



harve
escola de tecnologia

Airflow

Realize a orquestração de fluxos de trabalho com escalabilidade e segurança utilizando Airflow



AULA 1



INTRODUÇÃO AO APACHE AIRFLOW



Porque usar Apache Airflow

Quando os dados crescem, surgem perguntas novas:

A automação programada funcionou?

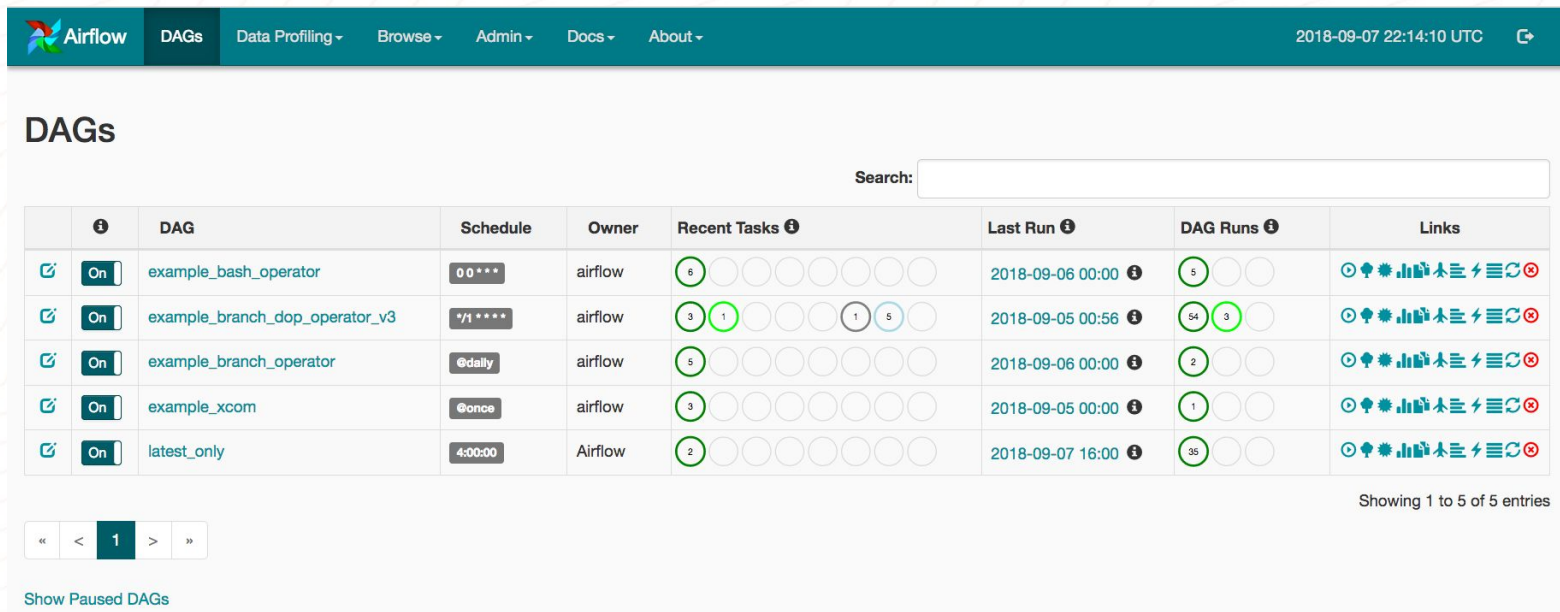
Quando ela para, ela afeta outras funções?

Conseguimos agendar ela sem causar novos problemas?

Qual foi o motivo do erro?

Por que usar Apache Airflow?

O Airflow é uma plataforma de código aberto para desenvolver, agendar e monitorar fluxos de trabalho orientados por lotes.



The screenshot displays the Apache Airflow web interface. At the top is a teal navigation bar with the Airflow logo, a menu (DAGs, Data Profiling, Browse, Admin, Docs, About), and a timestamp (2018-09-07 22:14:10 UTC). Below the navigation bar, the main heading "DAGs" is followed by a search bar. A table lists five DAGs with columns for status, name, schedule, owner, recent tasks, last run, DAG runs, and links. The first DAG, "example_branch_operator", is highlighted. At the bottom, there is a pagination control showing "1" of 5 entries and a link to "Show Paused DAGs".

	i	DAG	Schedule	Owner	Recent Tasks i	Last Run i	DAG Runs i	Links
	On	example_branch_operator	0 0 ***	airflow	8	2018-09-06 00:00 i	5	
	On	example_branch_dop_operator_v3	* / 1 * * * *	airflow	3 1	2018-09-05 00:56 i	54 3	
	On	example_branch_operator	@daily	airflow	5	2018-09-06 00:00 i	2	
	On	example_xcom	@once	airflow	3	2018-09-05 00:00 i	1	
	On	latest_only	4:00:00	Airflow	2	2018-09-07 16:00 i	35	

Showing 1 to 5 of 5 entries

« < 1 > »

Show Paused DAGs

Por que usar Apache Airflow?

O framework Python extensível do Airflow permite que você construa fluxos de trabalho conectando-se virtualmente a qualquer tecnologia.

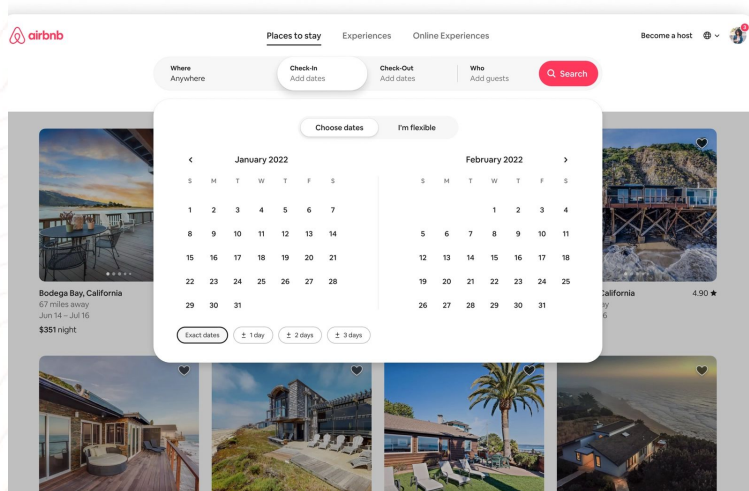


Apache
Airflow



Por que usar Apache Airflow?

Surgiu em 2014 como uma ferramenta interna do Airbnb para gerenciar e orquestrar fluxos de dados complexos. Em 2016, foi tornado open-source sob a Apache Software Foundation, ganhando popularidade rapidamente na comunidade de data engineering.



Visão Geral do Apache Airflow

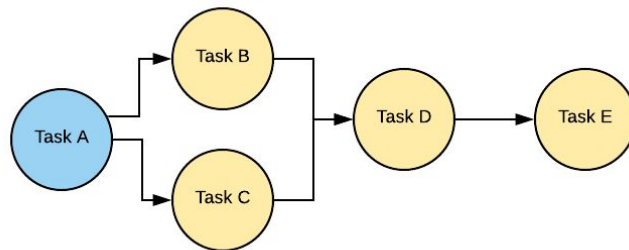
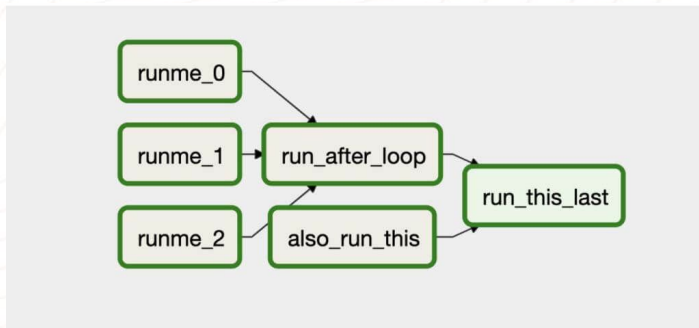
Por que usar o Airflow?

- **Flexibilidade:** Airflow oferece um framework Python extensível que permite construir workflows conectando-se a uma ampla variedade de tecnologias.
- **Escalabilidade:** Pode ser implantado de várias maneiras, desde uma configuração simples em um laptop até um ambiente distribuído para suportar workflows complexos.
- **Monitoramento:** Possui uma interface web que ajuda a visualizar e gerenciar o estado dos fluxos de trabalho, facilitando a identificação e resolução de problemas.
- **Comunidade ativa:** Como uma ferramenta open-source mantida pela Apache Software Foundation, o Airflow beneficia-se de uma comunidade engajada de desenvolvedores e usuários, garantindo suporte e atualizações regulares.

