

Relatório 11 - Pipelines de Dados I - Airflow

Guilherme Loan Schneider

Descrição da atividade

O curso do card atual tem como objetivo dar os passos iniciais na utilização do Apache Airflow, bem como exemplificar a utilização de DAGs (Directed Acyclic Graph) e elucidar várias questões importantes para o bom entendimento.

1.1 The basics of Apache Airflow

1.1.1 O que é o Apache Airflow?

Ele basicamente concentra vários processos de dados (Verificação de APIs, download de dados) utilizando a linguagem de programação Python, permitindo automatização, monitoramento, agendamento desses fluxos de trabalho (chamados de DAGs).

1.1.2 O que podemos fazer com o Apache Airflow?

- Permite criar e alterar fluxos de dados, sejam simples ou complexos;
- Permite gerenciar dependências entre tarefas;
- Permite repetir tarefas que falharam;
- Fornece registro extensivo;
- Permite acompanhar trabalhos por meio de sua interface gráfica;
- Disponibiliza logs para realização de auditoria de erros;
- Possui sistemas de alertas integrados com e-mails, Slacks, entre outros.

1.1.3 Onde ele é utilizado?

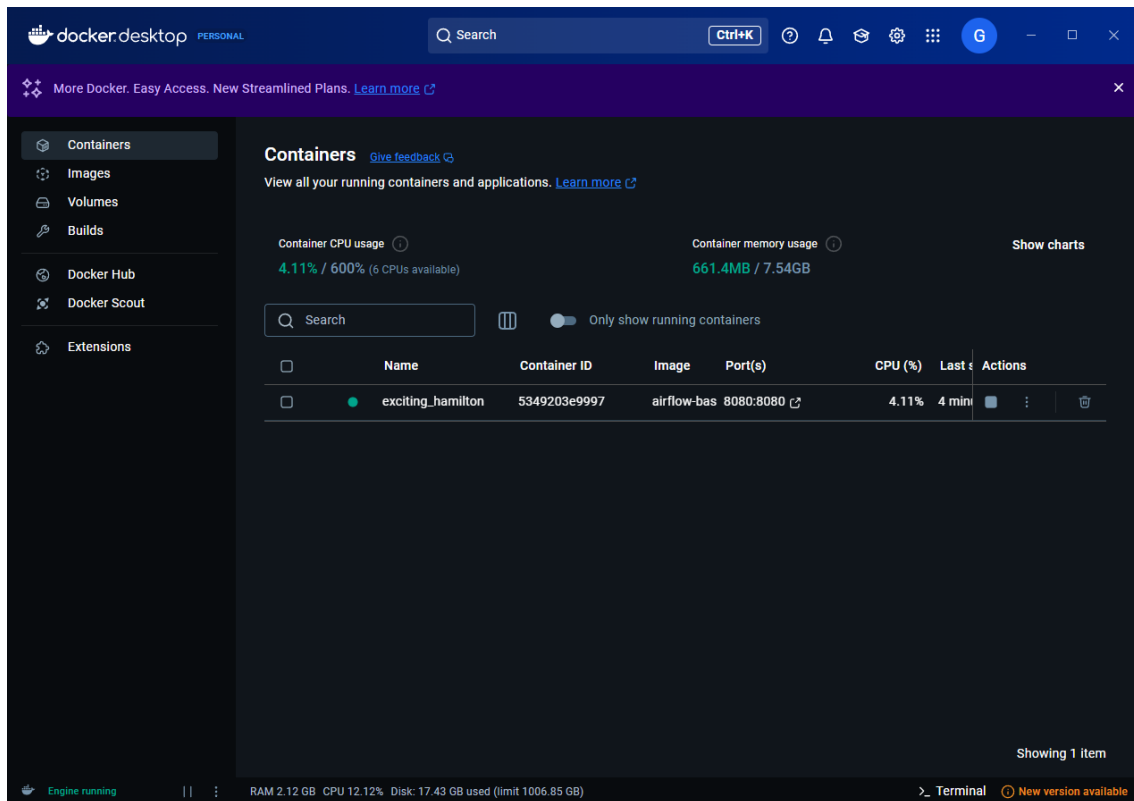
- a) ETL/ELT Pipelines – Extração, transformação e carga de dados entre diferentes fontes.
- b) Treinamento de Modelos de Machine Learning – Agendamento e execução de fluxos de ML.
- c) Automação de Processos de Dados – Como atualizações de relatórios ou sincronização de bases.
- d) Integração com Big Data – Execução de tarefas distribuídas em Spark, Hadoop, etc.

