Table of contents

Preface

Orchestration, or the routine scheduling and exection of dependent tasks, is a core component of modern data work. Orchestration continues to reach more and more data workers - it was originally a focus for data engineers, but it now permeates the work of data analysts, analytics engineers, data scientists, and machine learning engineers. The easier it is for any class of data worker to orchestrate their code, the easier it is for any member of an organization to derive value from the output of that code.

Flavors of Orchestration Code

Orchestration with Python is a vast and opinionated landscape, but there are three clear flavors of orchestration to have emerged over time:

1. **Object-oriented** orchestration, where tasks are objects and dependencies between tasks are handled with methods or operators (e.g. `>>`). Airflow's classic style is a good example of object-oriented orchestration.
2. **Decorative** orchestration, where tasks are functions and decorators are used to configure the tasks. Dependencies are often managed by passing the output of one function to the input of another. Airflow's taskflow API and Dagster's entire API are good examples of decorative orchestation.
3. **File-oriented** orchestration, where tasks are files, and dependencies are cleverly inferred or explicitly declared. Tools like Mage, dbt, and Orchest exemplify file-oriented orchestration.

What is gusty?

gusty is a file-oriented framework for Airflow, the absolute standard for orchestrators today. Airflow is a Top-Level Apache Project with sustained development, a gigantic ecosystem of provider packages, and is offered as a hosted service by major public clouds and other Airflow-focused companies. While other orchestrators natively support file-oriented orchestration, Airflow is such a good orchestrator that it was compelling to create a file-oriented framework for it. If you are reading this, you are likely already familiar with - or using - Airflow.

gusty exists to make file-oriented orchestration fun and easy using Airflow, allowing for file-oriented DAGs to be incorporated in existing Airflow projects without any need to change existing work or Airflow code. You can use any Airflow operator with gusty; gusty is simply a different way to write Airflow DAGs. This document hopes to serve as a guide for getting the most out of file-oriented orchestration in Airflow using gusty.

Part I

Getting Started

1 Basic DAG Structure

To familiarize ourselved with gusty, we'll start by making a simple DAG, called `hello_dag`.

A gusty DAG lives inside of your Airflow DAGs folder (by default `$AIRFLOW_HOME/dags`), and is comprised of a few core elements:

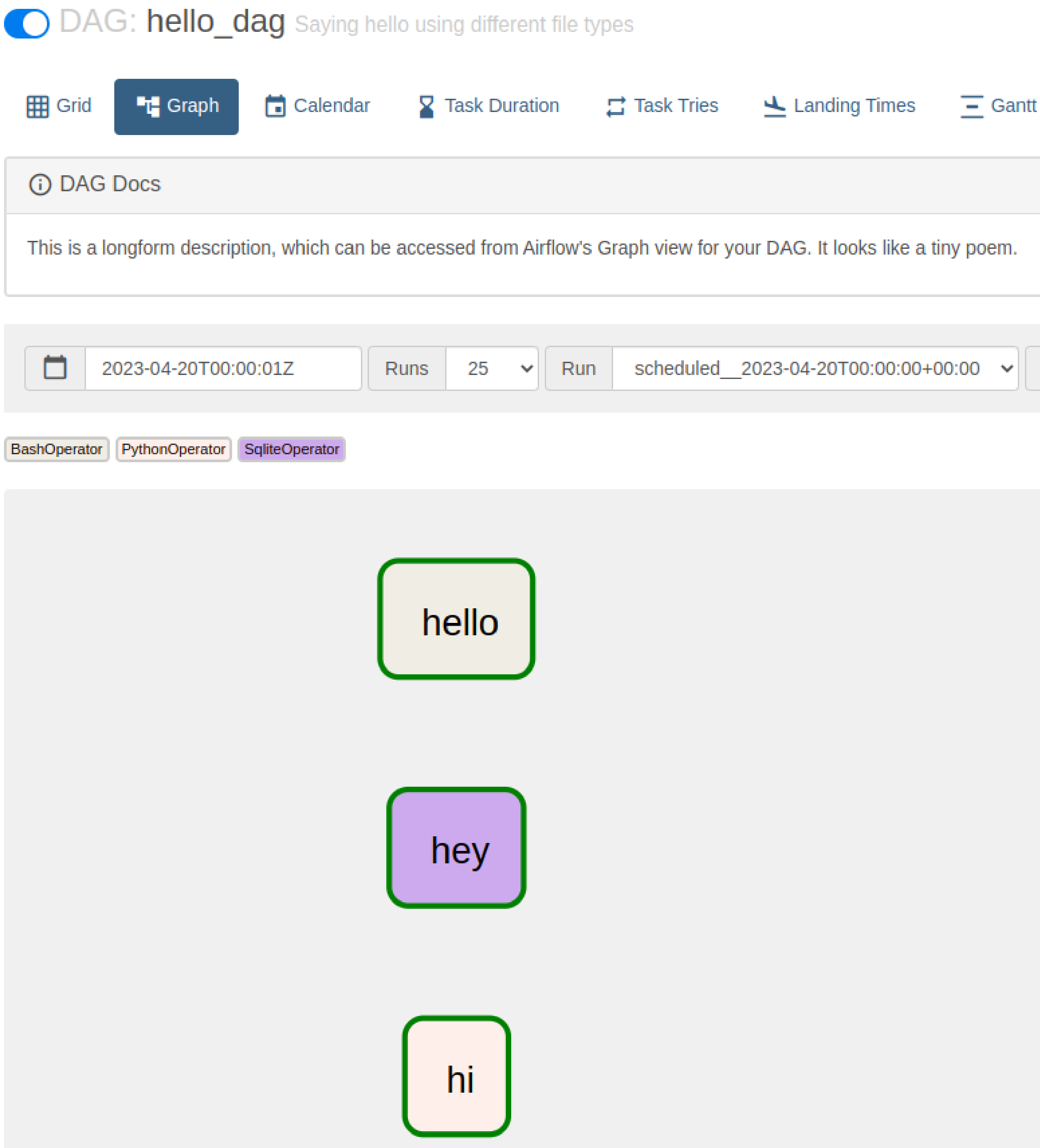
1. **Task Definition Files** - Each file hold specifications for a given task. In the example below, `hi.py`, `hey.sql`, and `hello.yml` are our Task Definition Files. These Task Definition Files are all stored inside our `hello_dag` folder.
2. **METADATA.yml** - This optional file contains any argument that could be passed to Airflow's DAG object, as well as some optional gusty-specifc argument. In the example below, `METADATA.yml` is stored inside of our `hello_dag` folder, alongside the Task Definition Files.
3. **DAG Generation File** - The file that turns a gusty DAG folder into an Airflow DAG. It's more or less like any other Airflow DAG file, and it will contain gusty's `create_dag` function. In the example below, `hello_dag.py` is our DAG Generation File. The DAG Generation File does *not* need to be named identically to the DAG folder.

```
$AIRFLOW_HOME/dags/

    hello_dag/
        METADATA.yml
        hi.py
        hey.sql
        hello.yml

    hello_dag.py
```

The contents of this `hello_dag` folder will produce the Airflow DAG seen below.

In the event you wanted to create a second gusty DAG, you can just repeat this pattern. For example, if we wanted to add `goodbye_dag`:

```
$AIRFLOW_HOME/dags/
```


DAG: hello_dag Saying hello using different file types
Grid
Graph
Calendar
Task Duration
Task Tries
Landing Times
Gantt
DAG Docs
This is a longform description, which can be accessed from Airflow's Graph view for your DAG. It looks like a tiny poem.
2023-04-20T00:00:01Z
Runs
25
Run
scheduled__2023-04-20T00:00:00+00:00
BashOperator
PythonOperator
SqliteOperator
hello
hey
hi

```
    goodbye_dag/
        METADATA.yml
        bye.py
        later.sql
        goodbye.yml
|
    hello_dag/
        METADATA.yml
        hi.py
        hey.sql
        hello.yml

    goodbye_dag.py
    hello_dag.py
```

1.1 Task Definition Files

The three primary file types used for Task Definition Files are Python, SQL, and YAML. gusty supports other file types, but these three are the most commonly used. The general pattern for Task Definition files is that they contain:

- **Frontmatter** - YAML which carries the specification and parameterization for the task. This can include which Airflow (or custom) operator to use, any keyword arguments to be passed to that operator, and any task dependencies the given task may have.
- **Body** - The primary contents of the task. For example, the Body of a SQL file is the SQL statement which will be executed; the body of a Python file can be the `python_callable` that will be ran by the operator. For YAML files, there is no Body because the whole Task Definition File is YAML.

gusty will pass any argument that can be passed to the `operator` specified (as well as any BaseOperator arguments) to the operator. The specified `operator` should be a full path to that operator.

