Laboratoire Apache Spark sur Kubernetes

Objectifs d'apprentissage

Dans ce laboratoire, vous allez :

- Créez ici un pod Kubernetes (un ensemble de conteneurs exécutés dans Kubernetes) contenant Apache Spark que nous utilisons pour soumettre des tâches à Kubernetes
- Soumettre des tâches Apache Spark à Kubernetes

Aperçu

Bienvenue dans le laboratoire sur la soumission d'applications Apache Spark à un cluster Kubernetes. Cet exercice est simple grâce au nouveau planificateur Kubernetes natif qui a été ajouté à Spark récemment.

Kubernetes est un orchestrateur de conteneurs qui permet de programmer des millions de conteneurs « docker » sur d'énormes clusters de calcul contenant des milliers de nœuds de calcul. Initialement inventé et open source par Google, Kubernetes est devenu le standard de facto pour le développement et le déploiement d'applications cloud natives au sein et en dehors d'IBM. Avec RedHat OpenShift, IBM est le leader du cloud hybride Kubernetes et fait partie des trois premières entreprises contribuant à la base de code open source de Kubernetes.

Prérequis

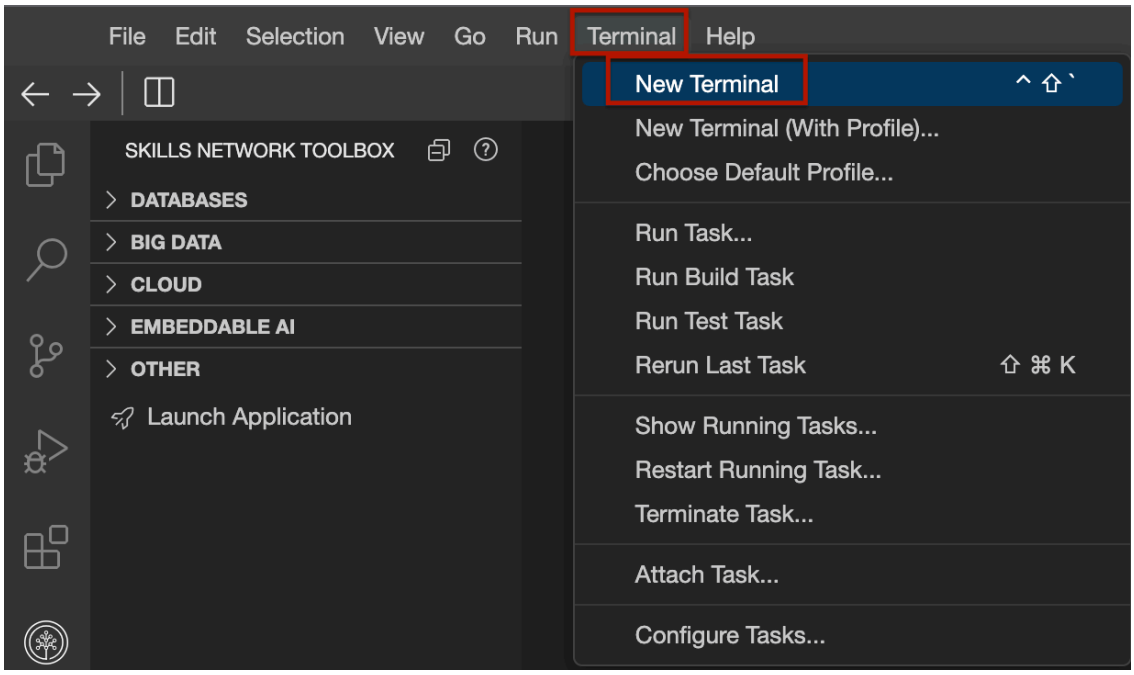
Remarque : si vous exécutez ce laboratoire dans l'environnement de laboratoire Skillsnetwort, tous les prérequis sont déjà installés pour vous

Les seuls prérequis pour ce laboratoire sont :

- Une installation *Docker fonctionnelle*
- Une installation *Kubernetes fonctionnelle*
- L' outil de ligne de commande *git*

Installation

1. Sur le côté droit de ces instructions, vous verrez l'IDE Theia. Sélectionnez l' onglet *Lab* . *Dans la barre de menu, sélectionnez Terminal>Nouveau terminal* .



2. Veuillez entrer la commande suivante dans le terminal pour obtenir le dernier code.

   ```
   git clone https://github.com/ibm-developer-skills-network/fgskh-new_horizons.git
   ```

3. Changez le répertoire vers le code téléchargé.

   ```
   cd fgskh-new_horizons
   ```

4. Ajoutez un alias à `kubectl`. Cela vous aidera à simplement taper `k` au lieu de `kubectl`.

   ```
   alias k='kubectl'
   ```

5. Enregistrez l'espace de noms actuel dans une variable d'environnement qui sera utilisée ultérieurement

   ```
   my_namespace=$(kubectl config view --minify -o jsonpath='{..namespace}')
   ```

Déployer le pod Apache Spark Kubernetes

1. Installer le POD Apache Spark

   ```
   k apply -f spark/pod_spark.yaml
   ```

2. Il est maintenant temps de vérifier l'état du Pod en exécutant la commande suivante.

   ```
   k get po
   ```

Si vous voyez le résultat suivant, cela signifie que le Pod n'est pas encore disponible et que vous devez attendre un peu.

```
NAME    READY   STATUS              RESTARTS   AGE
spark   0/2     ContainerCreating   0          29s
```

3. Attendez quelques secondes et émettez à nouveau la commande après un certain temps.

   ```
   k get po
   ```

   Veuillez répéter l'étape 2 jusqu'à ce que vous obteniez un `STATUS`qui reflète qu'il s'agit de `Running`.