Conceitos Fundamentais

DAGs (Directed Acyclic Graphs)

Em Apache Airflow, uma DAG (Directed Acyclic Graph) é uma representação de um fluxo de trabalho, composto por tarefas e suas dependências. Uma DAG é composta por nós (tarefas) e arestas (dependências entre tarefas), formando um grafo direcionado sem ciclos, onde cada tarefa só pode ser executada após a conclusão de suas tarefas predecessoras.



Conceitos Fundamentais

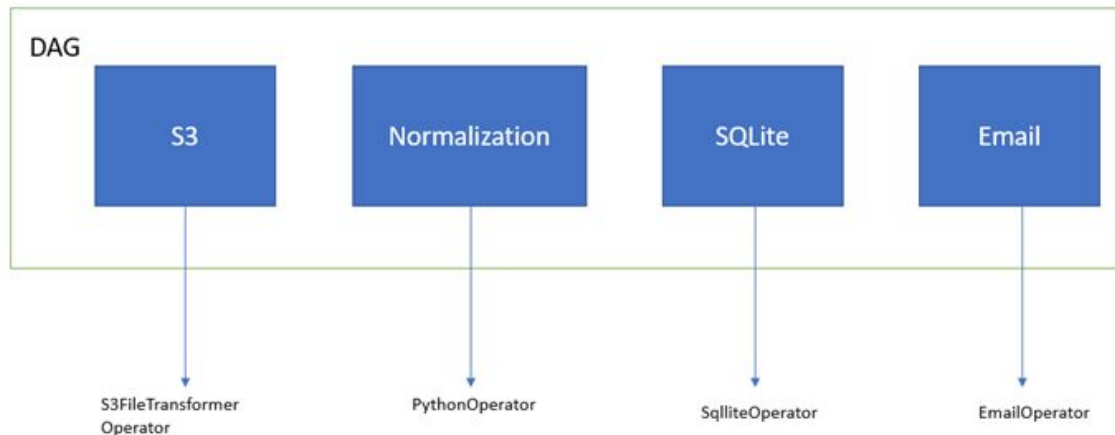
DAGs em python

```
1  from datetime import datetime
2
3  from airflow import DAG
4  from airflow.decorators import task
5  from airflow.operators.bash import BashOperator
6
7  # A DAG represents a workflow, a collection of tasks
8  with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
9      # Tasks are represented as operators
10     hello = BashOperator(task_id="hello", bash_command="echo hello")
11
12     @task()
13     def airflow():
14         print("airflow")
15
16     # Set dependencies between tasks
17     hello >> airflow()
```

Conceitos Fundamentais

Operators

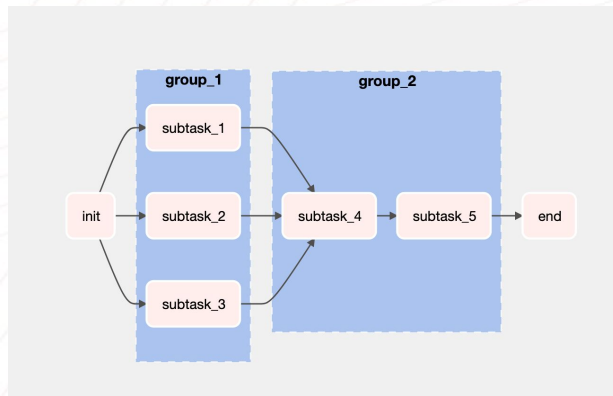
Um operador define uma unidade de trabalho para o Airflow dentro de uma DAG. Ele deve ser de um tipo específico de acordo com a sua função.



Conceitos Fundamentais

Tasks

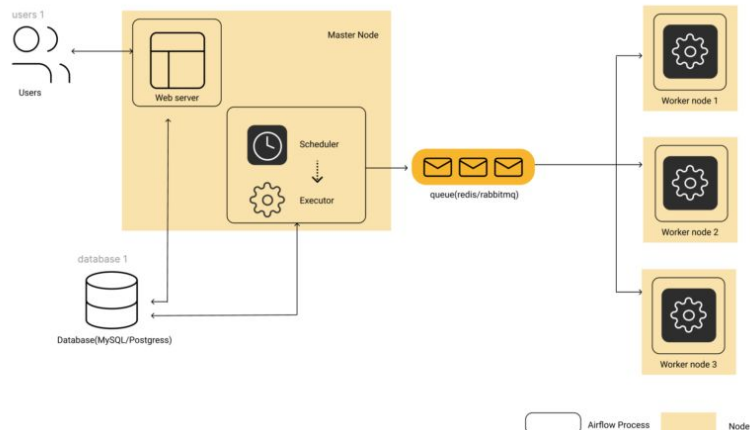
Em Apache Airflow, uma "Task" (tarefa) é uma unidade de trabalho específica que é executada como parte de um fluxo de trabalho definido em um DAG (Directed Acyclic Graph). Cada tarefa representa uma ação individual a ser realizada, como executar um script Python, executar um comando Bash, enviar um email, entre outras dependendo do operador relacionado.



Conceitos Fundamentais

Scheduler

O agendador do Airflow monitora todas as tarefas e DAGs e, em seguida, aciona as instâncias das tarefas assim que suas dependências são concluídas. Uma vez por minuto, por padrão, o agendador verifica se alguma tarefa ativa pode ser acionada. O agendador usa o Executor configurado para executar tarefas que estão prontas.



Conceitos Fundamentais

Executor

Os executores são o mecanismo pelo qual as instâncias de tarefa são executadas. Eles possuem uma API comum e são "plugáveis", o que significa que você pode trocar os executores com base nas necessidades da sua instalação.

O Airflow só pode ter um executor configurado por vez, isso é definido pela opção executor na seção [core] do arquivo de configuração.

Conceitos Fundamentais

Executor

Executores Locais

Sequential
Executor

Local Executor

Executores Remotos

CeleryExecutor

CeleryKubernetes
Executor

KubernetesExecut
or

KubernetesLocalE
xecutor

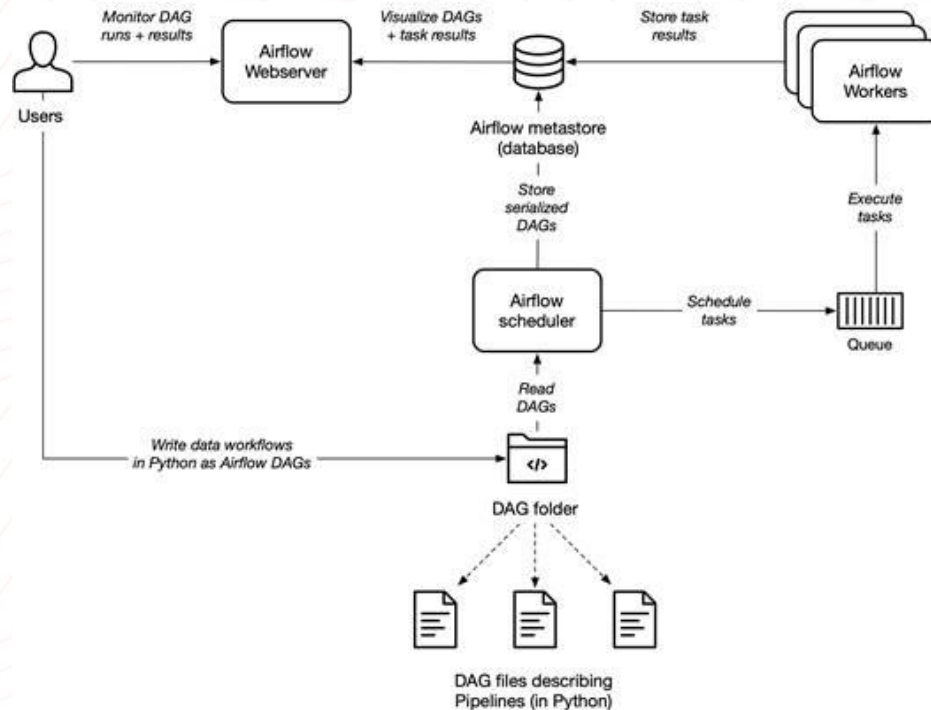
Conceitos Fundamentais

Metadatabase

O Metadatabase do Airflow é um banco de dados que armazena informações sobre tarefas, fluxos de trabalho e execuções. Ele registra o histórico de execução das tarefas, incluindo sucesso, falha ou conclusão. Esses dados são fundamentais para o monitoramento, agendamento e gerenciamento de fluxos de trabalho. O Metadatabase é essencial para garantir a integridade e rastreabilidade das operações realizadas pelo Airflow. Ele permite análises retrospectivas e insights em tempo real sobre o desempenho e a eficácia dos processos automatizados.