The file name of each Task Definition File will become the name of the Airflow task.

Let's explore these different file types by looking at the contents of these Task Definition Files in `hello_dag`.

1.1.1 YAML Files with `hello.yml`

Here are the contents of our `hello.yml` file:

```yaml
operator: airflow.operators.bash.BashOperator
bash_command: echo hello
```

The resulting task would contain a `BashOperator` with the task id `hello`.

Because the entire file is YAML, there is no separation of Frontmatter and Body.

1.1.2 SQL Files with `hey.sql`

Here are the contents of our `hey.sql` file:

```sql
---
operator: airflow.providers.sqlite.operators.sqlite.SqliteOperator
---

SELECT 'hey'
```

The resulting task would contain a `SqliteOperator` with the task id `hey`.

The Frontmatter of our SQL file is encased in a set of triple dashes (`---`). The Body of the file is everything below the second set of triple dashes. For SQL files, the Body of the file is passed to the `sql` argument of the underlying operator. In this case, `SELECT 'hey'` would be passed to the `sql` argument.

1.1.3 Python Files with `hi.py`

Here are the contents of our `hi.py` file:

```python
# ---
# python_callable: say_hi
# ---

def say_hi():
  greeeting = "hi"
  print(greeeting)
  return greeeting
```

The resulting task would contain a `PythonOperator` with the task id `hi`.

The Frontmatter of our Python file is also encased in a set of triple dashes (`---`), but you will also note that the entirety of the Frontmatter, including the triple dashes, is prefixed by comment hashes (`#`).

By default, gusty will specify specify Airflow's PythonOperator as the `operator`, when no `operator` argument is provided. As with any Task Definition File, you can specify whatever `operator` is available to you in your Airflow environment, so you could just as easily add `operator: airflow.operators.python.PythonVirtualenvOperator` to this Frontmatter to use the `PythonVirtualenvOperator` instead of the `PythonOperator`.

When a `python_callable` is specified in the Frontmatter of a Python file, gusty will search the Body of the Python file for a function with the name specified in the Frontmatter's `python_callable` argument. For the best results with Python files, it's recommended that you put all of the Body contents in a named function, as illustrated above.

1.2 METADATA.yml

The `METADATA.yml` file is a special file for passing DAG-related arguments to Airflow's DAG object. Airflow's DAG object takes arguments like `schedule` (when you want your DAG to run), `default_args.start_date` (how far back you want your DAG to start), `default_args.email` (who should be notified if a task in DAG fails), and more. The `METADATA.yml` file is a convenient way to pass this information to Airflow.

Let's look at the contents of the `METADATA.yml` file in our `hello_dag` folder:

```yaml
description: "Saying hello using different file types"
doc_md: |-
  This is a longform description,
  which can be accessed from Airflow's
  Graph view for your DAG. It looks
  like a tiny poem.
schedule: "0 0 * * *"
catchup: False
default_args:
    owner: You
    email: you@you.com
    start_date: !days_ago 28
    email_on_failure: True
    email_on_retry: False
    retries: 1
    retry_delay: !timedelta
      minutes: 5
```

```yaml
tags:
  - docs
  - demo
  - hello
```

The above METADATA.yml configures a DAG that runs once a day (`schedule: "0 0 * * *"`), has a start date of 28 days ago (`default_args.start_date: !days_ago 28`), and is tagged with the tags `docs`, `demo`, and `hello`. It also adds a `description`, a `doc_md`, and more, but every argument here is simply an argument in Airflow's DAG object.

The only thing that you might not have seen before are *YAML constructors*, as illustrated above in the `default_args.start_date` (using `!days_ago`) and `default_args.retry_delay` (using `!timedelta`) arguments, which are calling *functions* inside of YAML. In short, YAML constructors are just Python functions that are called when your YAML (or any Task Definition File Frontmatter) is loaded. We'll discuss YAML constructors more in later sections, but they are a powerful way to control file-oriented DAGs and tasks, and help ensure you have just as much control over your DAGs as writing them any other way.

We'll also cover gusty-specific `METADATA.yml` later on, but for now, all you need to know is that the `METADATA.yml` file is used for passing arguments to Airflow's DAG object.

1.3 DAG Generation File

Finally, let's look at the DAG Generation File that ultimately generates the Airflow DAG, `hello_dag.py`:

```python
import os
from gusty import create_dag

# There are many different ways to find Airflow's DAGs directory.
# hello_dag_dir returns something like: "/usr/local/airflow/dags/hello_dag"
hello_dag_dir = os.path.join(
  os.environ["AIRFLOW_HOME"],
  "dags",
  "hello_dag")

hello_dag = create_dag(hello_dag_dir, latest_only=False)
```

gusty's `create_dag` function takes as its first argument the path to a directory containing Task Definition Files, in our case the `hello_dag` direcory. Any keyword argument that can be passed to Airflow's DAG object can be passed to `create_dag`, where any arguments that are specified *both* in `create_dag` and `METADATA.yml` will take the value specified in `METADATA.yml`.

Additionally, `create_dag` takes some gusty-specific arguments, one of which is illustrated here: `latest_only=False`, which disables gusty's default behavior of installing a `LatestOnlyOperator` at the absolute root of an Airflow DAG. You can read more about the `LatestOnlyOperator` in Airflow's documentation, but setting `latest_only=False` will ensure a gusty-generated DAG mirrors Airflow's default behavior.

In subsequent chapters, we'll cover more of gusty's capabilities, but these are the core components of generating a file-oriented Airflow DAG with gusty!

2 Task Dependencies

Task orchestration often involves ensuring tasks run in a specific order. With gusty, there are three ways to specify task dependencies:

1. A **dependencies block** in a task's Frontmatter, where you can pass a list of task ids in the current dag upon which the current task depends.
2. An **external dependencies block** in a task's Frontmatter, where you can pass a list of `dag_id: task_id` combinations for tasks in *other* dags upon which the current task depends.
3. A **dependencies attribute** on your custom operator, which is a list of task ids in the current dag upon which the current task depends. This powerful option allows for task dependencies to generated dynamically and automatically.

In this section, we'll focus on the the dependencies external dependencies blocks, available for use in any Task Definition File's Frontmatter.

We'll continue using our `hello_dag` example from the previous chapter.

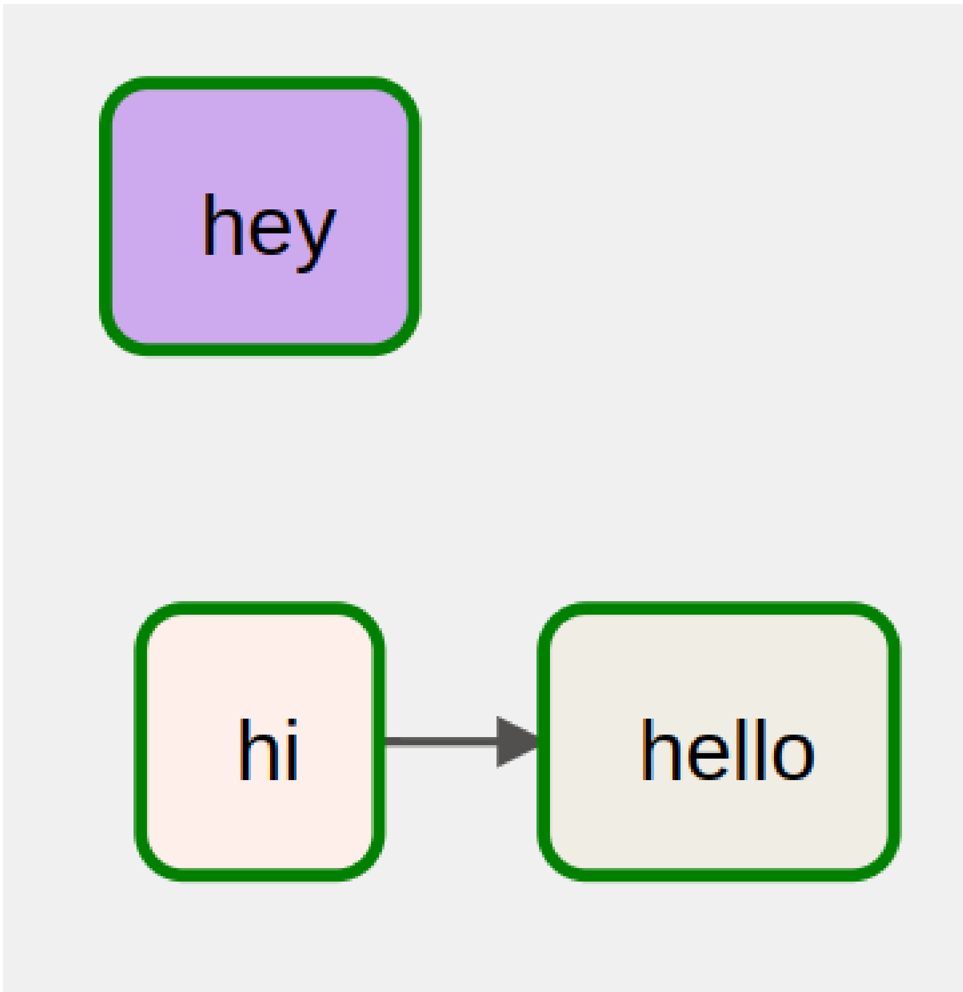
Let's say that our `hello` task depended on our `hi` task running before it. To specify this dependency, we would add the `hi` task to a list in the `dependencies` block of the `hello.yml` Task Definition File:

```yaml
operator: airflow.operators.bash.BashOperator
dependencies:
  - hi
bash_command: echo hello
```