1.1.4 Utilizando o Airflow

Essa seção faz uma pequena introdução sobre o Airflow, mostrando o que é, como funciona, além de fazer a instalação inicial com o Docker Desktop e utilizando o Terminal. Além disso, explica também os principais pontos da interface do Airflow.

Após ser feita a instalação do Docker Desktop e os comandos básicos passados nas aulas, ao abrir o Docker Desktop ou utilizar o comando “docker ps” no terminal, o usuário poderá ver os containers que estão rodando.

Visualização do Docker Desktop:



É possível verificar que o container “Exciting_hamilton” está em execução. Para acessar a UI do Airflow, basta entrar no link <http://localhost:8080/admin/>. A visualização da interface está mais abaixo.

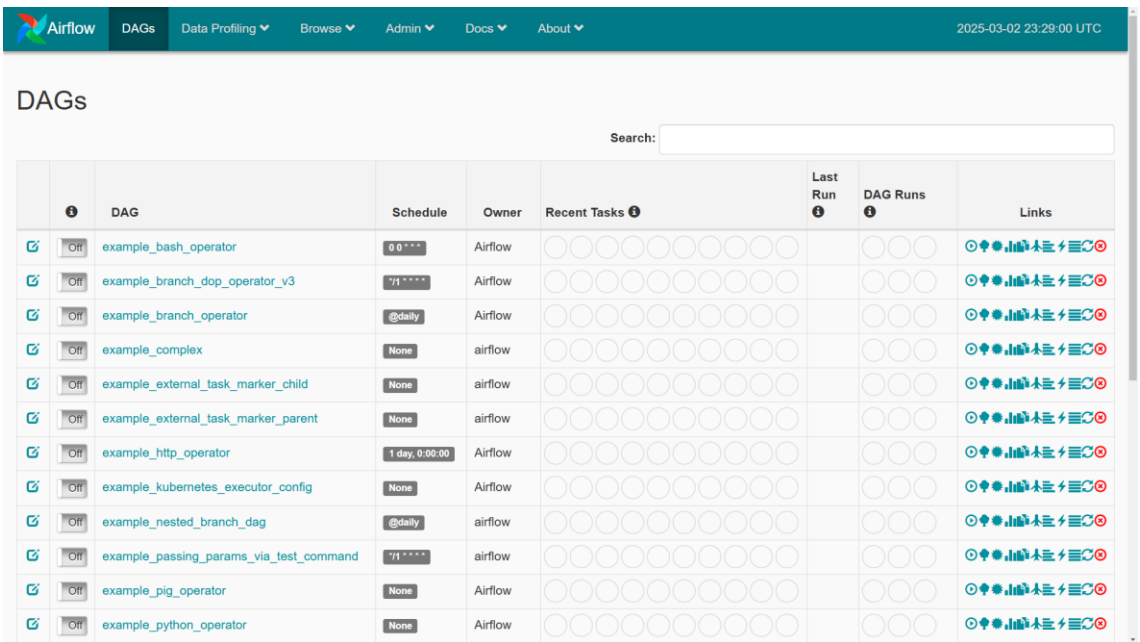
Visualização via terminal:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

