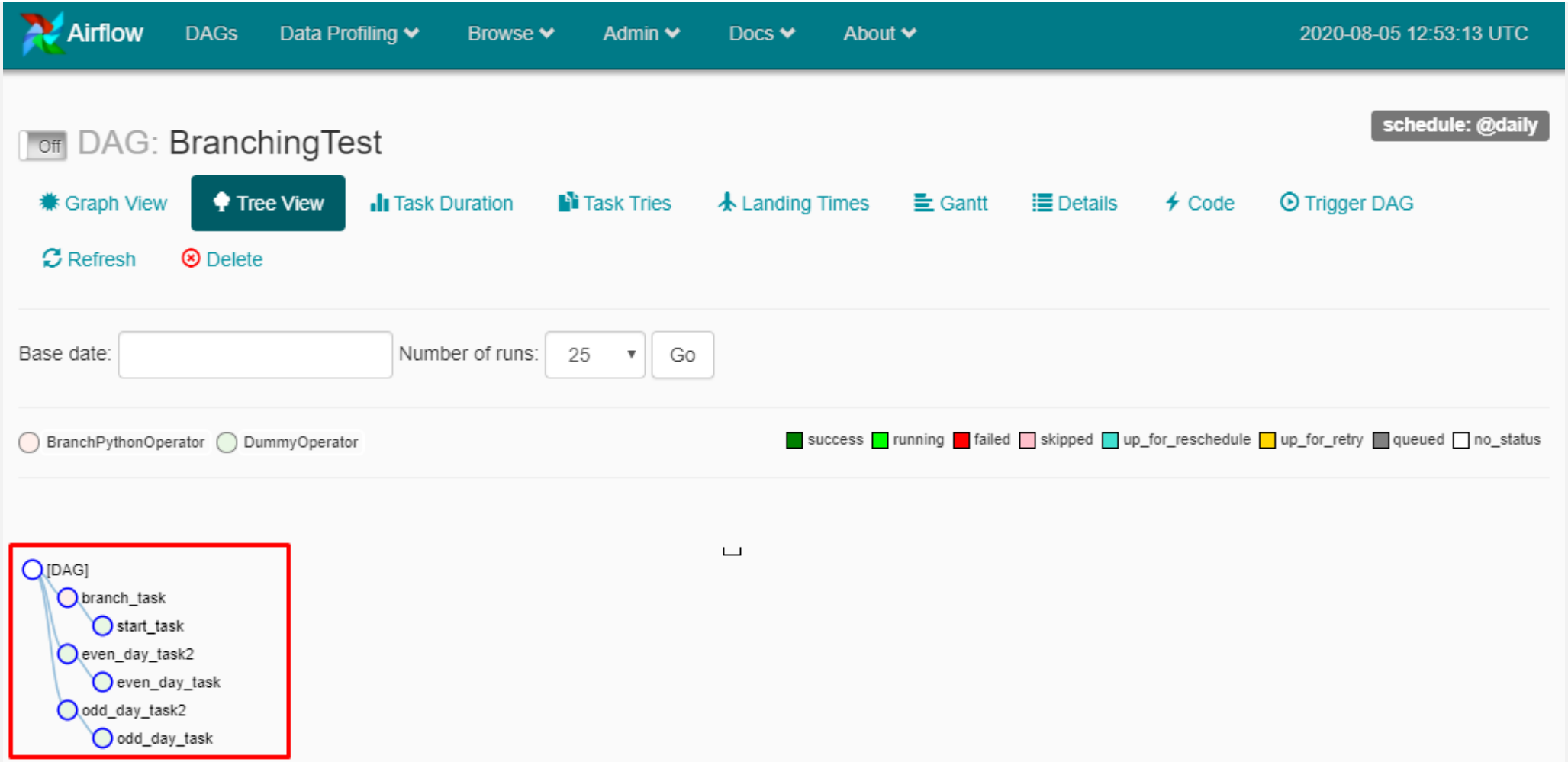
The DAG is defined for you, along with the tasks in question. Your current task is to implement the BranchPythonOperator.

```
# Create a function to determine if years are different
def year_check(**kwargs):
    current_year = int(kwargs['ds_nodash'][0:4])
    previous_year = int(kwargs['prev_ds_nodash'][0:4])
    if current_year == previous_year:
        return 'current_year_task'
    else:
        return 'new_year_task'

# Define the BranchPythonOperator
branch_task = BranchPythonOperator(task_id='branch_task',
                dag=branch_dag, python_callable=year_check,
                provide_context=True)

# Define the dependencies
branch_dag >> current_year_task
branch_dag >> new_year_task
```