Now, in our Airflow UI, our DAG graph will show that `hi` precedes `hello`.

Remember, in gusty, the file name (minus the file extension) becomes the task id, so you do not need to specify `hi.py`, just `hi`.

You can list as many dependencies as you need to for any task.


hey
hi
hello

2.1 External Dependencies

A common pattern in Airflow is to have tasks in one DAG depend on tasks in another DAG, or to have one DAG depend completely on another DAG. This behavior is possible in gusty by using the `external_dependencies` block. The `external_dependencies` block accepts a list of key-value pairs where each key is a DAG id and each value is a task id.

For each key-value pair listed in the `external_dependencies` block, gusty will generate an ExternalTaskSensor, a built-in Airflow sensor, and place the resulting sensor task upstream of the given dependent task. If the same external dependency is specified across multiple tasks, gusty will only create one sensor and place this one sensor upstream of all tasks with the specified external dependency.

There are a few ways to configure external dependencies, and we'll look at all of them below.

2.1.1 Single Task External Dependency

Let's keep building up our `hello.yml` Task Definition File.

To specify that our `hello` task depends on an upstream task, which we'll call `upstream_task`, in an upstream DAG, which we'll call `upstream_dag`, we add the following `external_dependencies` block:

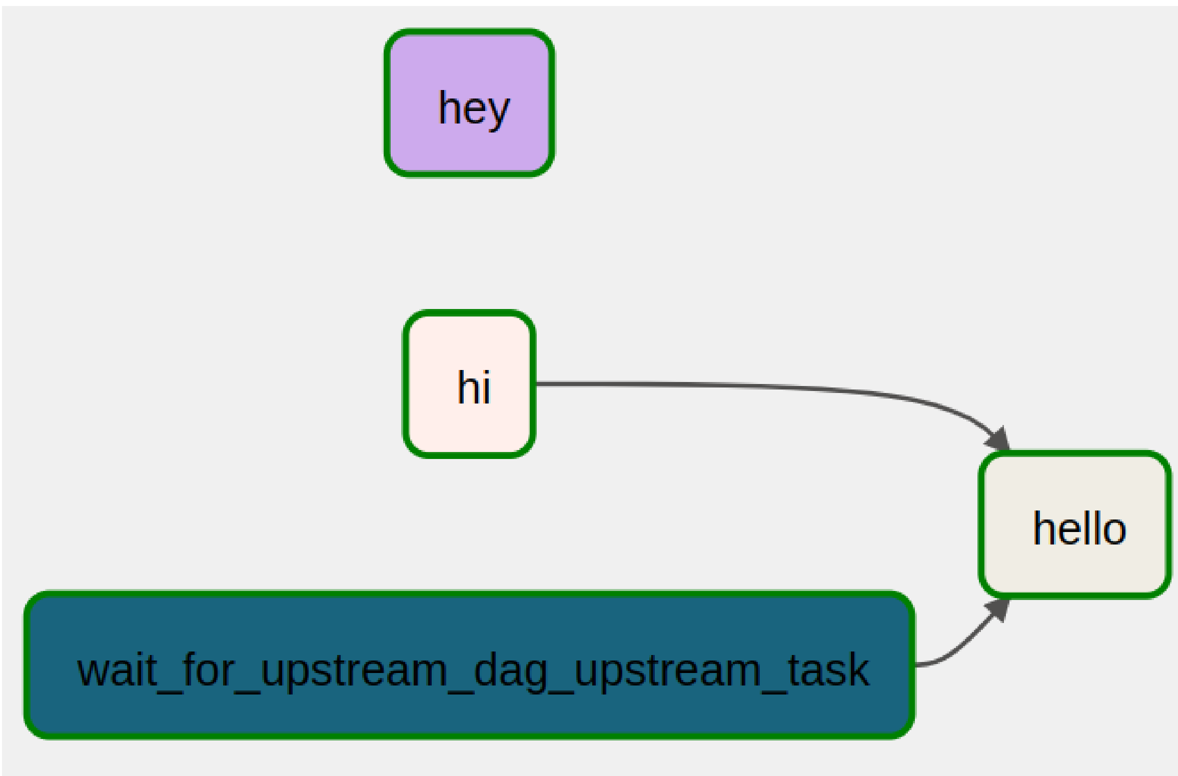
```yaml
operator: airflow.operators.bash.BashOperator
dependencies:
  - hi
external_dependencies:
  - upstream_dag: upstream_task
bash_command: echo hello
```

The result will be a new `ExternalTaskSensor` task with the task id `wait_for_upstream_dag_upstream_task`, preceding the existing `hello` task.

As with `dependencies`, you can list as many external dependecies as you require.

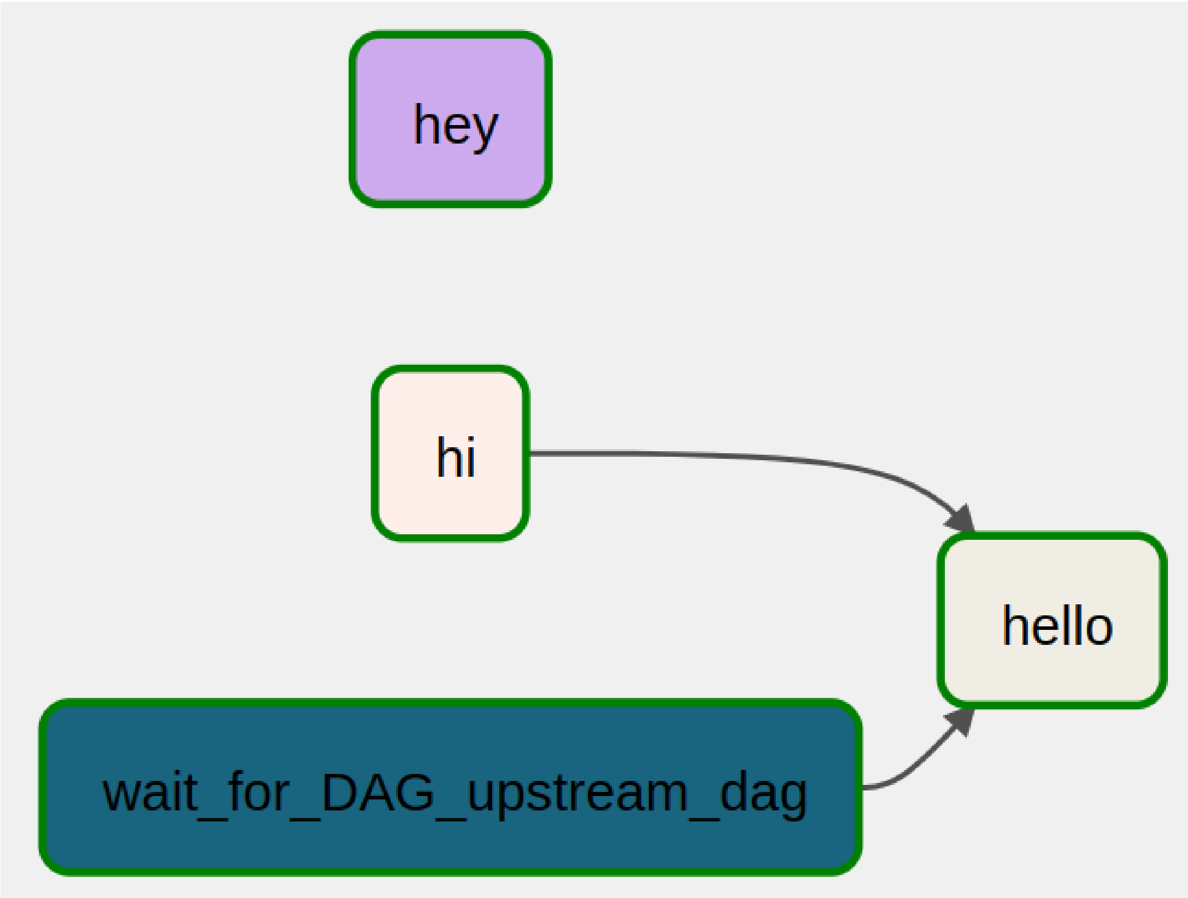
2.1.2 Whole DAG External Dependency

An alternative to speciying a single task for an external dependency is to specify that the *entire* upstream DAG is the dependency. In this case, we use the special keyword `all` to configure the `ExternalTaskSensor` to wait for the entire DAG:


hey
hi
hello
wait_for_upstream_dag_upstream_task

```yaml
operator: airflow.operators.bash.BashOperator
dependencies:
  - hi
external_dependencies:
  - upstream_dag: all
bash_command: echo hello
```

The result will be a new `ExternalTaskSensor` task with the task id `wait_for_DAG_upstream_dag`, preceding the existing `hello` task.

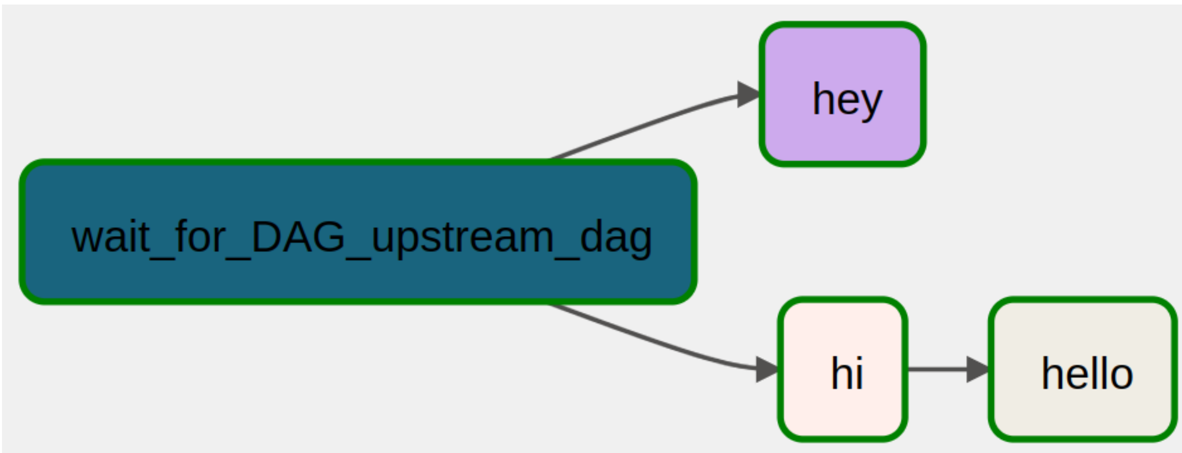


2.1.3 External Dependencies in METADATA.yml

As an Airflow project grows, you might find that more and more of your tasks have the same external dependency, or sometimes DAGs just logically *should* depend on one another (e.g. a DAG that ingests data should precede a DAG that transforms that data). For these cases, you can utilize the same exact same `external_dependencies` block in any `METADATA.yml` file.

When you specify an external dependency in a `METADATA.yml` file, the `ExternalTaskSensor` task will be placed at the root of the DAG, ensuring that no tasks in the DAG run before the `ExternalTaskSensor` task completes.

Here's what it would like like if we took the same external dependency from above and place it in an `external_dependencies` block in `METADATA.yml` instead:



The `ExternalTaskSensor` now precedes every other task in the graph.

2.1.4 Offset Schedules

Understandably, but frustratingly, the default behavior of Airflow's `ExternalTaskSensor` is to look for DAG runs that have that have ran at the same "logical date". This means that if you have one DAG scheduled to run daily at 00:00 UTC (`"0 0 * * *"`), let's call this DAG `earlier_dag`, and another DAG scheduled to run daily at 06:00 UTC (`"0 6 * * *"`), let's call this DAG `later_dag`, and you specify an external dependency between `later_dag` and `earlier_dag`, the default syntax for an `external_dependencies` block will not work, because - in the case where `later_dag` depends on `earlier_dag` - the `ExternalTaskSensor` in `later_dag` will be looking for an 06:00 UTC DAG run of `earlier_dag`, which does not exist.

Fortunately, the `external_dependencies` block accepts an alternative syntax for this scenario, where:

- The keys under `external_dependencies` are the external DAG ids.
- A `tasks` list is provided for a given external DAG.
- Additional configuration for the `ExternalTaskSensor` class, such as the `execution_delta`, can be passed in.

For example, to configure `later_dag` (06:00 UTC) to depend on `earlier_dag` (00:00 UTC), we could add the following block to `later_dag`'s `METADATA.yml`:

```yaml
external_dependencies:
  earlier_dag:
    execution_delta: !timedelta
      hours: 6
    tasks:
      - all
```

This will ensure the resulting `wait_for_DAG_earlier_dag` looks for a successful `earlier_dag` DAG run at 00:00 UTC (`later_dag`'s 06:00 UTC run minus 6 hours).

2.1.5 Alternative Approaches to Offset Schedules

2.1.5.1 Custom Sensors

It's possible to create a custom sensor that "doesn't care" about the logical date, and just looks at the last/latest DAG run. This ensures you don't have to worry about setting any offset schedules.

Here is a small snippet inspired by the cal-itp/data-infra repo (which they since deleted in this commit):

```python
from airflow.utils.db import provide_session
from airflow.sensors.external_task_sensor import ExternalTaskSensor


class LastDagRunSensor(ExternalTaskSensor):
    def __init__(self, external_dag_id, external_task_id=None, **kwargs):
        super().__init__(
          external_dag_id=external_dag_id,
          external_task_id=external_task_id,
          **kwargs)

        def dag_last_exec(crnt_dttm):
            return self.get_dag_last_execution_date(self.external_dag_id)

        self.execution_date_fn = dag_last_exec

    @provide_session
    def get_dag_last_execution_date(self, dag_id, session):
```

```python
        from airflow.models import DagModel

        q = session.query(DagModel).filter(DagModel.dag_id == self.external_dag_id)

        dag = q.first()
        return dag.get_last_dagrun().logical_date
```

In the event you wanted to use this `LastDagRunSensor` as the sensor class for the external dependencies in your gusty DAG, you could do so by using the `wait_for_class` argument available in `create_dag`. For example, here's what your `later_dag.py` DAG file might look like if you decided to do so:

```python
import os
from gusty import create_dag
# Wherever you store the code for the above sensor..
from plugins.sensors import LastDagRunSensor

later_dag_dir = os.path.join(
  os.environ["AIRFLOW_HOME"],
  "dags",
  "later_dag")

later_dag = create_dag(
  later_dag_dir,
  wait_for_class=LastDagRunSensor,
  latest_only=False)
```

Now all of the external dependencies defined in the `later_dag`'s Task Definition Files will use the custom `LastDagRunSensor` instead of the default `ExternalTaskSensor`.

2.1.6 Other External Dependency Considerations

You can configure your external dependencies further using the `wait_for_defaults` argument in `create_dag`, which accepts a dictionary of arguments that are available to Airflow's ExternalTaskSensor. Here is the subset of parameters available in `wait_for_defaults`:

- `poke_interval`
- `timeout`
- `retries`
- `mode`
- `soft_fail`
- `execution_delta`

- `execution_date_fn`
- `check_existence`

Additionally, anything available to BaseOperator will be passed through.

