실시간 빅데이터 처리 spark/flink
- Part5 에어플로우를 통한 배치 프로세싱

이 수업에서 다루고자 하는 것

이론

- Task & DAG
- Backfill & Catchup
- Timeout & Callback
- Retry & Alret
- Pool & Paralleism
- Hook
- SubDAG & TaskGroups
- Branching
- Trigger Rule
- XCOM

이 수업에서 다루고자 하는 것

실습

- PythonOperator
- BashOperator
- PostgresOperator
- BranchOperator
- BranchDateTimeOperator
- SubDagOperator
- TriggerDagRunOperator
- ExternalTaskSensor
- ShortCircuit Operator
- LatestOnlyOperator

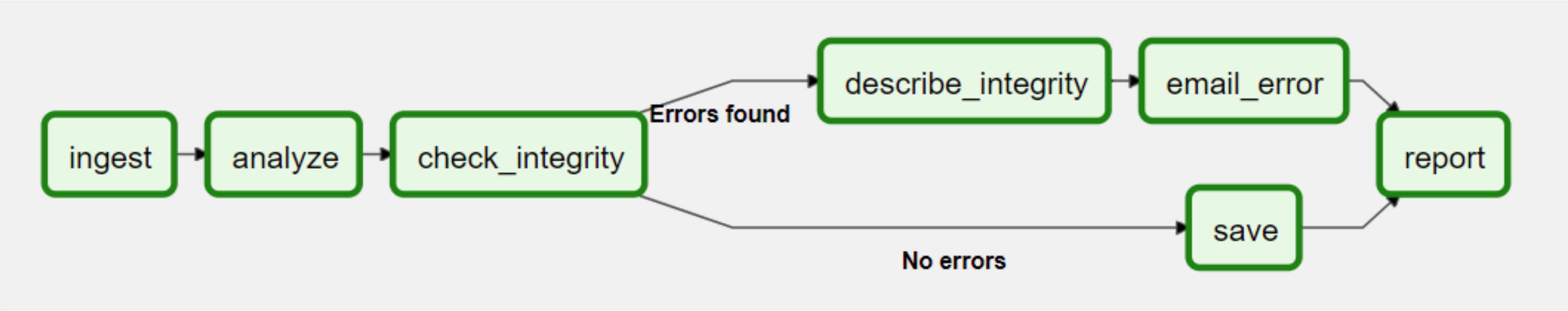
Chapter 1. Abstraction

Chapter 1.

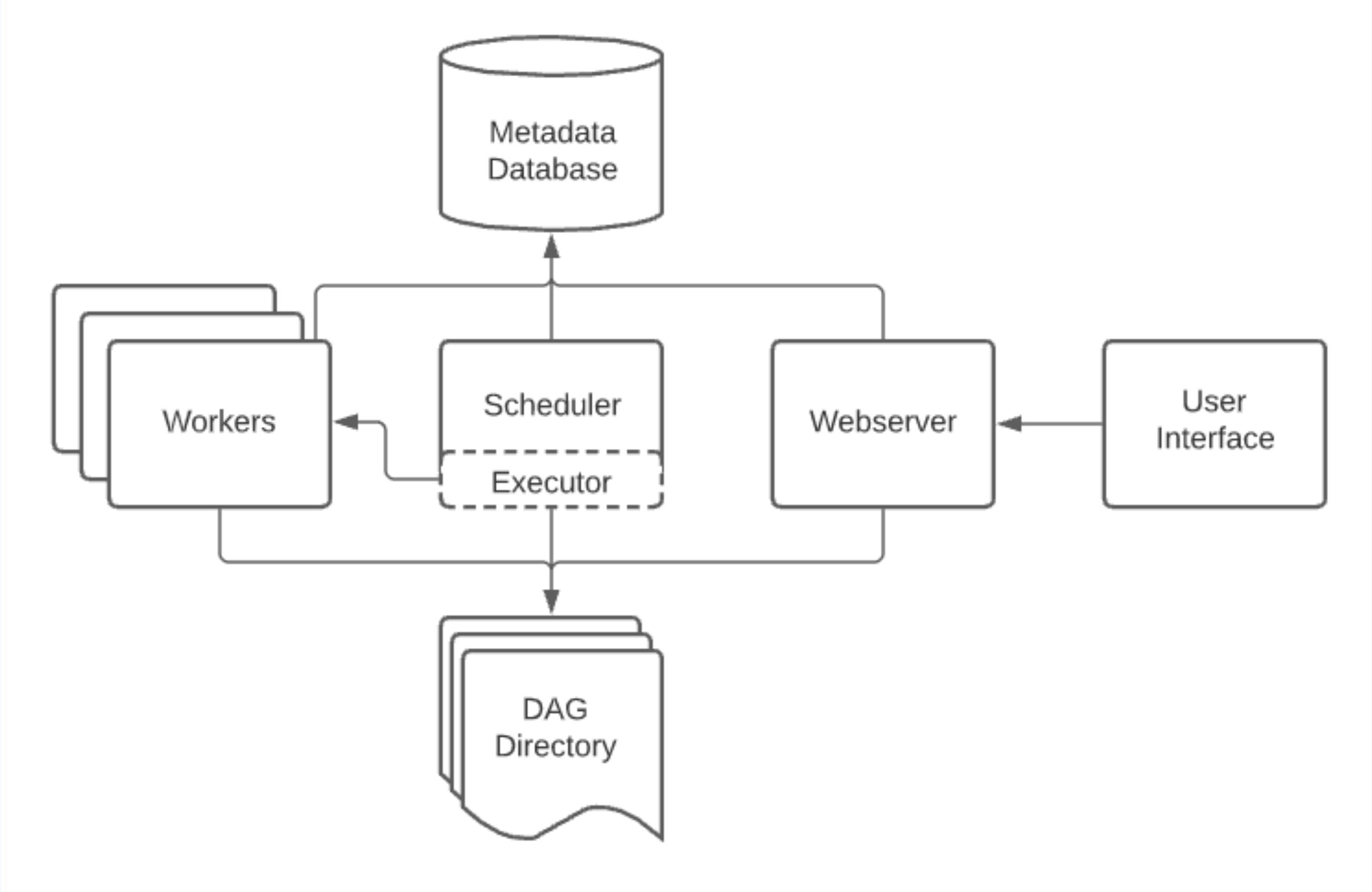
01. Architecture

01. Overview

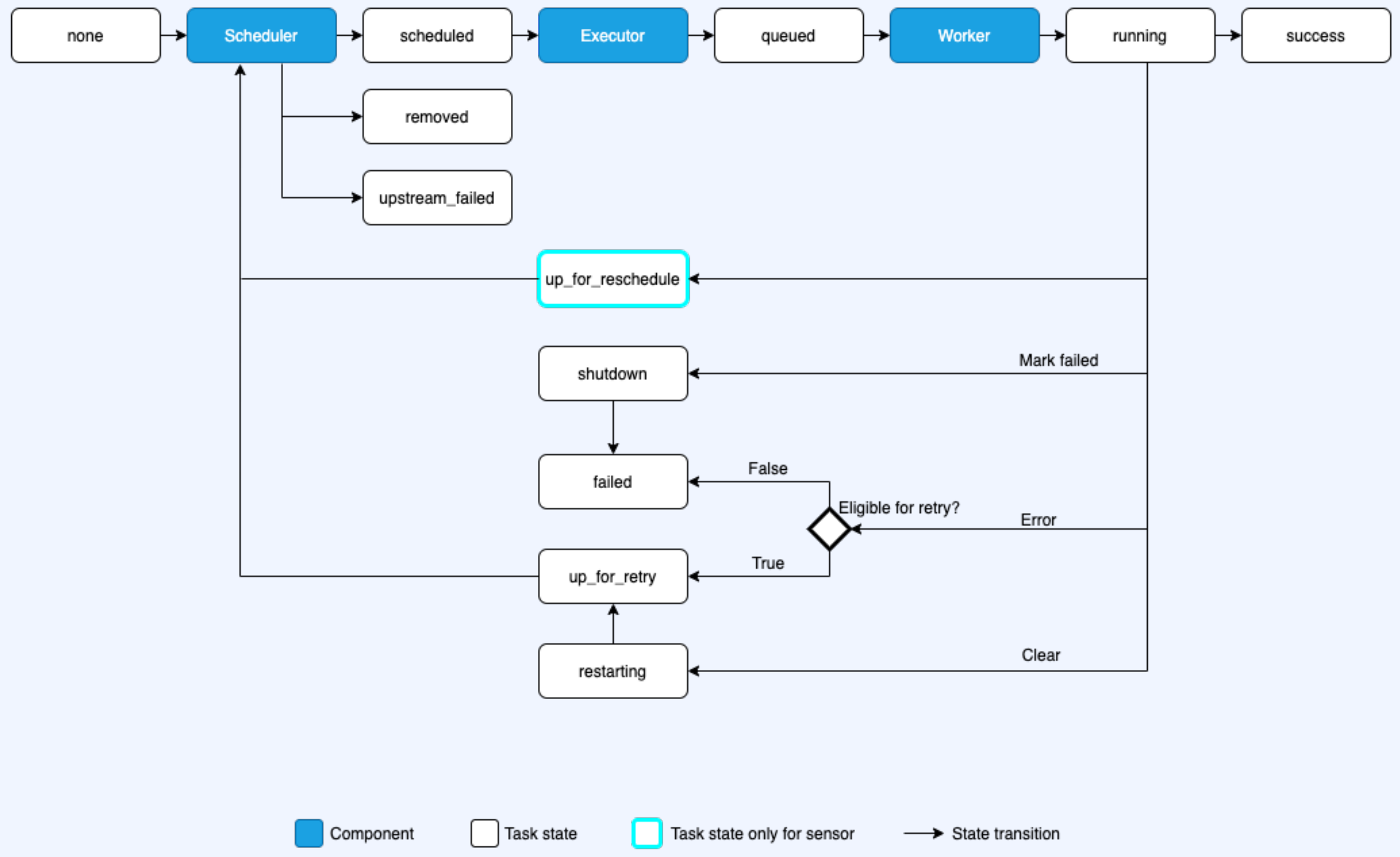
- Workflow == DAG(Directed Acyclic Graph)
  - Contains individual pieces of work == Tasks



