|  |  |
| --- | --- |
| **NOM** | Kihm |
| **Prénom** | Antonny |
| **Date de naissance** | 14/11/1978 |

**Copie à rendre**

**Bachelor DevOps 22/27**

**Documents à compléter et à rendre**

**Partie 1 : Automatisation du déploiement d’infrastructure dans le Cloud**

1. Écrivez en code la création de : (**6 points)**

* Apache Spark Cluster
* Mongodb

vous pouvez utiliser le fournisseur/plateforme que vous préférez. Ex : Terraforme avec aws/Azure/google cloud. CloudFormation avec aws, etc.

7

**Le code relatif au travail détaillé plus bas dans chacune des parties est présent sur*:***

<https://github.com/ravayak/studi-ecf>

*Veuillez vous référer aux dossiers correspondants à chacune des parties (Apache Spark, Application Python et mongodb).*

**Note sur les conventions typographiques adoptées dans ce document:**

Tous les exemples de code citéssont *en italique*

Toutes les sorties de console présentées sont en courrier new 10

1 **Automatisation du déploiement d’infrastructure dans le Cloud**

1. **Création d’apache Spark cluster**

Afin d’automatiser la création et le déploiement d’Apache Spark, notre choix s’est porté sur l’outil Terraform. Concernant la plateforme de déploiement, nous nous sommes orientés vers AWS [EMR](https://aws.amazon.com/fr/emr/).   
En quelques mots, Amazon EMR (Elastic MapReduce) est un service de cloud computing qui permet d'exécuter des frameworks de traitement de données, tels que Apache Spark sur des ressources évolutives dans le cloud. Ce service est donc parfaitement adapté à nous besoins.

EMR se compose de nodes qui sont de trois types:

- primary: coordonne la distribution des tâches et des données, suit le statut des tâches et monitore la santé du cluster.

- core: exécute des tâches et les stocke dans HDFS (hadoop Distributed File System)

- task: exécute des tâches mais ne stocke pas de données  
Un des avantages d’EMR est qu’il s’intègre très bien avec d’autres services AWS comme S3 pour stocker les datas, EC2 pour provisionner les machines virtuelles pour les nodes workers et, VPC pour la gestion de notre réseau.

Nous tirons parti de cette intégration via la configuration de notre cluster emr dans notre fichier principal terraform.

Comme nous le verrons, la mise en place de cette solution nécessite aussi la configuration indépendante de trois services (iam, s3 et security) afin d’assurer une mise en place cohérente et opérationnelle. Nous détaillerons chaque point par la suite.

**Fichiers principaux:**

**terraform.tfvars**

Ce fichier va alimenter les variables définies dans variables.tf.

Nous pouvons configurer tous les paramètres nécessaires pour le cluster EMR. ,La région, le nombre d'instances pour le nœud esclave, le type d'instance pour le maître/esclave, la version de Spark, l'ID de sous-réseau (<subnet\_id>), l'ID de VPC(<vpc\_id>, le nom de la paire de clés (key\_name). ingress\_cidr\_blocks est à définir avec une adresse IP plutôt que d''autoriser tout le traffic avec 0.0.0.0/0 pour des raisons de sécurité.

Nous partons du principe qu’au préalable un VPC configuré est disponible. Nous ne détaillerons pas ici la création de ce dernier, qui n’entre pas dans le cadre de l’exercice et alourdirait la présentation.

**main.tf**

C’est le script principal qui va exécuter nos 4 modules.

Rappelons que Terraform analyse le graphe des dépendances et détermine un ordre d'exécution optimal. Autrement dit, l'ordre d'exécution est déterminé par les dépendances explicites entre les ressources et les modules. Si un module ou une ressource dépend d'une sortie d'un autre module, Terraform respectera ces dépendances lors de l'exécution.

Nos 4 modules (s3, iam, security et emr) sont explicitement référencés par leurs propriétés «source».

Toutes les variables référencées ont normalement été définies dans terraform.tfvars.

**Modules**

**Module emr**

Ce script Terraform déploie un cluster Spark sur Amazon EMR (Elastic MapReduce).

Rappel:

AWS EMR est conçu pour prendre en charge plusieurs frameworks de traitement distribué, dont Apache Spark, Apache Hadoop MapReduce, Apache HBase, etc. Dans le cas spécifique de notre script, nous utilisons EMR pour déployer un cluster Spark, mais ce cluster EMR inclut également [Hadoop](https://fr.wikipedia.org/wiki/Hadoop), notamment le système de fichiers distribué [HDFS](https://fr.wikipedia.org/wiki/Hadoop" \l "Hadoop_Distributed_File_System) et les capacités de traitement [MapReduce](https://fr.wikipedia.org/wiki/MapReduce) (Hadoop et Apache Spark sont deux frameworks complémentaires qui peuvent être utilisés ensemble dans le domaine du traitement des données distribuées).

Configuration du Cluster EMR :

Le cluster EMR est configuré avec un nom, une version de distribution spécifique (release\_label), des applications (Spark dans ce cas), et d'autres paramètres.

La protection contre la «résiliation» (termination\_protection) est désactivée, et le cluster reste actif même en l'absence de tâches (keep\_job\_flow\_alive\_when\_no\_steps).

EMR prend en charge d'autres composants liés à Hadoop pour fournir un environnement complet et géré pour le traitement et le stockage distribués de données. Cela permet de bénéficier des fonctionnalités de Spark tout en tirant parti de la flexibilité et des avantages du service EMR.

Configuration des Instances EC2 :

Les attributs EC2, tels que le sous-réseau, la clé SSH, les groupes de sécurité EC2, et le profil IAM, sont définis.

La configuration des volumes EBS pour le nœud maître et les nœuds esclaves est également spécifiée.

Configuration des Rôles IAM :

Des rôles IAM pour le service EMR (emr\_service\_role) et l'auto-scaling (emr\_autoscaling\_role) sont spécifiés.

Bootstrap Actions :

Une action de démarrage (bootstrap action) est définie pour exécuter des actions spécifiques lors du lancement du cluster.

Deux «steps» Spark sont spécifiés.

La premier copie un script depuis S3 vers le répertoire home d'Hadoop, et le deuxième exécute le script en question pour configurer PySpark avec Conda. Nous verrons ces deux scripts plus en détail dans la section scripts.

Une petite précision concernant les 2 lignes 36 et 45:

*jar = "command-runner.jar"*

«command-runner.jar» est un utilitaire fourni par AWS EMR. Il est utilisé pour exécuter des commandes et des scripts sur les instances d'un cluster EMR. En l’occurence il est utilisé ici pour copier le fichier le fichier pyspark\_quick\_setup.sh (ligne 36) et l’exécuter (ligne 45).

Configuration JSON de Spark :

Une configuration JSON est spécifiée pour les paramètres de configuration Spark, tels que la maximisation des allocations de ressources (maximizeResourceAllocation) et l’allocation dynamique (spark.dynamicAllocation.enabled).

Instance Groups :

Deux groupes d'instances sont définis, un pour le nœud maître (emr-master) et l'autre pour les nœuds esclaves (emr-core), chacun avec ses propres configurations.

**Module s3:**

Le module s3 ne présente aucune complexité. Nous créons un bucket qui va contenir les 2 scripts de démarrage ("scripts/bootstrap\_actions.sh" et "scripts/pyspark\_quick\_setup.sh") qui sont détaillés plus bas.

**Module iam**

Ce script définit plusieurs rôles IAM pour différentes composantes liées à Amazon EMR.

Plus précisément, ces rôles IAM sont utilisés pour fournir des permissions spécifiques aux services et instances associés au cluster EMR, y compris le service EMR lui-même, l'auto-scaling d'EMR, et les instances EC2 utilisées par EMR.

Rôle IAM pour EMR Service:

Nous créons tout d’abord un aws\_iam\_policy\_document afin de définir les autorisations nécessaires pour que notre service elasticmapreduce.amazonaws.com assume le rôle IAM.

Rappel : Le rôle IAM est une entité AWS qui définit les autorisations et les politiques de sécurité pour les ressources AWS.