2.1.6.1 Set mode to reschedule

By default in Airflow, sensors run in `mode="poke"`, which means they take up a worker slot for the entire time they are waiting for the external task/DAG to complete. You can set `mode="reschedule"` to free up the worker slot in between "pokes". Building on the `create_dag` call in `later_dag.py` above:

```
later_dag = create_dag(
  later_dag_dir,
  wait_for_class=LastDagRunSensor,
  wait_for_defaults={
    "mode": "reschedule"
    },
  latest_only=False)
```

2.1.6.2 Set a timeout

By default in gusty, external dependencies will timeout after 1 hour, or 3600 seconds. If you want to wait longer, you can set your `timeout`, in seconds:

```
later_dag = create_dag(
  later_dag_dir,
  wait_for_class=LastDagRunSensor,
  wait_for_defaults={
    "mode": "reschedule",
    "timeout": 7200 # 2 hours in seconds
    },
  latest_only=False)
```

2.1.6.3 Learn More

If you want to learn more about sensors, check out Airflow's BaseSensorOperator and Airflow's BaseOperator.

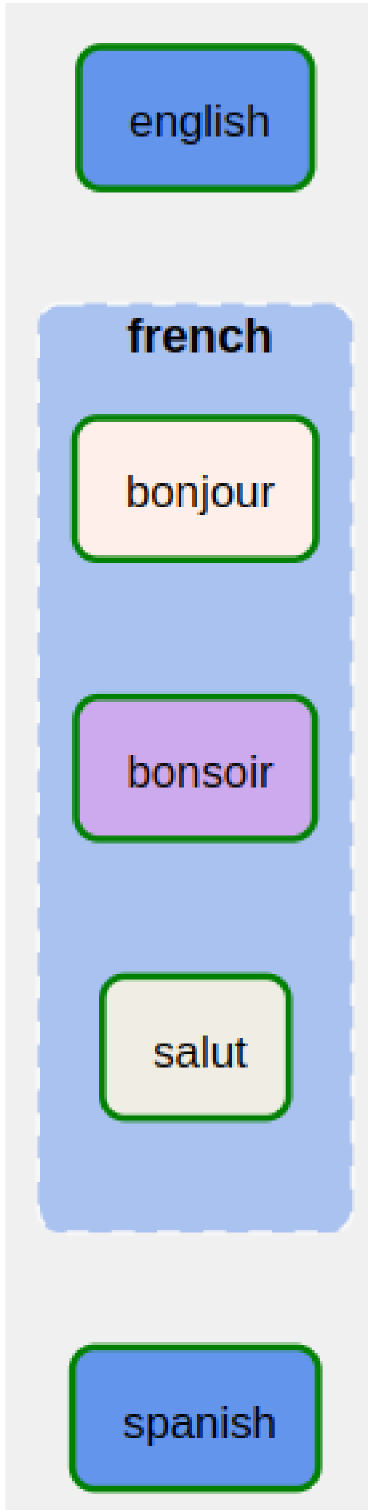
3 Task Groups

In Airflow, a TaskGroup is a fairly arbitrary - but often useful - grouping of tasks. In gusty, you can organize your Task Definition Files into Airflow task groups by simply adding subfolders to your DAG folder, and putting your Task Definition Files in each subfolder. As you might have guessed, the task group id is the same as the subfolder name.

For example, maybe we want our `hello_dag` to have it tasks organized into separate task groups based on the language of the greeting. Here's what the updated structure of our `hello_dag` DAG might look like:

```
$AIRFLOW_HOME/dags/

  hello_dag/

     english/
        hello.yml
        hey.sql
        hi.py

     french/
        bonjour.py
        bonsoir.sql
        salut.yml

     spanish/
        hola.py
        oye.sql
        saludos.yml

     METADATA.yml

  hello_dag.py
```

Now, our Airflow DAG will have three task groups: `english`, `french`, and `spanish`. Each task group will contain the tasks found in each folder. For example, the `french` task group will contain tasks `bonjour`, `bonsoir`, and `salut`.


english
french
bonjour
bonsoir
salut
spanish

By default, gusty does *not* prefix a task group's name on to the task name. Any altering of a task name inside of a task group is done so explcitly. How? Just like DAG folders, task group folders can also leverage their own `METADATA.yml` files.

You can add a `METADATA.yml` file to any task group folder. This is useful for when you want to have specific task group behavior, such as different `default_args` or if you want to prefix or suffix the task group id onto the task id.

Let's add a `METADATA.yml` file to our `spanish` task group subfolder:

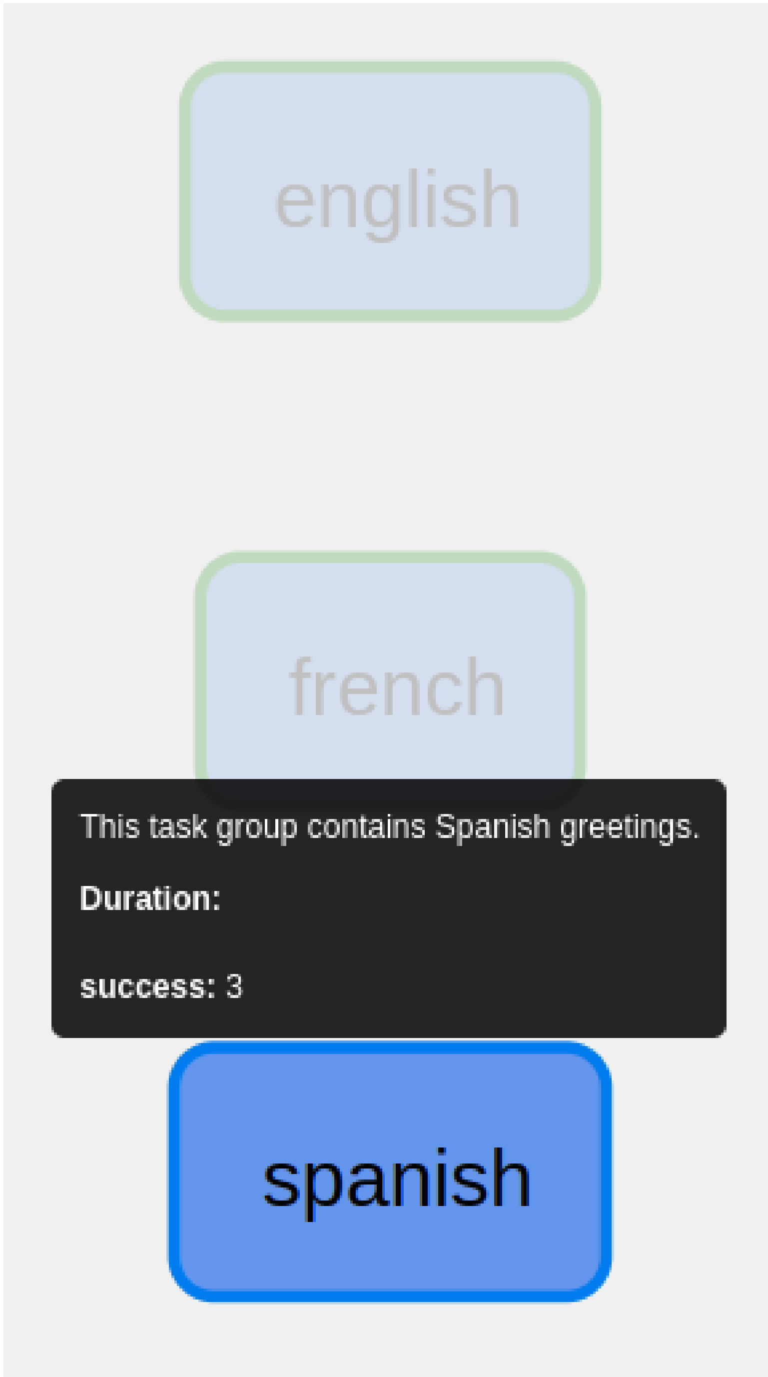
```
dags/

  hello_dag/

     english/
        hello.yml
        hey.sql
        hi.py

     french/
        bonjour.py
        bonsoir.sql
        salut.yml

     spanish/
        METADATA.yml
        hola.py
        oye.sql
        saludos.yml

     METADATA.yml

  hello_dag.py
```

The contents of this task group `METADATA.yml` file might look something like this:

```yaml
tooltip: "This task group contains Spanish greetings."
prefix_group_id: True
```

The above `METADATA.yml` will give the `spanish` task group a tooltip, when hoving over the node in the Airflow UI's graph view, and all tasks in the task group will be prefixed with `spanish_`, such as `spanish_oye` and `spanish_saludos`.

Here is a look at the tooltip on hover:


english
french
This task group contains Spanish greetings.
Duration:
success: 3
spanish

And here is look at the prefixed Spanish tasks:

Lastly, just like tasks, task group `METADATA.yml` can take advantage of `dependencies` blocks. So if a lot of tasks depend on the same upstream task, it might make sense to put them in the same task group folder, and set the upstream dependency in the `METADATA.yml`.

