Kedro Airflow K8S Plugin

Release 0.8.1

GetInData

Jun 22, 2023

CONTENTS:

INTRODUCTION

1.1 What is Airflow?

Airflow is a platform to programmatically author, schedule and monitor workflows. Workflows are represented as DAGs. Each DAG is represented by nodes, that define job to be executed. The DAGs are stored in the file storage, allowing user to run the pipeline once or schedule the recurring run.

1.2 What is Kubernetes?

Kubernetes is a platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

1.3 Why to integrate Kedro project with Airflow and Kubernetes?

Airflow's main attitude is the portability. Once you define a pipeline, it can be started on any Kubernetes cluster. The code to execute is stored inside docker images that cover not only the source itself, but all the libraries and entire execution environment. Portability is also one of key Kedro aspects, as the pipelines must be versionable and packagebale. Kedro, with Kedro-docker plugin do a fantastic job to achieve this and Airflow looks like a nice add-on to run the pipelines on powerful remote Kubernetes clusters.

TWO

INSTALLATION

2.1 Installation guide

2.1.1 Kedro setup

First, you need to install base Kedro package in `<17.0` version

> Kedro 17.0 is supported by kedro-airflow-k8s, but not by kedro-mlflow yet, so the latest version from 0.16 family is recommended.

```
$ pip install 'kedro<0.17'
```

2.1.2 Plugin installation

Install from PyPI

You can install `kedro-airflow-k8s` plugin from `PyPi` with `pip`:

```
pip install --upgrade kedro-airflow-k8s
```

Install from sources

You may want to install the develop branch which has unreleased features:

```
pip install git+https://github.com/getindata/kedro-airflow-k8s.git@develop
```

2.1.3 Available commands

You can check available commands by going into project directory and runnning:

```
$ kedro airflow-k8s

Usage: kedro airflow-k8s [OPTIONS] COMMAND [ARGS]...

Options:
-e, --env TEXT  Environment to use.
-p, --pipeline TEXT  Pipeline name to pick.
```

```
-h, --help      Show this message and exit.

Commands:
  compile         Create an Airflow DAG for a project
  init            Initializes configuration for the plugin
  list-pipelines  List pipelines generated by this plugin
  run-once        Uploads pipeline to Airflow and runs once
  schedule        Uploads pipeline to Airflow with given schedule
  ui              Open Apache Airflow UI in new browser tab
  upload-pipeline Uploads pipeline to Airflow DAG location
```

compile

`compile` command takes one argument, which is the directory name containing configuration (relative to `conf` folder). As an outcome, `dag` directory contains python file with generated DAG.

init

`init` command adds default plugin configuration to the project, based on Apache Airflow CLI input. It also allows optionally adding github actions, to streamline project build and upload.

list-pipelines

`list-pipelines` lists all pipelines generated by this plugin which exist in Airflow server. All generated DAGs are tagged with tag `generated_with_kedro_airflow_k8s:$PLUGIN_VERSION` and the prefix of this tag is used to distinguish among the other tags.

run-once

`run-once` command generates DAG from the pipeline, uploads it Airflow DAG location and triggers the DAG run as soon as the new DAG instance is available. It optionally allows waiting for DAG run completion, checking if `success` status is returned.

schedule

`schedule` command takes three arguments, one is the directory name containing configuration (relative to `conf` folder), the second one is the output location of generated dag, the third is cron like expression that relates to Airflow DAG `schedule_interval`.

`ui`

ui simplifies access to Apache Airflow console. It also allows open UI for the specific DAG.

`upload-pipeline`

upload-pipeline command takes two arguments, one is the directory name containing configuration (relative to conf folder), the second one is the output location of generated dag.

2.2 Configuration

Plugin maintains the configuration in the `conf/base/airflow-k8s.yaml` file.

```
# Base url of the Apache Airflow, should include the schema (http/https)
host: https://airflow.example.com

# Directory from where Apache Airflow is reading DAGs definitions
output: gs://airflow-bucket-example-com

# Configuration used to run the pipeline
run_config:

    # Name of the image to run as the pipeline steps
    image: airflow-k8s-plugin-demo

    # Pull policy to be used for the steps. Use Always if you push the images
    # on the same tag, or Never if you use only local images
    image_pull_policy: IfNotPresent

    # Pod startup timeout in seconds - if timeout passes the pipeline fails, default to
→600
    startup_time: 600

    # Namespace for Airflow pods to be created
    namespace: airflow

    # Name of the Airflow experiment to be created
    experiment_name: Airflow K8S Plugin Demo

    # Name of the dag as it's presented in Airflow
    run_name: airflow-k8s-plugin-demo

    # Apache Airflow cron expression for scheduled runs
    cron_expression: "@daily"

    # Optional start date in format YYYYMMDD, if not provided `days_ago(2)` is used
→instead
    start_date: "20210721"

    # Optional pipeline description
    description: "Very Important Pipeline"
```