Arquitetura do Airflow

Componentes principais



CHECK-POINT



Instalação e configuração do Airflow



INSTALAÇÃO

1. [Instalação via Docker](#)
2. [Acesso e login](#)
3. [Visão geral do painel](#)
 - a. Navegação pelas funcionalidades
4. [Conexões com sistemas externos](#)
5. [Variáveis globais](#)
6. Configuração de Celery Executor



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

Vamos realizar a instalação usando Docker Compose.

Tutorial: <https://airflow.apache.org/docs/apache-airflow/2.0.2/start/docker.html>



CHECK-POINT



CRIAÇÃO DE DAGS (DIRECTED ACYCLIC GRAPHS)





















































































































CRIAÇÃO DE DAGS (DIRECTED ACYCLIC GRAPHS)





 **DAGs** Data Profiling ▾ Browse ▾ Admin ▾ Docs ▾ About ▾ 2018-09-07 22:14:10 UTC 

DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks 	Last Run 	DAG Runs 	Links
	 On	example_bash_operator	00***	airflow	 6      	2018-09-06 00:00 	 5  	        
	 On	example_branch_dop_operator_v3	*/* ****	airflow	 3  1    1  5 	2018-09-05 00:56 	 54  3 	        
	 On	example_branch_operator	@daily	airflow	 5      	2018-09-06 00:00 	 2  	        
	 On	example_xcom	@once	airflow	 3      	2018-09-05 00:00 	 1  	        
	 On	latest_only	4:00:00	Airflow	 2      	2018-09-07 16:00 	 35  	        

Showing 1 to 5 of 5 entries

  1  

[Show Paused DAGs](#)

CRIAÇÃO DE DAGS (DIRECTED ACYCLIC GRAPHS)

Importando bibliotecas

```
from datetime import datetime, timedelta  
from airflow import DAG  
from airflow.operators.dummy_operator import DummyOperator  
from airflow.operators.python_operator import PythonOperator
```

CRIAÇÃO DE DAGS (DIRECTED ACYCLIC GRAPHS)

Definindo argumentos padrão para a DAG

```
default_args = {  
    'owner': 'airflow',  
    'depends_on_past': False,  
    'start_date': datetime(2024, 4, 27),  
    'email_on_failure': False,  
    'email_on_retry': False,  
    'retries': 1,  
    'retry_delay': timedelta(minutes=5),  
}
```


CRIAÇÃO DE DAGS (DIRECTED ACYCLIC GRAPHS)

Definindo a DAG.

```
dag = DAG(  
    'simple_dag',  
    default_args=default_args,  
    description='Uma DAG simples com dois operadores',  
    schedule_interval=timedelta(days=1),  
)
```

CRIAÇÃO DE DAGS (DIRECTED ACYCLIC GRAPHS)

Definindo operadores.

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

def print_hello():
    print('Olá, Airflow!')

hello_task = PythonOperator(
    task_id='hello_task',
    python_callable=print_hello,
    dag=dag,
)

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

CRIAÇÃO DE DAGS (DIRECTED ACYCLIC GRAPHS)

Definindo a ordem de execução das tarefas.

```
start_task >> hello_task >> end_task
```



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

Crie uma DAG conforme o exemplo dos slides.

CHECK-POINT



OPERADORES DO AIRFLOW



OPERADORES DO AIRFLOW

Os operadores herdam do BaseOperator, que inclui todos os argumentos necessários para executar trabalho no Airflow.

1. **PythonOperator:** Executa uma função Python definida pelo usuário como uma tarefa.
2. **BashOperator:** Executa um comando Bash como uma tarefa.
3. **DummyOperator:** Uma tarefa fictícia usada para fins de organização ou marcação.
4. **BranchPythonOperator:** Permite ramificar o fluxo de trabalho com base em condições Python.
5. **EmailOperator:** Envia e-mails como uma tarefa.
6. **HTTPOperator:** Realiza solicitações HTTP como uma tarefa.
7. **Sensor:** Espera até que uma determinada condição seja atendida antes de continuar o fluxo de trabalho. Existem sensores específicos para arquivos, banco de dados, API, etc.
8. **SubDagOperator:** Executa um fluxo de trabalho definido em um subDAG.
9. **DockerOperator:** Executa um contêiner Docker como uma tarefa.
10. **SparkSubmitOperator:** Submete um trabalho para execução no Apache Spark.
11. **KubernetesPodOperator:** Projetado para executar contêineres em um cluster Kubernetes.

...

OPERADORES DO AIRFLOW

Vamos criar um BashOperator para deletar os arquivos da pasta da lixeira.

OPERADORES DO AIRFLOW

BashOperator: Importando bibliotecas

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

OPERADORES DO AIRFLOW

BashOperator: Definindo argumentos padrões das DAGs

```
default_args = {  
    'owner': 'airflow',  
    'depends_on_past': False,  
    'start_date': datetime(2024, 4, 30),  
    'email_on_failure': False,  
    'email_on_retry': False,  
    'retries': 1,  
}
```

OPERADORES DO AIRFLOW

BashOperator: Definindo a DAG.

```
dag = DAG(  
    'exemplo_bash_operator_deletar_lixeira',  
    default_args=default_args,  
    description='Exemplo de uso do BashOperator para deletar arquivos da lixeira',  
    schedule_interval='@daily',  
)
```


OPERADORES DO AIRFLOW

BashOperator: Criando comando para deletar arquivos.

```
delete_trash_command = """  
rm -rf ~/.local/share/Trash/*  
"""
```

OPERADORES DO AIRFLOW

BashOperator: Definindo a tarefa.

```
delete_trash_task = BashOperator(  
    task_id='deletar_arquivos_lixeira',  
    bash_command=delete_trash_command,  
    dag=dag,  
)
```

OPERADORES DO AIRFLOW

BashOperator: Definindo a dependência das tarefas.

```
delete_trash_task
```

OPERADORES DO AIRFLOW

Vamos criar um `MySqlOperator` para criar uma tabela nova.

OPERADORES DO AIRFLOW

MySQLOperator: Importando bibliotecas

```
from airflow import DAG
from airflow.operators.mysql_operator import MySQLOperator
from datetime import datetime
```

OPERADORES DO AIRFLOW

MySQLOperator: Definindo os argumentos das DAG.

```
default_args = {  
    'owner': 'airflow',  
    'depends_on_past': False,  
    'start_date': datetime(2024, 4, 30),  
    'email_on_failure': False,  
    'email_on_retry': False,  
    'retries': 1,  
}
```

OPERADORES DO AIRFLOW

MySQLOperator: Definindo a DAG.

```
dag = DAG(  
    'exemplo_mysql_operator',  
    default_args=default_args,  
    description='Exemplo de uso do MySQLOperator',  
    schedule_interval='@daily',  
)
```

OPERADORES DO AIRFLOW

MySqlOperator: Definindo o comando sql.

```
sql_command = """
CREATE TABLE exemplo (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(255) NOT NULL
);
"""
```