Nous spécifions les permissions du rôle IAM appelé "EMR\_ServiceRole".

*assume\_role\_policy = "${data.aws\_iam\_policy\_document.emr\_assume\_role.json}" :* Cela spécifie la politique d'attribution de rôle (assume role policy) pour le rôle IAM. La valeur est obtenue en référençant le aws\_iam\_policy\_document IAM que nous avons défini précédemment avec data.aws\_iam\_policy\_document.emr\_assume\_role. Cette politique indique quel service est autorisé à assumer ce rôle (dans ce cas, "elasticmapreduce.amazonaws.com").

La policy AWS AmazonElasticMapReduceRole est ensuite attachée au rôle "EMR\_ServiceRole".

Rôle IAM pour Auto-scaling:

Nous créons à nouveau aws\_iam\_policy\_document afin de spécifier les permissions du rôle d'auto-scaling EMR.

Un rôle IAM appelé "EMR\_AutoScalingRole" est créé avec la politique d'assumption définie par le aws\_iam\_policy\_document plus haut.

La politique AWS AmazonElasticMapReduceforAutoScalingRole est attachée au rôle "EMR\_AutoScalingRole".

Rôle IAM pour EC2:

A nouveau, nous définissons un aws\_iam\_policy\_document afin de pour spécifier les permissions du rôle EC2.

Un rôle IAM appelé "EC2\_InstanceProfile" est créé avec la politique d'assumption définie par le aws\_iam\_policy\_document plus haut.

La politique AWS AmazonElasticMapReduceforEC2Role est attachée au rôle "EC2\_InstanceProfile".

Maintenant que nous avons défini un rôle, nous créons un «profil d'instance IAM» et nous le lions au rôle ("EC2\_InstanceProfile").

**Module Security**

Ce module contient la configuration des groupes de sécurité AWS pour les instances maître et esclave d'un cluster EMR.

Groupe de sécurité pour le maître EMR (aws\_security\_group.emr\_master) :

Nom et description : Le groupe de sécurité est nommé "${var.name} - EMR-master" avec une description indiquant qu'il s'agit d'un groupe de sécurité pour le maître EMR.

VPC : Le groupe de sécurité est associé au VPC spécifié par la variable var.vpc\_id.

Révocation des règles lors de la suppression : Les règles du groupe de sécurité seront révoquées lors de sa suppression.

Règles ingress (entrantes):

Port 22 (SSH) : Autorise le trafic TCP sur le port 22 depuis l'adresse IP spécifiée par la variable var.ingress\_cidr\_blocks.

Ports 4040, 8888, 20888 : Autorise le trafic TCP sur ces ports depuis l'adresse IP spécifiée par la variable var.ingress\_cidr\_blocks.   
Les ports 4040, 8888 et 20888 sont généralement associés à des services spécifiques dans l'écosystème Apache Spark, qui est utilisé dans le contexte d'un cluster EMR.

Port 4040 :

Ce port est souvent utilisé pour l'interface web Spark UI, qui fournit des informations détaillées sur les applications Spark en cours d'exécution sur le cluster.

Autoriser le trafic sur ce port permet aux utilisateurs d'accéder à l'interface web Spark UI depuis l'adresse IP spécifiée par la variable var.ingress\_cidr\_blocks.

Port 8888 :

Le port 8888 est couramment utilisé pour les notebooks Jupyter, qui peuvent être utilisés pour le développement interactif avec Spark en utilisant des langages comme Python (PySpark).

Autoriser le trafic sur ce port permet aux utilisateurs d'accéder aux notebooks Jupyter depuis l'adresse IP spécifiée par la variable var.ingress\_cidr\_blocks.

Port 20888 :

Ce port est parfois utilisé pour l'interface web JobHistoryServer, qui fournit des informations sur les travaux exécutés sur le cluster.

Autoriser le trafic sur ce port permet aux utilisateurs d'accéder à l'interface web JobHistoryServer depuis l'adresse IP spécifiée par la variable var.ingress\_cidr\_blocks.

Règles egress (sortantes):

Toutes les connexions sortantes sont autorisées ("-1") vers n'importe quelle adresse IP ("0.0.0.0/0").

Tags :

Un tag "Name" est définie avec la valeur "EMR\_master".

Groupe de sécurité pour l'esclave EMR (aws\_security\_group.emr\_slave) :

Nom et description : Le groupe de sécurité est nommé "${var.name} - EMR-slave" avec une description indiquant qu'il s'agit d'un groupe de sécurité pour l'esclave EMR.

VPC : Le groupe de sécurité est associé au VPC spécifié par la variable var.vpc\_id.

Révocation des règles lors de la suppression : Les règles du groupe de sécurité seront révoquées lors de sa suppression.

Règles ingress :

Port 22 (SSH) : Autorise le trafic TCP sur le port 22 depuis l'adresse IP spécifiée par la variable var.ingress\_cidr\_blocks.

Règles egress :

Toutes les connexions sortantes sont autorisées ("-1") vers n'importe quelle adresse IP ("0.0.0.0/0").

Balises :

Enfin un tag "Name" est définie avec la valeur "EMR\_slave".

**Scripts**

Avant d’entrer plus en détails, faisons un point de rappel général.

Conda est un gestionnaire de packages, un environnement virtuel et un système de gestion des dépendances. Il est principalement utilisé dans le domaine des data science et de l'apprentissage automatique pour créer des environnements isolés contenant différentes versions de bibliothèques et d'outils. Conda peut installer des packages depuis différentes sources, y compris le référentiel Anaconda, qui propose un ensemble étendu de packages scientifiques.

Miniconda est une version minimaliste de Conda. C'est un installateur minimal qui comprend uniquement Conda, Python et quelques autres utilitaires de base. L'avantage de Miniconda est qu'il est plus léger que l'installation complète d'Anaconda. Nous pourrons ajouter les packages necessaires au fur et à mesure, plutôt que d'installer un ensemble prédéfini de packages.

Nos scripts configurent un environnement Spark sur une machine où Miniconda est installé. Voici comment cela se déroule :

**bootstrap\_actions.sh:**

Le script installe Miniconda s'il n'est pas déjà installé.

Il ajoute le chemin de l'interpréteur Python de Conda à la configuration de Spark.

conda va alors configurer l’environnement pour utiliser Jupyter Notebook avec PySpark, pandas et seaborn.  
Rappellons que pandas est une bibliothèque open-source qu simplifie le chargement, la manipulation, la transformation et l'analyse des données, en offrant un ensemble de fonctionnalités similaire à celui des bases de données relationnelles, mais dans un contexte Python.

Seaborn est une bibliothèque de visualisation de données basée sur Matplotlib, une autre bibliothèque de visualisation Python. Seaborn offre une interface de haut niveau pour créer des graphiques statistiques informatifs et attrayants. Elle est particulièrement adaptée à la visualisation de jeux de données statistiques complexes, avec une syntaxe concise et des options de personnalisation étendues. Seaborn fonctionne bien avec les objets de données Pandas, ce qui en fait un choix populaire pour la visualisation de données dans le domaine de l'analyse de données et de l'apprentissage automatique.

Le script met ensuite à jour pip et installe findspark. Cette bibliothèque permet de localiser automatiquement l'installation Spark sur la machine, ajoutant le chemin approprié aux variables d'environnement, ce qui simplifie l'utilisation de Spark avec Python.

**pyspark\_quick\_setup.sh**

Ce script bash configure l'environnement Spark pour utiliser Conda avec PySpark.

*echo -e "\nexport PYSPARK\_PYTHON=/home/hadoop/conda/bin/python" >> /etc/spark/conf/spark-env.sh :*

Définit la variable d'environnement PYSPARK\_PYTHON pour indiquer à Spark le chemin vers l'interpréteur Python de Conda.

*echo "export PYSPARK\_DRIVER\_PYTHON=/home/hadoop/conda/bin/jupyter" >> /etc/spark/conf/spark-env.sh :*

Définit la variable PYSPARK\_DRIVER\_PYTHON pour indiquer à Spark le chemin vers l'interpréteur Python à utiliser pour le pilote (driver).