This is a simple but effective use of branching to perform an occasional set of tasks without requiring significant code changes. Make sure to remember the various capabilities with branching to make your workflows more robust.

Branch troubleshooting

While working with a workflow defined by a colleague, you notice that a branching operator executes, but there's never any change in the DAG results. You realize that regardless of the state defined by the branching operator, all other tasks complete, even as some should be skipped.



Use what you've learned to determine the most likely reason that the branching operator is ineffective.

- The `branch_test` method does not return the correct value.
- The DAG does not run often enough for the callable to work properly.
- The dependency is missing between the `branch_task` and `even_day_task` and `odd_day_task` .

Answer: The dependency is missing between the `branch_task` and `even_day_task` and `odd_day_task`. Always remember to look for the simple issues first before trying to modify your code or processes too deeply.

Creating a production pipeline

To run a specific task from command-line:

```
airflow run <dag_id> <task_id> <date>
```

To run a full DAG:

```
airflow trigger_dag -e <date> <dag_id>
```

Operators recap:

- BashOperator - expects a `bash_command`
- PythonOperator - expects a `python_callable`
- BranchPythonOperator - requires a `python_callable` and `provide_context=True` . The callable must accept `**kwargs` .
- FileSensor - requires `filepath` argument and might need `mode` or `poke_interval` attributes