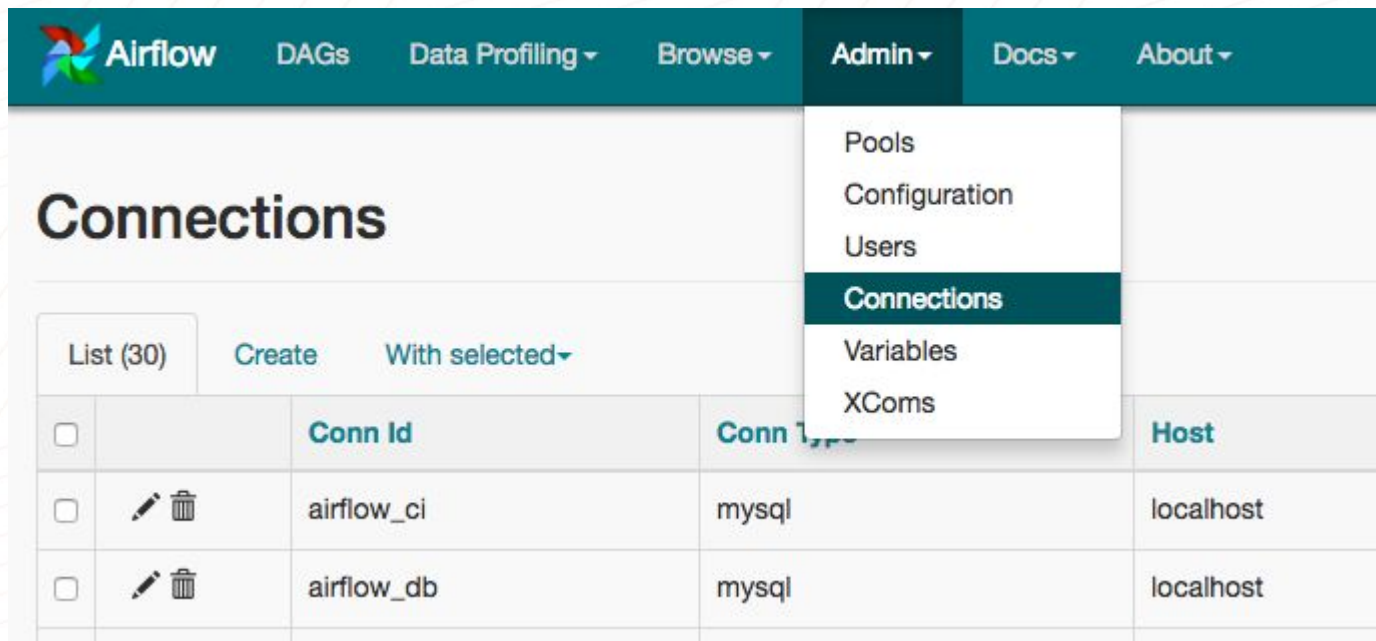

OPERADORES DO AIRFLOW

MySQLOperator: Especificando a tarefa e a conexão que será utilizada dentro dela.





```
mysql_task = MySQLOperator(  
    task_id='executar_comando_mysql',  
    mysql_conn_id='mysql_conn', # O nome da conexão MySQL definida no Airflow  
    sql=sql_command,  
    dag=dag,  
)
```

OPERADORES DO AIRFLOW

MySQLOperator: Especificando a tarefa e a conexão que será utilizada dentro dela.



The screenshot shows the Airflow web interface. The top navigation bar includes links for DAGs, Data Profiling, Browse, Admin, Docs, and About. The 'Admin' menu is expanded, showing options like Pools, Configuration, Users, Connections (highlighted), Variables, and XComs. The main content area is titled 'Connections' and features a table with 30 items. The table has columns for Conn Id, Conn Type, and Host. Two connections are visible: 'airflow_ci' and 'airflow_db', both using 'mysql' as the connection type and 'localhost' as the host.

		Conn Id	Conn Type	Host
<input type="checkbox"/>	 	airflow_ci	mysql	localhost
<input type="checkbox"/>	 	airflow_db	mysql	localhost

OPERADORES DO AIRFLOW

MySQLOperator: Associe a conexão ao operador no arquivo "airflow.cfg".

```
[mysql]  
mysql_conn_id = mysql_conn
```

OPERADORES DO AIRFLOW

Voltando ao python...

OPERADORES DO AIRFLOW

MySQLOperator: Definindo dependências.

```
mysql_task
```



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

1. Vamos criar uma DAG utilizando o operador airflow para apagar os arquivos dentro de uma pasta usando BashOperator.
2. Vamos criar uma segunda DAG usando o operador do mysql para criar uma tabela chamada "exemplo_[nome do aluno]".

DESCOBERTA GUIADA

1. Vamos criar uma DAG utilizando o operador airflow para apagar os arquivos dentro de uma pasta usando BashOperator.

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

# Define os argumentos padrão do DAG
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2024, 5, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Define o nome do DAG
dag = DAG(
    'delete_files_dag',
    default_args=default_args,
    description='Um exemplo de DAG que apaga arquivos de uma pasta',
    schedule_interval=timedelta(days=1),
)

# Define a tarefa para apagar os arquivos
delete_files_task = BashOperator(
    task_id='delete_files_task',
    bash_command='rm -rf /caminho/para/a/sua/pasta/*', # Substitua pelo caminho da sua pasta
    dag=dag,
)

# Define as dependências entre as tarefas
delete_files_task
```


DESCOBERTA GUIADA

Operador Mysql:

1. Adicionar a criação do container no docker compose do Airflow para o Mysql;
2. Usar o IP como host na configuração de conexão com o Airflow;
3. Acessar o container do Mysql e verificar a criação com "show create table xxxx;"

```
from airflow import DAG
from airflow.operators.mysql_operator import MySqlOperator
from datetime import datetime
```

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2024, 5, 1),
}
```

```
dag = DAG(
    'create_table_dag',
    default_args=default_args,
    description='Cria uma tabela no banco de dados harve _airflow',
    schedule_interval=None,
)
```

```
create_table_task = MySqlOperator(
    task_id='create_table_task',
    mysql_conn_id='mysql_default',
    sql="""
CREATE TABLE IF NOT EXISTS example_table (
    id INT PRIMARY KEY,
    name VARCHAR(50)
)
""",
    dag=dag,
)
```

```
create_table_task
```

Docker Compose

Services:

mysql:

image: mysql:latest

container_name: mysql-airflow

restart: always

environment:

MYSQL_ROOT_PASSWORD: mypassword

MYSQL_DATABASE: harve _airflow

ports:

- "3306:3306"

CHECK-POINT



AULA 2



AGENDAMENTO E EXECUÇÃO DE TAREFAS



AGENDAMENTO E EXECUÇÃO DE TAREFAS

Configuramos no `schedule_interval`:

```
dag = DAG(  
    'exemplo_mysql_operator',  
    default_args=default_args,  
    description='Exemplo de uso do MySQLOperator',  
    schedule_interval='@daily',  
)
```


AGENDAMENTO E EXECUÇÃO DE TAREFAS

Podemos colocar nos seguintes formatos:

Cron Expression: Você pode usar uma expressão cron para definir intervalos precisos. Por exemplo:

* * * * * para acionar a DAG a cada minuto.

0 * * * * para acionar a DAG a cada hora no início da hora.

0 0 * * * para acionar a DAG uma vez por dia à meia-noite.

E assim por diante, com as partes da expressão cron definindo minutos, horas, dias do mês, meses e dias da semana.

Intervals: Você também pode usar intervalos predefinidos, como @hourly, @daily, @weekly, @monthly, @yearly, @once, etc.

AGENDAMENTO E EXECUÇÃO DE TAREFAS

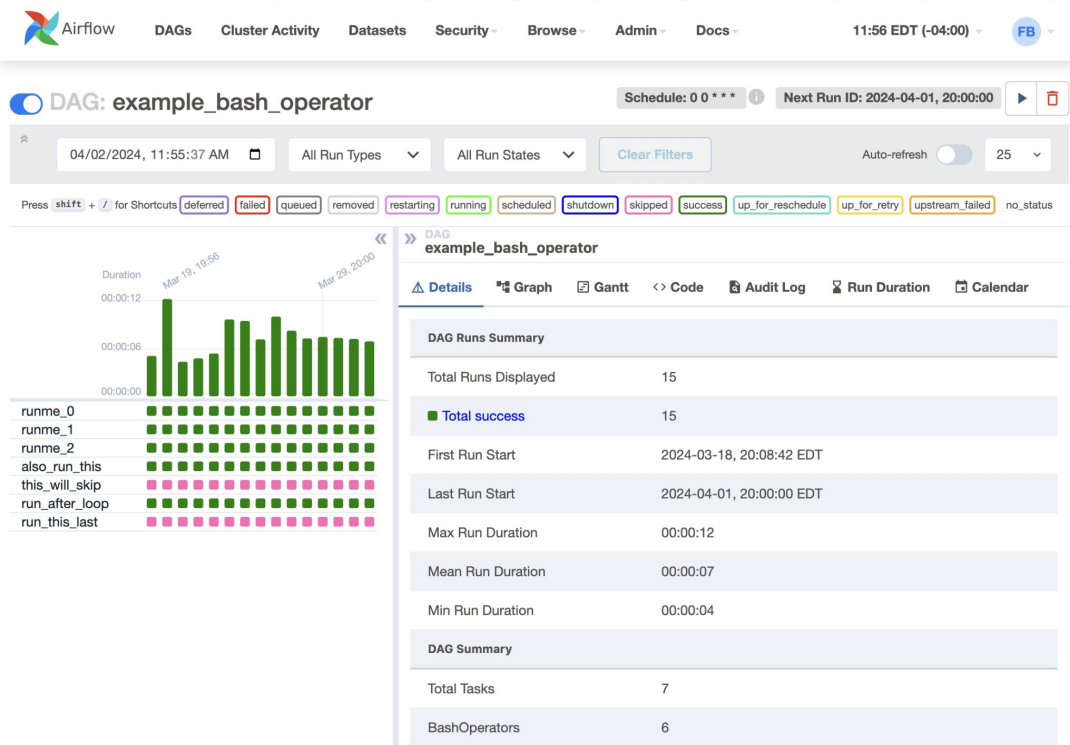
Podemos colocar nos seguintes formatos:

Timedelta Object: Você pode usar objetos `datetime.timedelta` para definir um intervalo de tempo relativo. Por exemplo, `timedelta(hours=1)`, `timedelta(days=1)`, `timedelta(weeks=1)`, etc.

