Nome: Fernando Buligon Antunes **Data:** 06/12/2024

Apache Airflow: The Hands-On Guide

1. Distributing Apache Airflow

a. Introduction

Rápida introdução sobre a seção, foi feito um resuminho sobre o que será visto.

b. Sequential Executor with PostgreSQL

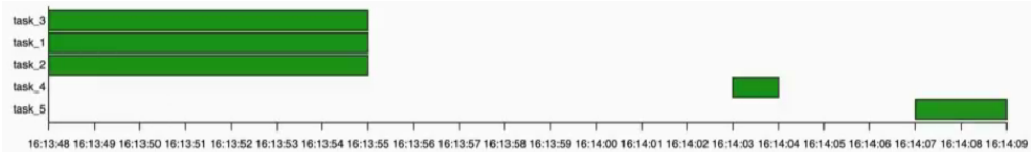
Primeiro ele explica o que é o SQLite, um banco de dados do tipo relacional que implementa a maioria das funcionalidades do SQL padrão e não tem que configurar muita coisa para rodar. Após isso, ele faz um lembrete sobre o que é um executor e um executor sequencial, o executor sequencial como o próprio nome sugere ele executa as tarefas sequencialmente, fazendo um por vez e respeitando a fila.

c. Local Executor with PostgreSQL

O executor local é o mais recomendado quando mover para produção com o airflow por dois motivos principais, ele é mais fácil de manusear que os outros então qualquer um consegue gerenciar sem muitos problemas e também ele é capaz de rodar múltiplas tarefas simultaneamente.

d. [Practice] Executing tasks in parallel with the Local Executor

Foi colocado em prática a teoria sobre o Local Executor, no AirflowUI foi possível observar o executor rodando três tarefas simultaneamente.



e. [Practice] Ad Hoc Queries with the metadata database

Nesse tópico foi visto sobre o banco de dados de metadata, no AirflowUI, foi criada uma nova conexão, depois em Ad Hoc Query foi selecionada essa conexão que acabou de ser criada e ali pôde ser rodado um comando SQL, o exemplo usado foi “select * from pg_catalog.pg_tables where schemaname != ‘pg_catalog’ and schemaname != ‘information_schema’ and tableowner = ‘airflow’;”, depois foram feitos mais alguns exemplos.

f. Scale out Apache Airflow with Celery Executors and Docker

Na maioria das ocasiões o Local Executor é o suficiente, mas caso o número de DAGs seja muito alto pode haver algum problema, e é aí que o Celery Executor entra, ele permite rodar tarefas de forma distribuída e assíncrona, então é ideal para aumentar a performance em projetos com muitas DAGs, acelera bastante o processo, os responsáveis por processar as tarefas são chamados de workers.

g. [Practice] Set up the Airflow cluster with Celery Executors and Docker

Nesse tópico foi feito um tour pelo arquivo “docker-compose-CeleryExecutor.yml” e depois foi rodado, após isso, através do navegador foi acessado o localhost só que com uma porta diferente do AirflowUI, dessa vez foi acessada à página da Flower através da porta 5555 e nela foi possível ver o worker da tarefa que acabou de ser criada.

h. [Practice] Distributing your tasks with the Celery Executor

Tendo um worker, ainda falta passar quais tarefas ele deverá processar, neste tópico foi configurada a distribuição das tarefas, basicamente acontece o seguinte: o scheduler envia as tarefas para a fila, e o worker buscas quais tarefas ele precisa

processar entre as presentes ali, deixando passar as que não foram designadas a ele.

i. [Practice] Adding new worker nodes with the Celery Executor

Nos tópicos anteriores foi criado apenas um worker, só que apenas um worker não dá conta de executar tudo sozinho de maneira otimizada, para isso, foram criados mais três workers para que as tarefas possam ser divididas em mais partes, tornando o processo mais rápido

j. [Practice] Sending tasks to a specific worker with Queues

Anteriormente foi ensinado a como dividir as tarefas de uma única fila para os workers processarem, aqui foi ensinado a como criar múltiplas filas para que fique mais organizado, tornando mais fácil de gerenciar e até mesmo mais otimizado, para demonstrar em quais casos isso seria útil, o autor usa uma situação hipotética em que há múltiplas tasks que possuem diferentes requisitos, uma precisa de mais CPU e a outra precisa de Spark, então ao invés de juntar todas as tarefas em uma fila, são criadas duas filas em que apenas workers adequados para o tipo das tarefas estarão designados para que não ocorra de um worker com baixo poder de CPU disponível se tornar responsável por uma tarefa que exige alto poder.

k. [Practice] Pools and priority weights Limiting parallelism - prioritizing tasks

Primeiramente o autor apresenta o conceito de pool, que consiste em uma maneira de limitar o número de instâncias simultâneas de um tipo específico de tarefa com o objetivo de evitar o sobrecarregamento. Após isso foi criada uma nova conexão no AirflowUI e para limitar o número de tarefas que chegam a API simultâneamente foi só ir em Admin -> Pools -> Create e colocar o nome da pool e limitar os slots no valor requisitado, nesse caso, foi um. Com a pool criada, foi necessário ir no código e adicionar o parâmetro pool e colocar o nome da pool criada, depois só rodar novamente e estará funcionando devidamente.