Creating a production pipeline #1

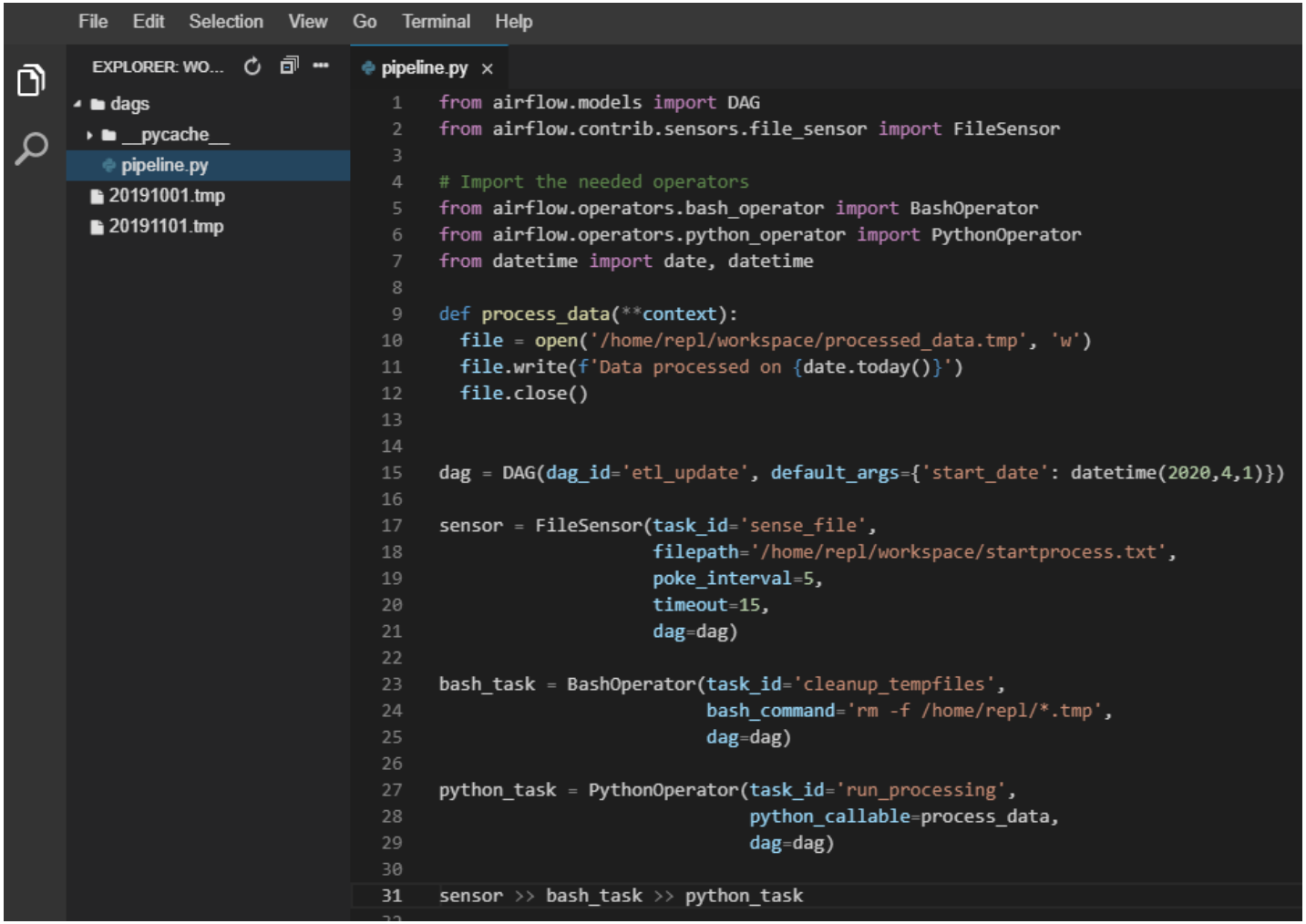
Now it's time to implement your workflow into a production pipeline consisting of many objects including sensors and operators. Your boss is interested in seeing this workflow become automated and able to provide SLA reporting as it provides some extra leverage for closing a deal the sales staff is working on. The sales prospect has indicated that once they see updates in an automated fashion, they're willing to sign-up for the indicated data service.

From what you've learned about the process, you know that there is sales data that will be uploaded to the system. Once the data is uploaded, a new file should be created to kick off the full processing, but something isn't working correctly.

Refer to the source code of the DAG to determine if anything extra needs to be added.