Experimente a nova plataforma cruzada PowerShell https://aka.ms/pscore6

PS G:\Meu Drive\UTFPR\LAMIA\Aula 11\arquivos aula\airflow-materials\airflow-section-3> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
5349203e9997   airflow-basic  "/entrypoint.sh"        6 days ago    Up 3 minutes  0.0.0.0:8080
->8080/tcp    exciting_hamilton
PS G:\Meu Drive\UTFPR\LAMIA\Aula 11\arquivos aula\airflow-materials\airflow-section-3> |
```

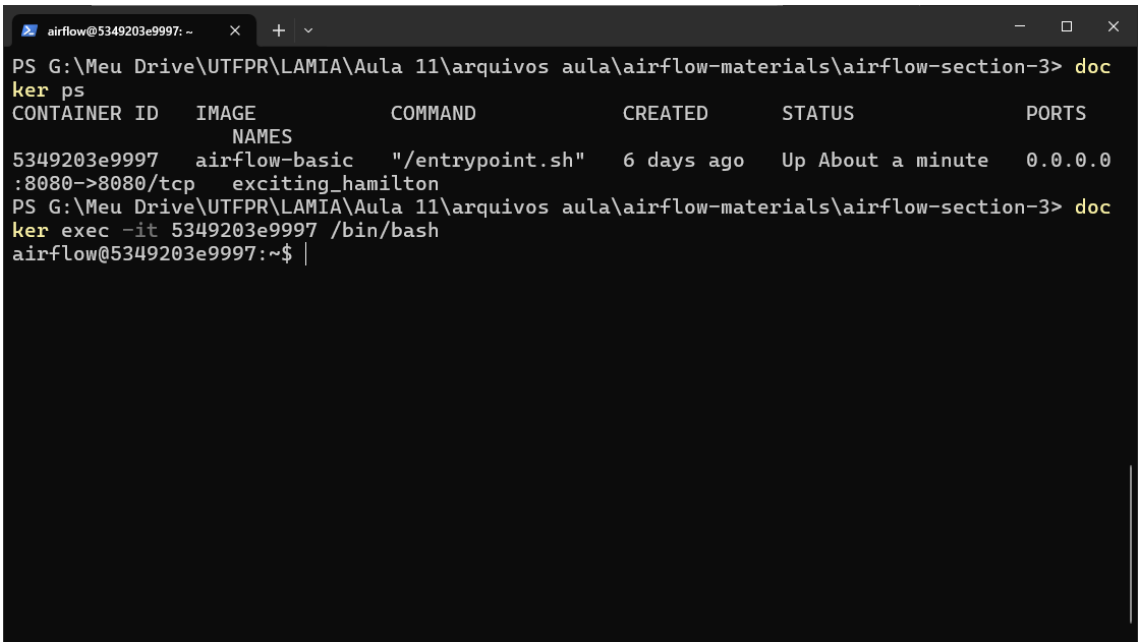
Visualização da UI do Airflow a partir do localhost:8080 (é possível alterar a porta que o airflow irá executar).



The screenshot shows the Apache Airflow web interface at localhost:8080. The top navigation bar includes links for DAGs, Data Profiling, Browse, Admin, Docs, and About. The main heading is "DAGs". Below it is a search bar. The table lists various DAGs with columns for DAG name, Schedule, Owner, Recent Tasks, Last Run, DAG Runs, and Links. The DAGs listed include example_bash_operator, example_branch_dop_operator_v3, example_branch_operator, example_complex, example_external_task_marker_child, example_external_task_marker_parent, example_http_operator, example_kubernetes_executor_config, example_nested_branch_dag, example_passing_params_via_test_command, example_pig_operator, and example_python_operator.

DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
example_bash_operator	0 0 * * *	Airflow				
example_branch_dop_operator_v3	* * * * *	Airflow				
example_branch_operator	@daily	Airflow				
example_complex	None	airflow				
example_external_task_marker_child	None	airflow				
example_external_task_marker_parent	None	airflow				
example_http_operator	1 day, 0:00:00	Airflow				
example_kubernetes_executor_config	None	Airflow				
example_nested_branch_dag	@daily	airflow				
example_passing_params_via_test_command	* * * * *	airflow				
example_pig_operator	None	Airflow				
example_python_operator	None	Airflow				

É importante salientar também o comando para iniciar uma sessão shell no container executado. “docker exec -it container_id /bin/bash”. É imprescindível para fazer as instalações básicas do Airflow.

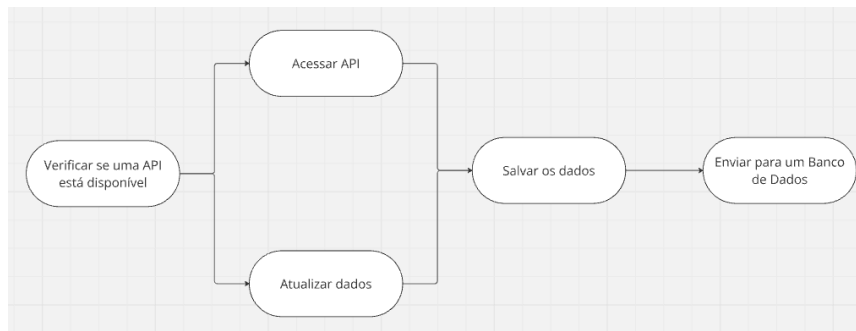