l. Kubernetes Reminder

Antes de dar início ao executor kubernetes o autor faz uma revisão sobre o que seria um kubernetes. É um projeto de código aberto fundado pela Google em 2014 que é capaz de automatizar tarefas de administradores de sistemas, da perspectiva de um desenvolvedor, ele reduz o tempo e o esforço necessários para implementar aplicações. Também possui alguns defeitos que acabam incomodando bastante, além de ser complicado de configurar também é difícil de manter.

m. Scaling Airflow with Kubernetes Executors

Um executor Kubernete roda suas tarefas em Kubernetes, cada tarefa roda em um novo “pod”, e isso é uma característica interessante pois assim que as tarefas se tornam totalmente isoladas elas podem crashar que somente o pod relacionado a tarefa será impactado e vai ser automaticamente restaurado pelo kubernetes.

n. [Practice] Set up a 3 nodes Kubernetes Cluster with Vagrant and Rancher

Nesse tópico foi colocado em prática a teoria vista acima. Com o VirtualBox já instalado, também foi necessário instalar o Vagrant que permite subir ambientes de desenvolvimento baseado em vms usando um arquivo de configuração nomeado background file, depois foi necessário adicionar mais dois plugins, com tudo preparado, foi possível iniciar o kubernetes cluster digitando “vagrant up”, para verificar se está rodando como deveria existe o comando “vagrant status”, para acessar o “mestre” basta digitar “vagrant ssh master”, depois foi instalado um rancher com o objetivo de gerenciar o kubernetes cluster, esse rancher simplifica o que tomaria muito tempo, para isso, ele nos provê uma interface, o comando para criar ele é meio comprido “docker run -d --restart=unless-stopped -p 80:80 -p 443:443 --name rancher rancher/rancher:v2.3.2”, depois é possível acessar a interface de usuário através do navegador.

o. [Practice] Installing Airflow with Rancher and the Kubernetes Executor

No tópico anterior foi configurado o kubernetes cluster, o que ficou faltando fazer foi instalar o Airflow nele, é possível fazer isso através da interface de usuário do rancher, basta ir em Airflow > Global > Apps > Manage Catalogs > Add Catalog, em Add Catalog é necessário colocar um nome que nesse caso foi “airflow-chart” e também é preciso informar um link, que foi disponibilizado pelo apresentador, o link leva a um diretório dele “https://github.com/marclamberti/airflow-helm-chart.git”, depois só clicar em Create.

p. [Practice] Running your DAGs with the Kubernetes Executor

Primeiramente é necessário checar se tudo feito até agora está ativo e rodando, depois se conectar através do cmd mesmo com o comando “vagrant ssh master”, estando conectado, é preciso uma linha de comando que é possível encontrar na interface de usuário do rancher em Apps > airflow-p-xccpd > Notes, arrastando para baixo até a última parte, a linha é export POD_NAME=$(kubectl get pods --namespace airflow-p-xccpd -l “component=web,app=airflow-k8s” -o jsonpath=”{.items[0].metadata.name}”), depois de encontrar esse comando, basta copiar e colar no cmd, essa linha nos dá acesso ao pod responsável por rodar o servidor web. Abaixo dessa linha, também a outra que precisa ser rodada também kubectl port-forward --namespace airflow-p-xccpd $POD_NAME 8080:8080, com esse comando, é feita uma conexão direta entre a máquina local e o pod airflow-p-xccpd, permitindo acessar o serviço no pod através do “http://localhost:8080”, acessando o localhost, vai ser direcionado para o AirflowUI, e de lá é possível rodar suas DAGs do airflow com o executor do kubernetes.

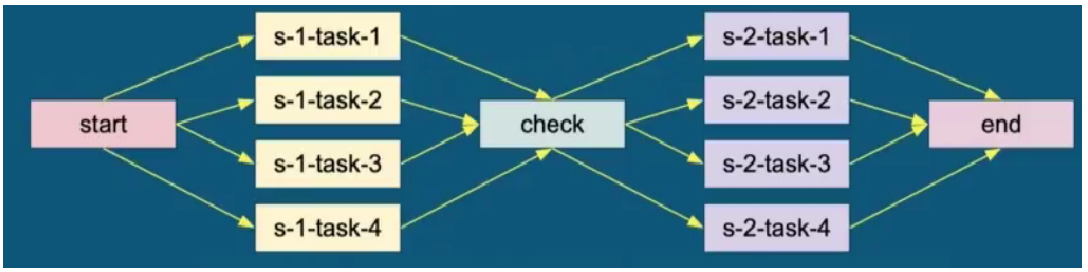
2. Improving your DAGs with advanced concepts