(continues on next page)

```
    # Comma separated list of image pull secret names
    image_pull_secrets: my-registry-credentials

    # Service account name to execute nodes with
    service_account_name: airflow

    # List of handlers executed after task failure
    failure_handlers:
      # type of integration, currently only slack is available
      - type: slack
        # airflow connection id with following parameters:
        # host - webhook url
        # password - webhook password
        # login - username
        connection_id: slack
        # message template that will be send. It can contains following parameters that␣
→will be replaced:
        # task
        # dag
        # execution_time
        message_template: |
            :red_circle: Task Failed.
            *Task*: {task}
            *Dag*: {dag}
            *Execution Time*: {execution_time}
            *Log Url*: {url}

    # Optional volume specification
    volume:
        # Storage class - use null (or no value) to use the default storage
        # class deployed on the Kubernetes cluster
        storageclass: # default
        # The size of the volume that is created. Applicable for some storage
        # classes
        size: 1Gi
        # Access mode of the volume used to exchange data. ReadWriteMany is
        # preferred, but it is not supported on some environemnets (like GKE)
        # Default value: ReadWriteOnce
        #access_modes: [ReadWriteMany]
        # Flag indicating if the data-volume-init step (copying raw data to the
        # fresh volume) should be skipped
        skip_init: False
        # Allows to specify fsGroup executing pipelines within containers
        # Default: root user group (to avoid issues with volumes in GKE)
        owner: 0
        # If set to True, shared persistent volume will not be created at all and all␣
→other parameters under
        # `volume` are discarded
        disabled: False

    # List of optional secrets specification
```

```
    secrets:
            # deploy_type: (Optional - default: 'env`) The type of secret deploy in␣
→Kubernetes, either `env` or `volume`
        -   deploy_type: "env"
            # deploy_target: (Optional) The environment variable when `deploy_type`␣
→`env` or file path when `deploy_type` `volume` where expose secret. If `key` is not␣
→provided deploy target should be None.
            deploy_target: "SQL_CONN"
            # secret: Name of the secrets object in Kubernetes
            secret: "airflow-secrets"
            # key: (Optional) Key of the secret within the Kubernetes Secret if not␣
→provided in `deploy_type` `env` it will mount all secrets in object
            key: "sql_alchemy_conn"


    # Apache Airflow macros to be exposed for the parameters
    # List of macros can be found here:
    # https://airflow.apache.org/docs/apache-airflow/stable/macros-ref.html
    macro_params: [ds, prev_ds]

    # Apache Airflow variables to be exposed for the parameters
    variables_params: [env]

    # Optional resources specification
    resources:
        # Default configuration used by all nodes that do not declare the
        # resource configuration. It's optional. If node does not declare the resource
        # configuration, __default__ is assigned by default, otherwise cluster defaults
        # will be used.
        __default__:
            # Optional labels to be put into pod node selector
            node_selectors:
              #Labels are user provided key value pairs
              node_pool_label/k8s.io: example_value
            # Optional labels to apply on pods
            labels:
              running: airflow
            # Optional annotations to apply on pods
            annotations:
              iam.amazonaws.com/role: airflow
            # Optional list of kubernetes tolerations
            tolerations:
                - key: "group"
                  value: "data-processing"
                  effect: "NoExecute"
                - key: "group"
                  operator: "Equal",
                  value: "data-processing",
                  effect: "NoSchedule"
            requests:
                #Optional amount of cpu resources requested from k8s
                cpu: "1"
```

```