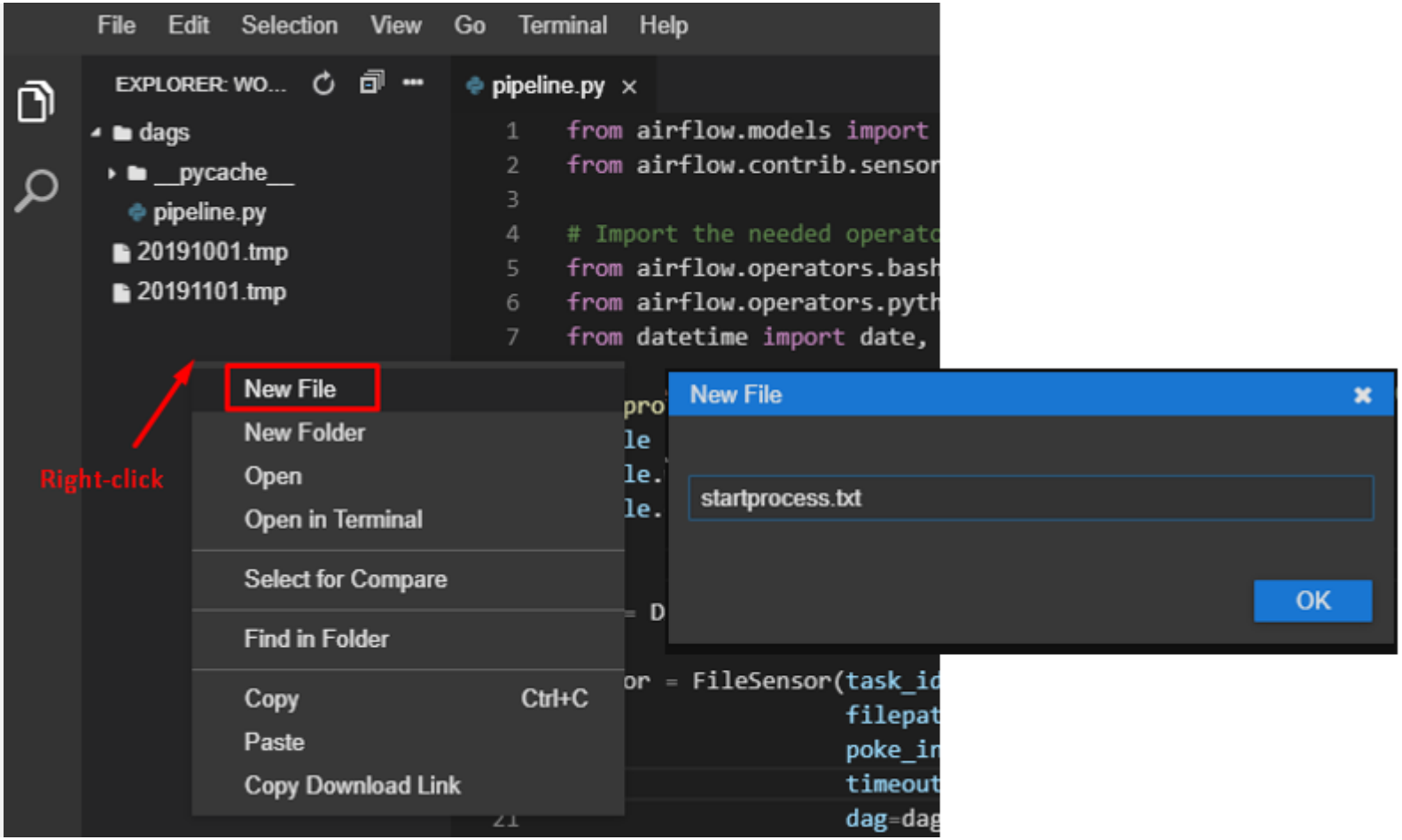
```
from airflow.models import DAG
from airflow.contrib.sensors.file_sensor import FileSensor

# Import the needed operators
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
from datetime import date, datetime

def process_data(**context):
    file = open('/home/repl/workspace/processed_data.tmp', 'w')
    file.write(f'Data processed on {date.today()}')
    file.close()

dag = DAG(dag_id='etl_update', default_args={'start_date': datetime(2020,4,1)} )

sensor = FileSensor(task_id='sense_file',
             filepath='/home/repl/workspace/startprocess.txt',
             poke_interval=5,
             timeout=15,
             dag=dag)
```

```
bash_task = BashOperator(task_id='cleanup_tempfiles',
                         bash_command='rm -f /home/repl/*.tmp',
                         dag=dag)

python_task = PythonOperator(task_id='run_processing',
                             python_callable=process_data,
                             dag=dag)

sensor >> bash_task >> python_task
```



Run this at command line: `airflow test etl_update sense_file -1`

Snap! Time is out. You ran the correct command though, to find out why the `sense_file` task would not complete. It's looking for a `startprocess.txt` file and it's not finding it, so it keeps poking every 5 seconds to see if it's there. You just need to create this file! You can use the `touch` command in the terminal, or right click and select "New File" in the menu on the left of the editor to create `startprocess.txt` (empty text file).