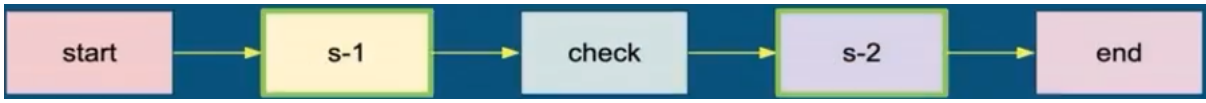
a. Introduction

Rápida introdução sobre a seção, foi feito um resuminho sobre o que será visto.

b. Minimizing Repetitive Patterns With SubDAGs

Quando for começar a construir DAGs mais complexas provavelmente vai acabar se deparando com várias tarefas rodando em paralelo com lógicas semelhantes, para deixar a DAG mais “limpa” uma das das possíveis soluções seria agrupar essas tarefas semelhantes, e para fazer isso, existem as SubDAGs, como possível ver nas imagens abaixo, é meramente visual, as tarefas continuam rodando separadamentes mas fica mais organizado, é possível diferenciar pela bordinha da tarefa, como na imagem dois, tem como observar isso no AirflowUI.

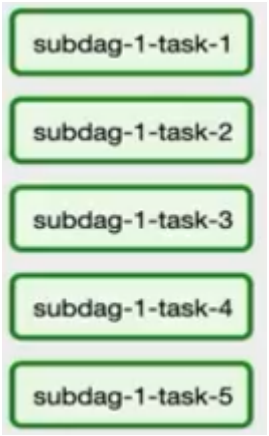




c. [Practice] Grouping your tasks with SubDAGs and Deadlocks

Para testar a teoria aprendida no tópico acima, foram criadas cinco tarefas vazias usando um for e também foi criada uma DAG, no AirflowUI é possível rodar essa DAG e depois clicando nela e depois indo em Graph View é possível ver as tarefas:

Clicando em “subdag-1” ou “subdag-2” e depois em “Zoom into Sub Dag” é possível observar todas as tarefas que foram colocadas dentro dela, a imagem abaixo representa o caso da subdag-1.



d. Making different paths in your DAGs with Branching

Desde o início do curso as DAGs foram vistas como uma série de tarefas linkadas juntas com um objetivo. Branching permite com que sua DAG escolha caminhos diferentes dependendo do resultado de uma tarefa, então basicamente uma tarefa é rodada, se a saída favorecer certa tarefa então a outra não vai ser rodada, diferentemente de antes em que todas eram rodadas, a função que permite isso é a BranchPythonOperator.

e. [Practice] Making Your First Conditional Task Using Branching

Foi feito um tour pelo código branch_dag.py, que basicamente cria uma DAG, um dicionário com URLs de três APIs de geolocalização, uma função que checa qual API mais se adequa, depois com o BranchPythonOperator ele define qual tarefa vai ser executada.

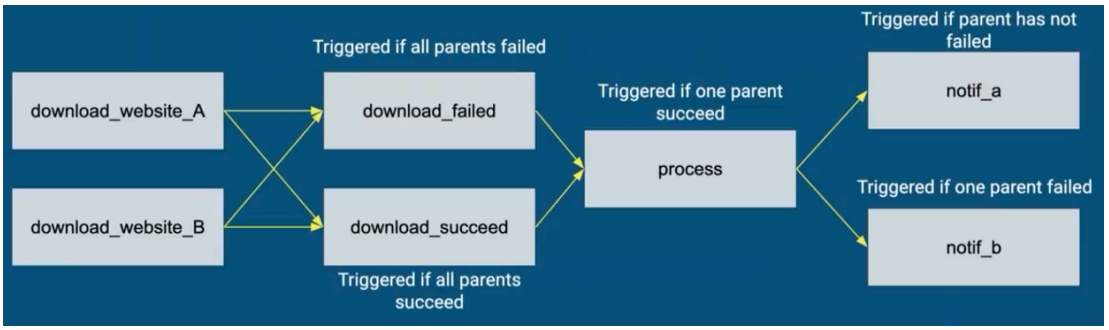
f. Trigger rules for your tasks

Quando as DAGs são criadas é necessário configurar dependências entre as tarefas que você deseja executar, por padrão, cada tarefa é executada depois de seus parâmetros serem concluídos. Existem diferentes opções de regras, mas em sua maioria se tratam de condicionais, por exemplo, se as tarefas 1, 2 e 3 rodarem com êxito, então a tarefa 4 pode ser executada, outro exemplo, dependendo do resultado da tarefa 4 é chamada

a tarefa 5 ou 6, mais um exemplo, se as tarefas 1 e 2 rodarem mas a 3 não, a tarefa 4 ainda pode ser executada.

g. [Practice] Changing how your tasks are triggered

Foi colocado em prática o que foi visto usando o exemplo do diagrama na imagem abaixo:



Nele é possível observar uma série de tarefas que dependem do resultado da anterior para acontecer, por exemplo, na primeira parte há duas tarefas, “download_website_A” e “download_website_B”, caso as duas sejam executadas com sucesso, somente a tarefa “download_succeed” vai ser chamada, e assim por diante até chegar na parte final, que é responsável por notificar o resultado.