                Optional amount of memory resource requested from k8s
                memory: "1Gi"
            limits:
                #Optional amount of cpu resources limit on k8s
                cpu: "1"
                #Optional amount of memory resource limit on k8s
                memory: "1Gi"
        # Other arbitrary configurations to use, for example to indicate some exception␣
→resources
        huge_machines:
            node_selectors:
                big_node_pool: huge.10x
            requests:
                cpu: "16"
                memory: "128Gi"
            limits:
                cpu: "32"
                memory: "256Gi"
    # Optional external dependencies configuration
    external_dependencies:
        # Can just select dag as a whole
        - dag_id: upstream-dag
        # or detailed
        - dag_id: another-upstream-dag
        # with specific task to wait on
          task_id: with-precise-task
        # Maximum time (minute) to wait for the external dag to finish before this
        # pipeline fails, the default is 1440 == 1 day
          timeout: 2
        # Checks if the external dag exists before waiting for it to finish. If it
        # does not exists, fail this pipeline. By default is set to true.
          check_existence: False
        # Time difference with the previous execution to look at (minutes),
        # the default is 0 meaning no difference
          execution_delta: 10

    # Optional authentication to MLflow API
    authentication:
      # Strategy that generates the tokens, supported values are:
      # - Null
      # - GoogleOAuth2 (generating OAuth2 tokens for service account provided by GOOGLE_
→APPLICATION_CREDENTIALS)
      # - Vars (credentials fetched from airflow Variable.get - specify variable keys,
      # matching MLflow authentication env variable names, in `params`,
      # e.g. ["MLFLOW_TRACKING_USERNAME", "MLFLOW_TRACKING_PASSWORD"])
      type: GoogleOAuth2
      #params: []

    # Optional custom kubermentes pod templates applied on nodes basis
    kubernetes_pod_templates:
    # Name of the node you want to apply the custom template to.
    # if you specify __default__, this template will be applied to all nodes.
```

```
    # Otherwise it will be only applied to nodes tagged with `k8s_template:<node_name>`
      spark:
  # Kubernetes pod template.
  # It's the full content of the pod-template file (as a string)
  # `run_config.volume` and `MLFLOW_RUN_ID` env are disabled when this is set.
  # Note: python F-string formatting is applied to this string, so
  # you can also use some dynamic values, e.g. to calculate pod name.
      template: |-
        type: Pod
        metadata:
          name: {PodGenerator.make_unique_pod_id('{{ task_instance.task_id }}')}
          labels:
            spark_driver: {'{{ task_instance.task_id }}'}

  # Optionally, you can also override the image
  #   image:

  # Optional spark configuration
  spark:
    # Type of spark clusters to use, supported values: dataproc, k8s, kubernetes,␣
→custom
    type: dataproc
    # Optional factory of spark operators class
    operator_factory: my_project.factories.OperatorFactory
    # Region indicates location of cluster for public cloud configurations, for␣
→example region in GCP
    region: europe-west1
    # Project indicates logical placement inside public cloud configuration, for␣
→example project in GCP
    project_id: target-project
    # Name of the cluster to be created
    cluster_name: ephemeral
    # Location where the spark artifacts are uploaded
    artifacts_path: gs://dataproc-staging-europe-west2-546213781-jabcdefp4/packages
    # Optional path in the project to the script portion preprended to generated init␣
→script
    user_init_path: relative_location_to_src/init_script.sh
    # Optional path in the project to the script portion appended to generated init␣
→script
    user_post_init_path: relative_location_to_src/post_init_script.sh
    # Optional configuration of the cluster, used during cluster creation, depends on␣
→type of the cluster
    cluster_config: # example dataproc configuration
      master_config:
        disk_config:
          boot_disk_size_gb: 35
      worker_config:
        disk_config:
          boot_disk_size_gb: 35
```