Run this at command line: `airflow test etl_update sense_file -1`

```
repl:~/workspace/dags$ airflow test etl_update sense_file -1
[2020-08-05 13:21:14,658] {__init__.py:51} INFO - Using executor SequentialExecutor
[2020-08-05 13:21:16,788] {dagbag.py:90} INFO - Filling up the DagBag from /home/repl/workspace/dags
[2020-08-05 13:21:16,816] {taskinstance.py:620} INFO - Dependencies all met for <TaskInstance: etl_update.sense_file 2020-08-0
1T00:00:00+00:00 [None]>
[2020-08-05 13:21:16,875] {taskinstance.py:620} INFO - Dependencies all met for <TaskInstance: etl_update.sense_file 2020-08-0
1T00:00:00+00:00 [None]>
[2020-08-05 13:21:16,876] {taskinstance.py:838} INFO -
--------------------------------------------------------------------------------
[2020-08-05 13:21:16,876] {taskinstance.py:839} INFO - Starting attempt 1 of 1
[2020-08-05 13:21:16,876] {taskinstance.py:840} INFO -
--------------------------------------------------------------------------------
[2020-08-05 13:21:16,876] {taskinstance.py:859} INFO - Executing <Task(FileSensor): sense_file> on 2020-08-01T00:00:00+00:00
[2020-08-05 13:21:17,225] {file_sensor.py:60} INFO - Poking for file /home/repl/workspace/startprocess.txt
[2020-08-05 13:21:17,226] {base_sensor_operator.py:123} INFO - Success criteria met. Exiting.
repl:~/workspace/dags$
```

Successful run!

You've just successfully modified and troubleshot a DAG within Airflow. Nice job verifying the `startprocess.txt` file existed to allow the DAG to continue. While this DAG is relatively simple, it implements many components of a production level workflow. These same troubleshooting principles can assist you when building a production system.

Creating a production pipeline #2

Continuing on your last workflow, you'd like to add some additional functionality, specifically adding some SLAs to the code and modifying the sensor components.

Refer to the source code of the DAG to determine if anything extra needs to be added. The default_args dictionary has been defined for you, though it may require further modification.

File Edit Selection View Go Terminal Help

EXPLORER: WO...

- dags
  - __pycache__
  - pipeline.py
  - process.py
  - processed_data.tmp
  - startprocess.txt

pipeline.py

```python
from airflow.models import DAG
from airflow.contrib.sensors.file_sensor import FileSensor
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
from dags.process import process_data
from datetime import timedelta, datetime

# Update the default arguments and apply them to the DAG
default_args = {
  'start_date': datetime(2019,1,1),
  'sla': timedelta(minutes=90)
}

dag = DAG(dag_id='etl_update', default_args=default_args)

sensor = FileSensor(task_id='sense_file',
                    filepath='/home/repl/workspace/startprocess.txt',
                    poke_interval=45,
                    dag=dag)

bash_task = BashOperator(task_id='cleanup_tempfiles',
                         bash_command='rm -f /home/repl/*.tmp',
                         dag=dag)

python_task = PythonOperator(task_id='run_processing',
                             python_callable=process_data,
                             provide_context=True,
                             dag=dag)

sensor >> bash_task >> python_task
```


```python
from airflow.models import DAG
from airflow.contrib.sensors.file_sensor import FileSensor
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
```

```
from dags.process import process_data
from datetime import timedelta, datetime

# Update the default arguments and apply them to the DAG
default_args = {
    'start_date': datetime(2019,1,1),
    'sla': timedelta(minutes=90)
}

dag = DAG(dag_id='etl_update', default_args=default_args)

sensor = FileSensor(task_id='sense_file',
                    filepath='/home/repl/workspace/startprocess.txt',
                    poke_interval=45,
                    dag=dag)

bash_task = BashOperator(task_id='cleanup_tempfiles',
                         bash_command='rm -f /home/repl/*.tmp',
                         dag=dag)

python_task = PythonOperator(task_id='run_processing',
                             python_callable=process_data,
                             provide_context=True,
                             dag=dag)

sensor >> bash_task >> python_task
```