h. Avoid hard coding values with Variables, Macros and Templates

O apresentador mostra que é importante usar variáveis ao invés de valores fixos, pois pode ocorrer desse valor fixo colocado não ser mais referente ao que era antes, então para evitar problemas futuros é sempre bom usar variáveis, um exemplo simples seria um select, em que ao invés de usar “select * from my_table where dt=”2019-01-01””, o correto seria “select * from my_table where dt=”{{execution_date}}””.

i. How to share data between your tasks with XCOMs

Em alguns casos pode acontecer de ser necessário um compartilhamento de dados entre as tarefas, para isso existe o XCOM, que significa cross communication, ou, comunicação cruzada, que permite múltiplas tarefas enviar e receber mensagens entre elas.

j. [Practice] Sharing (big) data with XCOMs

Acessando a interface de usuário do Airflow, a DAG xcom_dag foi rodada e depois indo em Admin, em seguida em XComs, é possível encontrar uma lista de todas as “conversas” armazenadas no banco de dados, um ponto importante de citar, é que o Airflow não vai limpar automaticamente essas conversas, então é trabalho do desenvolvedor apagar manualmente ou criar um bot que automatize esse serviço, se não fizer isso em algum momento vai ter ocupado um espaço importante do banco de dados, sem contar também que manter conversas desnecessárias vai deixar o ambiente poluído visualmente, tornando mais difícil de encontrar a conversa específica desejada.

k. TriggerDagRunOperator or when your DAG controls another DAG

Usando as noções de boas práticas, devemos sempre manter nossas DAGs o mais simples e organizadas o possível, uma das maneiras de fazer isso é colocando uma DAG para controlar a outra DAG através do TriggerDagRunOperator.

l. [Practice] Trigger a DAG from another DAG

Ele explicou o código do arquivo “triggerdagop_controller_dag.py”, em que a DAG é composta de duas tarefas que são “trigger_dag” e “last_task”, em que a trigger_dag possui uma função “conditionally_trigger” que basicamente recebe os parâmetros e se tudo estiver de acordo ela aciona outra DAG, permitindo uma flexibilidade ao decidir se a DAG de destino deve ser executada e com quais dados executar.

Depois também foi visto o conteúdo do arquivo “triggerdagop_target_dag.py”, nele há outra DAG que vai ser controlada pela DAG do primeiro código mostrado, essa DAG foi criada com três tarefas distintas, a primeira printa a mensagem no log, a segunda printa a mensagem no terminal, e a terceira pausa a execução por 30 segundos.

m. Dependencies between your DAGs dependent with the ExternalTaskSensor

O apresentador explica o ExternalTaskSensor com um caso de uso, em um cenário com duas DAGs, em que a primeira é responsável por extrair dados do banco de dados de produção e a segunda DAG agrega os dados para envio dos resultados para outro banco de dados, uma coisa que podemos fazer é esperar a primeira DAG terminar o processo dela antes de iniciar a segunda DAG, implementando o ExternalTaskSensor podemos fazer com que a segunda inicie sozinha assim que a primeira finalizar corretamente. Então basicamente ela permite que fazer com que uma tarefa interna aguarde uma tarefa externa sem finalizada com sucesso para que possa iniciar.

n. [Practice] Make your DAG dependent with the ExteranlTaskSensor

O professor fez uma explicação sobre o código do arquivo “externaltasksensor_dag.py”, nele há uma DAGs composta por duas tarefas, “sensor” e “last_task”, na imagem abaixo é possível ver a tarefa sensor fazendo uso da executor ExternalTaskSensor, em que devemos colocar o id da tarefa, o id da DAG externa e também o id da tarefa externa que queremos esperar.

```
sensor = ExternalTaskSensor(
    task_id='sensor',
    external_dag_id='sleep_dag',
    external_task_id='t2'
)
```

Essa DAG externa está localizada no arquivo “sleep_dag.py” e nela também há duas tarefas, a primeira é a “t1” e a segunda “t2”, como informado na imagem, a que vamos esperar é a t2, a única coisa que ela faz é pausar por 30 segundos, a t1 não faz nada, mas pela ordem de dependência criada ela tem que ser executada antes da t2. No teste depois ao tentar executar a tarefa sensor o que aconteceu foi que ela esperou a t2 ser executada, assim como previsto, e a t2 teve que esperar a t1.

3. Deploying Airflow on AWS EKS with Kubernetes Executors and Rancher

    a. Introduction

        Rápido resumo sobre a seção.

    b. Quick overview of AWS EKS

        O autor faz um rápido resumo sobre o que é o Elastic Kubernetes Service (EKS), que basicamente facilita o gerenciamento e desenvolvimento de aplicações em containers, também conta com entrosamento com o serviços da AWS (Amazon Web Services).

    c. [Practice] Set up an EC2 instance for Rancher

        Para que possamos fazer uso do Rancher é necessário configurar uma instância EC2. Para isso, primeiro foi preciso acessar o site da aws e criar uma conta caso ainda não tenha uma, depois de logado, tem que ir em EC2 services, acessando essa página é possível criar uma instância EC2, que é um server virtual na AWS para rodar aplicações, para isso, clique em instances e depois em launch instance, ali é possível encontrar inúmeros templates contendo aplicações para lançar sua instância, algumas são pagas mas a que foi usada é grátis, que foi a Amazon Linux 2, depois tem que escolher a configuração desejada, a escolhida foi a t2.small.