2.2.1 Indicate resources in pipeline nodes

Every node declared in `kedro` pipelines is executed inside pod. Pod definition declares resources to be used based on provided plugin configuration and presence of the tag `resources` in `kedro` node definition.

If no such tag is present, plugin will assign `__default__` from plugin `resources` configuration. If no `__default__` is given in plugin `resources` configuration or no `resources` configuration is given, pod definition will not be given any information on how to allocate resources to pod, thus default k8s cluster values will be used.

```
# train_model node is assigned resources from `huge_machines` configuration, if no such
→configuration exists,
# `__default__` is used, and if __default__ does not exist, k8s cluster default values
→are used
node(func=train_model, inputs=["X_train", "y_train"], outputs="regressor", name='train_
→model', tags=['resources:huge_machines'])
# evaluate_model node is assigned resources `__default__` configuration and if it does
→not exist,
# k8s cluster default values are used
node(func=evaluate_model, inputs=["X_train", "y_train"], outputs="regressor", name=
→'evaluate_model')
```

2.2.2 Custom kubernetes pod templates

You can provide custom kubernetes pod templates using `kubernetes_pod_templates`. Pod template to be used is based on the provided plugin configuration and presence of the tag `k8s_template` in `kedro` node definition.

If no such tag is present, plugin will assign `__default__.template` from plugin `kubernetes_pod_templates` configuration, if exists. If no `__default__` is given in plugin `kubernetes_pod_templates` configuration or no `kubernetes_pod_templates` configuration is provided at all, the following plugin's default minimal pod template will be used.

```
type: Pod
  metadata:
    name: {PodGenerator.make_unique_pod_id('{{ task_instance.task_id }}')}
spec:
  containers:
    - name: base
      env:
        - name: MLFLOW_RUN_ID
          value: {{ task_instance.xcom_pull(key="mlflow_run_id") }}
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: {self._pvc_name}
```

where environment and volumes sections are present only if kedro mflow is used in the project and/or `run_config.volume` is not disabled.

Note, that `claimName` is calculated the following way

```
pvc_name = '{{ project_name | safe | slugify }}.{% raw %}{{ ts_nodash | lower  }}{%
→endraw %}'
```

If you do use a custom pod template and you want to keep the built-in mlflow/volume support you need to include these sections in your template as well.

```
# train_model node is assigned a custom pod template from `spark` configuration, if no␣
↪such configuration exists,
# `__default__` is used, and if __default__ does not exist, the plugin's minimal pod␣
↪template is used
node(func=train_model, inputs=["X_train", "y_train"], outputs="regressor", name='train_
↪model', tags=['k8s_template:spark'])
# evaluate_model node is assigned a custom pod template `__default__` configuration and␣
↪if it does not exist,
# the plugin's default minimal pod template
node(func=evaluate_model, inputs=["X_train", "y_train"], outputs="regressor", name=
↪'evaluate_model')
```

When using custom kubernetes pod templates the resulting pod configuration is a merge between properties provided via plugin settings, e.g. `resources.__default__.annotations`, and those specified in a template. In case of a conflict, plugin settings precede that of the template.

2.2.3 Spark on Kubernetes configuration

In order to configure spark on kubernetes, custom `cluster_config` has to be provided. It has the following structure:

```
type: kubernetes # or k8s
cluster_name: spark_k8s # name of the Airflow connection id, that points to kubernetes␣
↪control plane, default `spark_default`
cluster_config:
  # Location of the script that initialize the kedro session and runs the project; is␣
↪invoked with `run --env=$ENV --node=$NODES --runner=ThreadRunner` kedro parameters
  # arguments
  run_script: local:///home/kedro/spark_run.py
  # Optional image to use for the driver and executor. If not provided, value from `run_
↪config.image` is used
  image: airflow-k8s-plugin-spark-demo
  # Optional spark configuration
  conf:
    spark.dynamicAllocation.enabled: true
    spark.dynamicAllocation.maxExecutors: 4
  # Optional port of the driver
  driver_port: 10000
  # Optional port of the block managers
  block_manager_port: 10001
  # Optional dictionary of secrets to be mounted into driver and executor from the␣
↪namespace of the project
  secrets:
    kedro-secret: /var/kedro_secret
  # Optional labels to be assigned to driver and executor
  labels:
    huge-machine: yes
  # Optional spark-dir local storage to be mounted from dynamically created PVC
  local_storage:
    class_name: standard
    size: 100Gi
  # Optional dictionary of environment variables to be injected into driver and executor␣
↪pods
```

(continues on next page)

```
env_vars:
  GOOGLE_APPLICATION_CREDENTIALS: /var/kedro_secret/sa
# Optional request for cpu resources, for driver and executor (memory request equals␣
↪memory limit)
requests:
  cpu: 2
# Optional limit for the cpu and memory resources, for driver and executor
limits:
  cpu: 4
  memory: 16Gi # enforces memory request
# Optional number of executors to spawn
num_executors: 4 # by default 1
# Optional list of jars used by the spark runtime, spark-submit format
jars: local:///home/kedro/jars/gcs-connector.jar
# Optional list of repositories used by the spark runtime, spark-submit format
repositories: https://oss.sonatype.org/content/groups/public
# Optional list of packages used by the spark runtime, spark-submit format
packages: org.apache.spark:spark-streaming-kafka_2.10:1.6.0,org.
↪elasticsearch:elasticsearch-spark_2.10:2.2.0
```

Additionally, the following parameters are used from the project configuration to create the pods: `run_config.image`, `run_config.image_pull_policy`, `run_config.namespace`, `run_config.service_account_name`.