DataFixtures: Para casos em que você deseja basear o agendamento em datas específicas, você pode usar os objetos `airflow.utils.dates.days_ago`, `airflow.utils.dates.weeks_ago`, `airflow.utils.dates.months_ago`, `airflow.utils.dates.days_after`, etc.

AGENDAMENTO E EXECUÇÃO DE TAREFAS

Monitoramento de execução:



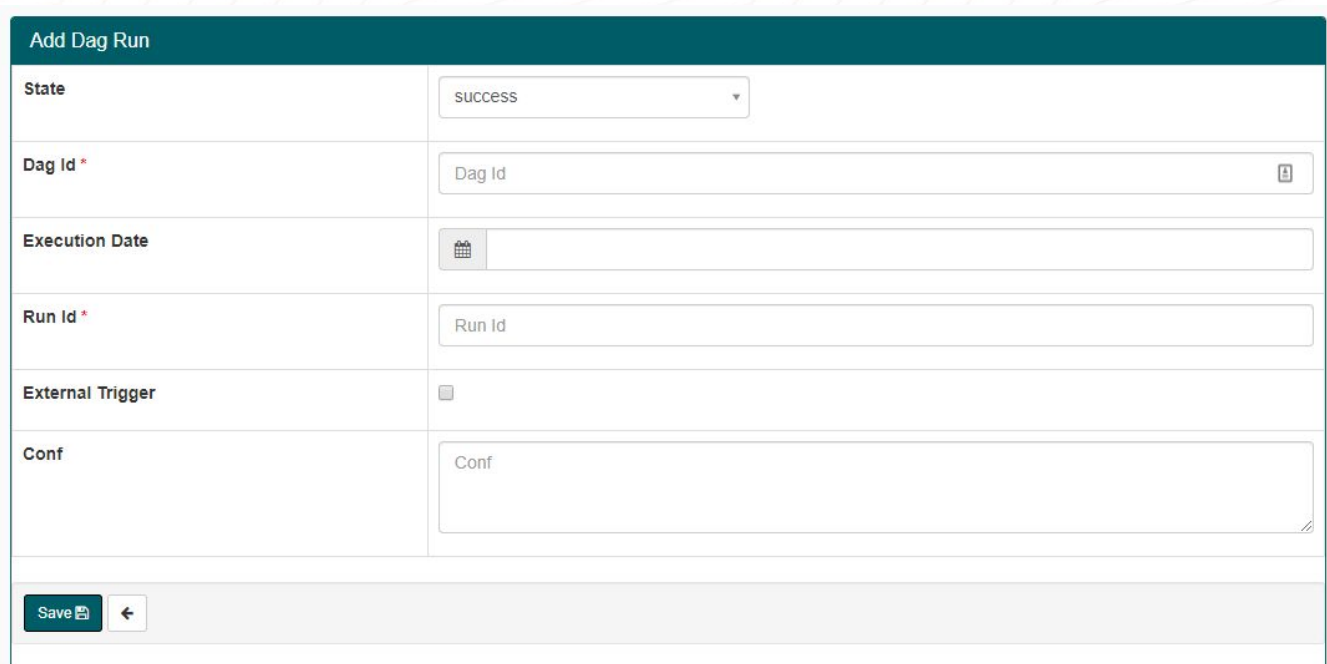
AGENDAMENTO E EXECUÇÃO DE TAREFAS

O reagendamento das tarefas pode ser realizado das seguintes formas:

- Via Interface web
- Via linha de comando
- Via python

AGENDAMENTO E EXECUÇÃO DE TAREFAS

Na Interface web do airflow, selecione a task que falhou e escolha “Trigger DAG Run”:



The screenshot shows the 'Add Dag Run' form in the Airflow web interface. The form has a dark teal header with the title 'Add Dag Run'. Below the header, there are six rows of form fields:

- State:** A dropdown menu with 'success' selected.
- Dag Id *:** A text input field with 'Dag Id' as a placeholder and a small icon on the right.
- Execution Date:** A date picker with a calendar icon and an empty input field.
- Run Id *:** A text input field with 'Run Id' as a placeholder.
- External Trigger:** A checkbox that is currently unchecked.
- Conf:** A large text area with 'Conf' as a placeholder and a small icon on the right.

At the bottom of the form, there is a grey bar containing a 'Save' button with a document icon and a back arrow button.

AGENDAMENTO E EXECUÇÃO DE TAREFAS

Use o comando “airflow clear” para limpar o estado de uma ou mais tarefas com falha e “airflow dags trigger” para programar.

```
airflow clear nome_da_dag -s START_DATE -t task_id
```

```
airflow dags trigger nome_da_dag
```

AGENDAMENTO E EXECUÇÃO DE TAREFAS

Você também pode reexecutar tarefas programaticamente usando o Python.

```
✓ from airflow import DAG, settings
  from airflow.api.common.experimental.trigger_dag import trigger_dag

dag_id = 'nome_da_dag'
execution_date = 'YYYY-MM-DD'
task_id = 'nome_da_tarefa'

dag = DAG(dag_id)

session = settings.Session()
✓ dr = trigger_dag(dag_id, run_id=f'triggered_{execution_date}', conf=None,
  |   execution_date=execution_date, replace_microseconds=False, session=session)
```



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

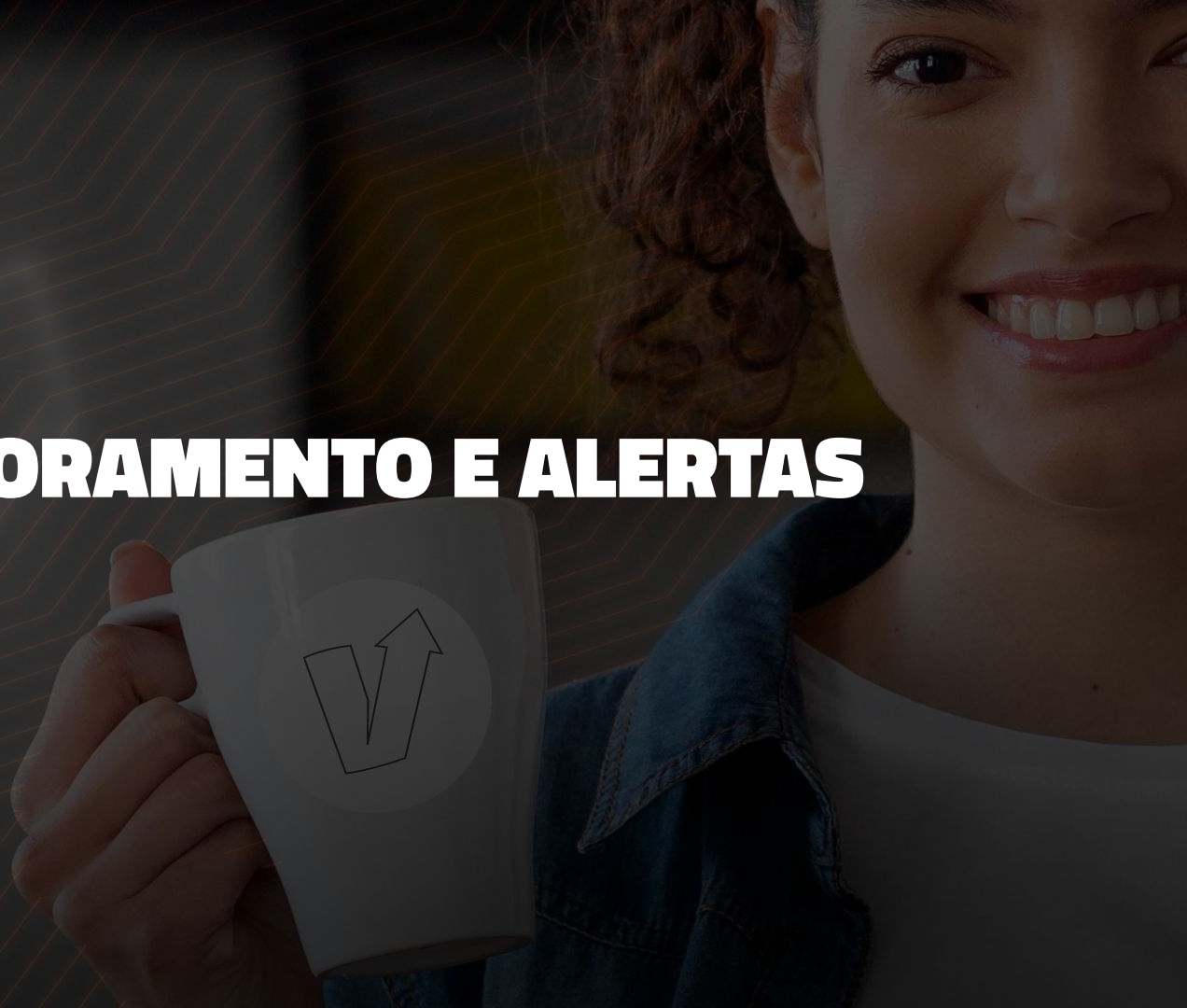
Vamos criar uma DAG simples que irá rodar terças e quintas às 20h.