*echo "export PYSPARK\_DRIVER\_PYTHON\_OPTS='notebook --no-browser --port 8888 --ip=\"0.0.0.0\"'" >> /etc/spark/conf/spark-env.sh :*

Définit les options pour le démarrage de Jupyter Notebook dans le pilote PySpark.

Spécifie que le notebook doit écouter sur toutes les interfaces (0.0.0.0), sur le port 8888, et sans l'ouverture automatique du navigateur.

En résumé, ce script ajuste la configuration de Spark pour utiliser Conda avec PySpark et configure l'environnement pour le lancement d'un notebook Jupyter dans le contexte PySpark.

**Conclusion**

Nous avons désormais un environnement qui nous permet à tout moment de déployer apache Spark automatiquement sur AWS EMR avec terraform.

Il nous suffit d’exécuter «terraform init» et «terraform apply» pour déployer notre cluster.

Le nom et l'ID du cluster ainsi que l'adresse IP du nœud maître devraient apparaître dans la sortie si tout se passe comme prévu.

Nous pouvons par la suite nous connecter en SSH sur le nœud maître et commencer immédiatement à travailler avec PySpark.

1. **Création de MongoDB**

Le déploiement de MongoDB sera également réalisé sur AWS.

Nous répliquerons l’instance maitre avec deux autres serveurs, idéalement dans une zone géographique différente pour assurer une haute disponibilité.

Nous utiliserons également Terraform pour l'automatisation de la création des machines EC2, ainsi que [Ansible](https://www.ansible.com/) pour le provisionnement.

Nous aurons un cluster MongoDB composé de trois instances EC2 (de type t2.large mais adaptable selon les besoins).

Parmi ces instances, MongoDB va élire un serveur master ou primaire ainsi que deux serveurs appelés secondaires.

Afin que ces trois serveurs se partagent les mêmes données, nous allons devoir créer ce que MongoDB appelle un [replica set](https://docs.mongodb.com/manual/tutorial/deploy-replica-set/), soit un ensemble de données.

**Création des serveurs**

Le script principal (main.tf) ne présente pas de difficultés de compréhensions majeures cependant nous tâcherons de décrire globalement ce qui est configuré afin de clarifier le processus général de déploiement.

Globalement, le fichier déploie une instance EC2 avec MongoDB installé, et configure les règles de sécurité associées :

Instance EC2 MongoDB :

Type : t2.large

une instance avec des performances moyennes comme indiqué par Amazon:

«Les instances T2 sont un bon choix pour diverses charges de travail polyvalentes, notamment les micro-services, les applications interactives à faible latence, les bases de données petites et moyennes, les bureaux virtuels, les environnements de développement et de développement, les référentiels de code et les prototypes de produits.»

Bien entendu, l’instance sera à mettre à l’échelle en fonction de la charge de travail.

Système d'exploitation : Ubuntu Server 22.04 LTS

Volume de stockage racine : 100 Go (gp2)

Sécurité : Une règle de groupe de sécurité MongoDB est associée à l'instance, autorisant le trafic SSH, le trafic MongoDB sur le port 27017, et le trafic de réplication MongoDB sur le port 27019.

Une clé («TF\_KEY») est associée à l’instance.

Gestion des clés SSH :

Une paire de clés SSH est générée avec une clé privée RSA de 4096 bits.

La clé privée générée est stockée dans un fichier local nommé "TF\_key".

La clé publique est attachée à la paire de clés et utilisée pour accéder à l'instance EC2 grâce à: key\_name = "TF\_key" dans le bloc «resource "aws\_instance" "mongodb\_one"»

Groupe de sécurité MongoDB :

Un groupe de sécurité MongoDB est créé avec des règles entrantes et sortantes (ingress / egress).

Règle autorisant le trafic sortant vers toutes les destinations (0.0.0.0/0).

Règle autorisant le trafic SSH entrant depuis toutes les sources (0.0.0.0/0).

Règle autorisant le trafic MongoDB entrant depuis toutes les sources sur le port 27017.

Règle autorisant le trafic de réplication MongoDB entrant depuis toutes les sources sur le port 27019.

**Remarque**: Notons que l'on autorise ici toutes les provenances dans l'entrée cidr\_blocks du bloc egress. Il faut biensur dans les faits restreindre au maximum ces accès.

Notons au passage que nous avons défini deux variables AWS\_REGION et ENVIRONMENT dans variables.tf. D’autres variables auraient pu être définies bien entendu.

**Provisionning avec Ansible**

Nous allons maintenant provisionner le serveur MongoDB avec notre playbook.

Rappelons brièvement que Ansible est un outil d'automatisation de configurations. Il est principalement utilisé pour déployer des applications et configurer des serveurs de manière reproductible.

Ansible fonctionne sur la base d’un «playbook» qui est un fichier qui applique des rôles à un «inventory» pouvant contenir des variables, des tasks et des conditions. L’inventory étant l'inventaire des machines.

Inventory:

Notre inventory est très simplement composé du serveur maître et deux serveurs secondaires. Les adresses IP de chacun des serveurs restent bien entendu à compléter après avoir exécuté terraform apply.

Nous définissons deux groupes d’hôtes «db-mongodb-master» et «db-mongodb-slave» chaucun composé de leurs hôtes respectifs.

Playbook:

Notre playbook, composé seulement de quelques lignes, exécute le rôle «project.provision.mongodb».

Que réalise ce rôle ?

**Le rôle project.provision.mongodb réalise:**

· l’installation et création d'un service système MongoDB

· la préparation du fichier de configuration MongoDB

· l’activation de la réplication avec les autres hosts

· le démarrage de l'instance MongoDB

Installation et création d'un service système MongoDB

Nous utilisons le module apt pour installer le paquet mongodb-org.

Nous avons omis les phases préalables à l’installation de mongo, c’est à dire l’installation d’éventuelles dépendances, la mise à jour du cache apt, l’ajout de la clé d'authentification APT de MongoDB au trousseau de clés du système, l’ajout du référentiel MongoDB au fichier de sources APT uniquement s'il n'existe pas déjà, et ansi de suite, par soucis d’alléger la présentation.

Activation du service Mongod :

Nous activons et démarrons le service MongoDB.

Autorisation des connexions distances:

Nous autorisons toutes les connexions distantes en modifiant le fichier mongodb.conf via le module lineinfole.

Autorisation de l’authentification:

Nous autorisons l’authentification à MongoDB.

Réplication:

Afin d'activer la réplication, nous avons besoin de spécifier dans ce fichier, un nom de replica set (ici, rs0)

Cette réplication fonctionnera uniquement dans le cas où les serveurs peuvent communiquer entre eux.

Il est également important de sécuriser les échanges. C'est pourquoi nous allons également créer une clé qui aura pour but d'authentifier les serveurs discutant entre eux :

Préparation du fichier clé d'autorisation :

Utilisation de l'action locale pour générer un fichier de clé (avec OpenSSL) lorsque le type de réplica est "master".

Création du répertoire principal de MongoDB :

Utilisation du module file pour créer le répertoire "/home/mongodb" avec les permissions appropriées.

Copie de la clé vers les maîtres et les esclaves :

Utilisation du module copy pour copier la clé vers "/home/mongodb/mongodb-key" sur les maîtres et esclaves, avec les propriétés et permissions appropriées.

Ajout de la clé à la configuration MongoDB :

Utilisation du module lineinfile pour ajouter la référence au fichier de clé dans le fichier de configuration "/etc/mongod.conf" sur les serveurs

Redémarrage de MongoDB :

Nous redémarrons MongoDB afin de prendre en compte les changements réalisés.

**Conclusion**

L'ensemble de ces tâches contribue à la configuration et à l'activation de MongoDB sur les hôtes spécifiés, en prenant en compte la réplication et une sécurisation minimale.

Nous avons désormais une infrastructure mongoDB opérationnelle basée sur un serveur maître et deux serveurs secondaires.