Further details on configuring spark jobs on k8s at https://spark.apache.org/docs/latest/running-on-kubernetes.html.

2.2.4 Dynamic configuration support

kedro-airflow-k8s contains hook that enables TemplatedConfigLoader. It allows passing environment variables to configuration files. It reads all environment variables following KEDRO_CONFIG_ pattern, which you can later inject in configuration file using ${name} syntax.

There are two special variables KEDRO_CONFIG_COMMIT_ID, KEDRO_CONFIG_BRANCH_NAME with support specifying default when variable is not set, e.g. ${commit_id|dirty}

GETTING STARTED

3.1 Quickstart

3.1.1 Prepequisites

Although the plugin does not perform deployment, it's recommended to have access to Airflow DAG directory in order to test run the generated DAG.

3.1.2 Install the toy project with Kedro Airflow K8S support

It is a good practice to start by creating a new virtualenv before installing new packages. Therefore, use `virtalenv` command to create new env and activate it:

```
$ virtualenv venv-demo
created virtual environment CPython3.8.5.final.0-64 in 145ms
  creator CPython3Posix(dest=/home/mario/kedro/venv-demo, clear=False, no_vcs_
→ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle,␣
→via=copy, app_data_dir=/home/mario/.local/share/virtualenv)
    added seed packages: pip==20.3.1, setuptools==51.0.0, wheel==0.36.2
  activators BashActivator,CShellActivator,FishActivator,PowerShellActivator,
→PythonActivator,XonshActivator
$ source venv-demo/bin/activate
```

Then, `kedro` must be present to enable cloning the starter project, along with the latest version of `kedro-airflow-k8s` plugin and kedro-docker.

```
$ pip install 'kedro<0.17' kedro-airflow-k8s kedro-docker
```

With the dependencies in place, let's create a new project:

```
$ kedro new --starter=git+https://github.com/getindata/kedro-starter-spaceflights.git --
→checkout allow_nodes_with_commas
Project Name:
=============
Please enter a human readable name for your new project.
Spaces and punctuation are allowed.
 [New Kedro Project]: Airflow K8S Plugin Demo

Repository Name:
```

```
================
Please enter a directory name for your new project repository.
Alphanumeric characters, hyphens and underscores are allowed.
Lowercase is recommended.
 [airflow-k8s-plugin-demo]:


Python Package Name:
====================
Please enter a valid Python package name for your project package.
Alphanumeric characters and underscores are allowed.
Lowercase is recommended. Package name must start with a letter or underscore.
 [airflow_k8s_plugin_demo]:


Change directory to the project generated in ${CWD}/airflow-k8s-plugin-demo


A best-practice setup includes initialising git and creating a virtual environment␣
↪before running
 `kedro install` to install project-specific dependencies. Refer to the Kedro
  documentation: https://kedro.readthedocs.io/
```

TODO: switch to the official `spaceflights` starter after https://github.com/quantumblacklabs/kedro-starter-spaceflights/pull/10 is merged

Finally, go the demo project directory and ensure that kedro-airflow-k8s plugin is activated:

```console
$ cd airflow-k8s-plugin-demo/
$ kedro install
(...)
Requirements installed!
$ kedro airflow-k8s --help
```console
$ kedro airflow-k8s

Usage: kedro airflow-k8s [OPTIONS] COMMAND [ARGS]...

Options:
-e, --env TEXT  Environment to use.
-p, --pipeline TEXT  Pipeline name to pick.
-h, --help      Show this message and exit.

Commands:
  compile        Create an Airflow DAG for a project
  init           Initializes configuration for the plugin
  list-pipelines List pipelines generated by this plugin
  run-once       Uploads pipeline to Airflow and runs once
  schedule       Uploads pipeline to Airflow with given schedule
  ui             Open Apache Airflow UI in new browser tab
  upload-pipeline  Uploads pipeline to Airflow DAG location
```

3.1.3 Build the docker image to be used on Airflow K8S runs

First, initialize the project with `kedro-docker` configuration by running:

```
$ kedro docker init
```

This command creates a several files, including `.dockerignore`. This file ensures that transient files are not included in the docker image and it requires small adjustment. Open it in your favourite text editor and extend the section `# except the following` by adding there:

```
!data/01_raw
```

This change enforces raw data existence in the image. Also, one of the limitations of running the Kedro pipeline on Airflow (and not on local environment) is inability to use MemoryDataSets, as the pipeline nodes do not share memory, so every artifact should be stored as file. The `spaceflights` demo configures four datasets as in-memory, so let's change the behaviour by adding these lines to `conf/base/catalog.yml`:

```
X_train:
  type: pickle.PickleDataSet
  filepath: data/05_model_input/X_train.pickle
  layer: model_input

y_train:
  type: pickle.PickleDataSet
  filepath: data/05_model_input/y_train.pickle
  layer: model_input

X_test:
  type: pickle.PickleDataSet
  filepath: data/05_model_input/X_test.pickle
  layer: model_input

y_test:
  type: pickle.PickleDataSet
  filepath: data/05_model_input/y_test.pickle
  layer: model_input
```

Finally, build the image:

```
kedro docker build
```

When execution finishes, your docker image is ready. If you don't use local cluster, you should push the image to the remote repository:

```
docker tag airflow_k8s_plugin_demo:latest remote.repo.url.com/airflow_k8s_plugin_
→demo:latest
docker push remote.repo.url.com/airflow_k8s_plugin_demo:latest
```

3.1.4 Setup GIT repository

Plugin requires project to be under git repository. Perform repository initialization and commit project files

3.1.5 Compile DAG

Plugin requires configuration to be present. It's best to use:

```
kedro airflow-k8s init --with-github-actions --output ${AIRFLOW_DAG_FOLDER} https://
→airflow.url
```

This command creates configuration file in `conf/pipelines/airflow-k8s.yaml` with some custom values and reference to Airflow passed in arguments. It also creates some default github actions.

> When using this command, pay attention that the configuration expects `commit_id` and `google_project_id` to be present. Set them up by setting environment variable `KEDRO_CONFIG_COMMIT_ID` and `KEDRO_CONFIG_GOOGLE_PROJECT_ID`.

Also mlflow configuration has to be set up (if required by the project) as described in mlflow section.

Having configuration ready, type:

```
kedro airflow-k8s -e pipelines compile
```

This command compiles pipeline and generates DAG in `dag/airflow_k8s_plugin_demo.py`. This file should be copied manually into Airflow DAG directory, that Airflow periodically scans. After it appears in airflow console, it is ready to be triggered.

As an alternative, one cas use the following:

```
kedro airflow-k8s -e pipelines upload-pipeline -o ${AIRFLOW_DAG_HOME}
```

in order to get DAG copied directly to Airflow DAG folder. Google Cloud Storage locations are also support with `gcs://` or `gs://` prefix in the parameter (this requires plugin to be installed with `pip install kedro-airflow-k8s[gcp]`).

In order to use AWS S3 as storage, prefix output with `s3://` (this requires plugin to be installed with `pip install kedro-airflow-k8s[aws]`).

> It's optional to indicate which pipeline to pick, with `-p` option. By default, pipeline name `__default__` is used. Option `-p` can refer to other pipeline by name it's registered inside kedro hook.

3.1.6 Diagnose execution

Every kedro node is transformed into Airflow DAG task. DAG also contains other, supporting tasks, which are handled by a set of custom operators. In order to diagnose DAG run, every task is logging information with standar python logging library. The outcome is available in Airflow `Log` tab.

3.2 GCP AI Platform support

Google Cloud's AI Platform offers couple services that simplify Machine Learning tasks.

3.2.1 Using `kedro` with AI Platform Notebooks

AI Platform Notebooks provides an easy way to manage and host JupyterLab based data science workbench environment. What we've found out is that the default images provided by a service cause some dependency conflicts. To avoid this issues make sure you use isolated virtual environment, e.g. virtualenv. New virual environment can be created by simply invoking `python -m virtualenv venv` command.

3.3 Mlflow support

If you use MLflow and kedro-mlflow for the Kedro pipeline runs monitoring, the plugin will automatically enable support for:

- starting the experiment when the pipeline starts,
- logging all the parameters, tags, metrics and artifacts under unified MLFlow run.

To make sure that the plugin discovery mechanism works, add `kedro-mlflow` as a dependencies to `src/requirements.in` and run:

```
$ pip-compile src/requirements.in > src/requirements.txt
$ kedro install
$ kedro mlflow init
```

Then, adjust the kedro-mlflow configuration and point to the mlflow server by editing `conf/local/mlflow.yml` and adjusting `mlflow_tracking_uri` key. Then, build the image:

```
$ kedro docker build
```

And re-push the image to the remote registry.

> If `kedro-mlflow` is not installed as dependency and configuration is not in place (missing `kedro mlflow init`), the MLflow experiment will not be initialized and available for pipeline tasks in Apache Airflow DAG.

3.4 Authentication to MLflow API

3.4.1 GoogleOAuth2

Given that Airflow has access to `GOOGLE_APPLICATION_CREDENTIALS` variable, it's possible to configure plugin to use Google service account to authenticate to secured MLflow API endpoint, by generating OAuth2 token.

All is required to have `GOOGLE_APPLICATION_CREDENTIALS` environment variable setup in Airflow installation and MLflow to be protected by Google as an issuer. The other thing is to have environment variable `GOOGLE_AUDIENCE` which indicates OAuth2 audience the token should be issued for.

Also, plugin configuration requires the following:

```
run_config:
  authentication:
    type: GoogleOAuth2
```

3.4.2 Vars

If you store your credentials in Airflow secrets backend, e.g. HashiCorp vault, it's possible to configure the plugin to use Airflow Variables as MLFlow API credentials.

Names of the variables need to match expected MLflow environment variable names, e.g. `MLFLOW_TRACKING_TOKEN`. You specify them in the authentiation config. For instance, setting up Basic Authentication requires the following:

```
run_config:
  authentication:
    type: Vars
    params: ["MLFLOW_TRACKING_USERNAME", "MLFLOW_TRACKING_PASSWORD"]
```

> NOTE: Authentication is an optional element and is used when starting MLflow experiment, so if MLflow is enabled in project configuration. It does not setup authentication inside Kedro nodes, this has to be handled by the project. Check GoogleOAuth2Handler class for details.

3.5 Authentication to Airflow API

Most of the operations provided by plugin uses Airflow API to either list dags or trigger them. By default, access to Airflow API is blocked and in order to enable it you need to modify `api.auth_backend` config variable as described in the documentation. Suggested setting for best plugin usage experience is to disable authentication on Airflow by setting value `airflow.api.auth.backend.default` and install middleware proxy blocking access to the API paths to users without expected JWT token in the header.

Sample configuration for istio filter and token issued by gcloud SDK can look like:

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: jwt-token-verification
spec:
  selector:
    matchLabels:
      component: webserver
  jwtRules:
  - issuer: https://accounts.google.com
    jwksUri: https://www.googleapis.com/oauth2/v3/certs
    audiences:
    - 32555940559.apps.googleusercontent.com # google token generator
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: airflow-api-access
spec:
  selector:
```

(continues on next page)

(continued from previous page)

```
    matchLabels:
      component: webserver
  rules:
  # allow all users to access UI, but not API
  # UI has its own access management
  - to:
    - operation:
        notPaths: ["/api/*"]
  # enforce JWT token on API
  - when:
    - key: request.auth.audiences
      values:
      - 32555940559.apps.googleusercontent.com # issued by gcloud sdk
    - key: request.auth.presenter
      values:
      - [service-account]@[google-project].iam.gserviceaccount.com
    to:
    - operation:
        paths: ["/api/*"]
```

This setup ensures that all requests to the API paths are validated by Istio by checking the content of JWT token issued by Google (using gcloud auth print-identity-token). In order to validate other tokens, modify `audiences` and `jwtRules` accordingly.

Token can be passed to `kedro airflow-k8s` commands by using environment variable `AIRFLOW_API_TOKEN`, for example:

```
$ AIRFLOW_API_TOKEN=eyJhbGci... kedro airflow-k8s list-pipelines 2> /dev/null
2021-08-13 14:59:13,635 - root - INFO - Registered hooks from 3 installed plugin(s):␣
→kedro-kubeflow-0.3.1, kedro-mlflow-0.7.2
2021-08-13 14:59:13,680 - root - INFO - Registered CLI hooks from 1 installed plugin(s):␣
→kedro-telemetry-0.1.1
2021-08-13 15:05:38,800 - kedro_telemetry.plugin - INFO - You have opted into product␣
→usage analytics.
2021-08-13 14:59:14,764 - kedro.framework.session.store - INFO - `read()` not␣
→implemented for `BaseSessionStore`. Assuming empty store.
Name     ID
-------  ------------------
model1   model1-branch-name
```

3.6 Spark integration

Kedro allows integration with pyspark as described in kedro documentation. kedro-airflow-k8s allows running such projects with Airflow, using the external Spark cluster as a computation environment. Not every pipeline node is executed on spark cluster, but only the ones which require spark environment.