3.1 Why Use Task Group Folders?

Task groups folders serve a few powerful purposes at scale:

- They help keep Task Definition Files organized.
- They can help keep parts of a DAG logically compartmentalized.
- They can help keep dependencies between sets of tasks easier to manage.


english
french
spanish
spanish_hola
spanish_oye
spanish_saludos

4 Many DAGs

While `create_dag` is great for creating a single DAG, part of what makes gusty so convenient is that creating any number of new DAGs can be as easy as just making a new folder in a directory. Once you get to the point where you're just creating new folders full of Task Definition Files and metadata, you and your team get to think about Airflow less and can instead focus on defining the core components of your workflows.

To help facilitate this growth, gusty also provides a `create_dags` function, for generating multiple DAGs. With `create_dags`, instead of passing a path to a *single* DAG folder, you'll pass in a directory path where *many* DAG folders reside.

In the example below, we'll make a "home" for all of our gusty DAGs inside a directory named `gusty_dags`. Inside the `gusty_dags` directory are two DAGs, `hello_dag` and `goodbye_dag`.

```
$AIRFLOW_HOME/dags/

  gusty_dags/

     goodbye_dag/
        METADATA.yml
        goodbye.yml

     hello_dag/
        METADATA.yml
        hello.yml


  gusty_dags.py
```

Now, we'll use the `create_dags` function in `gusty_dags.py` to generate *multiple* DAGs in a single file! Here's what our `gusty_dags.py` file looks like:

```
import os
from gusty import create_dags
from gusty.utils import days_ago
```

```python
# gusty_dags_dir returns something like: "/usr/local/airflow/dags/gusty_dags"
gusty_dags_dir = os.path.join(
  os.environ["AIRFLOW_HOME"],
  "dags",
  "gusty_dags")

create_dags(
  gusty_dags_dir,
  globals(),
  schedule="0 0 * * *",
  catchup=False,
  default_args={
    "owner": "you",
    "email": "you@you.com",
    "start_date": days_ago(1)
  },
  wait_for_defaults={
    "mode": "reschedule"
  },
  extra_tags=["gusty_dags"],
  latest_only=False)
```

The above will create both `hello_dag` and `goodbye_dag` DAGs, which reside inside of the `gusty_dags_dir` defined in `gusty_dags.py`.

The second argument, `globals()`, assigns the DAGs to the global environment, so Airflow can find the DAGs.

`schedule`, `catchup`, `default_args` are arguments available in the Airflow DAG object.

`wait_for_defaults`, `extra_tags`, and `latest_only` are all gusty-specific `create_dag` arguments. `wait_for_defaults` and `latest_only` were previously discussed here and here. `extra_tags` are additional tags appended to any existing `tags` specified in either `create_dag` or a `METADATA.yml` file.

4.1 The Power of `create_dags`

The value in `create_dags` is that multiple DAGs can be created with common schedules, default arguments, tags, and more, *plus* each DAG can contain DAG-specific information, such as documentation (e.g. `description` and `doc_md`) and tags, inside their own `METADATA.yml`.

In gusty, `METADATA.yml` takes precedence over any `create_dag` argument, so you can override anything set in `create_dags` with the DAG-specific `METADATA.yml`.

Now you have the building blocks to use file-oriented orchestration in Airflow with gusty!

Part II

Doing More

5 Using Constructors

Constructors are functions you can invoke in your YAML. These functions are invoked every time your Task Definition File is loaded during gusty's DAG creation process.

Constructors are available to us thanks the PyYAML package.

To better understand constructors, let's orient ourselves around a simple Python function, called `double_it`:

```
def double_it(x):
  return x + x
```

If we were to run `double_it(2)`, we'd get back `4`.

To invoke `double_it` from YAML, we begin our value entry with an exclaimation point (`!`), as illustrated below:

```
some_argument: !double_it 2
```

When this YAML is loaded, the argument `some_argument` in our YAML will be assigned the value `4`.

You can also use keyword arguments (i.e. `double_it(x=2)`) with constructors:

```
some_argument: !double_it
  x: 2
```

The above will still result in `some_argument` taking on the value of `4`.

5.1 Using Constructors with gusty

gusty makes it easy for you to leverage YAML contructors. The simplest way to leverage your functions as YAML constructors within gusty is to use the Airflow DAG object's built-in `user_defined_macros` argument. When you pass a dictionary of functions/macros to `user_defined_macros`, gusty will make all of those functions/macros available to you as YAML constructors.

Your call to `create_dag` might look something like this:

```
create_dag(
  ...,
  user_defined_macros={
    "double_it": double_it
  }
)
```

Then, in a Task Definition File, you could leverage `double_it` both as a YAML constructor, as well as - just as in any other Airflow task - using Jinja. Here's a `BashOperator` example below.

```
operator: airflow.operators.bash.BashOperator
retries: !double_it 4
bash_command: echo {{ double_it("hello") }}
```

The above would result in a task with 8 retries and a bash command that (when executed) would echo `hellohello`.

An important note on the timing of function evaluation: `double_it` is used twice above, once as a YAML constructor in the `retries` argument and once as a Jinja macro in the `bash_command` argument. The YAML constructor will be evaluated every time the DAG is generated, which is once every few minutes by default (in Airflow). The Jinja macro will only be evaluated when the task is executed.

5.2 Built-in Constructors

5.2.1 gusty

There are a few built-in constructors gusty contains, primarily to make creating a DAG using `METADATA.yml` easy. The three built-in constructors are datetime, timedelta, and `days_ago`, which simply provides a datetime object for as many days ago you specify.

5.2.2 ABSQL

The YAML loading functionality for gusty is maintained in a separate, lightweight project called ABSQL.

The ABSQL package ships with a handful of default functions, which are also available to you as both YAML constructors and macros within gusty DAGs.

6 Multi-tasking

Sometimes orchestration involves repetition. For example, you might have a DAG with 3 different tasks that fetch stock data for 3 different stock symbols. To create this DAG, you'd likely use a `for` loop. You can achieve this `for` loop style task generation within gusty using some special frontmatter blocks:

- `multi_task_spec` - For iterating over arguments to be passed to an operator.
- `python_callable_partials` - For iterating over arguments to be passed to the function assigned to a `python_callable` argument.

In both `multi_task_spec` and `python_callable_partials`, the keys below each block will be the task id for a given task, and the arguments below each task id will be passed to that operator or callable, respectively.

We'll look at examples of each below.

Imagine we want three `BashOperator` tasks to echo "hi", "hey", and "hello world".

We can define all three tasks in a single Task Definition File, which we'll call `multi_greeing.yml`. The name of the Task Definition File is arbitrary. Here are its contents:

```yaml
operator: airflow.operators.bash.BashOperator
bash_command: echo $GREETING
multi_task_spec:
  say_hi:
    env:
      GREETING: hi
  say_hey:
    env:
      GREETING: hey
  say_hello_world:
    env:
      GREETING: hello
    bash_command: echo $GREETING world
```

The above Task Definition File will create three tasks: `say_hi`, `say_hey`, and `say_hello_world`. The tasks `say_hi` and `say_hey` both inherit the same `bash_command`, but have different `env` arguments. The `say_hello_world` task also contains its own `env` argument, but goes a step further as to define its own `bash_command`.

This `multi_task_spec` produces the following graph:

This powerful syntax allows you to keep your task definitions DRY. In this example, every task has a dedicated, unique `env` argument. In the case of `say_hi` and `say_hey`, they share a common `bash_command`. In the case of `say_hello_world`, it gets its own `env` and `bash_command`. Very flexible!