**Partie 2 : Déploiement d’une application en continu**

1. Créez une application Python (hello word) avec Pyspark intégré **(1 point)**.
2. Dockerizez votre application Python **(2 points)**.
3. Écrivez le script qui build/test le Python et déployez le dans un container de votre choix **(3 points)** (vous pouvez utiliser Kubernetes ou DockerEE ou autre container services).
4. Création de l’application python avec spark intégré

from pyspark.sql import SparkSession

def main():

*# Initialisation*

spark = SparkSession.builder \

.appName("HelloWorld") \

.master("spark://hostname:7077") \

.getOrCreate()

*# Création d'un un ensemble de données simple*

data = [("Hello",), ("World",)]

columns = ["message"]

df = spark.createDataFrame(data, columns)

*# Affichage du contenu de l'ensemble de données*

df.show()

*# Arrêt de la session Spark*

spark.stop()

if \_\_name\_\_ == "\_\_main\_\_":

main()

PySpark est l'API Python pour Apache Spark. Il permet d'effectuer un traitement de données en temps réel à grande échelle dans un environnement distribué en utilisant Python. Il offre également une interface interactive appelée PySpark shell pour analyser les données de manière interactive.

Quelques commentaires sur le code même s’il ne présente aucune difficulté:

*SparkSession.builder* crée un constructeur de sessions Spark.

*.appName("HelloWorld")* définit le nom de l'application Spark.

*df = spark.createDataFrame(data, columns)* crée un DataFrame à partir des données et des colonnes spécifiées («hello world»).

*df.show()* affiche le contenu du DataFrame à la console. Dans cet exemple, il affichera quelque chose comme :

+-------+

|message|

+-------+

| Hello|

| World |

+-------+

*spark.stop()* arrête la session Spark, libérant les ressources associées.

1. Dockerisation de l’application

Le Dockerfile ne présente lui non plus aucune difficulté majeure de compréhension.

Un mot sur la ligne:

RUN pip install --no-cache-dir -r requirements.txt

ou -*-no-cache-dir* indique à pip de ne pas utiliser de cache lors de l'installation des dépendances, ce qui peut être utile pour éviter des problèmes potentiels liés à des versions obsolètes ou incorrectes dans le cache.

Le cache est généralement utilisé pour stocker les fichiers téléchargés lors des installations précédentes afin d'accélérer les installations ultérieures.

Avec *-r,* on spécifie un fichier texte contenant la liste de dépendances à installer (ici en l’occurence pyspark)

1. Script de build et déploiement

Le script principal gère le build et le déploiement de l’application dans kubernetes.   
Il build d’abord l’image sous le tag «ynno/python-app:latest» et push l’image sur le registry.

Kubernetes «applique» ensuite le déploiement décrit dans le fichier «deployment.yaml»

Quelques commentaires sur ce fichier:

Metadata:

name: python-app-deployment: Nom du déploiement.

labels: Étiquettes associées au déploiement (app: python-app, tier: backend, phase: production).

Spécifications du déploiement:

replicas: 1: Indique qu'il doit y avoir un seul replica de l'application.

selector: Définit comment le déploiement trouve les Pods à gérer.

matchLabels: Les Pods gérés doivent avoir les mêmes étiquettes que celles spécifiées ici.

template: Spécifie le modèle pour créer les Pods.

metadata.labels: Étiquettes pour les Pods créés.

spec: Configuration des Pods créés.

securityContext: Configurations de sécurité pour les Pods.

runAsUser: 1000 et runAsGroup: 3000: L'utilisateur et le groupe sous lesquels l'application s'exécutera.

Conteneur:

name: python-app: Nom du conteneur.

image: ynno/python-app:latest: Image Docker à utiliser.

imagePullPolicy: Always: Politique pour toujours pull la dernière version de l'image.

resources: Limites et demandes de ressources pour le conteneur (mémoire et CPU).

securityContext: Configurations de sécurité spécifiques au conteneur.

**Conclusion**

Ces configurations définissent donc un déploiement avec un seul replica de l'application Python. L'application s'exécute avec des restrictions de sécurité telles que l'exécution en tant qu'utilisateur non root et la limitation des «privilèges d'escalade» (c’est à dire la restriction de la capacité d'un conteneur à augmenter ses privilèges au-delà de ce qui lui est initialement accordé).

Bien entendu l’application nécessiterait d’être paramétrée correctement dans un véritable environnement de production. Les ressources que nous avons ici définies (processeur, mémoire, nombre de replicas) le sont à titre d’exemple mais ne réflètent pas un paramétrage destiné à un environnement de production réel!

**Partie 3 : Supervision des services déployés**

1. Mettez en place deux systèmes de monitoring (montrez des exemples pour chacun) :

* Monitoring de spark jobs (ex. : spark web UI) **(3 points)**.
* Monitoring de mongodb **(3 points)**.

1. Monitoring spark avec Spark web UI

Rappelons avant tout que dans un cluster EMR, le nœud principal est une instance Amazon EC2 comme nous l’avons vu dans la partie 1 (**Automatisation du déploiement d’infrastructure dans le Cloud**).

Nous pouvons donc nous connecter à notre instance ec2 en ssh (le port 22 a été ouvert dans « Apache Spark > modules > emr > security > main.tf») grâce au dns public exposé via la variable «master\_public\_dns» (définie dans le dossier Apache Spark > modules > emr > outputs) avec la clé référencée dans la variable «key\_name» incluse dans le fichier «variables».

Une fois connecté à notre instance, quand on exécute Spark localement, son interface utilisateur peut être consultée à l'adresse <http://localhost:4040>.

Remarque:

En effet, par défaut, l'interface utilisateur de Spark s'exécute sur le port 4040. Ce port a d’ailleurs été également ouvert dans « Apache Spark > modules > emr > security > main.tf» afin de pouvoir y accéder à distance.

Apache Spark fournit une suite d'interfaces utilisateur Web (Jobs, Stages, Tasks, Storage, Environment, Executors, et SQL) pour surveiller l'état de notre application (Spark/PySpark), la consommation de ressources du cluster et les configurations Spark.

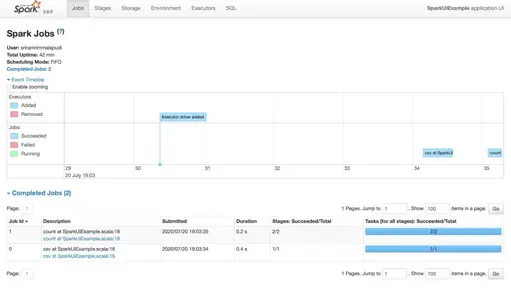
Pour mieux comprendre comment Spark exécute les tâches, cette série d'interfaces utilisateur est très utile.

Précisons que d’autres urls sont disponibles:

* Resource Manager: http://localhost:9870
* Spark Job Tracker: http://localhost:8088/
* Node Specific Info: http://localhost:8042/

Note : Pour accéder à ces URL, l'application Spark doit être dans un état «en cours d'exécution» (running state). Si nous souhaitons accéder à cette URL indépendamment de l'état de notre application Spark et qu’on souhaite accéder à l'interface utilisateur Spark en permanence, il faut démarrer le serveur Spark History.  
Spark History est un composant d'Apache Spark qui permet de visualiser et d'analyser les détails d'exécution des applications Spark précédemment exécutées. C'est un outil utile pour rétrospectivement examiner les performances, les erreurs, les tâches effectuées et d'autres informations liées à l'exécution d'une application Spark.

Une fois connecté, nous devrions accéder à une interface utilisateur semblable:



**Conclusion**

Nous n’avons rien de plus à faire pour accéder à Spark Web UI et ainsi surveiller l’état de notre application. Cette étape ne présente aucune difficulté particulière.

1. Monitoring MongoDB

Afin de monitorer notre serveur MongoDB, nous avons choisi une solution basée sur [Prometheus](https://prometheus.io/) et [Grafana](https://grafana.com/).

**Prometheus** est une plateforme de surveillance et d'alerte open source qui collecte et stocke des métriques sous forme de données émise en série de manière temporelle.

**Grafana** est une application web open source dédiée à la visualisation et à l'analyse interactive. Elle permet d'ingérer des données depuis un grand nombre de sources, d'interroger ces données et de les afficher sur des graphiques personnalisables pour une analyse facilitée. Il est également possible de définir des alertes pour être rapidement informé de comportements inattendus.