Vamos simular um erro em uma DAG e rodá-la novamente.

CHECK-POINT

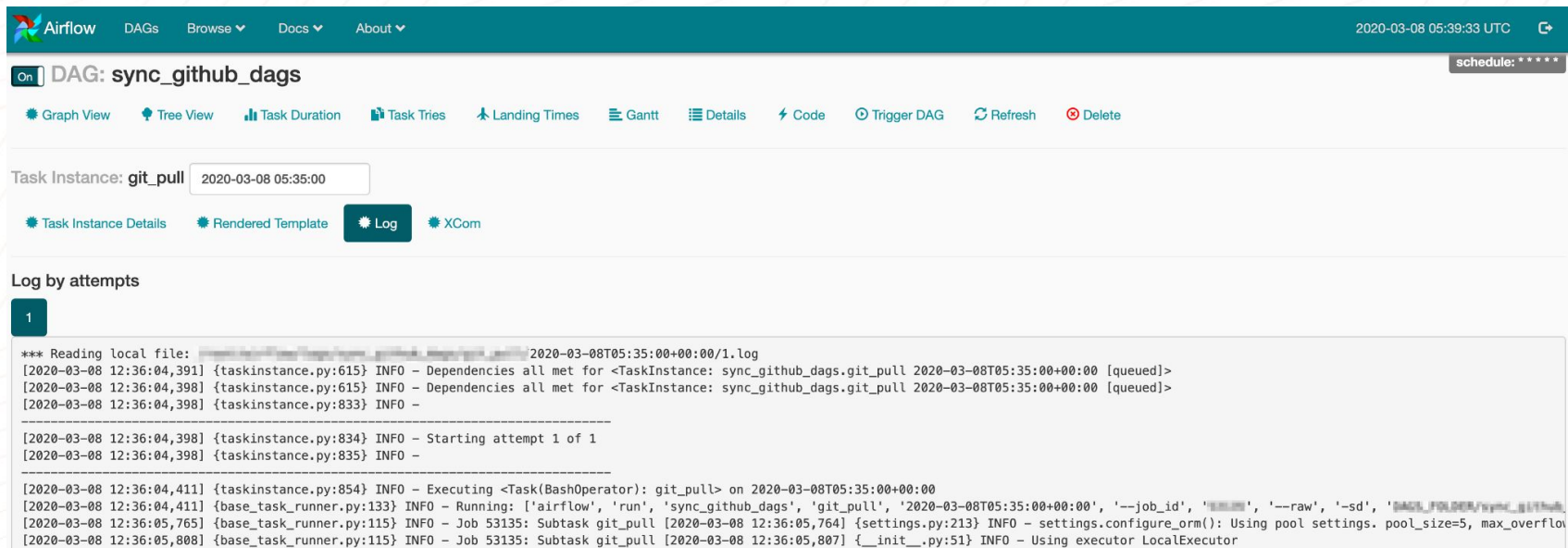


MONITORAMENTO E ALERTAS



MONITORAMENTO E ALERTAS

Logs do Airflow: O Airflow registra informações detalhadas sobre a execução de DAGs e tarefas.



The screenshot displays the Apache Airflow web interface. At the top, there's a navigation bar with links for DAGs, Browse, Docs, and About. The current view is for a DAG named 'sync_github_dags'. Below the navigation bar, there are tabs for Graph View, Tree View, Task Duration, Task Tries, Landing Times, Gantt, Details, Code, Trigger DAG, Refresh, and Delete. The 'Details' tab is selected, showing the task instance 'git_pull' for the date '2020-03-08 05:35:00'. Below this, there are buttons for Task Instance Details, Rendered Template, Log, and XCom. The 'Log' button is highlighted. The log content shows the execution details of the task instance, including dependencies, starting attempt, and the execution of the task using the LocalExecutor.

On DAG: sync_github_dags schedule: *****

Task Instance: git_pull 2020-03-08 05:35:00

Task Instance Details Rendered Template Log XCom

Log by attempts

1

```
*** Reading local file: 2020-03-08T05:35:00+00:00/1.log
[2020-03-08 12:36:04,391] {taskinstance.py:615} INFO - Dependencies all met for <TaskInstance: sync_github_dags.git_pull 2020-03-08T05:35:00+00:00 [queued]>
[2020-03-08 12:36:04,398] {taskinstance.py:615} INFO - Dependencies all met for <TaskInstance: sync_github_dags.git_pull 2020-03-08T05:35:00+00:00 [queued]>
[2020-03-08 12:36:04,398] {taskinstance.py:833} INFO -
-----
[2020-03-08 12:36:04,398] {taskinstance.py:834} INFO - Starting attempt 1 of 1
[2020-03-08 12:36:04,398] {taskinstance.py:835} INFO -
-----
[2020-03-08 12:36:04,411] {taskinstance.py:854} INFO - Executing <Task(BashOperator): git_pull> on 2020-03-08T05:35:00+00:00
[2020-03-08 12:36:04,411] {base_task_runner.py:133} INFO - Running: ['airflow', 'run', 'sync_github_dags', 'git_pull', '2020-03-08T05:35:00+00:00', '--job_id', '1', '--raw', '-sd', 'DAGS_FOLDER/sync_github_dags']
[2020-03-08 12:36:05,765] {base_task_runner.py:115} INFO - Job 53135: Subtask git_pull [2020-03-08 12:36:05,764] {settings.py:213} INFO - settings.configure_orm(): Using pool settings. pool_size=5, max_overflow=10
[2020-03-08 12:36:05,807] {base_task_runner.py:115} INFO - Job 53135: Subtask git_pull [2020-03-08 12:36:05,807] {__init__.py:51} INFO - Using executor LocalExecutor
```