```
PS G:\Meu Drive\UTFPR\LAMIA\Aula 11\arquivos aula\airflow-materials\airflow-section-3> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
5349203e9997   airflow-basic  "/entrypoint.sh"        6 days ago    Up About a minute    0.0.0.0:8080->8080/tcp
exciting_hamilton
PS G:\Meu Drive\UTFPR\LAMIA\Aula 11\arquivos aula\airflow-materials\airflow-section-3> docker exec -it 5349203e9997 /bin/bash
airflow@5349203e9997:~$
```

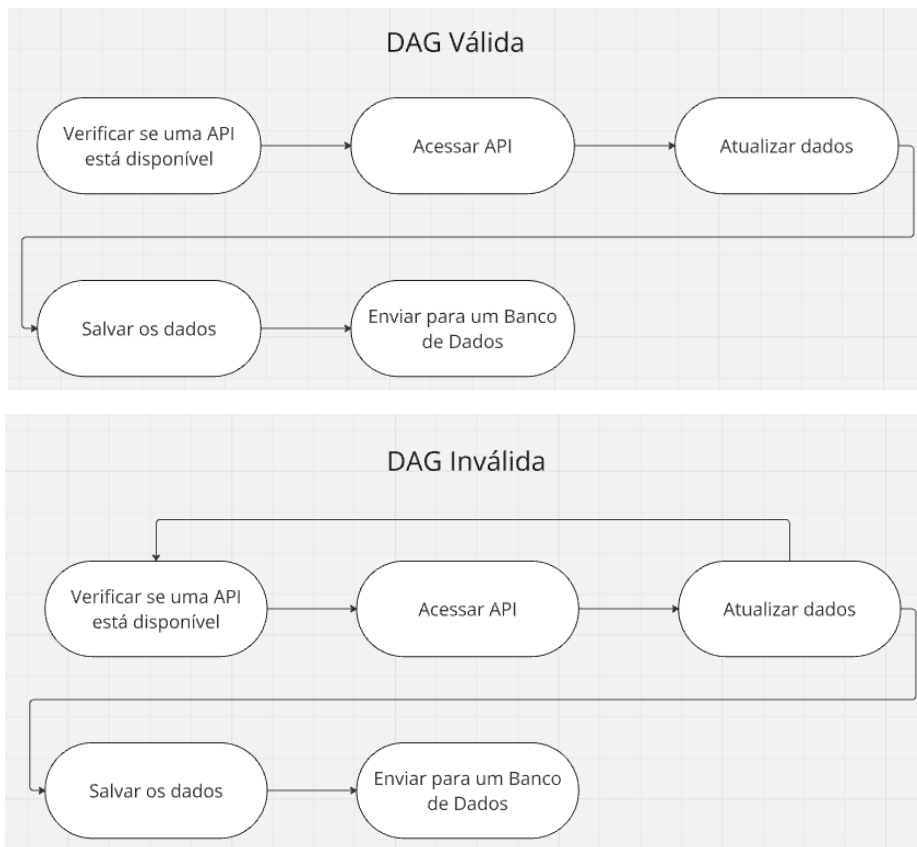
1.2 The Forex Data Pipeline

Esse modulo traz o primeiro conjunto de DAGs que serão implementadas, chamado de Forex Data Pipeline, onde em cada aula é abordado uma DAG individual.

1.2.1 O que é uma DAG?



Uma DAG é um conjunto de tasks que deverão ser feitas, em uma determinada ordem (chamado de dependências) e não deverão possuir loops. Uma única task (representada na figura abaixo) simboliza uma função que deverá ser feita, onde a Task 1 deverá verificar se é possível acessar um determinado site, a Task 2 acessa informações, e a Task 4 baixa essas informações e salva. A Task 3 poderia ser ao invés de acessar todas as informações novamente, apenas atualizar as que já existem.



É importante destacar também que as DAGs possuem campos default na hora de cria-las, como o `dag_id` que é o nome da sua DAG, `description` que pode ser utilizado para descrever como funciona a sua DAG, `start_date` que define a data que sua DAG deve começar, `schedule_interval` que indicará a frequência de execução (a cada hora, dia, semana), `default_args` que são os argumentos padrão de operadores e por fim o `catchup`, que indicará se deverá ser executado operações “atrasadas”.

Nas imagens abaixo foram implementadas DAGs com argumentos padrões citados anteriormente, bem como outros que não foram comentados até então.