Ainsi, l’utilisation conjointe de Prometheus permettra de collecter, surveiller, analyser et visualiser les données de notre instance MongoDB.

Nous basculerons vers un utilisateur non-root disposant de privilèges sudo et nous nous appuyons sur un pare-feu configuré avec [ufw](https://doc.ubuntu-fr.org/ufw)

MongoDB a été installé préalablement à l’étape 1 sur notre instance ec2. Nous démarons ici le processus d’installation et de configuration après s’être connecté à notre instance en ssh.

Installation de Prometheus:

Mise à jour des paquets:

*- sudo apt update*

Création du répertoire de configuration:

*- sudo mkdir -p /etc/prometheus*

Création du répertoire ‘datas’:

*- sudo mkdir -p /var/lib/prometheus*

Téléchargement des fichiers d’installations (binaires précompilés) (version 2.45.2 au *moment de la rédaction du document):*

*- wget https://github.com/prometheus/prometheus/releases/tag/v2.45.2/prometheus-2.45.2.linux-amd64.tar.gz*

Après extraction du fichier , nous naviguons dans le répertoire prometheus-2.45.2.linux-amd64, nous copions les binaires dans le dossier /usr/local/bin

- sudo mv prometheus promtool /usr/local/bin/

ainsi que les autres fichiers de configuration nécessaires dans /etc/prometheus créé avant:

*- sudo mv consoles/ console\_libraries/ /etc/prometheus/*

Les fichiers de console et de bibliothèque de console sont utilisés pour lancer l'interface graphique de Prometheus. Ces fichiers seront conservés avec les fichiers de configuration afin de pouvoir être utilisés lors du démarrage du service.

On déplace également le fichier de template:

*- sudo mv prometheus.yml /etc/prometheus/prometheus.yml*

Ce fichier est le fichier de template qui permet de configurer les ports et les fichiers nécessaires au démarrage du service.

Nous pouvons maintenant vérifier la version de prometheus par:

*- prometheus --version*

qui devrait normalement afficher une sortie ressemblante à ceci:

prometheus, version 2.45.2 (branch: HEAD, revision: b41e0750abf5cc18d8233161560731de05199330)

build user: root@0aa1b7fc430d

build date: 20220714-15:13:18

go version: go1.18.4

platform: linux/amd64

Configuration de Prometheus en tant que service:

Nous allons créer un groupe et un utilisateur Prometheus. Nous aurons besoin d'un utilisateur dédié avec un accès propriétaire aux répertoires nécessaires.

*- sudo groupadd --system prometheus*

*- sudo useradd -s /sbin/nologin --system -g prometheus prometheus*

Nous modifions les autorisations du répertoire prometheus comme suit afin que l’utilisateur dédié ait les permissions correctes:

*- sudo chown -R prometheus:prometheus /etc/prometheus/ /var/lib/prometheus/*

*- sudo chmod -R 775 /etc/prometheus/ /var/lib/prometheus/*

Nous créons le fichier nécessairre pour exécuter Prometheus en tant que service.

- sudo nano /etc/systemd/system/prometheus.service

et nous ajoutons la descriptions du service:

*[Unit]*

*Description=Prometheus*

*Wants=network-online.target*

*After=network-online.target*

*[Service]*

*User=prometheus*

*Group=prometheus*

*Restart=always*

*Type=simple*

*ExecStart=/usr/local/bin/prometheus \*

*--config.file=/etc/prometheus/prometheus.yml \*

*--storage.tsdb.path=/var/lib/prometheus/ \*

*--web.console.templates=/etc/prometheus/consoles \*

*--web.console.libraries=/etc/prometheus/console\_libraries \*

*--web.listen-address=0.0.0.0:9090*

*[Install]*

*WantedBy=multi-user.target*

Ici nous configurons Prometheus pour utiliser les fichiers répertoriés dans le bloc ExecStart pour exécuter le service.

Le fichier de service indique à systemd d'exécuter Prometheus en tant qu'utilisateur prometheus avec le fichier de configuration /etc/prometheus/prometheus.yml et de stocker ses données dans le répertoire /var/lib/prometheus.

Nous configurons également Prometheus pour fonctionner sur le port 9090.

Nous pouvons désormais démarrer le service:

*- sudo systemctl start prometheus*

puis l’autoriser:

- sudo systemctl enable prometheus

et vérifier son état:

- sudo systemctl status prometheus

avec une sortie ressemblante à ceci:

prometheus.service - Prometheus

Loaded: loaded (/etc/systemd/system/prometheus.service; enabled; vendor preset: enabled)

Active: active (running) since Fri 2022-08-05 18:06:05 UTC; 13s ago

Main PID: 7177 (prometheus)

Tasks: 6 (limit: 527)

Memory: 21.0M

CGroup: /system.slice/prometheus.service

└─7177 /usr/local/bin/prometheus --config.file=/etc/prometheus/prometheus.yml --storage.tsdb.path=/var/lib/prometheus/ --web.console.template>

Accès à la console Web:

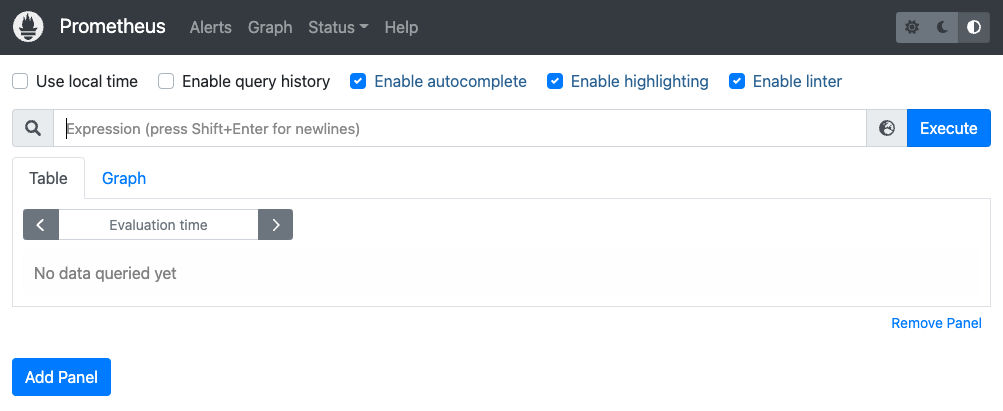
Pour accéder à la console web, il peut être nécessaire d’autoriser le port 9090.  
Pour celà, nous pouvons checker l’ensemble de règles ufw avec:

*- sudo ufw status*

Si le port n’est pas autorisé, nous l’ajoutons:

*- sudo ufw allow 9090*

Pour accéder à la console web, nous pouvons nous rendre en local sur <http://localhost:9090:>



Maintenant que Prometheus est opérationnel, nous devons le binder à MongoDB en utilisant mongoDB exporter.

Configuration de mongoDB exporter

L'exportateur MongoDB (MongoDB Exporter), est un outil qui permet de collecter des métriques et des statistiques à partir d'une instance MongoDB et de les exposer à un système de surveillance ou de gestion. Cet exportateur est souvent utilisé en conjonction avec des outils de monitoring comme Prometheus.

Fonctionnement global:

- Collecte des Métriques : L'exportateur MongoDB se connecte à une instance MongoDB et récupère diverses métriques liées à la performance et à l'utilisation de la base de données.

- Exposition des Métriques : Une fois les métriques collectées, l'exportateur expose ces données sous une forme standard, souvent au format accepté par les systèmes de surveillance comme Prometheus. Il peut exposer ces métriques via une interface HTTP ou d'autres protocoles.

- Intégration avec le Système de Monitoring : Les métriques exposées peuvent ensuite être collectées par un système de surveillance comme Prometheus, qui peut les stocker, les interpréter et générer des alertes en cas de conditions anormales ou de problèmes de performance.

Mettons en place cet exportateur.