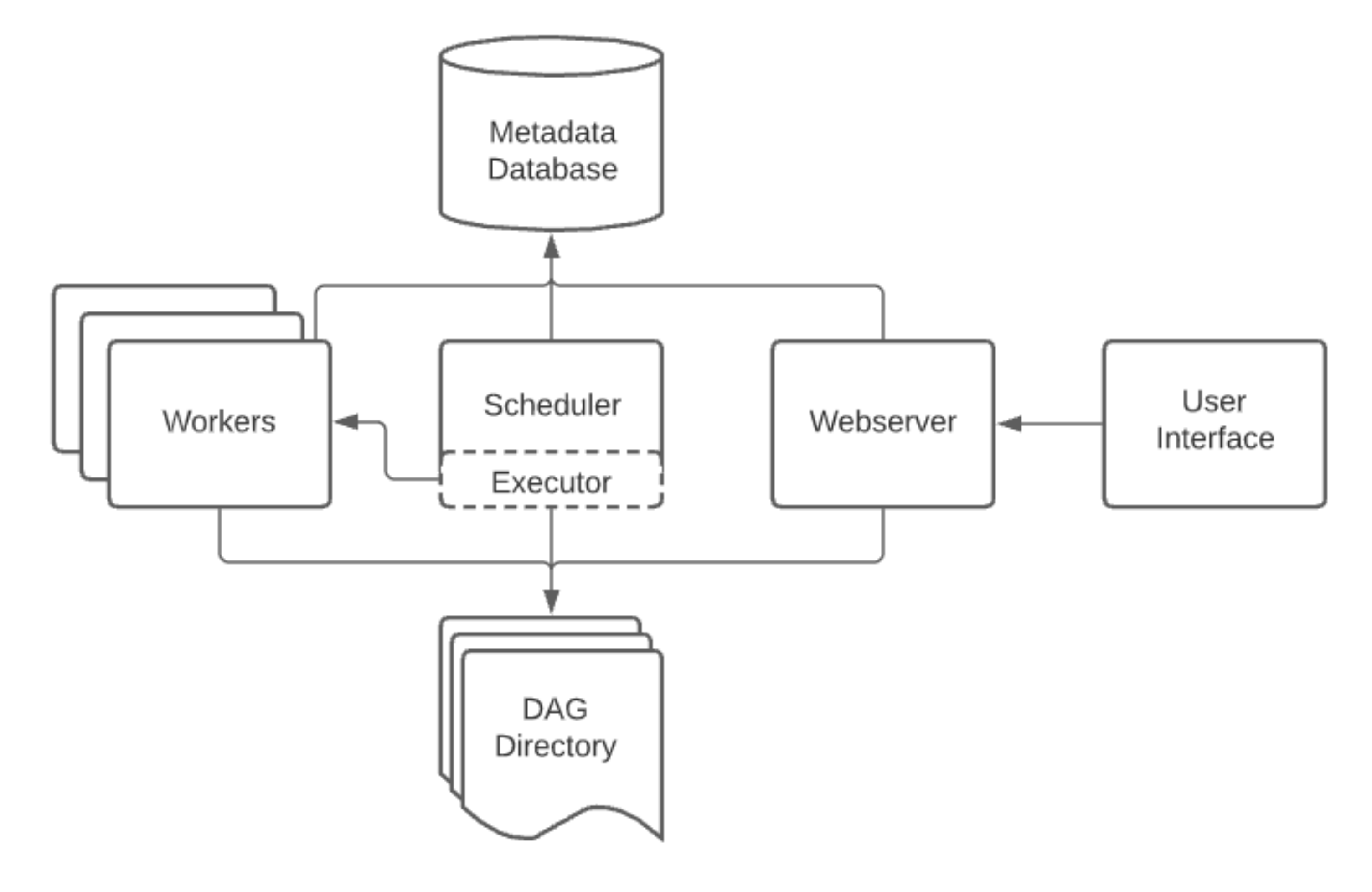
01. Components



01. Task Lifecycle



01. Components



01. Workloads

- Task의 모든 type은 내부적으로 BaseOperator의 subclass
- Three common types
  - Operators
  - Sensors
  - TaskFlow-decorated @task

01. TaskFlow 예제

```
@task
def get_ip():
    return my_ip_service.get_main_ip()


@task
def compose_email(external_ip):
    return {
        'subject':f'Server connected from {external_ip}',
        'body': f'Your server executing Airflow is connected from the external IP
{external_ip}<br>'
    }
```

01. TaskFlow 예제

```
email_info = compose_email(get_ip())

EmailOperator(
    task_id='send_email',
    to='example@example.com',
    subject=email_info['subject'],
    html_content=email_info['body']
)
```

01. Control Flow

- DAGs
  - run many times, and multiple runs of them can happen in parallel
  - always including an interval they are “running for” (the data interval)
- Tasks
  - have dependencies declared on each other
    - first_task >> second_task

01. Circular Dependency

```
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime

with DAG('circular_dependency_dag', start_date=datetime(2022, 1, 1)) as dag:
    task1 = DummyOperator(task_id='task1')
    task2 = DummyOperator(task_id='task2')

    task1 >> task2 >> task1
```

01. Circular Dependency

```
with DAG('DAG1', default_args=default_args, schedule_interval=None) as dag:
    trigger_dag2 = TriggerDagRun(
                  task_id='trigger_dag2’,
                  trigger_dag_id="DAG2",
                  conf={"message": "Hello from DAG1"})

with DAG('DAG2', default_args=default_args, schedule_interval=None) as dag:
    trigger_dag1 = TriggerDagRun(
                  task_id='trigger_dag1’,
                  trigger_dag_id="DAG1",
                  conf={"message": "Hello from DAG2"} )
```