        Com a instância já criada, é necessário conectar ela ao EC2, para isso, é só seguir os seguintes passos: Selecionar ela > Actions > Connect > EC2 Instance Connect > Connect, e assim a conexão está feita.

    d. [Practice] Create an IAM User with permissions

        Foi necessário criar um novo usuário para usar o Rancher, no console da AWS, clicando em services e buscando por IAM, depois seleciona essa opção e vai até Users, depois em add user é possível criar um novo usuário com o nome que quiser, mas o tipo de acesso é necessário colocar como programmatic access,

depois disso tem como adicionar os acessos ao usuário, nesse caso foi adicionado o acesso de administrador para que o usuário tenha acesso a tudo.

e. [Practice] Create an ECR repository

Nos tópicos anteriores foram criados uma instância EC2 e um usuário novo, agora é vez de criar e configurar um repositório ECR, que permite armazenar e gerenciar imagens do docker. No console da AWS, em services tem que pesquisar pro ecr, depois clicar em get started, depois só colocar um nome e clicar em create repository. Com o repositório criado, foi preciso instalar o aws cli 2, a instalação é bem simples e rápida, depois no terminal é só digitar “aws2 configure” e logar com o usuário criado anteriormente.

f. [Practice] Create an EKS cluster with Rancher

Com tudo já configurado, é possível criar um cluster EKS no Rancher acessando a interface de usuário dele, indo em Add Cluster > Amazon EKS, depois só preencher alguns campos com dados pessoais e a maioria das configurações foram mantidas nos padrões. Depois de terminar de configurar demora de 10 a 15 minutos para ser criado, quando criado, é possível acessar a página dele clicando em cima.

g. How to access your applications from the outside

Quando é rodado uma aplicação como o airflow no kubernetes cluster o web server dele vai rodar dentro de um pod, para que você consiga acessar o servidor web de fora do cluster, o kubernetes oferece algumas maneira de fazer isso, a maneira mais “primitiva” de fazer isso é usando o NodePort, que basicamente abre uma porta para todos os nodes e qualquer dado que for enviado para aquela porta é passada para o node referente, outra maneira é o Load Balancer, que faz a mesma coisa só que mais eficiente, o grande problema dela é que é paga, então dependendo do número de serviços pode acabar saindo caro, a última maneira mostrada foi a Ingress, que ao contrário das outras maneira, essa não é um serviço, é um ponto de entrada que passa os dados de acordo com a rota usada.

h. [Practice] Deploy Nginx Ingress with Catalogs (Helm)

Nessa parte foi instalado o Nginx Ingress no kubernetes cluster, para fazer isso é preciso estar na interface de usuário do Rancher, ir em airfloweks cluster > Global > Apps > Manage Catalogs, depois clicar nos três pontos que ficam no lado direito do helm e ativar ele. Depois é necessário voltar em Apps > Launch > nginx-ingress, acessando essa parte é só manter as configurações padrões, nomear e clicar em Launch. Terminando esses passos o Nginx Ingress já vai estar baixado e disponível para uso

i. [Practice] Deploy and run Airflow with the Kubernetes Executor on EKS

Nesse vídeo foi instalada e executada a primeira DAG usando airflow pelo executor do kubernetes dentro do cluster EKS, assim como foi feito no Nginx, foi usado catalogs para instalar o airflow no conjunto, então foi seguido os mesmo passo até a parte dos catálogos, só que dessa vez ao invés de ativar um, foi criado um novo em Add Catalog, o nome colocado foi “airflow-chart” e na url foi colocada uma disponibilzada no github do autor “https://github.com/marclamberti/airflow-helm-chart.git”, as outras configurações são mantidas com os valores padrões. Agora é possível instalar o airflow, clicando em Apps > Launch > airflow-k8s-eks, depois só nomear e trocar o papel de projeto para cluster, para finalizar basta clicar em Launch, depois disso tem como acessar a interface de usuário do airflow rodando dentro do eks cluster com o executor do kubernetes configurado.

j. [Practice] Cleaning your AWS

Nessa parte foi feito a limpeza de tudo feito até agora, se não fizer isso vai ficar cada vez mais pesado e lento até que os serviços sejam terminados. Na interface de usuário do Rancher tem que ir em airfloweks cluster > Global > selecionar airfloweks cluster > Delete > Delete, depois no console da aws é possível observar tudo sendo deletado, foi checado a maioria do

que foi criado, como o EKS, CloudFormation (usado pelo Rancher) e EC2, é necessário fazer essa checagem porque tem a chance de acontecer algum erro em meio ao deletamento, aí tem que deletar de volta.