Pour celà, nous allons créer un dossier pour l’exportateur:

**-mkdir mongodb-exporter**

L'exportateur MongoDB peut être téléchargé depuis Github. Il est fourni sous forme d'un fichier binaire dans une archive. Nous le configurerons également en tant que service.   
Nous téléchargeons le fichier:

- wget <https://github.com/percona/mongodb_exporter/releases/download/v0.40.0/mongodb_exporter-0.40.0.linux-amd64.tar.gz>

- Après extrait le dossier, nous déplaçons les binaires dans /usr/local/bin:

*- sudo mv mongodb\_exporter /usr/local/bin/*

Activation de l'authentification MongoDB

Dans cette section, nous configurons l'authentification MongoDB pour l'exportateur et créons un utilisateur pour surveiller les métriques du cluster.

Nous commençons par nous connecter à notre instance mongoDB avec mongo:

- mongo

- > use admin

Après avoir basculé vers la base de données admin, nous créons un utilisateur avec le rôle clusterMonitor :

*- db.createUser({user: "test",pwd: "testing",roles: [{ role: "clusterMonitor", db: "admin" },{ role: "read", db: "local" }]})*

La sortie suivante devrait s’afficher:

Successfully added user: {

"user" : "test",

"roles" : [

{

"role" : "clusterMonitor",

"db" : "admin"

},

{

"role" : "read",

"db" : "local"

}

]

}

Nous quittons l’interpréteur mongo pour définir la variable d'environnement MONGODB\_URI avec les informations d'authentification appropriées :

- export MONGODB\_URI=mongodb://test:testing@localhost:27017

La variable utilise les informations d'authentification définies comme étant l'utilisateur test et le mot de passe testing. 27017 est le port par défaut pour une instance mongodb.

Après avoir créé ce rôle clusterMonitor qui aide à surveiller les métriques du cluster, nous allons maintenant configurer l'exportateur MongoDB pour qu'il fonctionne en tant que service.

Configuration de mongoDB exporter en tant que service

Nous créons un fichier système pour l'exportateur.

En accédant à /lib/systemd/system, nous créons un nouveau fichier de service:

*- cd /lib/systemd/system/*

*- sudo nano mongodb\_exporter.service*

Dans le fichier de service, nous ajoutons les lignes suivantes:

*[Unit]*

*Description=Exportateur MongoDB*

*User=prometheus*

*[Service]*

*Type=simple*

*Restart=always*

*ExecStart=/usr/local/bin/mongodb\_exporter*

*[Install]*

*WantedBy=multi-user.target*

Les lignes indiquent à systemd d'exécuter l'exportateur MongoDB en tant que service sous l'utilisateur prometheus.

ExecStart exécutera le binaire mongodb\_exporter depuis usr/local/bin

Nous redémarrons le démon pour recharger les fichiers d'unité :

- sudo systemctl daemon-reload

et nous démarrons l’exportateur en tant que service:

*- sudo systemctl start mongodb\_exporter.service*

Nous checkons son état:

*- sudo systemctl status mongodb\_exporter.service*

● mongodb\_exporter.service - Exportateur MongoDB

Loaded: loaded (/lib/systemd/system/mongodb\_exporter.service; disabled; vendor preset: enabled)

Active: active (running) since Fri 2022-08-05 18:18:38 UTC; 1 weeks 3 days ago

Main PID: 7352 (mongodb\_exporte)

Tasks: 5 (limit: 527)

Memory: 14.2M

CGroup: /system.slice/mongodb\_exporter.service

└─7352 /usr/local/bin/mongodb\_exporter

Pour s’assurer que tout fonctionne comme prévu, nous exécutons:

*- sudo curl <http://localhost:9216/metrics>*

Nous devrions avoir une longue sortie contenant des lignes similaires à :

# HELP go\_gc\_duration\_seconds A summary of the GC invocation durations.

# TYPE go\_gc\_duration\_seconds summary

go\_gc\_duration\_seconds{quantile="0"} 0

go\_gc\_duration\_seconds{quantile="0.25"} 0

go\_gc\_duration\_seconds{quantile="0.5"} 0

go\_gc\_duration\_seconds{quantile="0.75"} 0

go\_gc\_duration\_seconds{quantile="1"} 0

go\_gc\_duration\_seconds\_sum 0

go\_gc\_duration\_seconds\_count 0

# HELP go\_goroutines Number of goroutines that currently exist.

# TYPE go\_goroutines gauge

go\_goroutines 11

# HELP go\_memstats\_alloc\_bytes Number of bytes allocated and still in use.

# TYPE go\_memstats\_alloc\_bytes gauge

go\_memstats\_alloc\_bytes 1.253696e+06

# HELP go\_memstats\_alloc\_bytes\_total Total number of bytes allocated, even if freed.

# TYPE go\_memstats\_alloc\_bytes\_total counter

go\_memstats\_alloc\_bytes\_total 1.253696e+06

# HELP go\_memstats\_buck\_hash\_sys\_bytes Number of bytes used by the profiling bucket hash table.

# TYPE go\_memstats\_buck\_hash\_sys\_bytes gauge

go\_memstats\_buck\_hash\_sys\_bytes 3054

# HELP go\_memstats\_frees\_total Total number of frees.

# TYPE go\_memstats\_frees\_total counter

go\_memstats\_frees\_total 2866

...

La sortie confirme que l'exportateur MongoDB collecte des métriques, telles que la version de MongoDB, les métriques du document et les détails des connexions.

Configuration de l'Exportateur MongoDB en tant que Cible pour Prometheus

Dans cette section, nous configurons l'exportateur MongoDB en tant que cible pour Prometheus.

Nous nous rendons dans le dossier de configuration de Prometheus:

*- cd /etc/prometheus/*

*- sudo nano prometheus.yml*

Nous ajoutons l'exportateur en tant que cible en copiant la ligne mise en évidence dans le fichier /etc/prometheus/prometheus.yml

# A scrape configuration containing exactly one endpoint to scrape:

# Here it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- *job\_name: "prometheus"*

*static\_configs:*

*- targets: ["localhost:9090", "localhost:9216"]*

9216 est le port par défaut de l'exportateur MongoDB.

Après avoir ajouté la cible, nous redémarrons Prometheus pour qu’il prenne en compte nos changements:

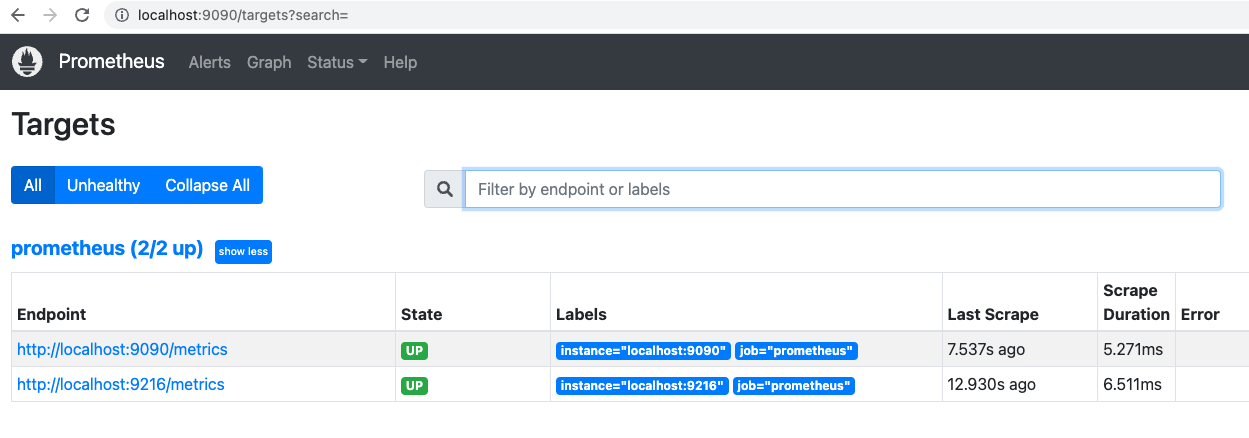
*- sudo systemctl restart prometheus*

et nous accédons à http://localhost:9090/targets pour vérifier que Prometheus récupère l’exportateur nouvellement ajouté.