```
with DAG(dag_id="forex_data_pipeline_final",
         schedule_interval="@daily",
         default_args=default_args,
         catchup=False) as dag:
```

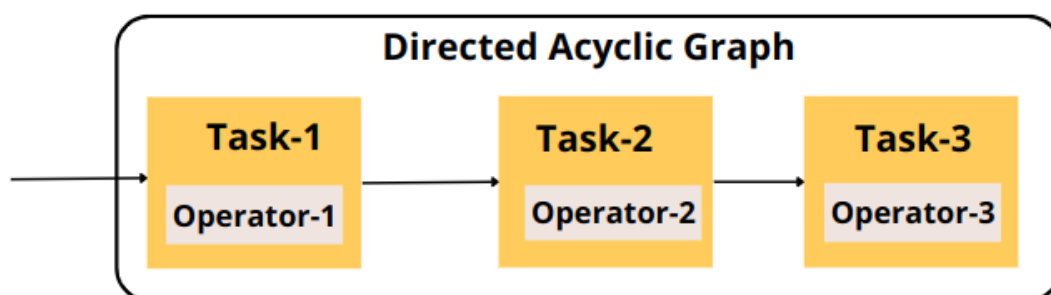
```
is_forex_rates_available = HttpSensor(
    task_id="is_forex_rates_available",
    method="GET",
    http_conn_id="forex_api",
    endpoint="latest",
    response_check=lambda response: "rates" in response.text,
    poke_interval=5,
    timeout=20
)
```

Acessando o arquivo no GitHub “forex_data_pipeline_pratica.py”, é possível verificar a ordem em que as Tasks são executadas, sendo indicada a ordem pelo operador de fluxo “>>”.

```
is_forex_rates_available >> is_forex_currencies_file_available >> downloading_rates >> saving_rates
saving_rates >> sending_email_notification >> sending_slack_notification
```

1.2.2 O que são operadores?

Um operador é a parte lógica de como os dados serão processados em cada Task de uma DAG. Abaixo é mostrado um exemplo implementado em aula de um operador. Cada operador representa uma unidade de trabalho, como rodar um script Python, executar uma consulta SQL ou chamar uma API.



```
with DAG(dag_id='start_and_schedule_dag', schedule_interval=timedelta(hours=1), default_args=default_args) as dag:

    # Task 1
    dummy_task_1 = DummyOperator(task_id='dummy_task_1')

    # Task 2
    dummy_task_2 = DummyOperator(task_id='dummy_task_2')

    dummy_task_1 >> dummy_task_2
```

1.2.3 Utilizando Operadores e Sensores

Ao nos depararmos com a utilização dos operadores, é importante também compreendermos o que podemos acessar/executar/fazer com esse tipo de funcionalidade. No curso utilizamos tipos diferentes, tanto de operadores, quanto de sensores.

Os sensores (Sensors) são utilizados para quando deve-se esperar por um evento acontecer, como verificar se um arquivo específico esteja disponível em uma determinada pasta (FileSensor). Abaixo estão os sensors utilizados:

HttpSensor	Verifica se uma URL ou API está respondendo antes de continuar o fluxo.
FileSensor	Aguarda até que um arquivo específico esteja disponível em um diretório antes de prosseguir.

Em seguida, estão os principais operadores utilizados:

PythonOperator	Executa uma função Python dentro do fluxo de trabalho.
BashOperator	Roda comandos de terminal (bash), como scripts shell ou comandos do sistema.
HiveOperator	Executa consultas SQL no Apache Hive (usado para armazenar e consultar grandes volumes de dados).
SparkSubmitOperator	Envia um job para ser executado no Apache Spark, útil para processar grandes volumes de dados.
EmailOperator	Envia emails como parte do fluxo de trabalho, útil para notificações ou relatórios.
SlackAPIPostOperator	Envia mensagens para canais do Slack, útil para alertas ou acompanhamento de tarefas.

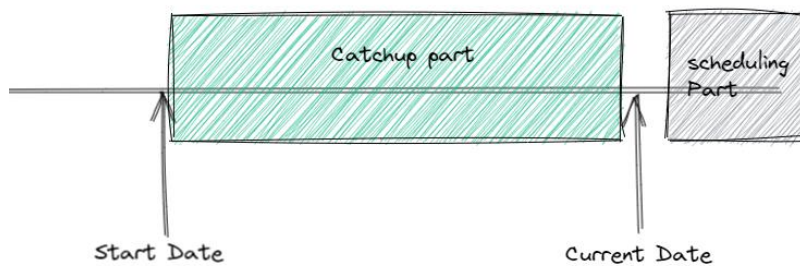
1.3 Mastering your DAGs

Essa seção tem como objetivo mostrar o funcionamento das DAGs, juntamente com parâmetros que podemos alterar nelas.

1.3.1 DagRun, Backfill e Catchup

- DagRun – Representa uma instância de uma DAG, contendo as suas tasks a serem executadas;
- Catchup – Esse processo consiste na execução de tasks anteriores que não foram executadas por algum motivo na DAG. (Exemplo: Uma dag foi criada no dia 01/03/2025 com schedule_date definido para diário, e ela foi ativada apenas

no dia 14/03/2025, o parâmetro Catchup quando em True faz com que ela seja executada desde o dia 01/03, até o dia 14/03).



- c) Backfill – Processo de executar um dag ou tarefa específica em um dag nos últimos dias. Por exemplo, se um dag estiver em execução desde o início de um mês e uma nova tarefa tiver sido adicionada a ele, e essa tarefa recém-adicionada precisar ser executada nos últimos dias, preenchendo as DAGs executadas anteriormente.

1.3.2 Pontos importantes ao utilizar datetime

- a) Sempre especificar o tempo e o fuso horário, isso fará com que o python receba um objeto “aware” (esse tipo de objeto leva em conta que o fuso horário está definido), evitando o tipo “naive”, que não leva em consideração o fuso horário;
- b) Ao criar um datetime sem o fuso horário definido, não significa que ele estará no UTC;
- c) Uma solução é importar o timezone do Airflow para criar esses objetos aware.

1.3.3 Tornando as tasks dependentes

A configuração `depends_on_past` é definida a nível da task e determina que, se a instância anterior da mesma task falhar, a próxima não será executada. Como consequência, a task atual não terá um status definido. No entanto, a primeira instância da task, correspondente ao `start_date`, poderá ser executada normalmente.

Já a configuração `wait_for_downstream`, também definida a nível da task, faz com que uma instância de uma determinada task espere a finalização bem-sucedida ("successful") de todas as tasks subsequentes antes de ser executada. Isso garante que a execução ocorra de maneira ordenada e evita possíveis inconsistências no fluxo de trabalho.