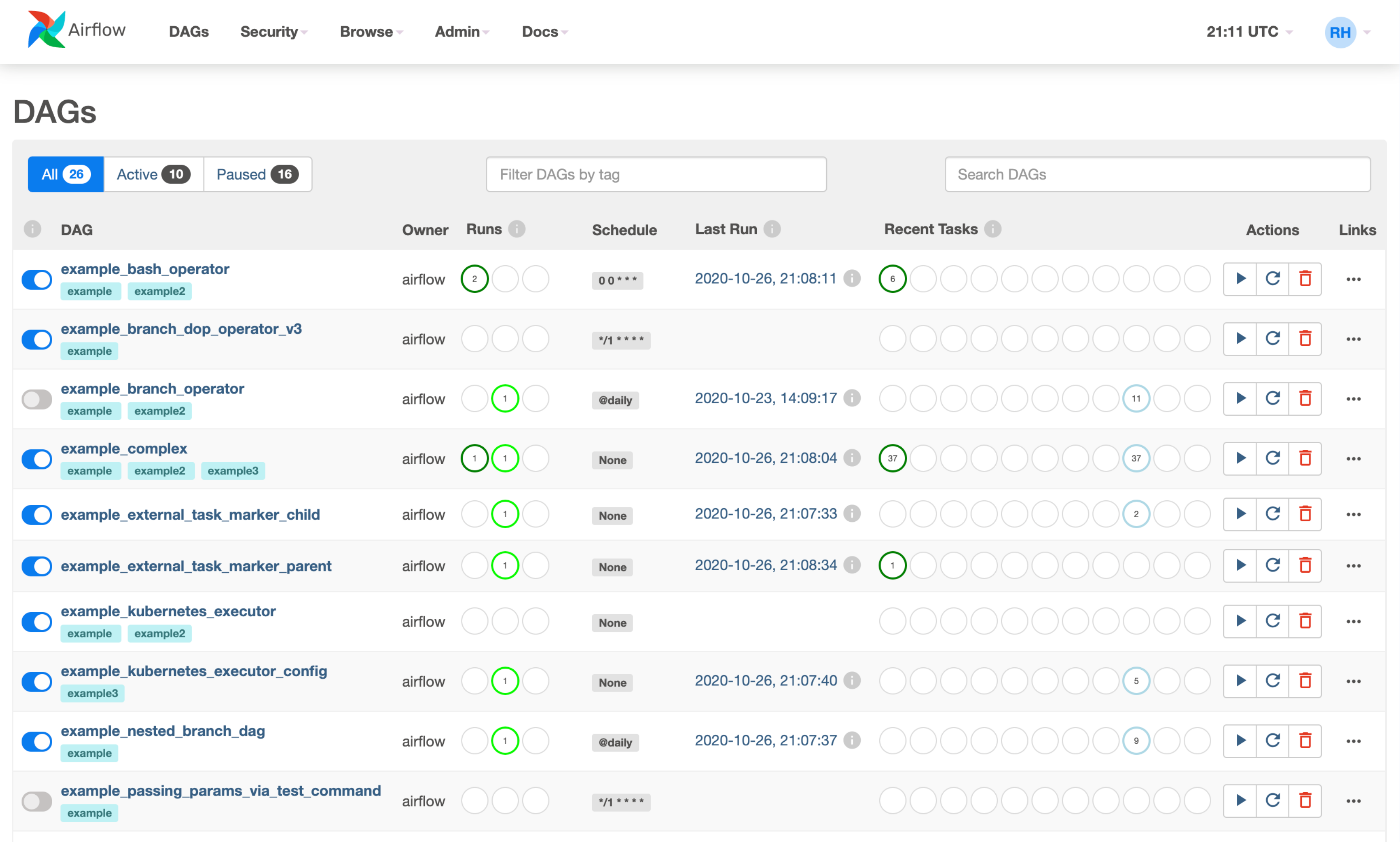
01. Pass data between tasks

- Xcoms
- Storage Service

01. Communication with external services

- Connections & Hooks
- Pools

01. User Interface



DAGs

All 26 | Active 10 | Paused 16

Filter DAGs by tag

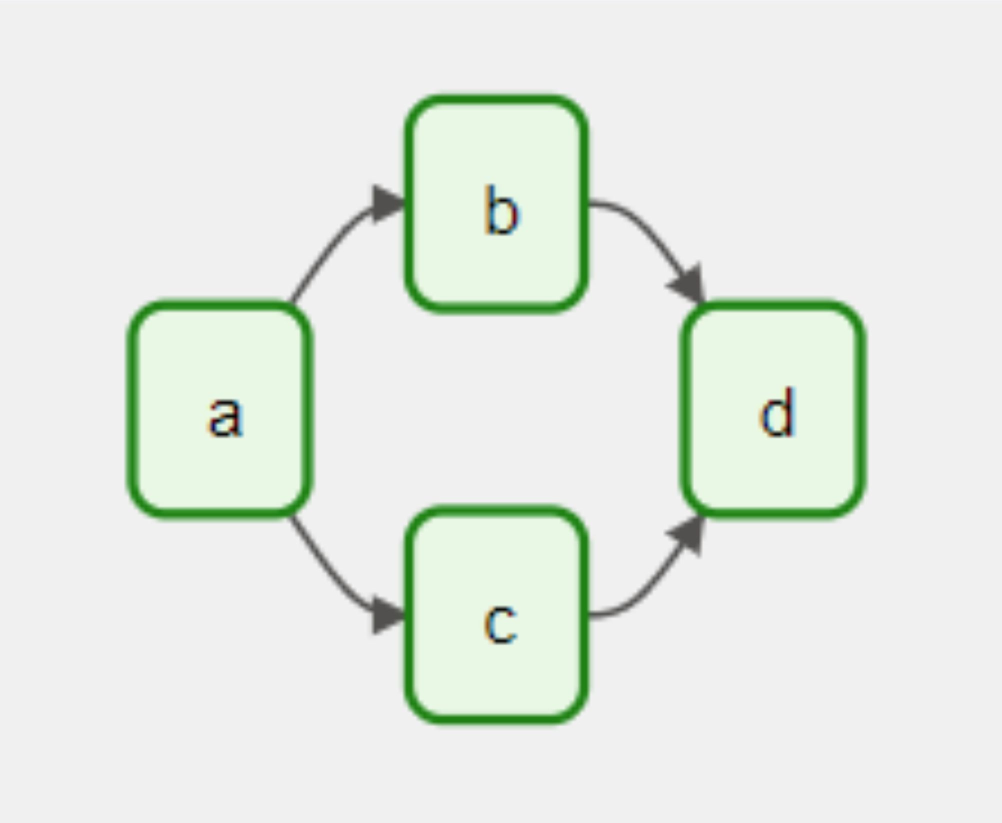
Search DAGs

	DAG	Owner	Runs	Schedule	Last Run	Recent Tasks	Actions	Links
●	example_bash_operator example example2	airflow	2	0 0 * * *	2020-10-26, 21:08:11	6	▶ ↻ 🗑	•••
●	example_branch_dop_operator_v3 example	airflow		*/1 * * * *			▶ ↻ 🗑	•••
○	example_branch_operator example example2	airflow	1	@daily	2020-10-23, 14:09:17	11	▶ ↻ 🗑	•••
●	example_complex example example2 example3	airflow	1 1	None	2020-10-26, 21:08:04	37 37	▶ ↻ 🗑	•••
●	example_external_task_marker_child	airflow	1	None	2020-10-26, 21:07:33	2	▶ ↻ 🗑	•••
●	example_external_task_marker_parent	airflow	1	None	2020-10-26, 21:08:34	1	▶ ↻ 🗑	•••
●	example_kubernetes_executor example example2	airflow		None			▶ ↻ 🗑	•••
●	example_kubernetes_executor_config example3	airflow	1	None	2020-10-26, 21:07:40	5	▶ ↻ 🗑	•••
●	example_nested_branch_dag example	airflow	1	@daily	2020-10-26, 21:07:37	9	▶ ↻ 🗑	•••
○	example_passing_params_via_test_command example	airflow		*/1 * * * *			▶ ↻ 🗑	•••

Chapter 2.

01. DAG

01. DAG



01. Declaring a DAG – Context Manager

```
with DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
):
    EmptyOperator(task_id="task")
```

01. Declaring a DAG – Context Manager

```
my_dag = DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
)
EmptyOperator(task_id="task", dag=my_dag)
```

01. Declaring a DAG – Context Manager

```
@dag(start_date=datetime.datetime(2021, 1, 1), schedule="@daily")
def generate_dag():
    EmptyOperator(task_id="task")


generate_dag()
```

01. Loading DAGs

```
dag_1 = DAG('this_dag_will_be_discovered')

def my_function():
    dag_2 = DAG('but_this_dag_will_not')

my_function()
```

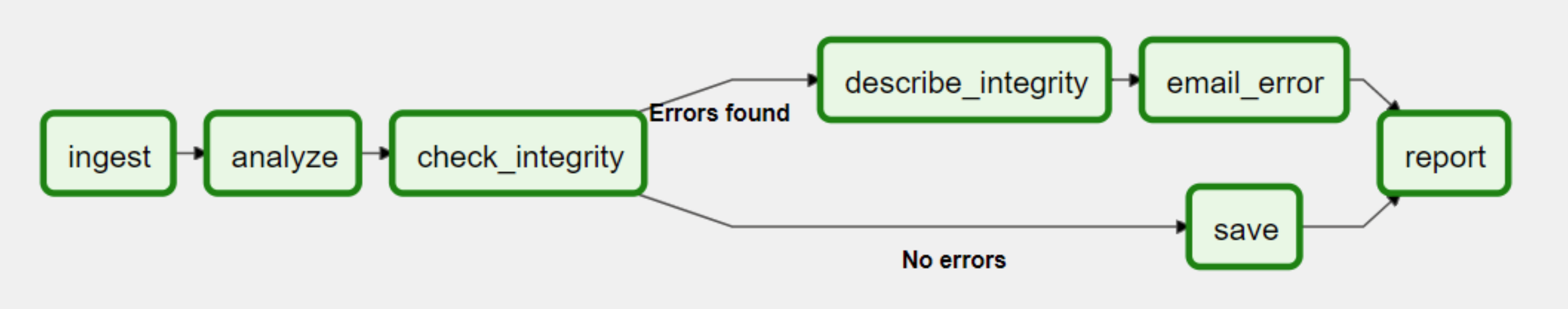
01. Running DAGs

```
with DAG("my_daily_dag", schedule="@daily"):
    ...
```

01. DAG Assignment

```
with DAG("my_dag"):
    some_operator = SomeOperator(task_id="some_task")
```

01. Edge Labels



01. Edge Labels

```
ingest = EmptyOperator(task_id="ingest")
analyse = EmptyOperator(task_id="analyze")
check = EmptyOperator(task_id="check_integrity")
describe = EmptyOperator(task_id="describe_integrity")
error = EmptyOperator(task_id="email_error")
save = EmptyOperator(task_id="save")
report = EmptyOperator(task_id="report")

ingest >> analyse >> check
check >> Label("No errors") >> save >> report
check >> Label("Errors found") >> describe >> error >> report
```

Chapter 2.

02. Tasks

02. Tasks

- Operators
- Sensors
- Taskflow – decorated @task

02. Relations

>> or <<

- first_task >> second_task >> [third_task, fourth_task]

upstraeam() or downstream()

- first_task.set_downstream(second_task)
- third_task.set_upstream(second_task)