Remarque:

On pourra dans notre cas visualiser les cibles en accédant à <http://addresse_ip_serveur:9090> ou «adresse\_ip\_serveur» est l’adresse ip publique de l’instance ec2 sur laquelle est installée notre serveur donc par exemple: http://goldenline.com:9090

Nous devrions désormais accéder à liste des cibles via l’interface web:



Rappel

Le port 9090 correspond à Prometheus qui récupère ses propres données. Le port 9216 est l'exportateur MongoDB, ce qui confirme que notre configuration fonctionne comme prévu.

À cette étape, nous avons installé l'exportateur MongoDB et l’avons configuré en tant que cible pour Prometheus afin de collecter des métriques. Nous avons donc un système de monitoring opérationel. Nous allons maintenant en dernière étape créer un tableau de bord MongoDB dans la console web de Grafana pour visualiser et analyser ces métriques.

Visualisation des données

Création d'un tableau de bord MongoDB dans Grafana

Nous allons dans cette étape créer un tableau de bord pour visualiser nos données MongoDB dans Grafana. Pour ce faire, nous ajouterons Prometheus en tant que source de données dans Grafana et nous importerons un tableau de bord MongoDB de Percona (Percona propose plusieurs tableaux de bord pour MongoDB, que vous pouvez trouver dans [la documentation produit](https://docs.percona.com/percona-monitoring-and-management/index.html)). Dans le cadre de ce travail, nous importerons le tableau de bord MongoDB Overview dans notre instance Grafana.

Dans le cadre des prérequis, nous considérons que nous avons installé et sécurisé Grafana\*.

Un mot quand même. Pour installer Grafana, il faudra:

- un nom de domaine (par exemple le nom de domaine fictif goldenline.com)

- les enregistrements au niveau du dns (records A avec goldenline.com et [www.goldenline.com](http://www.goldenline.com) pointant vers l’adresse IP)

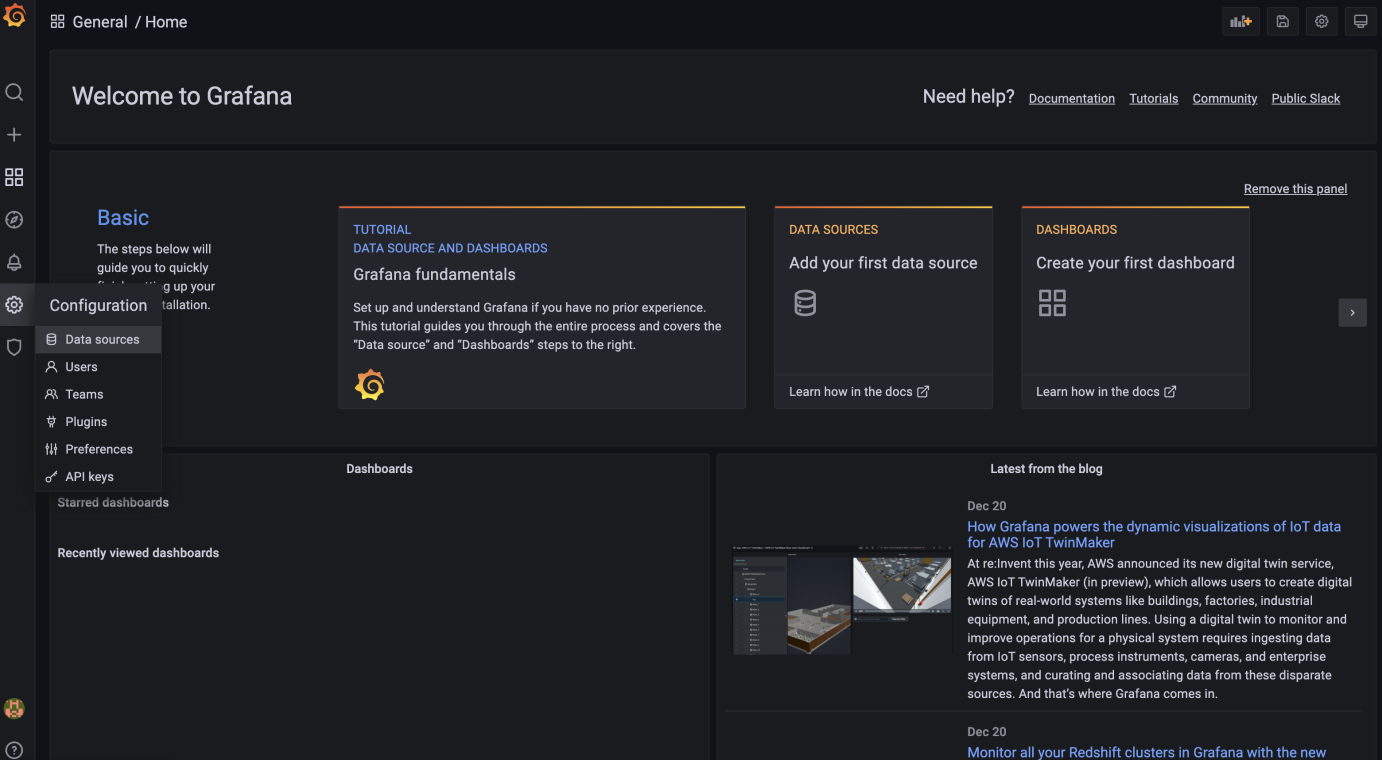
- un serveur nginx installé préalablement avec Let’s Encrypt afin de configurer correctement les accès https

Considérant que Grafana est installé et disponible, nous accédons à notre instance via localhost:3000 avec nos identifiants.

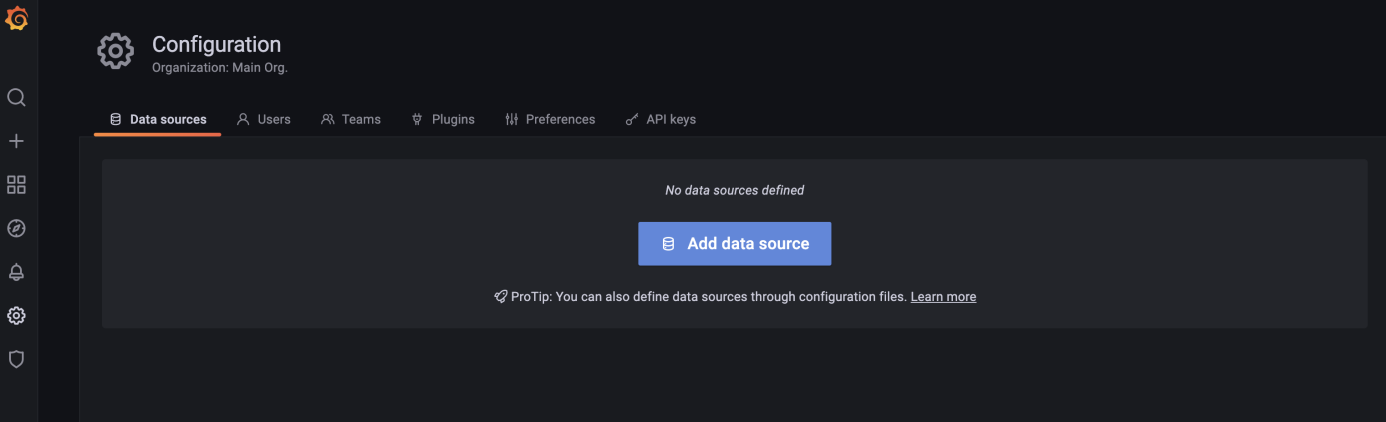
\* *En effet, il ne nous a pas semblé très pertinent d’alourdir la présentation en rajoutant toutes les étapes précises de configuration de Grafana. De nombreux tutoriels documentés permettent déjà de configurer et d’installer cet outil. Par exemple ici: <https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-grafana-on-ubuntu-20-04>*

Pour commencer, définissons Prometheus comme source de données dans Grafana.