4. Vous devriez voir un résultat tel que celui indiqué ci-dessous. L' `AGE`attribut peut être différent, selon le temps qu'il vous a fallu pour le faire fonctionner.

```
NAME  READY   STATUS    RESTARTS   AGE
spark 2/2     Running   0          10m
```

   In case you see the following status you need to delete the pod and start over again later as this usually happens when the image registry is unreliable or offline.

```
NAME   READY   STATUS              RESTARTS   AGE
spark  0/2     ImagePullBackOff    0          29s
```

5. Just in this case please delete the pod:

```
k delete po spark
```

Then start over:

```
k apply -f spark/pod_spark.yaml
```

Again, regularly check status:

```
k get po
```

Note that this Pod is called *spark* and contains two containers *(2/2)* of which are both in status *Running*. Please also note that Kubernetes automatically *RESTARTS* failed pods - this hasn't happened here so far. Most probably because the *AGE* of this pod is only 10 minutes.

Submit Apache Spark jobs to Kubernetes

Now it is time to run a command inside the *spark* container of this Pod.
The command *exec* is told to provide access to the container called *spark* (-c). With – we execute a command, in this example we just echo a message.

```
k exec spark -c spark  -- echo "Hello from inside the container"
```

You just ran a command in *spark* container residing in *spark* pod inside Kubernetes. We will use this container to submit Spark applications to the Kubernetes cluster. This container is based on an image with the Apache Spark distribution and the *kubectl* command pre-installed.

If you are interested you can have a look at the Dockerfile to understand what's really inside.

You can also check out the pod.yaml. You'll notice that it contains two containers. One is Apache Spark, another one is providing a Kubernetes Proxy - a so called side car container - allowing to interact with the Kubernetes cluster from inside a Pod.

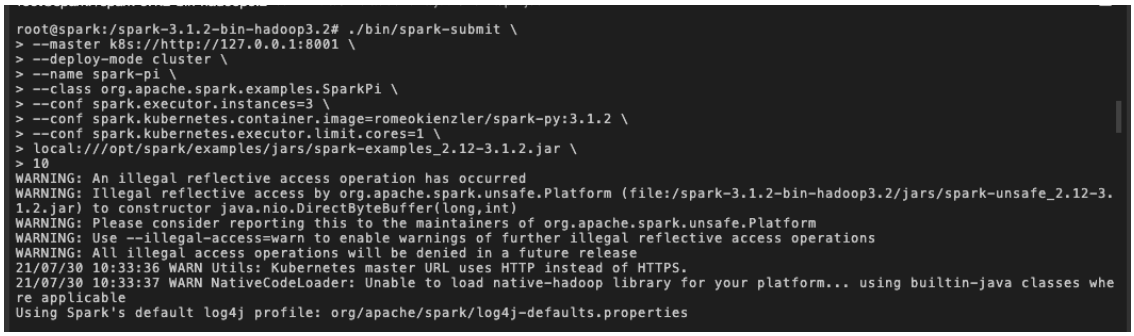
Inside the container you can use the *spark-submit* command which makes use of the new native Kubernetes scheduler that has been added to Spark recently.

The following command submits the *SparkPi* sample application to the cluster. SparkPi computes Pi and the more iterations you run, the more precise it gets:

```
k exec spark -c spark -- ./bin/spark-submit \
--master k8s://http://127.0.0.1:8001 \
--deploy-mode cluster \
--name spark-pi \
--class org.apache.spark.examples.SparkPi \
--conf spark.executor.instances=1 \
--conf spark.kubernetes.container.image=romeokienzler/spark-py:3.1.2 \
--conf spark.kubernetes.executor.request.cores=0.2 \
```

```
    --conf spark.kubernetes.executor.limit.cores=0.3 \
    --conf spark.kubernetes.driver.request.cores=0.2 \
    --conf spark.kubernetes.driver.limit.cores=0.3 \
    --conf spark.driver.memory=512m \
    --conf spark.kubernetes.namespace=${my_namespace} \
    local:///opt/spark/examples/jars/spark-examples_2.12-3.1.2.jar \
    10
```

You should see output like below, please ignore the WARNINGS. Unless you don't see ERRORS all is fine:



Understanding the spark-submit command

So let's have a look what's going on here:

- *./bin/spark-submit* is the command to submit applications to a Apache Spark cluster
- *–master k8s://http://127.0.0.1:8001* is the address of the Kubernetes API server - the way *kubectl* but also the Apache Spark native Kubernetes scheduler interacts with the Kubernetes cluster
- *–name spark-pi* provides a name for the job and the subsequent Pods created by the Apache Spark native Kubernetes scheduler are prefixed with that name
- *–class org.apache.spark.examples.SparkPi* provides the canonical name for the Spark application to run (Java package and class name)
- *–conf spark.executor.instances=1* tells the Apache Spark native Kubernetes scheduler how many Pods it has to create to parallelize the application. Note that on this single node development Kubernetes cluster increasing this number doesn't make any sense (besides adding overhead for parallelization)
- *–conf spark.kubernetes.container.image=romeokienzler/spark-py:3.1.2* tells the Apache Spark native Kubernetes scheduler which container image it should use for creating the driver and executor Pods. This image can be custom build using the provided Dockerfiles in *kubernetes/dockerfiles/spark/* and *bin/docker-image-tool.sh* in the Apache Spark distribution
- *–conf spark.kubernetes.executor.limit.cores=0.3* tells the Apache Spark native Kubernetes scheduler to set the CPU core limit to only use 0.3 core per executor Pod
- *–conf spark.kubernetes.driver.limit.cores=0.3* tells the Apache Spark native Kubernetes scheduler to set the CPU core limit to only use 0.3 core for the driver Pod
- *–conf spark.driver.memory=512m* tells the Apache Spark native Kubernetes scheduler to set the memory limit to only use 512MBs for the driver Pod
- *–conf spark.kubernetes.namespace=${my_namespace}* tells the Apache Spark native Kubernetes scheduler to set the namespace to *my_namespace* environment variable that we set before.
- *local:///opt/spark/examples/jars/spark-examples_2.12-3.1.2.jar* indicates the *jar* file the application is contained in. Note that the *local://* prefix addresses a path within the container images provided by the *spark.kubernetes.container.image* option. Since we're using a *jar* provided by the Apache Spark distribution this is not a problem, otherwise the *spark.kubernetes.file.upload.path* option has to be set and an appropriate storage subsystem has to be configured, as described in the documentation
- *10* tells the application to run for *10* iterations, then output the computed value of *Pi*

Please see the documentation for a full list of available parameters.

Monitor the Spark application in a parallel terminal

Once this command runs you can *open a second terminal window* within Theia and issue the following command:

> **Note:** To see at least one executor, run the below-mentioned command while the other terminal is still executing spark-submit command

```
kubectl get po
```

This will show you the additional Pods being created by the Apache Spark native Kubernetes scheduler - one driver and at least one executor. Note that with only one executor the driver may run the executor within its own pod. Here's an example when using one executor running separately from the driver pod (exact IDs replaced by X and Y for readability):

```
NAME                READY STATUS    RESTARTS AGE
spark               2/2   Running   0        28m
spark-pi-X-exec-1 1/1   Running   0        33s
spark-pi-X-driver 1/1   Running   0        44s
spark-pi-Y-driver 0/1   Completed 0        12m
```

You can see that Pod *spark-pi-Y-driver* is in status *Completed*, from a single executor run twelve minutes ago and that there are one driver and three executors actually running for job *spark-pi-X- ...*

To check the job's elapsed time just execute (you need to replace the Pod name of course with the one on your system):

Please make sure you run the following code in the newly created terminal window which allows you to execute commands within the Spark driver running in a POD.

> **Note:** Replace the ID in the Spark-pi-ID-driver with the one which is created by you. For example: if your pod is spark-pi-6f62d17a800beb3e-driver then replace ID with 6f62d17a800beb3e

```
kubectl logs spark-pi-6f62d17a800beb3e-driver |grep "Job 0 finished:"
```

You should get something like:

```
Job 0 finished: reduce at SparkPi.scala:38, took 8.446024 s
```

If you are interested in knowing what value for *Pi* the application came up with just issue:

> **Note:** Replace the ID in the Spark-pi-ID-driver with the one which is created by you. For example: if your pod is spark-pi-6f62d17a800beb3e-driver then replace ID with 6f62d17a800beb3e

```
kubectl logs spark-pi-6f62d17a800beb3e-driver |grep "Pi is roughly "
```

And you'll see something like:

```
Pi is roughly 3.1416551416551415
```

Experiment yourself

Now you can play around with values for *spark.executor.instances*, *spark.kubernetes.executor.limit.cores=0.5* (0.1 is also a valid number) and number of iterations and see how it affects runtime and precision of the

outcome. Just make sure you don’t exceed SkillsNetwork resource quota limit. Watch `Kubectl logs [driver pod]` to check logs for exceeding quota.

This concludes this lab.

Summary

In this lab you’ve learned how to create an Apache Spark client POD within the kubernetes cluster to submit jobs. Then, you’ve used the spark-submit command to create a job running inside this Kubernetes cluster. You are now able to scale your Apache Spark jobs on any Kubernetes cluster running in the cloud or in your data center to thousands of nodes, CPUs and GB of main memory.

Credits

Thanks a lot to Aije Egwaikhide for testing and her feedback to improve the lab.