```
# Task 1
# wait_for_downstream=True fará com que a task 2 só seja executada se a task 1 for bem sucedida
bash_task_1 = BashOperator(task_id='bash_task_1', bash_command="echo 'first task'", wait_for_downstream=True)

# Task 2
# depends_on_past=True fará com que a task 2 só seja executada se a task 1 for bem sucedida
# python_task_2 = PythonOperator(task_id='python_task_2', python_callable=second_task, depends_on_past=True)
python_task_2 = PythonOperator(task_id='python_task_2', python_callable=second_task)

# Task 3
python_task_3 = PythonOperator(task_id='python_task_3', python_callable=third_task)

bash_task_1 >> python_task_2 >> python_task_3
```

1.3.4 Como estruturar os diretórios DAG

Deve-se haver essa preocupação na organização dos arquivos por conta de existir muitas DAGs, tornando difícil seu manuseio, e DAGs utilizando muitos arquivos externos.

Utilizando arquivos .zip:

1. Deve-se criar um arquivo zip com todas as DAGs e seus arquivos extras;
2. As DAGs devem estar no root do arquivo zip;
3. Em seguida, o Airflow irá escanear e carregar os arquivos DAGs.

Utilizando o DagBag:

```
import os
from airflow.models import DagBag
dags_dirs = [
    '/usr/local/airflow/project_a',
    '/usr/local/airflow/project_b'
]

for dir in dags_dirs:
    dag_bag = DagBag(os.path.expanduser(dir))

    if dag_bag:
        for dag_id, dag in dag_bag.dags.items():
            globals()[dag_id] = dag
```

Uma DagBag é uma coleção de DAGs, analisados de uma árvore de pastas e tem configurações de alto nível. (é como um "coletor" de todas as DAGs registradas no Airflow).

1. A dagbag torna mais fácil de utilizar diferentes pastas de desenvolvimento (dev/staging/prod);
2. Um sistema consegue rodar diferentes sets independentes de configuração;
3. Permite adicionar novas pastas a partir de um script na pasta padrão das DAGs.

1.3.5 Erros nas DAGs

Ao configurar DAGs no Apache Airflow, é fundamental definir mecanismos para gerenciar erros, garantindo que falhas sejam monitoradas e tratadas corretamente. Esses mecanismos podem ser configurados tanto a nível da DAG quanto das tasks individuais.

A nível da DAG:

- `dagrun_timeout`: Define um tempo limite para a execução da DAG. Se o tempo for excedido, a DAG falha.
- `sla_miss_callback`: Especifica uma função de callback a ser executada quando uma SLA (Service Level Agreement) não for cumprida.
- `on_failure_callback`: Permite definir uma função que será executada caso a DAG falhe. Pode ser usada para notificações ou logs.
- `on_success_callback`: Define uma função que será acionada quando a DAG for concluída com sucesso.

A nível da task:

- `email`: Lista de emails para os quais notificações devem ser enviadas sobre a execução da task.

- `email_on_failure`: Se True, um email será enviado quando a task falhar.
- `email_on_retry`: Se True, um email será enviado quando a task for reexecutada após falha.
- `retries`: Número de tentativas de reexecução da task em caso de falha.
- `retry_delay`: Tempo de espera entre tentativas de reexecução. Aceita valores do tipo `timedelta`.
- `retry_exponential_backoff`: Se True, aplica um tempo de espera crescente entre tentativas de reexecução.
- `max_retry_delay`: Define o tempo máximo de espera entre reexecuções quando o `retry_exponential_backoff` está ativado.
- `execution_timeout`: Tempo máximo permitido para a execução da task antes que ela falhe automaticamente.
- `on_failure_callback`: Função de callback executada quando a task falha, útil para logs, notificações ou ações corretivas.
- `on_success_callback`: Função de callback chamada quando a task é concluída com sucesso.
- `on_retry_callback`: Função de callback acionada sempre que a task for reexecutada após uma falha.

Conclusões

A partir das aulas, foi possível compreender os conceitos básicos até a implementação e gerenciamento avançado de DAGs. Além disso, a estrutura e funcionamento das DAGs, a utilização de operadores e sensores, além de práticas essenciais para o agendamento e monitoramento eficiente dos fluxos de trabalho. Por fim, destaca-se boas práticas no uso do `datetime`, organização de diretórios e tratamento de erros.

Referencias

[Apache Airflow: The Hands-On Guide \(Seção 1 à 4\)](#)