You’ve correctly added support for SLAs in this DAG and modified the file sensor object to only look for its file every 45 seconds. These types of incremental improvements are often used when creating workflows in production. You may have also noticed that we’re using the `provide_context` entry with the PythonOperator, rather than just the BranchPythonOperator. Most operators within Airflow can accept the `provide_context` argument for the intended purpose.

Adding the final changes to your pipeline

To finish up your workflow, your manager asks that you add a conditional logic check to send a sales report via email, only if the day is a weekday. Otherwise, no email should be sent. In addition, the email task should be templated to include the date and a project name in the content.

```
from airflow.models import DAG
from airflow.contrib.sensors.file_sensor import FileSensor
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
from airflow.operators.python_operator import BranchPythonOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.email_operator import EmailOperator
from dags.process import process_data
from datetime import datetime, timedelta

# Update the default arguments and apply them to the DAG.
default_args = {
    'start_date': datetime(2019,1,1),
    'sla': timedelta(minutes=90)
}

dag = DAG(dag_id='etl_update', default_args=default_args)sensor = FileSensor(task_id='sense_file',
             filepath='/home/repl/workspace/startprocess.txt',
             poke_interval=45,
             dag=dag)

bash_task = BashOperator(task_id='cleanup_tempfiles',
                         bash_command='rm -f /home/repl/*.tmp',
                         dag=dag)

python_task = PythonOperator(task_id='run_processing',
                             python_callable=process_data,
                             provide_context=True,
                             dag=dag)

email_subject="""  Email report for {{ params.department }} on {{ ds_nodash }}  """
```

```
email_report_task=EmailOperator(task_id='email_report_task',
              to='sales@mycompany.com',
              subject=email_subject,
              html_content='',
              params={'department':'Data subscription services'},
              dag=dag)

no_email_task = DummyOperator(task_id='no_email_task', dag=dag)

def check_weekend(**kwargs):
    dt = datetime.strptime(kwargs['execution_date'],'%Y-%m-%d')
    #If dt.weekday() is 0-4, it's Mon-Fri. If 5-6, it's Sat/Sun
    if (dt.weekday() < 5):
        return 'email_report_task'
    else:
        return 'no_email_task'

branch_task = BranchPythonOperator(task_id='check_if_weekend',
                                   python_callable=check_weekend,
                                   provide_context=True,
                                   dag=dag)

sensor >> bash_task >> python_taskpython_task >> branch_task >> [email_report_task, no_email_task]
```

You've completed building a complex workflow using almost everything we've learned during this course — Operators, tasks, sensors, conditional logic, templating, SLAs, dependencies, and even alerting!

Summary

Congratulations! Let’s recap what we have learnt:

- Workflows / DAGs / Tasks
- Operators (BashOperator, PythonOperator, BranchPythonOperator, EmailOperator)
- Dependencies between tasks / Bitshift operators
- Sensors (to react to workflow conditions and state)
- Scheduling DAGs
- SLAs / Alerting to maintain visibility on workflows
- Templates for maximum flexibility when defining tasks
- Branching, to add conditional logic to DAGs
- Airflow interfaces: command line / UI
- Airflow executors
- Debugging / Troubleshooting

Next steps

- Set up your own environment for practice
- Explore other operators (eg. Amazon's S3, Postgresql) and sensors (eg. HDFS)
- Experiment with dependencies with a large number of tasks
- Look into parts of Airflow: XCom, Connections, etc
- Refer to Airflow documentations
- Keep building workflows

Happy learning Airflow!