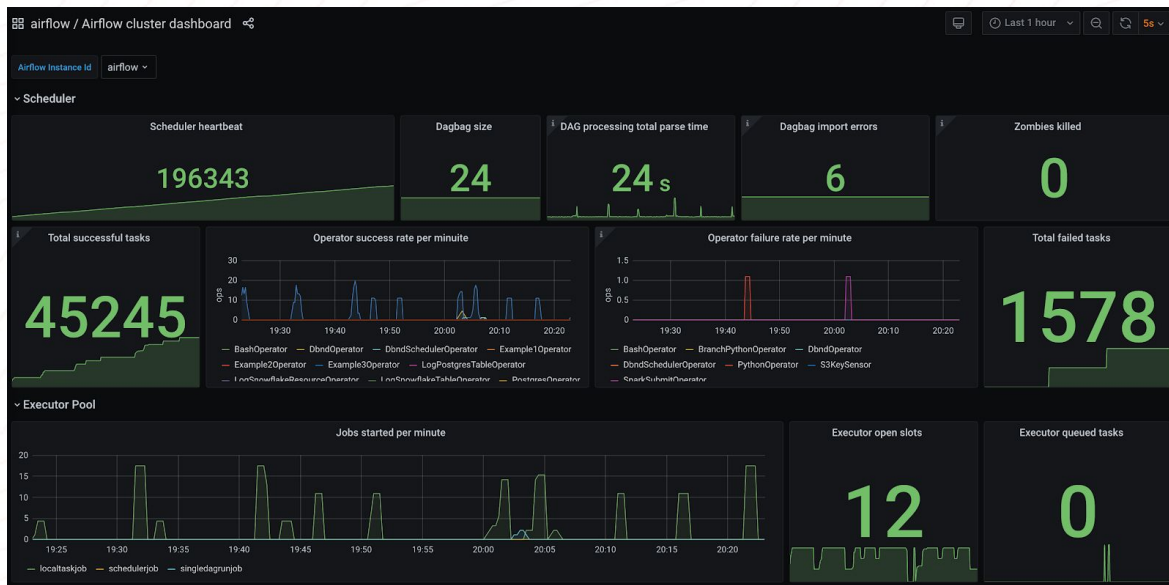
MONITORAMENTO E ALERTAS

Alertas por e-mail: O Airflow suporta o envio de e-mails para alertar sobre o status das execuções de DAGs.

```
default_args = {  
    'owner': 'airflow',  
    'email': ['you@example.com'], # Insira o seu endereço de e-mail aqui  
    'email_on_failure': True,      # Enviar e-mail em caso de falha  
    'email_on_retry': False,  
    'start_date': datetime(2024, 4, 30),  
}
```

MONITORAMENTO E ALERTAS

Integração com ferramentas de monitoramento externas: Você pode integrar o Airflow com ferramentas de monitoramento externas, como Prometheus, Grafana, Datadog, entre outras.





DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

Vamos identificar o log de uma DAG com erro.

Vamos programar envio de email de notificação em uma DAG.

CHECK-POINT



PARÂMETROS E VARIÁVEIS



PARÂMETROS E VARIÁVEIS

Como passar informações entre as tasks?

PARÂMETROS E VARIÁVEIS

Parâmetros (Parameters):

São valores que podem ser passados dinamicamente para uma DAG ou para uma execução de tarefa:

- Especificar a data de início ou término para uma execução de DAG.
- Definir um valor de limiar para o qual uma tarefa deve ser executada.
- Passar configurações específicas para uma tarefa, como uma URL de API ou credenciais de acesso.

Variáveis (Variables):

São uma maneira de armazenar e compartilhar informações globalmente entre os DAGs e as tarefas. Elas são armazenadas no banco de dados do Airflow e podem ser acessadas por ***todas as instâncias do Airflow em execução***:

- Credenciais de autenticação para serviços externos.
- Configurações globais que são compartilhadas entre vários DAGs.
- Informações de estado ou configuração que precisam ser acessadas de várias partes do seu fluxo de trabalho.

PARÂMETROS

Veja esse exemplo onde temos 2 tasks, uma que recupera a lista dos clientes no sql e outra task que chama a api para cada um. Criamos a função que chama a api e especificamos o parâmetro.

```
def call_api_for_each_client(client_data, **kwargs):  
    # Endpoint da API  
    api_endpoint = "http://exemplo.com/api/clientes"  
    # Parâmetros da requisição  
    payload = {  
        "id": client_data['id'],  
        "email": client_data['email']  
    }  
    # Fazendo a requisição à API  
    response = requests.post(api_endpoint, json=payload)  
    # Exibindo o resultado da requisição  
    print(response.json())
```

PARÂMETROS

Passamos o parâmetro na definição da task, nesse caso, ela virá do retorno da task "selct_clients_task".

```
call_api_for_each_client_task = PythonOperator(  
    task_id='call_api_for_each_client_task',  
    python_callable=call_api_for_each_client,  
    op_args=['[{ ti.xcom_pull(task_ids="select_clients_task") }]'],  
    provide_context=True,  
    dag=dag,  
)
```

PARÂMETROS

A tarefa **select_clients_from_db** (tarefa 1) que foi executada antes, retornou os dados anteriormente, permitindo usarmos ele no futuro.

```
def select_clients_from_db():  
    # Data atual menos um dia  
    date_threshold = datetime.now() - timedelta(days=1)  
    # Convertendo a data para o formato de string para usar na consulta SQL  
    date_threshold_str = date_threshold.strftime('%Y-%m-%d')  
    # Construindo a query SQL para selecionar clientes  
    query = f"SELECT id, email FROM clientes WHERE data_criacao > '{date_thres  
    # Retornando a query para ser usada como parâmetro na tarefa MySQLOperator  
    return query
```

VARIÁVEIS

Setando uma variável global.

```
from airflow.models import Variable

# Definindo o valor da variável
custom_message_value = "Bom dia, mundo!"

# Definindo a variável no Airflow
Variable.set("custom_message", custom_message_value)
```


VARIÁVEIS

Podemos também setar via linha de comando.

```
airflow variables set custom_message "Bom dia, mundo!"
```


VARIÁVEIS

Dando get em uma variável.

```
def print_custom_message():  
    custom_message = Variable.get("custom_message", default_var="Olá, mundo!")  
    print(custom_message)
```



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

Vamos criar um contexto nas DAGs já criadas onde faça sentido usar um parâmetro e uma variável.

CHECK-POINT



SENSORES



SENSORES

Os sensores no Airflow são operadores especiais que esperam até que uma determinada condição seja atendida:

FileSensor: Para verificar a existência de arquivos.

HttpSensor : Para verificar a disponibilidade de serviços HTTP / Webhook.

SqlSensor: Para aguardar determinados resultados em consultas SQL.

VARIÁVEIS

Exemplo de uso do FileSensor:

```
wait_for_file = FileSensor(  
    task_id='wait_for_file',  
    filepath='/path/to/your/file.txt', # Caminho do arquivo a ser monitorado  
    poke_interval=30, # Intervalo de verificação (em segundos)  
    timeout=600, # Tempo máximo de espera (em segundos)  
    dag=dag,  
)
```

VARIÁVEIS

Exemplo de uso do HTTPSensor:

```
from airflow.operators.http_sensor import HttpSensor

wait_for_webhook = HttpSensor(
    task_id='wait_for_webhook',
    http_conn_id='webhook_connection', # Conexão HTTP definida no Airflow
    endpoint='/webhook', # Endpoint do webhook
    method='GET', # Método HTTP a ser usado
    response_check=lambda response: True if response.status_code == 200 else False,
    dag=dag,
)
```



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

Crie uma dag que seja acionada somente se determinado arquivo for criado na pasta.

AULA 3



GERENCIAMENTO DE DEPENDÊNCIAS



GERENCIAMENTO DE DEPENDÊNCIAS

As dependências garantem que determinada tarefa irá ser executada somente após a finalização da outra.

GERENCIAMENTO DE DEPENDÊNCIAS

Tipos de dependência:

Dependências de fluxo de controle: Determinam a ordem em que as tarefas são executadas. Por exemplo, uma tarefa B pode depender da conclusão bem-sucedida da tarefa A antes de poder ser executada.

Controle de execução condicional: Às vezes, é necessário que uma tarefa seja executada somente se determinadas condições forem atendidas. O Airflow fornece operadores como BranchPythonOperator ou ShortCircuitOperator para lidar com esse tipo de dependência.

Dependências externas: Dependência de fatores externos, como a disponibilidade de recursos de computação, a conclusão de tarefas em sistemas externos, ou a disponibilidade de dados de fontes externas. O Airflow fornece mecanismos para lidar com essas dependências externas, como os sensores.

GERENCIAMENTO DE DEPENDÊNCIAS

Dependências de fluxo de controle

```
task_A = PythonOperator(  
    task_id='task_A',  
    python_callable=lambda: print("Executando a tarefa A"),  
    dag=dag,  
)  
  
task_B = PythonOperator(  
    task_id='task_B',  
    python_callable=lambda: print("Executando a tarefa B"),  
    dag=dag,  
)  
  
task_C = PythonOperator(  
    task_id='task_C',  
    python_callable=lambda: print("Executando a tarefa C"),  
    dag=dag,  
)  
  
# Definindo as dependências de fluxo de controle  
task_A >> task_B >> task_C
```


GERENCIAMENTO DE DEPENDÊNCIAS

Controle de execução condicional: Usamos o BranchPythonOperator e como código, uma função que retorna a string com o id da task.

```
decision_task = BranchPythonOperator(  
    task_id='decision_task',  
    python_callable=lambda: 'file_exists_task' if check_file_existence() else 'file_does_not_exist_task',  
    dag=dag,  
)
```

GERENCIAMENTO DE DEPENDÊNCIAS

Controle de execução condicional: E colocamos as tasks de opção entre colchetes (lista) na hora de definir as dependências.

```
check_file_task >> decision_task
```

```
decision_task >> [file_exists_task, file_does_not_exist_task]
```



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

Crie uma condição onde uma task é executada somente se determinado arquivo existir.