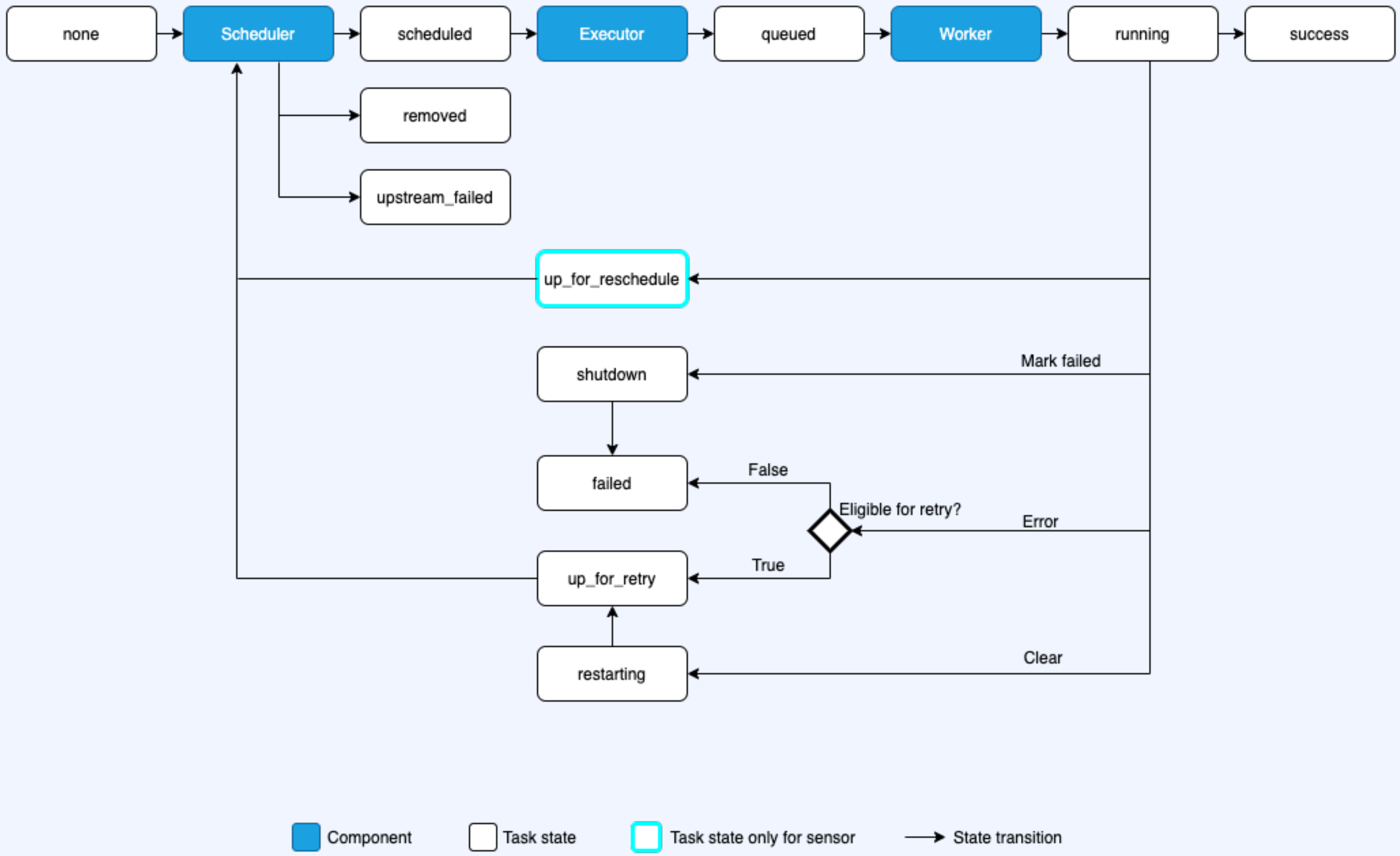
02. Task Instances

- none
- scheduled
- queued
- running
- success
- shutdown
- restarting

02. Task Instances

- failed
- skipped
- upstream_failed
- up_for_retry
- up_for_reschedule
- deferred
- removed

02. Task Instances



02. Special Exceptions

```
def skip_task():
    raise AirflowSkipException("This task is skipped!")
def fail_task():
    raise AirflowFailException("This task has failed!")

skip_task_operator = PythonOperator( task_id='skip_task',
python_callable=skip_task, dag=dag, )
fail_task_operator = PythonOperator( task_id='fail_task',
python_callable=fail_task, dag=dag, )
```

02. Zombie/Undead Tasks

- Zombie tasks
- Undead tasks

02. Executor Configuration

```
MyOperator(...,
    executor_config={
        "KubernetesExecutor":
            {"image": "myCustomDockerImage"}
    }
)
```

02. DAG & Task Documentation

```
""" ### My great DAG """

dag = DAG( "my_dag", start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
schedule="@daily", catchup=False, )
dag.doc_md = __doc__

t = BashOperator("foo", dag=dag)
t.doc_md = """
  \ #Title" Here's a [url](www.airbnb.com)
"""
```

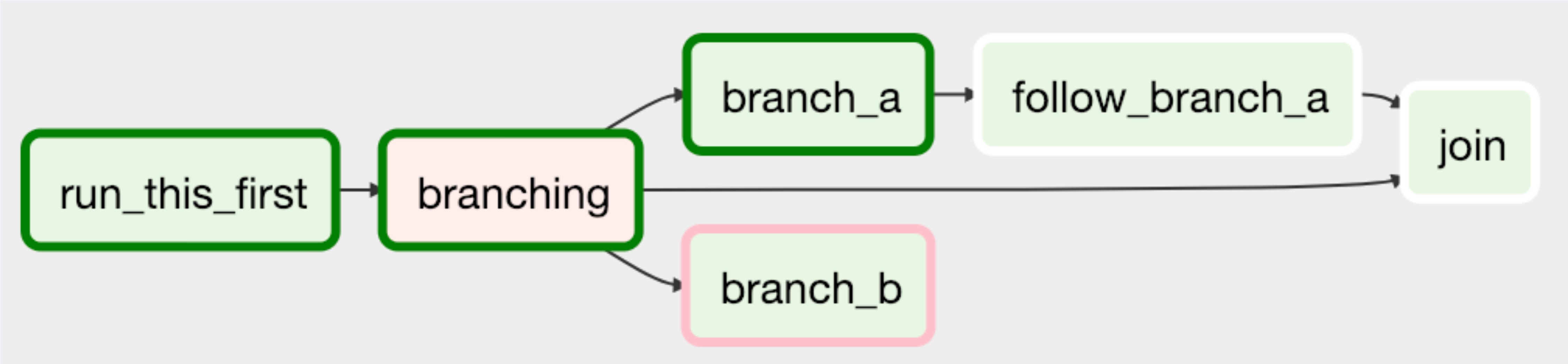
02. DAG & Task Documentation

- doc
- doc_json
- doc_yaml
- doc_md
- doc_rst

Chapter 2.

03. Control Flow

03. Branching



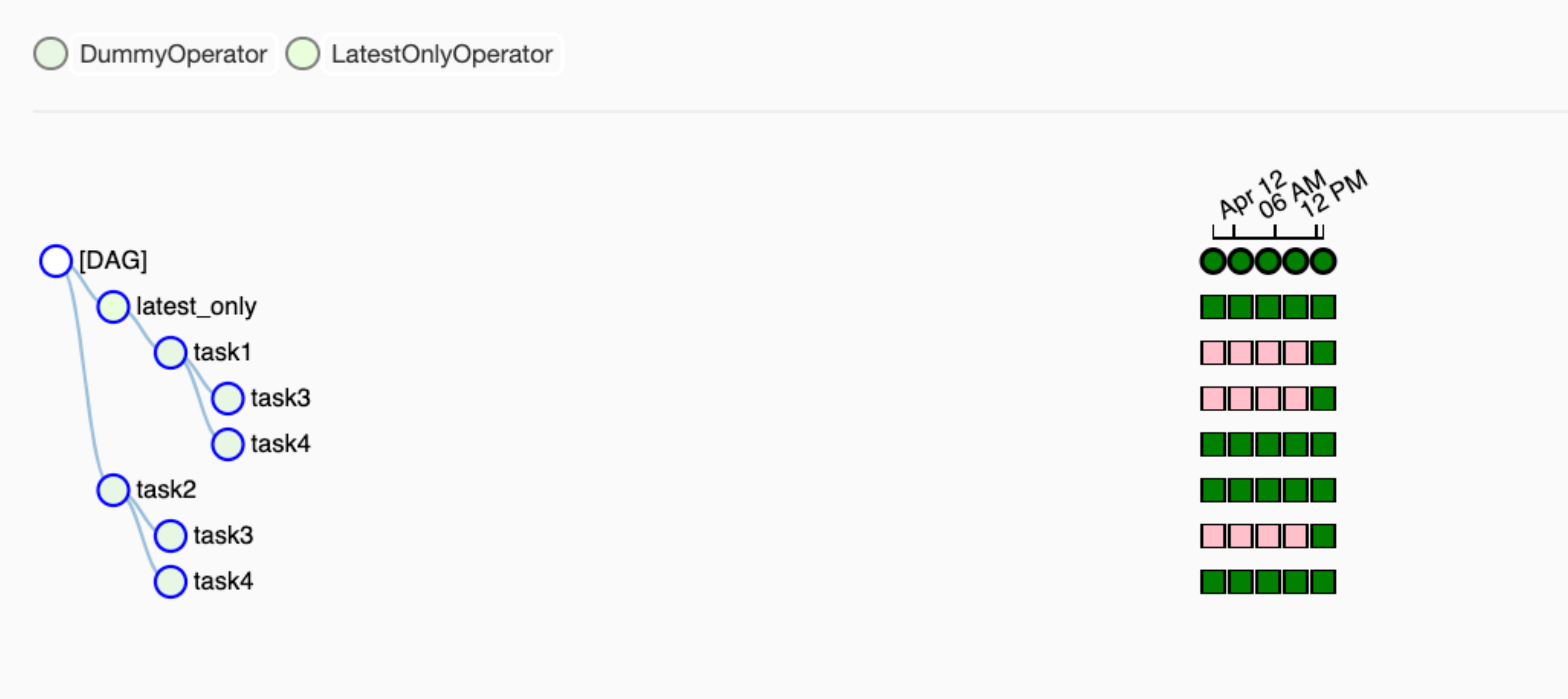
03. Branching

```
@task.branch(task_id="branch_task")
def branch_func(ti=None):
    xcom_value = int(ti.xcom_pull(task_ids="start_task"))
    if xcom_value >= 5:
            return "continue_task"
    elif xcom_value >= 3:
            return "stop_task"
    else: return None

branch_op = branch_func()
start_op >> branch_op >> [continue_op, stop_op]
```

03. Branching

```
class MyBranchOperator(BaseBranchOperator):
    def choose_branch(self, context):
        if context['data_interval_start'].day == 1:
            return ['daily_task_id', 'monthly_task_id']
        elif context['data_interval_start'].day == 2:
            return 'daily_task_id'
        else:
            return None
```

03. Latest Only



03. Branching

```
latest_only = LatestOnlyOperator(task_id="latest_only")
task1 = EmptyOperator(task_id="task1")
task2 = EmptyOperator(task_id="task2")
task3 = EmptyOperator(task_id="task3")
task4 = EmptyOperator(task_id="task4", trigger_rule=TriggerRule.ALL_DONE)

latest_only >> task1 >> [task3, task4]
task2 >> [task3, task4]
```