6.1 `python_callable_partials`

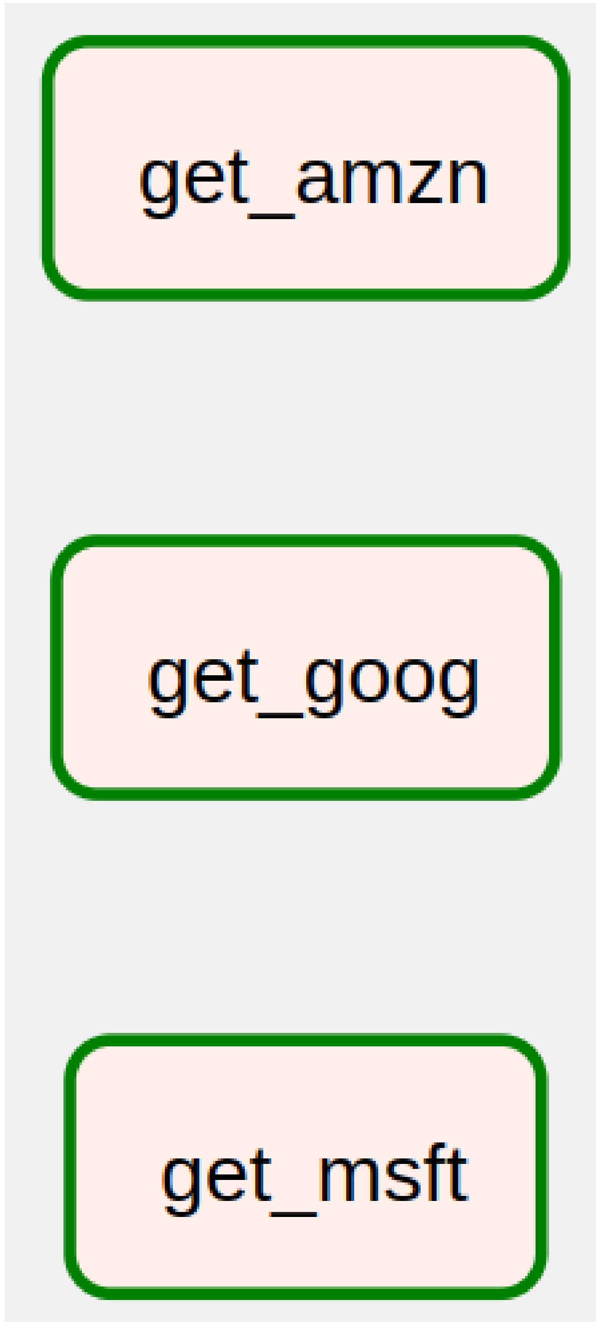
Similar to `multi_task_spec`, `python_callable_partials` allows you to generate multiple tasks in a single file, except instead of passing arguments to an operator, you pass arguments directly to a `python_callable`.

In the example Task Definition File below, we'll create a few tasks to fetch the past year's stock data from yfinance for three different stock symbols: `AMZN`, `GOOG`, and `MSFT`.

```
# ---
# python_callable: main
# python_callable_partials:
#   get_amzn:
#     symbol: AMZN
#   get_goog:
#     symbol: GOOG
#   get_msft:
#     symbol: MSFT
# ---

def main(symbol):
  from yfinance import Ticker

  stock = Ticker(symbol)
  history = stock.history(period='1y', interval='1d').reset_index()
  history["Symbol"] = symbol
  print(history.head())
```

The above Task Definition File will create three tasks: `get_amzn`, `get_goog`, and `get_msft`. Each task will have its respective `symbol` passed to the the `main` function.


say_hello_world
say_hey
say_hi


get_amzn
get_goog
get_msft

6.2 Mixing It Up

`multi_task_spec` and `python_callable_partials` are non-exclusive, so you can mix and match configuration as needed.

Let's build upon our `yfinance` example, and instead of using the default `PythonOperator`, let's use the PythonVirtualenvOperator, and change the requirements for our `get_amzn` task.

```
# ---
# operator: airflow.operators.python.PythonVirtualenvOperator
# python_callable: main
# python_callable_partials:
#   get_amzn:
#     symbol: AMZN
#   get_goog:
#     symbol: GOOG
#   get_msft:
#     symbol: MSFT
# multi_task_spec:
#   get_amzn:
#     requirements:
#       - yfinance==0.1.96
# ---

def main(symbol):
  from yfinance import Ticker

  stock = Ticker(symbol)
  history = stock.history(period='1y', interval='1d').reset_index()
  history["Symbol"] = symbol
  print(history.head())
```

In the above example, we changed two things:

- We are now explicitly using the `PythonVirtualenvOperator` via the `operator` entry.
- Our `get_amzn` task also gets an entry in the `multi_task_spec` block, specifying a list of `requirements` just for our `get_amzn` task.

With both `multi_task_spec` and `python_callable_partials` working together, you can pretty much iterate over anything!

7 Custom Operators

Custom operators are a great way to get even more out of gusty. Two great use cases for custom operators are:

- **Auto-detecting dependencies** - Have your tasks depend on one another without having to explicity set a `dependencies` block. This is very useful for SQL tasks, and can allow you to achieve a dbt-like dependency graph.
- **Running notebooks** - Just take a notebook and run it as a pipeline task. It's pretty much that simple!

7.0.1 How It Works

1. In gusty, the task name is the file name.
2. gusty (optionally) makes available to your custom operators a `task_id`, which is the task name. Just specify `task_id` as an argument in the operator's `__init__` method, and gusty will pass it in when building your task.
3. If you name your SQL tables after the `task_id`, you can detect table names in your SQL, which in turn can be leveraged as a list of dependencies.
4. If you attach this list of dependencies as an attribute on your custom operator, gusty automatically wires up these dependencies for you.

Let's make an example custom SQL operator that takes advantage of this.

7.0.2 Example Operator

To make our custom operator we will use:

- The `PostgresOperator` from Airflow's Postgres Provider, as the parent class for our custom operator.
- The Parser from the sql-metadata package, for detecting table names.

A common purpose of SQL tasks is to create tables, so we will have our users provide `SELECT` statements, and will wrap their statements in a `CREATE OR REPLACE TABLE` statement from within the operator.

This custom operator can be stored in our Airflow plugins folder, maybe under `plugins/custom_operators/__init__.py`. We'll store a function for detecting tables, `detect_tables`, in this file, as well, for this example.

```python
from sql_metadata import Parser
from airflow.providers.postgres.operators.postgres import PostgresOperator

# ---------------------- #
# Detect Tables Function #
# ---------------------- #

def detect_tables(sql):
  """Detects tables in a sql query."""

  # Remove any Jinja syntax to improve table detection
  jinjaless_sql = sql.replace("{{", "").replace("}}", "")

  # Can return "schema.table", but we just want the "table"
  tables_raw = Parser(jinjaless_sql).tables

  # Only take "table" if "schema.table" is in tables_raw
  tables = [t.split('.')[-1] for t in tables_raw]

  return tables

# --------------- #
# Custom Operator #
# --------------- #

class CustomPostgresOperator(PostgresOperator):

    def __init__(
            self,
            # gusty automatically passes in task_id when creating the task
            task_id,
            schema,
            sql,
            postgres_conn_id = "postgres_default",
```

```
            **kwargs):

        # gusty uses self.dependencies to create task dependencies
        self.dependencies = detect_dependencies(sql)

        # Always name your table after the task_id / file name
        table = task_id

        create_sql = f"CREATE OR REPLACE TABLE {schema}.{table} AS ({sql})"

        super(CustomPostgresOperator, self).__init__(
            task_id = task_id,
            sql = create_sql,
            postgres_conn_id = postgres_conn_id,
            **kwargs)
```

7.0.3 Example Usage

7.0.3.1 `users` Table

Now that we have our custom operator, we can invoke it in a Task Definition File. Let's use this customer operator to create a users table, in a Task Definition File named `users.sql`.

```
---
operator: plugins.custom_operators.CustomPostgresOperator
schema: app_data
---

SELECT
  id AS user_id,
  created_at
FROM raw_data.users_raw
```

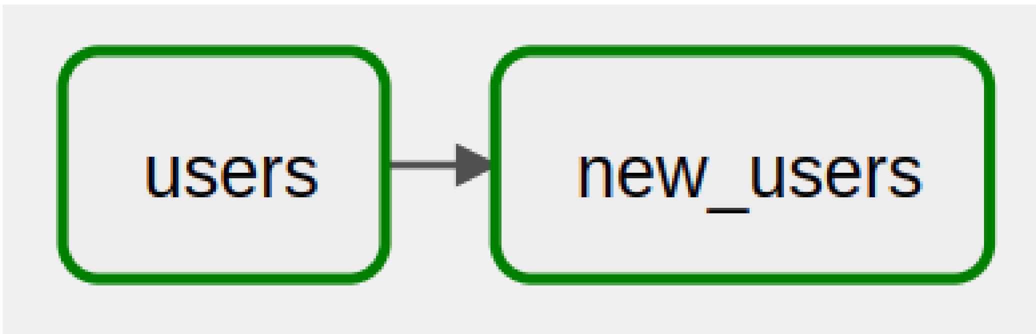
Per our custom operator, this will create a `users` table in our `app_data` schema.