CHECK-POINT



GERENCIAMENTO DE ERROS E RETRIES

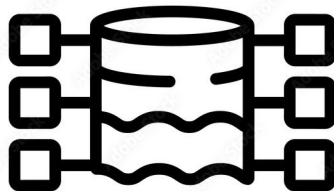
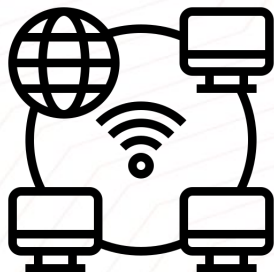
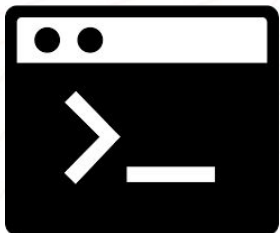


GERENCIAMENTO DE ERROS E RETRIES

Quando um problema acontece em uma task, como o airflow deve se comportar?

GERENCIAMENTO DE ERROS E RETRIES

Muitas vezes as tarefas dependem de **serviços externos** que podem estar fora do ar no momento e queremos que nesses casos, o airflow continue novamente.



Streaming



harve

GERENCIAMENTO DE ERROS E RETRIES

Tipos de Retry:

Retries Automáticos:

Se uma tarefa falhar durante a execução, o Airflow automaticamente a reiniciará de acordo com as configurações de retry definidas.

Política de Retries Exponenciais:

Nesse caso, o intervalo entre as tentativas de retry aumenta exponencialmente a cada retry. Isso ajuda a evitar a sobrecarga de recursos quando há problemas temporários no sistema.

GERENCIAMENTO DE ERROS E RETRIES

Tipos de Retry:

Políticas de Retry Personalizadas:

Além das políticas de retry padrão, você pode implementar políticas de retry personalizadas para lidar com casos específicos de erro. Isso pode incluir a implementação de lógica de retry personalizada com base em critérios específicos, como o tipo de erro ou o estado do sistema.

GERENCIAMENTO DE ERROS E RETRIES

Retries Automáticos: Configurado na DAG.

```
default_args = {  
    'owner': 'airflow',  
    'depends_on_past': False,  
    'start_date': datetime(2024, 5, 3),  
    'email_on_failure': False, # Desativando e-mail em caso de falha para simplificar  
    'email_on_retry': False, # Desativando e-mail em caso de retry para simplificar  
    'retries': 3, # Número máximo de retries  
    'retry_delay': timedelta(minutes=5) # Intervalo entre cada retry  
}
```

GERENCIAMENTO DE ERROS E RETRIES

Retries Automáticos: Configurado na task.

```
minha_tarefa = PythonOperator(  
    task_id='minha_tarefa',  
    python_callable=minha_tarefa,  
    retries=3, # Número máximo de retries  
    retry_delay=timedelta(minutes=5), # Intervalo entre cada retry (5 minutos)  
    dag=dag,  
)
```

GERENCIAMENTO DE ERROS E RETRIES

Política de Retries Exponenciais: O intervalo entre cada retry aumentará exponencialmente a cada tentativa de retry.

```
default_args = {  
    'owner': 'airflow',  
    'depends_on_past': False,  
    'start_date': datetime(2024, 5, 3),  
    'email_on_failure': False, # Desativando e-mail em caso de falha para simpl  
    'email_on_retry': False, # Desativando e-mail em caso de retry para simplif  
    'retries': 3, # Número máximo de retries  
    'retry_delay': timedelta(minutes=5), # Intervalo inicial entre cada retry  
    'retry_exponential_backoff': True # Habilitando retries exponenciais  
}
```

GERENCIAMENTO DE ERROS E RETRIES

Políticas de Retry Personalizadas: Definindo a função para calcular o valor do retry.

```
def calcular_intervalo_retry(ti):  
    # Obtendo o número de tentativas  
    tentativa = ti.try_number  
    # Definindo um intervalo base de retry (em minutos)  
    intervalo_base = 5  
    # Aumentando o intervalo de retry com base no número de tentativas  
    intervalo_personalizado = intervalo_base * (2 ** tentativa)  
    return timedelta(minutes=intervalo_personalizado)
```

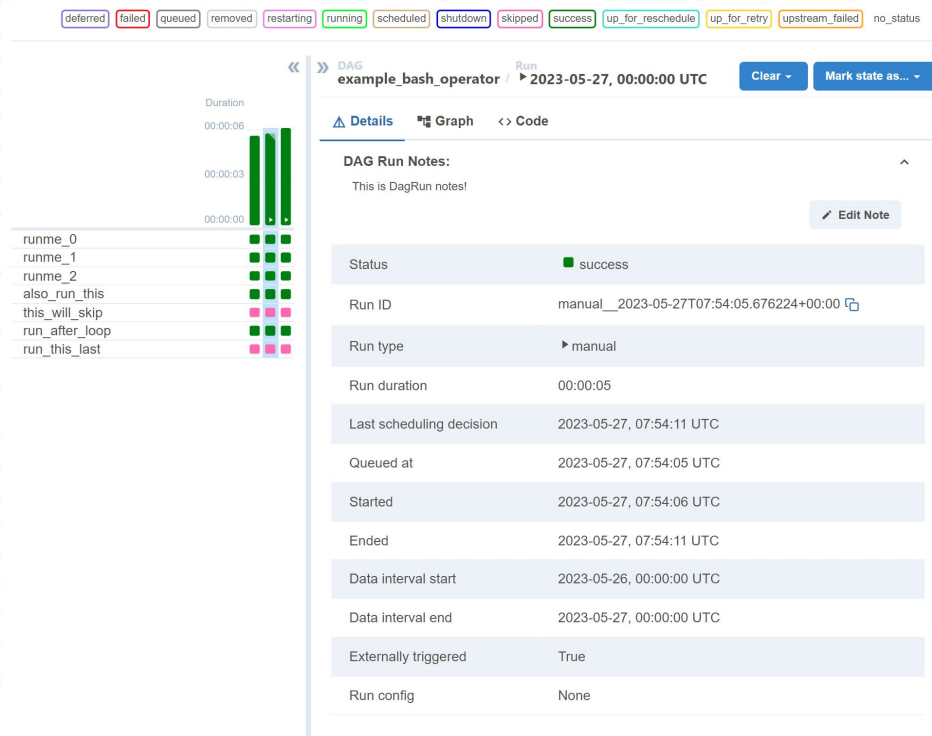

GERENCIAMENTO DE ERROS E RETRIES

Políticas de Retry Personalizadas: Definindo a função para calcular o valor do retry.

```
minha_tarefa = PythonOperator(  
    task_id='minha_tarefa',  
    python_callable=minha_tarefa,  
    retries=3, # Número máximo de retries  
    retry_delay=calcular_intervalo_retry, # Função  
    dag=dag,  
)
```


GERENCIAMENTO DE ERROS E RETRIES

Monitoramento de task



GERENCIAMENTO DE ERROS E RETRIES

Gerenciamento de erro: Os prints são impressos no log.

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime

# Definindo a função que será executada pela tarefa
def minha_tarefa():
    try:
        # Coloque aqui o código que pode gerar uma exceção
        resultado = 10 / 0 # Isso irá gerar uma exceção ZeroDivisionError
        print("Resultado:", resultado)
    except ZeroDivisionError as e:
        # Trate a exceção aqui, por exemplo, registrando o erro
        print("Erro:", str(e))

# Definindo as configurações da DAG
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
```



DESCOBERTA GUIADA

harve

DESCOBERTA GUIADA

Vamos criar um retry exponencial para uma task e registrar o problema no log.

CHECK-POINT





Conexão

—  com o  —

Mercado Digital

Versão base: {{versaobase}}

Gerado por: {{emailgerador}}

Data de geração: {{data}}