3.6.1 Project handling

In order to make this happen, the following applies. Firstly plugin detects if any of the kedro nodes are pyspark related. All such nodes are logically grouped in a way that dependencies between all nodes in the pipeline are maintained. Such construction keeps data management between kedro nodes local within the cluster for performance matter, while enforcing order correctness. Secondly, plugin creates spark submit nodes inside the DAG to reflect spark related tasks, grouped in the previous step. Additionally, the cluster create and delete operator is setup so that the dedicated spark instance is ready for the sake of given job run. As the last step, the artifacts required by spark, namely cluster initialization shell script, project archive and kedro run python script are prepared.

3.6.2 Configuration

`spark` configuration is a part of a `run_config`. This plugin supports Google Dataproc, but it's also possible to provide custom operators via external factory.

Google Dataproc

To configure Dataproc with the project, set `run_config.spark.type` as `dataproc`. Use `cluster_config` to provide dictionary that describes the cluster as required by Airflow Dataproc operators. Checking with Google Dataproc REST API is helpful.

Kubernetes

It's possible to execute kedro spark jobs on K8S. In this case, there's no external cluster that's started for the purpose of the task. Instead, user has to provide image, containing both spark and kedro project. Usually such image is created with `docker-image-tool.sh`, which is a part of spark distribution. Example image building process may look like this:

```
cp -r $SPARK_HOME /tmp
cd $KEDRO_PROJECT_HOME
cp -r . /tmp/spark-3.1.2-bin-hadoop3.2
/tmp/spark-3.1.2-bin-hadoop3.2/bin/docker-image-tool.sh -t $COMMIT_SHA -p Dockerfile -r
→$REGISTRY_IMAGE -b java_image_tag=14-slim build
```

There's a manual work to be done with the `Dockerfile` first. One approach is to use one of the templates provided by spark distribution and merge it with the `Dockerfile` of kedro project. It's important that image contain kedro project with all of it's contents and it's installed as a package on a system level together with all the requirements. Any supplementary jars required by the project can be included as well, unless should be fetched during the job execution from the external location.

It's also required to provide runner script inside the image. This script is provided as an entry point to `spark-submit`. The script should accept `run` command, `--env` kedro application argument, `--node` as a comma-separated list of kedro nodes names to be executed and `--runner=ThreadRunner`. It should initialize kedro session and run project with given arguments.

> `Dockerfile` should execute spark entrypoint on start. Script is provided as an argument to
> `spark-submit`

Example script template is provided inside the plugin sources in `src/kedro_airflow_k8s/templates/spark_run.py.tpl`. The script is delegating invocation directly to kedro.

Custom configuration

In order to provide one's own operators it's sufficient to mark `run_config.spark.type` as `custom`, and provide `run_config.spark.operator_factory` with the name of the custom class that acts as the operator factory. The class has to be available on the path when executing `kedro airflow-k8s` commands.

The easiest way to start is to derive from `kero_airflow_k8s.template_helper.SparkOperatorFactoryBase`. The following methods have to be provided:

- create_cluster_operator - returns string with the create cluster operator
- delete_cluster_operator - returns string with the delete cluster operator
- submit_operator - returns string with the submit job operator
- imports_statement - returns string with the full import statement of all required items from the previous methods

Custom initialization script

`run_config.spark.user_init_path` allows configuring the way the cluster is initialized. Plugin delivers initialization script that's aligned with the project artifacts. The script can be prepended with custom logic, to support the cases like custom package repository setup. `run_config.spark.user_post_init_path` additionally allows appending to initialization script part of the shell script. Scripts can use environment variable `PROJECT_HOME` in order to refer to project location on the cluster. It's required the paths to be relative to the project `src` path.

3.6.3 Detection of spark nodes

As the part of the plugin's process is to detect spark based nodes, the following rules apply:

- if the node is tagged with `kedro-airflow-k8s:group:pyspark` it's considered as a spark node - this allows arbitrary user selection of node to be executed by spark
- if any of the node's input or output is of type `pyspark.sql.dataframe.DataFrame` it's considered as a spark node - detection happens based on the type hints
- if any of the node's input or output is present in the data catalog as one of the `SparkDataSet`, `SparkHiveDataSet`, `SparkJDBCDataSet` it's considered as a spark node
- if none of the above applies, but logical group of spark nodes provide data as input to the node and the node provides the data as the input to the group it's considered as a spark node
- if none of the above applies, the node is considered as the `default` and it's put into DAG as usual

FOUR

INDICES AND TABLES

- genindex
- modindex
- search