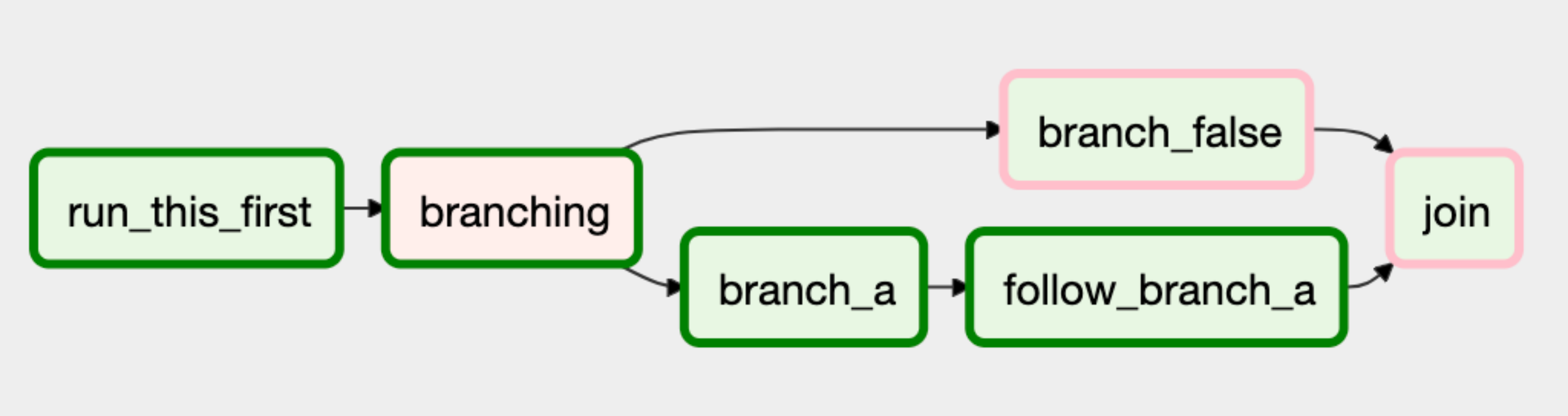
03. Depends On Past

- ‘depends_on_past’: True

03. Trigger Rules

- all_success (기본값)
- all_failed
- all_done
- all_skipped
- one_failed
- one_success
- one_done
- none_failed
- none_failed_min_one_success
- none_skipped
- always

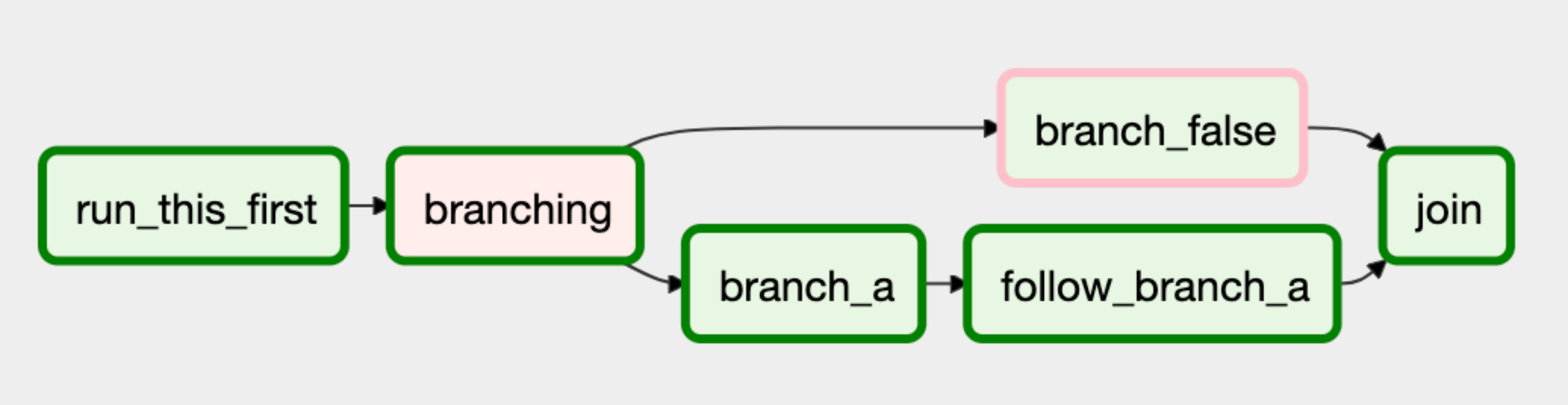
03. Trigger Rules



03. Trigger Rules

```
join = EmptyOperator(task_id="join",
    trigger_rule=TriggerRule.NONE_FAILED_MIN_ONE_SUCCESS,
    dag=dag)

run_this_first >> branching
branching >> branch_a >> follow_branch_a >> join
branching >> branch_false >> join
```

03. Trigger Rules



Chapter 2.

04. Timetables

04. DAG Runs

- An instantiation of the DAG **in time**
- The status of the DAG Run depends on the tasks states

04. DAG Run Status

- success if all of the leaf nodes states are either success or skipped
- failed if any of the leaf nodes state is either failed or upstream_failed

04. Cron Presets

- None
- @once
- @continuously
- @hourly: 0 * * * *
- @daily: 0 0 * * *
- @weekly: 0 0 * * 0
- @monthly: 0 0 1 * *
- @quarterly: 0 0 1 */3 *
- @yearly: 0 0 1 1 *

04. Data Interval

- execution_date(-> logical date)
- data_interval_start
- data_intercal_end

04. Data Interval

- 매일 실행되는 DAG의 경우:
  - logical_date (execution_date): 2021-01-02 00:00:00
  - data_interval_start: 2021-01-02 00:00:00
  - data_interval_end: 2021-01-03 00:00:00

- 매일 실행되는 DAG가 2일 전의 데이터를 처리
  - logical_date (execution_date): 2021-01-03 00:00:00
  - data_interval_start: 2021-01-01 00:00:00
  - data_interval_end: 2021-01-02 00:00:00

04. Timetables

- DAGs are driven by its internal “timetable”
- The timetable determines the data interval and the logical date

04. Custom Timetables

- Data intervals with “holes” between.
- Run tasks at different times each day.
- Schedules not following the Gregorian calendar.
- Rolling windows, or overlapping data intervals.

04. Implement Custom Timetables

```
class PassWeekendTimetable(Timetable):
  def next_dagrun_info( self, *, last_automated_dagrun: Optional[datetime],
**kwargs ) -> Optional[DagRunInfo]:
          if not last_automated_dagrun:
                  return self._first_dagrun()
          next_start = last_automated_dagrun + timedelta(days=1)
          if next_start.weekday() >= 5:
                  next_start += timedelta(days=2)
                  return DataInterval(
                          start=next_start, end=next_start + timedelta(days=1), )
```

04. Built-in Timetables – Events Timetable

```
@dag(
 schedule=EventsTimetable(
  event_dates=[
   pendulum.datetime(2022, 4, 5, 8, 27, tz="America/Chicago"),
   pendulum.datetime(2022, 4, 17, 8, 27, tz="America/Chicago"),
   pendulum.datetime(2022, 4, 22, 20, 50, tz="America/Chicago"),
  ],
  description="My Team's Baseball Games",
  restrict_to_events=False, ), ..., ) def example_dag(): pass
```

04. Differences between the two cron timetables

- CronTriggerTimetable: does not care the idea of data interval
- CronDataIntervalTimetable: does care the idea of data interval

- If catchup is False
  - CronTriggerTimetable: a new DAG run after the current time
  - CronDataIntervalTimetable: a new DAG run before the current time

Chapter 2.

05. Catchup & Backfill

05. Re-run DAG

- Re-run DAG != Create a new DAG run

05. Catchup

- start_date, end_date, none-dataset schedule

  -> defines a series of intervals: the scheduler turns into individual DAG runs
- scheduler kick off a DAG Run for any data interval

  that has not been run since the last data interval

- In code, catch_up=False
- In config, catch_up_default=False

05. Catchup

```
dag = DAG(
    "tutorial",
    default_args={ "depends_on_past": True,
                   "retries": 1,
                   "retry_delay": datetime.timedelta(minutes=3), },
    start_date=pendulum.datetime(2015, 12, 1, tz="UTC"),
    description="A simple tutorial DAG",
    schedule="@daily",
    catchup=False, )
```

05. Backfill

```
airflow dags backfill \
    --start-date START_DATE \
    --end-date END_DATE \
    dag_id
```

05. To re-run Tasks

- Re-run the tasks by clearing them for the scheduled date
- Clearing a task instance doesn’t delete the task instance record.
  - Instead, it updates **max_tries to 0**
  - sets the current **task instance state to None**, which causes the task to re-run.
- In the Tree or Graph views, click **Clear**

05. Re-run Options

- Past
- Future
- Upstream
- Downstream
- Recursive
- Failed

05. Re-run CLI

```
airflow tasks clear dag_id \
    --task-regex task_regex \
    --start-date START_DATE \
    --end-date END_DATE
```

Chapter 2.

06. Timeout

06. execution_time

- execution_timeout attribute to a **datetime.timedelta** value
- execution_timeout is breached,

  the task times out and **AirflowTaskTimeout** is raised.

06. timeout (Sensor / Reschedule mode)

- timeout controls the maximum time allowed for the sensor to succeed
- If timeout is breached, **AirflowSensorTimeout** will be raised

06. SFTPSensor 예제

```
sensor = SFTPSensor(
    task_id="sensor",
    path="/root/test",
    execution_timeout=timedelta(seconds=60),
    timeout=3600,
    retries=2,
    mode="reschedule",
)
```

06. SLAs

- Send the alarm when running the task

06. Set an SLA

```
def sla_miss_callback(context):
    pass

task_with_sla = DummyOperator(
    task_id='task_with_sla',
    sla=timedelta(hours=2),.
    sla_miss_callback=sla_miss_callback,
    dag=dag
)
```

06. Disable SLA

```
[core]
check_slas = False
```

Chapter 2.