7.0.3.2 `new_users` Table

Now we can make a second Task Definition File, `new_users.sql`, which references the `users` table.

```
---
operator: plugins.custom_operators.CustomPostgresOperator
schema: app_data
---

SELECT
  user_id
FROM app_data.users
WHERE DATE(created_at) = CURRENT_DATE()
```

In our Airflow DAG graph, the `new_users` task now automatically depends on the `users` task!



7.1 Running Notebooks

7.1.1 How It Works

1. Just like `task_id` is a special keyword you can add to your custom operator's `__init__` method, so is `file_path`.
2. `file_path` will be an absolute path to your Task Definition File, in this case a Jupyter Notebook.
3. Declare a YAML cell in your Jupyter Notebook, specifying the `operator` that should run the cell.
4. Have your custom operator run the cell.

7.1.2 Example Operator

We'll more or less take this example directly from the gusty demo.

To make our custom operator we will use:

- The built-in BashOperator, for running the command that renders the notebook.
- The nbconvert package to render the notebook.

Our operator will simply render the notebook as HTML and then delete it.

```
from airflow.operators.bash_operator import BashOperator

command_template = """
jupyter nbconvert --to html --execute {file_path} || exit 1; rm -f {rendered_output}
"""

class JupyterOperator(BashOperator):
    """
    The LocalJupyterOperator executes the Jupyter Notebook.
    Note that it is up to the notebook itself to handle connecting
    to a database. (But it can grab this from Airflow connections)
    """

    def __init__(
      self,
      # gusty automatically passes in file_path when creating the task
      file_path,
      *args,
      **kwargs):

        self.file_path = file_path
        self.rendered_output = self.file_path.replace('.ipynb', '.html')

        command = command_template.format(file_path = self.file_path,
                                          rendered_output = self.rendered_output)

        super(JupyterOperator, self).__init__(bash_command = command, *args, **kwargs)
```

7.1.3 Example Usage

See the Juypter Notebook Task Definition File example in the gusty demo, in the stock_predictions DAG. Notice how the first cell in the notebook is a YAML cell (see Raw notebook).

A create_dag Arguments

Both `create_dag` and `create_dags` can take any keyword arguments available to Airflow's DAG object. Additionally, there are some gusty-specific arguments for these functions.

Below we will cover all gusty-specific arguments available in `create_dag` and `create_dags`, followed by specific `create_dag` and `create_dags` considerations. The gusty-specific arguments can also be used in a DAG's `METADATA.yml`.

For the best results, it's recommended to always use keyword arguments with `create_dag` and `create_dags`.

A.0.1 `latest_only`

By default, gusty adds a `LatestOnlyOperator` at the absolute root of your Airflow DAG, which means that - by default - the tasks is your DAG will not run except for the latest DAG run. You can read more about the `LatestOnlyOperator` in Airflow's documentation, but setting `latest_only=False` will ensure a gusty-generated DAG mirrors Airflow's default behavior.

A.0.2 `extra_tags`

In addition to any tags set via an Airflow DAG's `tags` argument (available - as with any Airflow DAG parameter - in both `create_dag` and `METADATA.yml`), gusty will append any tags set in the `extra_tags` list to the provided tags.

To set `extra_tags` in your call to `create_dag`, provide a list like so:

```
extra_tags=["your", "extra", "tags"]
```

A.0.3 `root_tasks`

You can assign certain tasks to be at the beginning of the DAG by declaring `root_tasks`, a list of task ids. Any task id that is designated as a root task cannot have a `dependencies` block.

A.0.4 `leaf_tasks`

You can assign certain tasks to be at the end of the DAG by declaring `leaf_tasks`, a list of task ids. Any task id that is designated as a leaf task cannot have a `dependencies` block.

A.0.5 `external_dependencies`

A list of key value pairs in the format of `dag_id: task_id`, where the `dag_id` is some upstream DAG and the `task_id` is the task in that upstream DAG. When set, gusty will create ExternalTaskSensor tasks and place them at the root of the DAG. Set the `task_id` to `all` to wait for the entire upstream DAG to complete. See the section on external dependencies for more details.

A.0.6 `dag_constructors`

Provide either a list of functions or a dictionary of function names names and functions (much like what you would pass to an Airflow DAG's `user_defined_macros`) to have your functions available to you both as YAML constructors with gusty as well as in Airflow anywhere `user_defined_macros` are accepted.

gusty will consolidate your `user_defined_macros` and your `dag_constructors` so that all are available anywhere you'd expect. Really, you can just use the Airflow DAG object's `user_defined_macros` for everything.

A.0.6.1 list format

The list format for `dag_constructors` would look like this:

```
dag_constructors=[your_first_func, your_second_func]
```

The functions would be accessible based on their function name.

A.0.6.2 dictionary format

The dictionary format for dag_constructors would look like this:

```
dag_constructors={
  "your_first_func": your_first_func,
  "your_renamed_func": your_second_func
  }
```

The functions would be accessible by the key name, allowing you to - as illustrated above - renamed your functions if you so desire.

Again, you can just use Airflow's built-in `user_defined_macros` argument to achieve this same functionality, of having your macros available to you anywhere.

A.0.7 `wait_for_defaults`

A dictionary of values that can be passed to an Airflow ExternalTaskSensor (or BaseOperator).

A.0.8 `task_group_defaults`

A dictionary of values that can be passed to Airflow TaskGroup object.

A.0.9 `leaf_tasks_from_dict`

A dictionary of tasks that you want at the end of your DAG, where the key is the name of the task, and the value is a spec for that task.

```
leaf_tasks_from_dict={
  "my_dag_is_done": {
    "operator": "airflow.operators.bash.BashOperator",
    "bash_command": "echo done"
    }
}
```

A.0.10 `parse_hooks`

If you want to parse another file type, or want to override how gusty parses supported file types, you can pass a dictionary of file extensions and functions to parse those extensions. Your functions should take a `file_path` argument.

```
parse_hooks={
  ".sh": your_shell_file_parsing_function
}
```

See gusty's built-in parsers here.

A.0.11 `ignore_subfolders`

Will disable the creation of task groups from subfolders when set to `True`.

A.0.12 `render_on_create`

Disabled by default. If you want any Jinja in your spec to rendered on creation, set to `True`. Note that this will process everything every time the DAG is processed, which by default in Airflow is every few minutes. In general you don't want this on.

A.1 `create_dag` Specific Notes

The first argument to `create_dag` is a path to single DAG directory containing Task Definition Files.

A.2 `create_dags` Specific Notes

The first argument to `create_dags` is a path to a directory containing multiple DAG directories, each with their own Task Definition Files.

The second argument to `create_dag` should always be `globals()`, which will ensure the resulting DAG objects are discoverable by Airflow.

B Supported file types

Below is a list of supported file types and how they work out of the box.

You can always use parse hooks to add additional file types for your use cases, or override gusty's default parsers.

All Airflow task ids are the Task Definition Files' file names.

B.0.1 Behavior

Declare an `operator` and pass in any operator parameters using YAML.

B.0.2 Example

```yaml
operator: airflow.operators.bash.BashOperator
bash_command: echo hello world
```

B.1 `.py`

B.1.1 Behavior

For starters, you can just write Python code and by default gusty will execute your file using a `PythonOperator`.

To expand, you can declare a `python_callable` in the Frontmatter and define the function in the body.

While default behavior for `.py` files specifies `PythonOperator` as the `operator`, as with any Task Definition File, you can specify any `operator`.

B.1.2 Example

A Task Definition File, `hello_world.py`, with no Frontmatter:

```python
print("hello world")
```

A task_definition file, `hello_world.py`, with Frontmatter:

```python
# ---
# python_callable: main
# ---

def main():
  print("hello world")
```

The callable name is up to you, but it must match the function name in the Body.

B.2 `.sql`

B.2.1 Behavior

Declare an `operator` in a YAML header, then write SQL in the main .sql file. The SQL automatically gets sent to the operator.

B.2.2 Example

```sql
---
operator: airflow.providers.sqlite.operators.sqlite.SqliteOperator
---

SELECT 'hello world'
```

B.3 `.ipynb`

B.3.1 Behavior

Put a YAML block at the top of your notebook and specify an `operator` that renders your Jupyter Notebook.

B.3.2 Example

See the gusty demo Jupyter Notebook Example and sample JupyterOperator.

B.4 `.Rmd`

B.4.1 Behavior

Use the YAML block at the top of your notebook and specify an `operator` that renders your R Markdown Document.

B.4.2 Example

See the gusty demo Rmd Example and sample RmdOperator.