4. Monitoring Apache Airflow

a. Introduction

Rápido resumo sobre os conteúdos que serão vistos nessa seção.

b. How the login system works in Airflow

O sistema de login do Airflow é baseado na biblioteca padrão logging do Python e é bem flexível em termos de configuração, permitindo que o desenvolvedor deixe configurado de uma maneira que se encaixe melhor no que ele precisa.

c. [Practice] Setting up custom logging

Primeiro todos os containers da seção oito foram inicializados, depois o autor explicou da linha 20 até a 43 do arquivo “airflow.cfg”, essas linhas se tratam de configurações para os logs, desde formatos até cores. Depois foi acessado um diretório do github com um código com a configuração dos logs que vai ser usada (https://github.com/apache/airflow/blob/v1-10-stable/airflow/config_templates/airflow_local_settings.py), depois foi criado um novo arquivo chamado “log_config.py” e nele foi colocado o código tirado do github, depois no arquivo airflow.cfg foi necessário instanciar o “`logging_config_class`“ para “`log_config.DEFAULT_LOGGING_CONFIG`“ essa função está no código do outro arquivo, em que é possível ver como está configurada.

d. [Practice] Storing your logs in AWS S3

AWS S3 é um espaço em nuvem totalmente gratuito para guardar objetos que consistem em dados. Por padrão o Airflow armazena seus dados localmente, então se você roda muitos trabalhos grandes ou pequenos mas frequentemente, seu

espaço livre na memória vai desaparecer bem rápido, então armazenando seus dados no S3 faz com que seus custos em armazenamento caiam e também vão ter mais garantia de durabilidade, já que a nuvem pode ser acessada de qualquer máquina.

Sabendo a teoria, na prática para baixar basta ir no console da AWS, e pesquisar por S3 na aba de serviços, depois em Create Bucket, a única coisa que foi preciso fazer foi colocar um nome, o resto das configurações foram mantidas nas padrões, depois de finalizar a criação, é necessário criar um novo usuário com permissão apenas para ler e escrever naquele espaço, a maioria do processo de criação do novo usuário foi a mesma dos anteriores, com a diferença de que na hora de escolher o serviço ele foi colocado com S3, e nos nível de acesso as únicas opções selecionadas foram ListBucket, GetObject, PutObject, DeleteObject, ReplicateObject e RestoreObject.

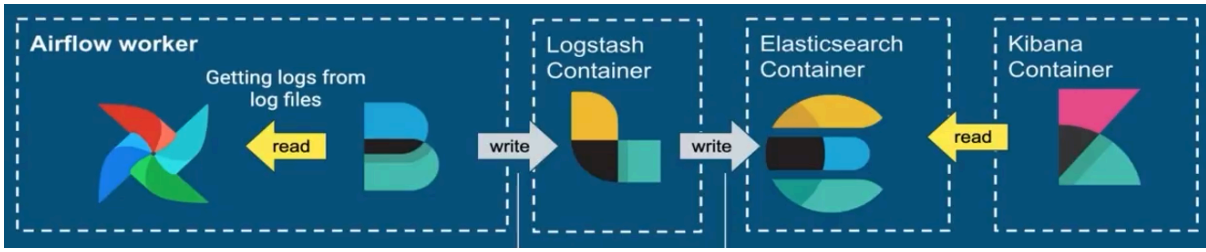
Depois para tudo funcionar foi só ir na interface de usuário do Airflow > Admin > Connections > Create > Conn Id: AWSS3LogStorage > Conn Type: S3 > Save. Depois foi aberto o arquivo airflow.cfg o `remote_logging` foi setado para True, permitindo que o airflow escreva e leia logs de uma localização remota, depois em `remote_log_conn_id` foi necessário colocar o id da conexão que acabou de ser criada (AWSS3LogStorage), depois em `remote_base_log_folder` tem que colocar a pasta na qual os logs vão ser armazenados. Para testar foi só rodar a DAG logger_dag, depois de rodar ir na parte do Log e na primeira linha vai ter um aviso mostrando que a leitura foi feita remotamente:
*** Reading remote log from s3://fba-airflow-logs/airflow-logs/logger_dag/t1.

e. Elasticsearch Reminder

Elasticsearch é um mecanismo open source de pesquisa que permite você armazenar, pesquisar e analisar grandes volumes de dados através de uma interface web, então permite armazenar os dados do log para que depois seja possível encontrar tendências, estatísticas, sumarização e mais algumas coisas em tempo real.

f. [Practice] Configuring Airflow with Elasticsearch

O airflow foi configurado usando o elasticsearch, seguindo uma arquitetura de um airflow worker e três containers, Logstash, Elasticsearch e Kibana. Ao contrário do S3 da AWS, não é possível escrever os eventos dos logs diretamente no elasticsearch, a única coisa que é possível fazer diretamente é ler os logs armazenados lá, no final, quando uma DAG for acionada, os eventos de log de um tarefa específica vão ser armazenadas em um arquivo de log local no formato JSON, cada vez que uma novo arquivo log for gerado, ele vai ser processado e enviado para o logstash, que vai aplicar algumas alterações para que seja gerado um id do log requerido pelo airflow, depois de todo esse processo, o arquivo finalmente vai ser enviado para o elasticsearch, e assim que os dados chegarem nele, vai ser possível fazer o monitoramento pelo kibana. A estrutura fica assim:



Depois foi dado uma olhada no final do código do arquivo “docker-compose-CeleryExecutorELK.yml”, nele é possível ver as configurações dos containers, também foi visto o arquivo “airflow-logs.conf”, é nele que vai acontecer o processamento dos logs para serem enviados para o elasticsearch, tanto que é possível ver que a estrutura está separada em entrada, filtro e saída.