07. Callbacks & Notifier

07. Callbacks

- 중요성
  - 상태 모니터링
  - 자동화

- 주의사항
  - Worker 실행에 의한 상태 변화만 적용
  - Callback의 오류 Logging
    - $AIRFLOW_HOME/logs/scheduler/latest/PROJECT/DAG_FILE.py.log

07. Callback Types

- on_success_callback
- on_failure_callback
- sla_miss_callback
- on_retry_callback
- on_execute_callback

07. Callback 예제

```
with DAG( dag_id="example_callback", schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    dagrun_timeout=datetime.timedelta(minutes=60), catchup=False,
    on_success_callback=None,
    on_failure_callback=task_failure_alert,
    tags=["example"], ):
task1 = EmptyOperator(task_id="task1")
task2 = EmptyOperator(task_id="task2")
task3 = EmptyOperator(task_id="task3",
    on_success_callback=[dag_success_alert])
task1 >> task2 >> task3
```

07. BaseNotifier

- An abstract class
- Sending notifications using the various **on_*_callback**

07. Extend the BaseNotifier

```
class MyNotifier(BaseNotifier):
    template_fields = ("message",)

    def __init__(self, message):
        self.message = message

    def notify(self, context):
        # Send notification here, below is an example
        title = f"Task {context['task_instance'].task_id} failed"
        send_message(title, self.message)
```

07. Using a notifier

```
with DAG(dag_id="example_notifier",
    start_date=datetime(2022, 1, 1),
    schedule_interval=None,
    on_success_callback=MyNotifier(message="Success!"),
    on_failure_callback=MyNotifier(message="Failure!"), ):

    task = BashOperator(
        task_id="example_task",
        bash_command="exit 1",
        on_success_callback=MyNotifier(message="Task Succeeded!"),
    )
```

07. Notifications

- Amazon: ChimeNotifier
- Apprise: AppriseNotifier
- Discord: DiscordNotifier
- Pagerduty: PagerdutyNotifier
- Slack: SlackNotifier
- Simple Mail Transfer Protocol (SMTP): SmtpNotifier

Chapter 2.

08. Pools

08. Pools

- **limit the execution parallelism** on arbitrary sets of tasks
- managed in the UI (Menu -> Admin -> Pools)
  - a name and assigning it a number of worker slots

08. pool parameter

```
aggregate_db_message_job = BashOperator(
    task_id="aggregate_db_message_job",
    execution_timeout=timedelta(hours=3),
    pool="ep_data_pipeline_db_msg_agg",
    bash_command=aggregate_db_message_job_cmd,
    dag=dag,
)
aggregate_db_message_job.set_upstream(wait_for_empty_queue)
```

08. Pools

- Number of slots occupied by a task == pool_slots
- As slots free up, queued tasks start running based on the **Priority Weights**
- If tasks are not given a pool -> default_pool(128 slots)

08. Priority Weights

- **priority_weight**(default: 1)
    - defines priorities in the executor queue
    - calculated based on its **weight_rule**

08. Weighting Methods

- airflow.utils.WeightRule
    - downstream (default)
    - upstream
    - absolute

08. Weighting Methods - Example

```
task_1 = EmptyOperator( task_id='task_1’,
    weight_rule=WeightRule.DOWNSTREAM, )
task_2 = EmptyOperator( task_id='task_2’,
    weight_rule=WeightRule.UPSTREAM, )
task_3 = EmptyOperator( task_id='task_3’,
    weight_rule=WeightRule.ABSOLUTE,
    priority_weight=10, )

task_1 >> task_2 >> task_3
```

08. Weighting Methods

- a task -> a single pool slot (default)
- pool_slots: can be configured to occupy more

08. Weighting Methods - Example

```
BashOperator( task_id="heavy_task",
    bash_command="bash backup_data.sh",
    pool_slots=2, pool="maintenance", )
BashOperator( task_id="light_task1",
    bash_command="bash check_files.sh",
    pool_slots=1, pool="maintenance", )
BashOperator( task_id="light_task2",
    bash_command="bash remove_files.sh",
    pool_slots=1, pool="maintenance", )
```

Chapter 2.

09. Connections & Hook

09. Connections

- A set of parameters(username, password and hostname)
  with the type of system
  and a unique name(conn_id)
- Managed via the UI or the CLI
- Customizable connection storage and backend options

09. Managing Connections

- In environment variables
  - AIRFLOW_CONN_{CONN_ID}
- In an external Secrets Backend
  - AWS Secrets Manager, HashiCorp Vault, Google Cloud Secret Manager
- In the Airflow metadata database (using the CLI or web UI)

09. Using Connection

```
conn = BaseHook.get_connection('<conn_id>')
print(conn.host)
```

09. Using Connection

```
pg_hook = PostgresHook('<conn_id>’)

records = pg_hook.get_records('SELECT * FROM your_table’)
print(records)
```

09. Using Connection

```
print_host = BashOperator(
    task_id='print_host', bash_command='echo {{ conn.<conn_id>.host }}’
)
```

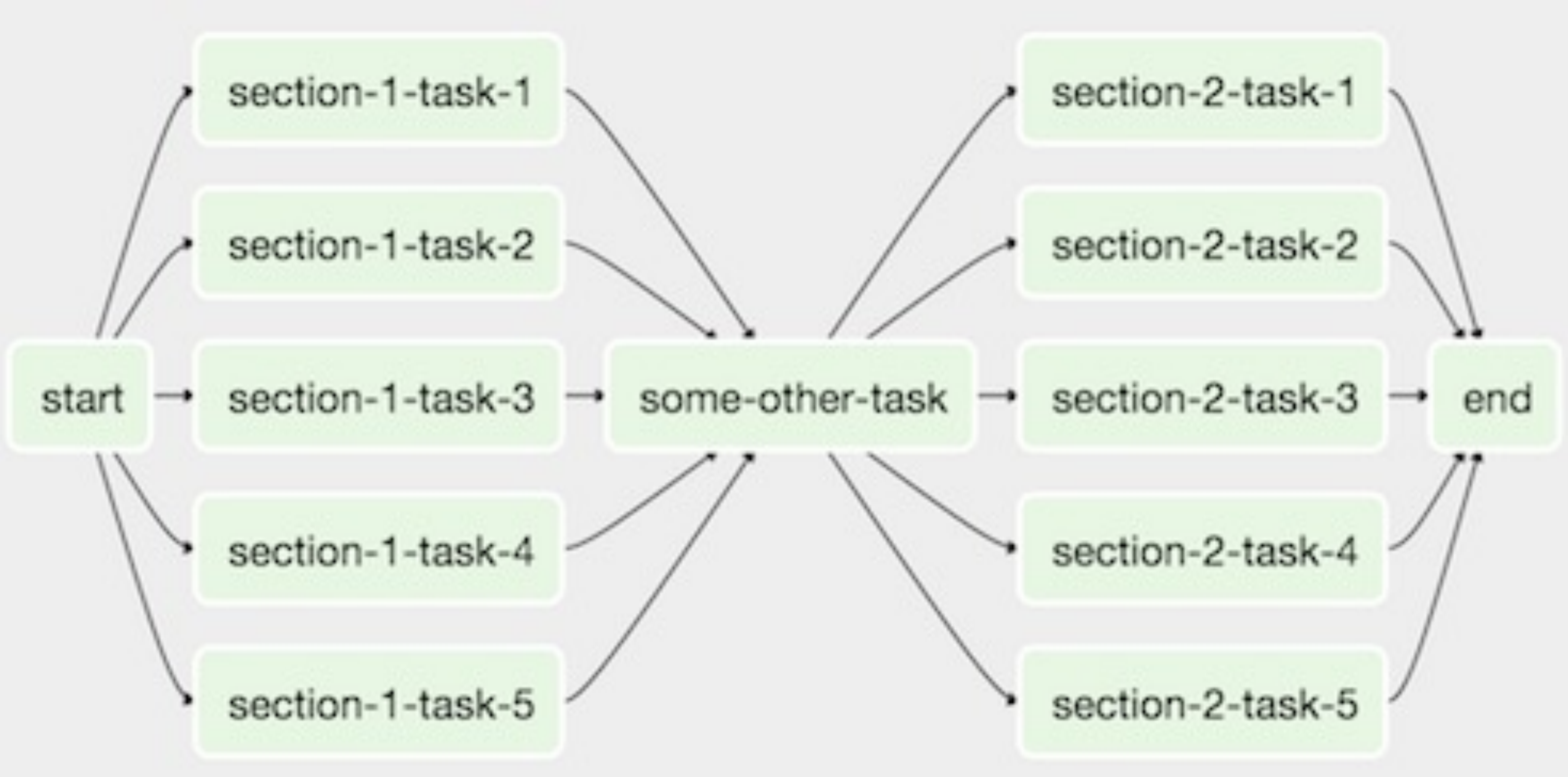
09. Hook

- A **high-level interface** to an external platform
- **Building blocks** that Operators are built out of
- Have a **default conn_id**
  - In PostgresHook, default conn_id is postgres_default

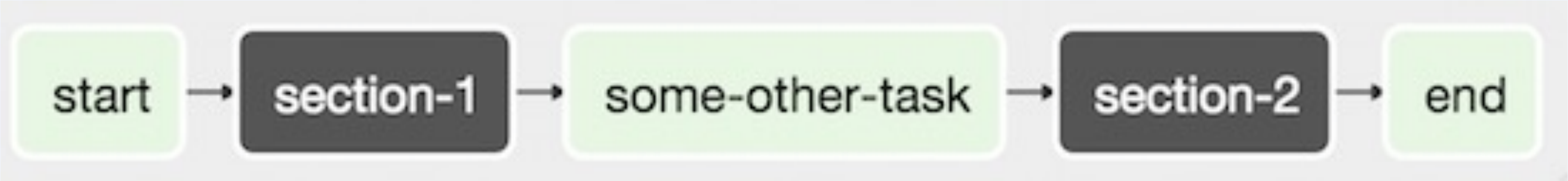
Chapter 2.

10. SubDAGs & TaskGroups

10. SubDAGs


start
section-1-task-1
section-1-task-2
section-1-task-3
section-1-task-4
section-1-task-5
some-other-task
section-2-task-1
section-2-task-2
section-2-task-3
section-2-task-4
section-2-task-5
end

10. SubDAGs



10. subdag() factory method

```
def subdag(parent_dag_name, child_dag_name, args) -> DAG:
    dag_subdag = DAG(
                dag_id=f"{parent_dag_name}.{child_dag_name} ”,
                ··· )
    for i in range(5):
                EmptyOperator(···, dag=dag_subdag, )
    return dag_subdag

subdag_task = SubDagOperator( task_id='subdag_task’,
    subdag=subdag('main_dag', 'subdag_task', args),
    dag=main_dag, )
```

10. TaskGroups

- Organize tasks into hierarchical groups in Graph
- Purely a UI grouping concept
  - https://airflow.apache.org/docs/apache-airflow/stable/_images/task_group.gif

10. TaskGroups Example

```
from airflow.decorators import task_group


@task_group()
def group1():
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")

task3 = EmptyOperator(task_id="task3")

group1() >> task3
```

10. TaskGroups Example – default_args

```
with DAG( dag_id="dag1", start_date=datetime.datetime(2016, 1, 1),
   schedule="@daily", default_args={"retries": 1}, ):
   @task_group(default_args={"retries": 3})
   def group1():
              """This docstring will become the tooltip for the TaskGroup."""
              task1 = EmptyOperator(task_id="task1")
              task2 = BashOperator(task_id="task2", bash_command="echo Hello
World!", retries=2)
              print(task1.retries) # 3
              print(task2.retries) # 2
```

10. TaskGroups vs SubDAGs

- SubDAG is deprecated hence TaskGroup is **always the preferred choice**.

Chapter 2.

11. XComs & Variables

11. XComs

- Short for ‘cross-communication’
- Let Tasks talk to each other

11. XComs

- Identifier
  - key(name), task_id, dag_id
- Value
  - serializable
  - only designed for small amounts of data
- Push & Pull
  - xcom_push, xcom_pull
  - If do_xcom_push == True (default)
    - retrun_value: auto-push result into XCom key
      - @task

11. XComs

- Templates
  - SELECT * FROM {{ task_instance.xcom_pull(task_ids='foo', key='table_name') }}
- vs Variables
  - XComs: per-task-instance, designed for communication
  - Variables: global, designed for overall configuration and value sharing

11. Custom Xcoms Backends

- Airflow metadate db (default)

11. Custom Xcoms Backends

- Interchangeable
    - xcom_backend
- Implement
    - BaseXCom: serialize_value, deserialize_value
- Rendered for UI
    - orm_deserialized_value
- Lifecycle
    - clear

11. Custom Xcom Backends - Container

```
from airflow.models.xcom import XCom
print(XCom.__name__)

from airflow.settings import conf
conf.get("core", "xcom_backend")
```

11. Variables

- **Runtime** configuration concept
  - A general key/value store **that is global**
  - **Can be quried** from tasks
- Set via Airflow's user interface or bulk uploaded as a JSON file

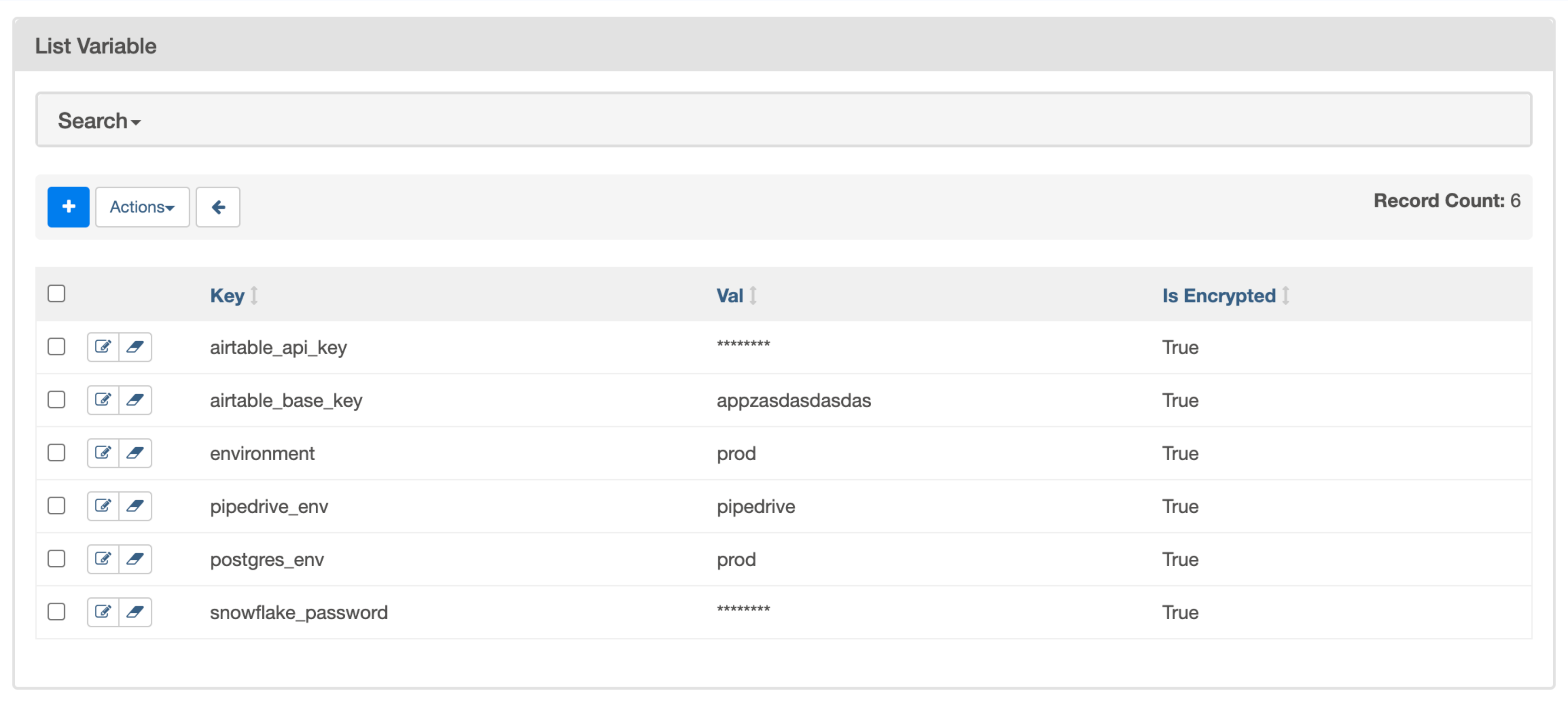
11. Use Variables – get()

```
from airflow.models import Variable

# Normal call style
foo = Variable.get("foo")

# Auto-deserializes a JSON value
bar = Variable.get("bar", deserialize_json=True)

# Returns the value of default_var (None) if the variable is not set
baz = Variable.get("baz", default_var=None)
```

11. Use Variables – templates

```
# Raw value
echo {{ var.value.<variable_name> }}

# Auto-deserialize JSON value
echo {{ var.json.<variable_name> }}
```

11. Managing Variables



List Variable

Search

Record Count: 6

□		Key	Val	Is Encrypted
□		airtable_api_key	********	True
□		airtable_base_key	appzasdasdasdas	True
□		environment	prod	True
□		pipedrive_env	pipedrive	True
□		postgres_env	prod	True
□		snowflake_password	********	True

11. Storing Variables in Environment Variables

```
export AIRFLOW_VAR_FOO=BAR

# To use JSON, store them as JSON strings
export AIRFLOW_VAR_FOO_BAZ='{"hello":"world"}’

from airflow.models import Variable

foo = Variable.get("foo")
foo_json = Variable.get("foo_baz", deserialize_json=True)
```

11. Securing Variables

- Fernet
- Secrets Backend
  - Amazon
    - SecretsManagerBackend
    - SystemsManagerParameterStoreBackend
  - Google
    - CloudSecretManagerBackend
  - Hashicorp
    - VaultBackend
  - Microsoft Azure
    - AzureKeyVaultBackend

Chapter 2.