Para que funcione tudo devidamente, foi necessário seguir alguns passos, como setando o `remote_logging` para true, `host` para http://elasticsearch:9200, deixar o `write_stdout` vazio (antes estava definido para False), `json_format` para true. Com todas essas configurações feitas no airflow.cfg, agora é possível iniciar a arquitetura.

g. [Practice] Monitoring your DAGs with Elasticsearch

A primeira coisa feita foi uma limpeza para liberar espço feita pela interface de usuário do airflow, Browse > Task Instances > Select all > With selected > Clear > Browse > DAG Runs > Select > With selected > Delete, agora acessando a interface de usuário do kibana, seguindo para a área de configuração > Index Management > Select > Manage Index > Confirm > Index Patterns > Select > Delete.

Com a limpeza feita, foram gerados alguns eventos de log para criar um dashboard rodando a DAG data_dag, ela possui três tarefas que basicamente não fazem nada, com exceção da última que vai falhar na metade do tempo baseado na data atual de execução. Agora no kibana indo Index Management é possível ver que o log foi salvo no elasticsearch, para fazer com que você tenha total visão sobre o que está acontecendo, é só ir em Index Patterns > Create index pattern > preencher o campo com airflow-logs > Next step > Selecionar @timestamp > Create index pattern, agora para criar o dashboard no kibana, >Visualize > Create new visualization > Gauge > airflow-logs, com ele é possível ver múltiplos dados que são baseados nas requisições do elasticsearch.

Para testar foi rodado duas DAGs, data_dag e logger_dag, voltando no dashboard, é possível ver o número de logs depois que foram executadas, aí caso queira tem como filtrar e configurar como você deseja que apareça.

h. Introduction to metrics

Todas as métricas do Airflow são baseadas em três tipos que são Counter, Gauges e Timers. O tipo a ser usado depende muito da situação, por exemplo, quando queremos o número total de tarefas que falharam o Counter é usado, o número de tarefas na fila do executor é o Gauges, e o número de milissegundos que a tarefa demorou para conseguir terminar é um Timer.

i. [Practice] Monitoring Airflow with TIG stack

Foi dado uma olhada no arquivo docker-compose-CeleryExecutorTIG.yml, na última parte dele é

possível encontrar os serviços do TIG stack, que são telegraf, influxfb e grafana, depois foi aberto o arquivo telegraf.conf para descomentar algumas linhas contendo configurações, seguindo os seguintes passos > [[outputs.influxdb]] > linha 104 e trocar o ip por influxdb > linha 108 > linha 120 e setar para true > linha 131 > linha 134 > linha 135 e trocar a senha para telegrafpass > linha 138. Tendo o output configurado, ainda é necessário configurar o input > [[inputs.statsd]] > descomentar todas as linhas de comando. Também foi preciso definir alguns parâmetros no arquivo airflow.cfg > setar `statsd_on` para True > `statsd_host` para telegraf.

Com tudo configurado, no cmd foi criado uma banco de dados e um usuário com os comandos “create database telegraf” e “create user telegraf with password ‘telegrafpass’”. Agora na interface do Grafana > Add data source > InfluxDB > URL: http://influxdb:8086 > Database: telegraf > User: telegraf > Password > telegrafpass > Save & test > Página inicial > New dashboard > Add Query > Airflow UI > rodar as DAGs data_dag e logger_dag > Grafana UI > FROM: airflow-dagbag-size > SELECT: remove mean() > + > Selectors > last > remove GROUP BY > Visualization > Gauge > Calc: Last > General > Title: Number of imported DAGs > Save dashboard > New name: airflow > Save. E assim foi criado um dashboard para monitorar as DAGs com Grafana, InfluxDB e Telegraf

j. [Practice] Triggering alerts for Airflow with Grafana

Aqui foi ensinado a como fazer com que o Grafana avise automaticamente quando algo errado acontecer, basta seguir os seguintes passos > abrir o arquivo grafana.ini > descomentar as linhas > 451 e setar `enabled` para true > 452 e trocar

localhost:25 por smp.gmail.com:587 > 453, colocar o email que deseja enviar a notificação > 455, colocar a senha que foi obtida gerando o app com o gmail. > voltar para o Grafana UI > Alerting > Notification channels > Add Channel > Name: email-channel > Type: email > Adresses: email que deseja receber a notificação > Save. Depois é necessário criar um novo painel no dashboard criado anteriormente indo em Dashboards > airflow > Add panel > Add Query > FROM: airflow_dag_logger_dag_t2_duration > SELECT: field(100_percentile) > remove mean() > + > Aggregations: distinct > Visualization > Axes > Unit: time: duration (s) > General > Title: logger_dag_t2_duration > Alert > Create Alert > WHEN: last () > IS ABOVE: 5 > Send to: email-channel > Save dashboard > Save. Fazendo isso, para testar é necessário ir até a tarefa 2 da DAG logger_dag e colocar pra ela pausar por mais de 5 segundos antes de terminar, depois só rodar a DAG de volta e ver se funcionou através do dashboard.