12. Params

12. Params

- Provide runtime configuration to tasks

12. Params vs Variables

- Params:
  - Specific DAG or Task's runtime configuration
  - Charateristic:
    - Provide different values each time you run a DAG or Task
    - Validation is possible using JSON Schema.
  - Store: In DAG or Task definition
- Variables:
  - Global key/value
  - Characteristic: Create, modify and delete via web UI, code, CLI
  - Store: Metadata db (can be secured)

12. DAG-level Params

```
from airflow import DAG
 from airflow.models.param import Param

 with DAG(
     "the_dag",
     params={
         "x": Param(5, type="integer", minimum=3),
         "my_int_param": 6
     },
 ):
```

12. Task-level Params

```
def print_my_int_param(params):
  print(params.my_int_param)

PythonOperator(
    task_id="print_my_int_param",
    params={"my_int_param": 10},
    python_callable=print_my_int_param,
)
```

12. Referencing Params in a Task

```
PythonOperator(
    task_id="from_template",
    op_args=[
        "{{ params.my_int_param + 10 }}",
    ],
    python_callable=(
        lambda my_int_param: print(my_int_param)
    ),
)
```

12. Referencing Params in a Task

```
with DAG(
    "the_dag",
    params={"my_int_param": Param(5, type="integer", minimum=3)},
    render_template_as_native_obj=True
):
```

12. Referencing Params in a Task

```
prints <class 'str'> by default
# prints <class 'int'> if render_template_as_native_obj=True
PythonOperator(
    task_id="template_type",
    op_args=[
        "{{ params.my_int_param }}",
    ],
    python_callable=(
        lambda my_int_param: print(type(my_int_param))
    ),
)
```

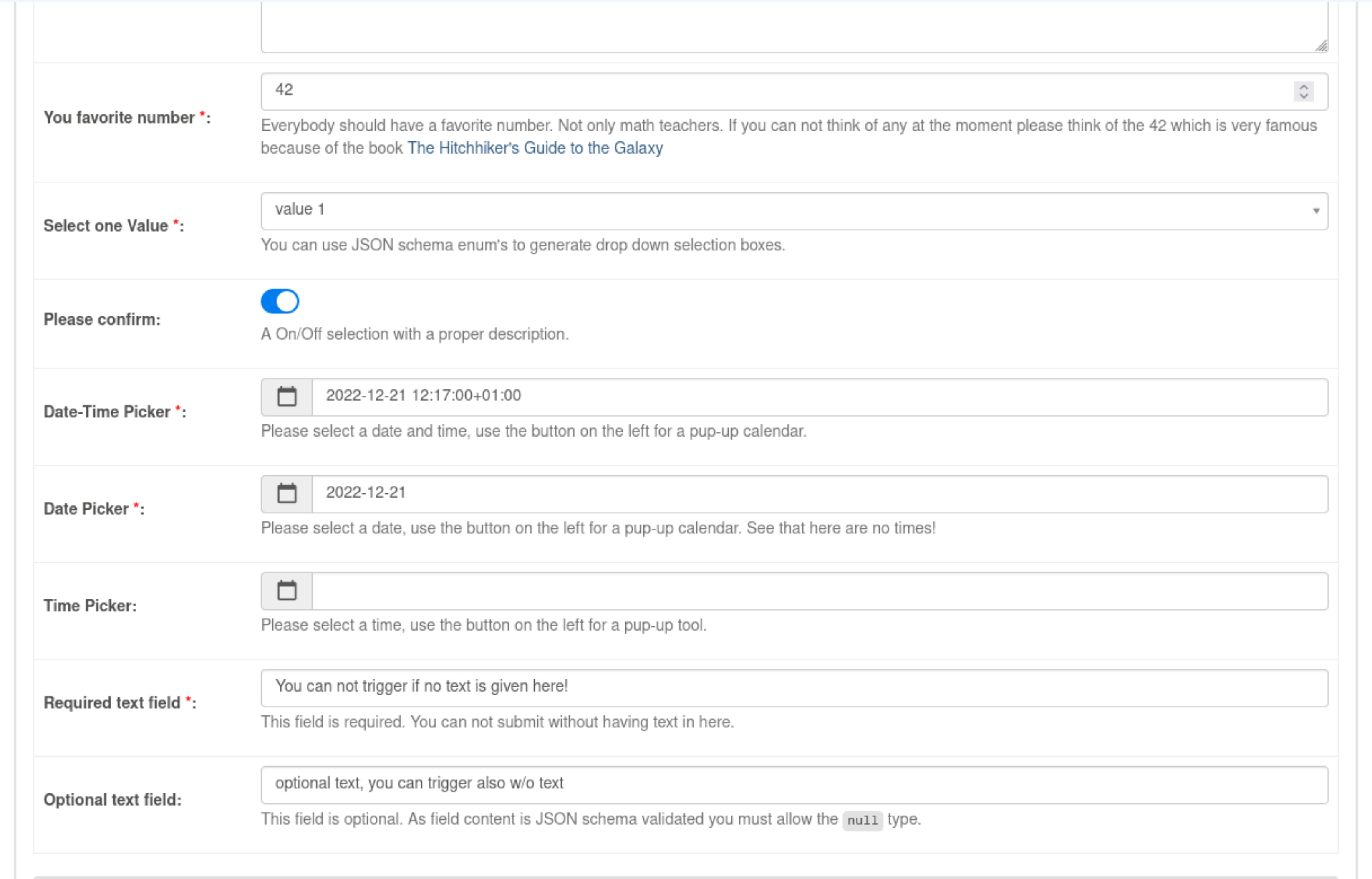
12. Referencing Params in a Task

```
def print_x(**context):
    print(context["params"]["my_int_param"])

 PythonOperator(
    task_id="print_my_int_param",
    python_callable=print_my_int_param,
 )
```

12. JSON Schema Validation

```
with DAG(
    "my_dag",
    params={
        # a required param which can be of multiple types
        # a param must have a default value
        "multi_type_param": Param(5, type=["null", "number", "string"]),

        # an enum param, must be one of three values
        "enum_param": Param("foo", enum=["foo", "bar", 42]),
    }
):
```

12. Trigger UI Form



You favorite number *: 42

Everybody should have a favorite number. Not only math teachers. If you can not think of any at the moment please think of the 42 which is very famous because of the book The Hitchhiker's Guide to the Galaxy

Select one Value *: value 1

You can use JSON schema enum's to generate drop down selection boxes.

Please confirm:

A On/Off selection with a proper description.

Date-Time Picker *: 2022-12-21 12:17:00+01:00

Please select a date and time, use the button on the left for a pup-up calendar.

Date Picker *: 2022-12-21

Please select a date, use the button on the left for a pup-up calendar. See that here are no times!

Time Picker:

Please select a time, use the button on the left for a pup-up tool.

Required text field *: You can not trigger if no text is given here!

This field is required. You can not submit without having text in here.

Optional text field: optional text, you can trigger also w/o text

This field is optional. As field content is JSON schema validated you must allow the `null` type.

Chapter 2.

13. Configuration

13. Celery

- pool
  - Celery Pool 구현

    가능한 선택사항은 prefork (기본값), eventlet, gevent 또는 solo
  - 타입: 문자열
  - 기본값: prefork
  - 환경 변수: AIRFLOW__CELERY__POOL

13. Celery

- worker_concurrency
  - Worker가 가져올 task instance의 수를 정의
  - 타입: 문자열
  - 기본값: 16
  - 환경 변수: AIRFLOW__CELERY__WORKER_CONCURRENCY

13. Celery

- worker_autoscale
    - Worker를 시작할 때 사용될 최대 및 최소 동시성
    - 타입: 문자열
    - 기본값: None
    - 환경 변수: AIRFLOW__CELERY__WORKER_AUTOSCALE
    - 예제: 16,12

13. Celery

- worker_prefetch_multiplier
  - 성능을 향상시키기 위해 Worker가 미리 가져오는 task의 수를 늘리는 데 사용
  - 타입: 정수
  - 기본값: 1
  - 환경 변수: AIRFLOW__CELERY__WORKER_PREFETCH_MULTIPLIER

13. Core

- default_pool_task_slot_count
    - Airflow 2.2.0 버전에서 새롭게 추가
    - 기본 pool에 대한 task slot 수
    - default_pool이 이미 생성된 기존 배포에서는 영향을 주지 않음
        - 기존 배포에 대해서는 웹서버, API 또는 CLI를 사용하여 변경
    - 타입: 문자열
    - 기본값: 128
    - 환경 변수: AIRFLOW__CORE__DEFAULT_POOL_TASK_SLOT_COUNT

13. Core

- max_active_runs_per_dag
  - DAG당 active DAG Run의 최대 수
  - 타입: 문자열
  - 기본값: 16
  - 환경 변수: AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG

13. Core

- max_active_tasks_per_dag
  - Airflow 2.2.0 버전에서 새롭게 추가
  - 각 DAG에서 동시에 실행할 수 있는 task instance의 최대 수
    - 새로운 DAG가 클러스터의 모든 Executor slot을 차지하는 걸 방지
  - 타입: 문자열
  - 기본값: 16
  - 환경 변수: AIRFLOW__CORE__MAX_ACTIVE_TASKS_PER_DAG

13. Core

- parallelism
  - Scheduler당 동시에 실행할 수 있는 task instance의 최대 수
    - 해당 값과 클러스터의 Scheduler 수를 곱한 값이 ”running” 상태를 가진 task instance의 최대 수
  - 타입: 문자열
  - 기본값: 32
  - 환경 변수: AIRFLOW__CORE__PARALLELISM

13. Core

- task_runner
  - 하위 프로세스에서 task instance를 실행하는 데 사용할 클래스
    - StandardTaskRunner
    - CgroupTaskRunner
    - Custom TaskRunner
  - 타입: 문자열
  - 기본값: StandardTaskRunner
  - 환경 변수: AIRFLOW__CORE__TASK_RUNNER

13. Scheduler

- parsing_processes
    - Scheduler는 DAG를 파싱하기 위해 여러 프로세스를 동시에 실행
        - 실행될 프로세스의 수를 정의
    - 타입: 문자열
    - 기본값: 2
    - 환경 변수: AIRFLOW__SCHEDULER__PARSING_PROCESSES

13. Summary

- Airflow 구성:
    - Executor 선택
        - executor 설정을 통해 선택
    - 병렬성 및 동시성
        - parallelism: 전체 시스템에서 동시에 실행할 수 있는 task의 최대 수
        - max_active_tasks_per_dag: 각 DAG에서 동시에 실행할 수 있는 task의 최대 수
        - worker_concurrency: 각 Worker에서 동시에 실행할 수 있는 task의 최대 수
- 데이터베이스
    - 풀링
        - sql_alchemy_pool_size 및 sql_alchemy_max_overflow 설정을 통해 조절