k. Airflow maintenance DAGs

O autor compartilhou algumas DAGs criadas por Robert Sanders disponíveis em https://github.com/teamclairvoyant/airflow-maintenance-dags.git, no total são cinco DAGs que podem acabar sendo úteis, por exemplo, log-cleanup vai remover todas os logs mais velhos que trinta dias, db-cleanup vai limpar algumas coisas como DagRun, TaskInstance, Log, XCom, Job DB e entradas SlaMiss. Contém mais que essas duas, cada uma bem organizadinha e explicada passo a passo.

5. Security in Apache Airflow

a. Introduction

Rápido resumo sobre a seção

b. [Practice] Encrypting sensitive data with Fernet

Até agora tudo que foi feito foi criar e manipular dados, mas algo que ainda falta é uma maneira de proteger eles, para

fazer isso, primeiro é necessário criar uma conexão no Airflow através do Airflow UI > Admin > Connections > Create > Extra: {“access_key”:”my_key”, “secret_key”:”my_secret”} > Save. Depoisd no arquivo airflow.cfg setar o `secure_mode` para true, depois de ter feito essas passo, é necessário baixar o pacote crypto, então no arquivo Dockerfile, linha 65, tem que adicionar “crypto” junto com os outros pacotes e reiniciar o programa.

De volta no arquivo airflow.cfg, na linha 121 tem que instanciar o `fernet_key` como a palavra chave, para conseguir essa palavra chave basta rodar o código do arquivo “generate_fernet_key”. Depois só reiniciar novamente o servidor e testar.

c. [Practice] Rotating the Fernet Key

No tópico anterior a senha da conexão foi encriptada, e uma maneira de deixar ela ainda mais segura é trocar de tempos em tempos, então tem que rodar o código do arquivo “generate_fernet_key” novamente, copiar a nova chave gerada e colar ela no atrás da senha criada anteriormente, adicionando uma vírgula ao final. Depois tem que abrir o terminal, e executar a seguinte linha “airflow rotate_fernet_key”, depois disso pode apagar a senha antiga.

d. [Practice] Hiding variables

Aqui foi ensinado a como esconder o valor de algumas variáveis de acordo com algum nome definido, para fazer isso é só ir na interface do Airflow > Admin > Variables > Create > Key: my_var > Val: my_value > Save, ainda é possível ver o valor inserido na variável, mas se na chave tivesse contido alguma das palavras da imagem abaixo, o valor não seria mostrado.

```
DEFAULT_SENSITIVE_VARIABLE_FIELDS = (
    'password',
    'secret',
    'passwd',
    'authorization',
    'api_key',
    'apikey',
    'access_token',
)
```

e. [Practice] Password authentication and filter by owner

Para fazer com que tenha que passar por uma autenticação antes de acessar o airflow, é preciso ir na linha 262 do arquivo airflow.cfg e setar para `authenticate` True, logo abaixo também foi adicionado um novo parâmetro necessário “auth_backend = airflow.contrib.auth.backends.password_auth”, depois tem que ir no arquivo Dockerfile, e adicionar um novo comando para instalar um pacote que será responsável por encriptar a senha, “&& pip install flask-bcrypt \”, depois tem que reiniciar, aí quando tentar abrir a interface do Airflow ela vai requisitar login antes que tenha acesso às ferramentas, para gerar um usuário basta rodar o código do arquivo generate_user, esse código pode ser rodado infinitas vezes para criar infinitos usuários, por isso é importante setar `filter_by_owner` para True no arquivo airflow.cfg, caso esteja como falso os dados de outros usuários também vão aparecer no seu usuário.

f. [Practice] RBAC UI

O autor começa o vídeo explicando que existem diferentes tipos de usuário, primeiro o Admin que tem acesso a todas as permissões possíveis (incluindo adicionar ou retirar permissões de outros usuários), Public que são usuário anônimos e não tem permissões, e o Viewer que tem permissões limitadas.

Depois no arquivo airflow.cfg, o parâmetro `rbac` da linha 299 foi setado para True, e no terminal foi criado uma nova conta executando o seguinte código “airflow create_user -r Admin --username admin --password admin --email admin@airflow.com

--firstname primeiro--lastname ultimo”, quando for tentar acessar a interface do airflow, ela vai pedir login, assim como no tópico anterior, mas como o rbac está ativado, aquele usuário criado não é válido mais. Quando logar, é possível observar uma nova seção na aba de ferramentas “Security” em que o administrador tem acesso a todos os usuários e suas estatísticas, conseguindo também trocar permissões. O administrador também é capaz de adicionar novos usuários através dessa aba, aí foi feito um teste, em que foi criado um usuário com cargo de viewer, quando o airflow UI foi acessado com ele, a aba Security não existia.