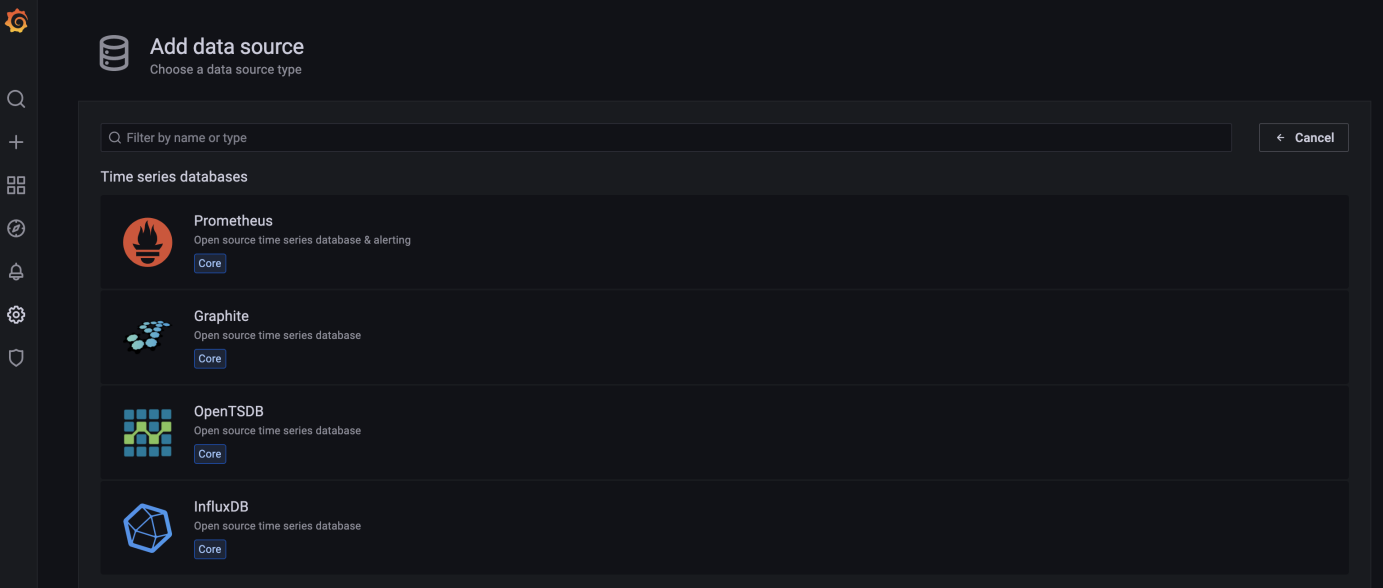
Dans le panneau de gauche, nous cliquons sur l'icône d'engrenage pour Configuration, puis sélectionnez Sources de données :

**

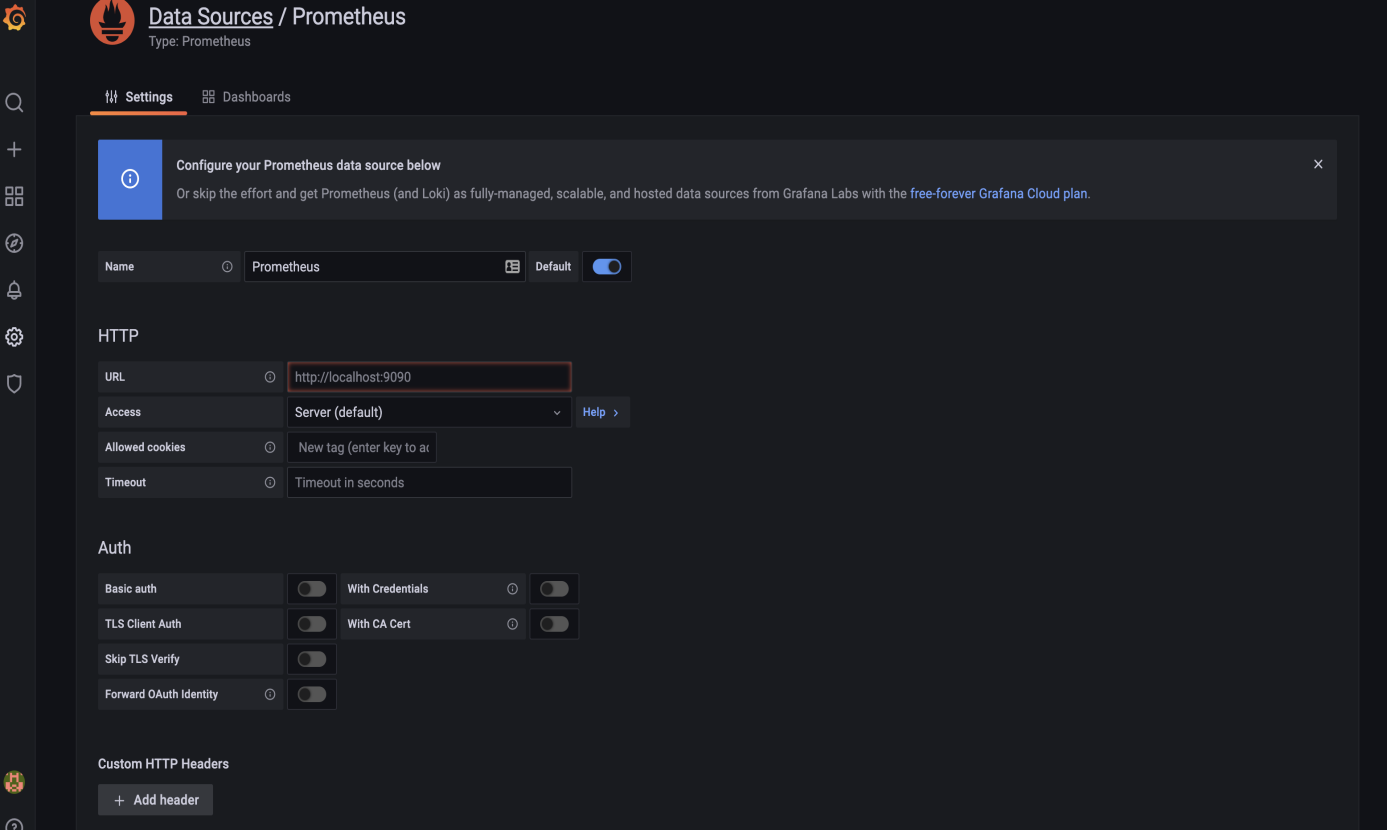
Nous ajoutons une source de données en cliquant sur ‘add data source’



Nous sélectionnons alors Prometheus:



Nous configurons les paramètres pour notre source de données Prometheus:



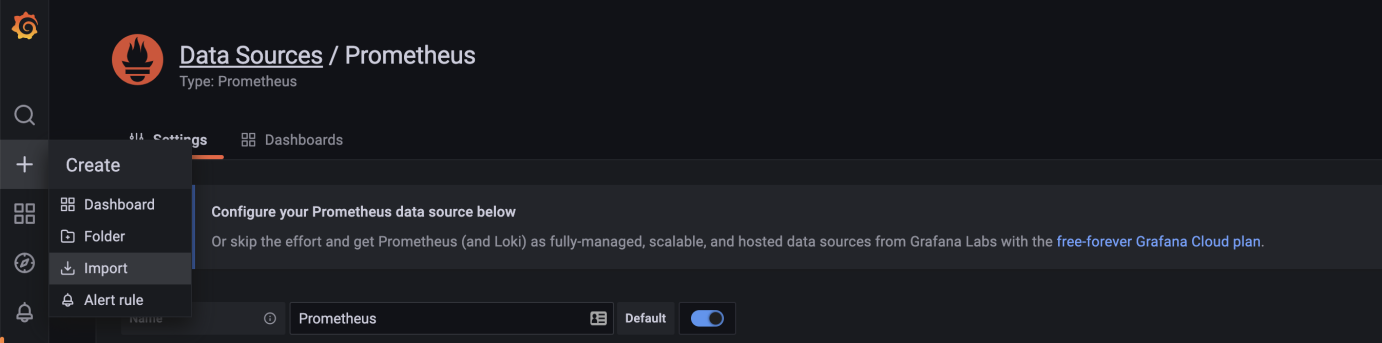
Dans le champ URL, nous fournissons l'URL de notre instance Prometheus, en l’occurence:

<http://localhost:9090>

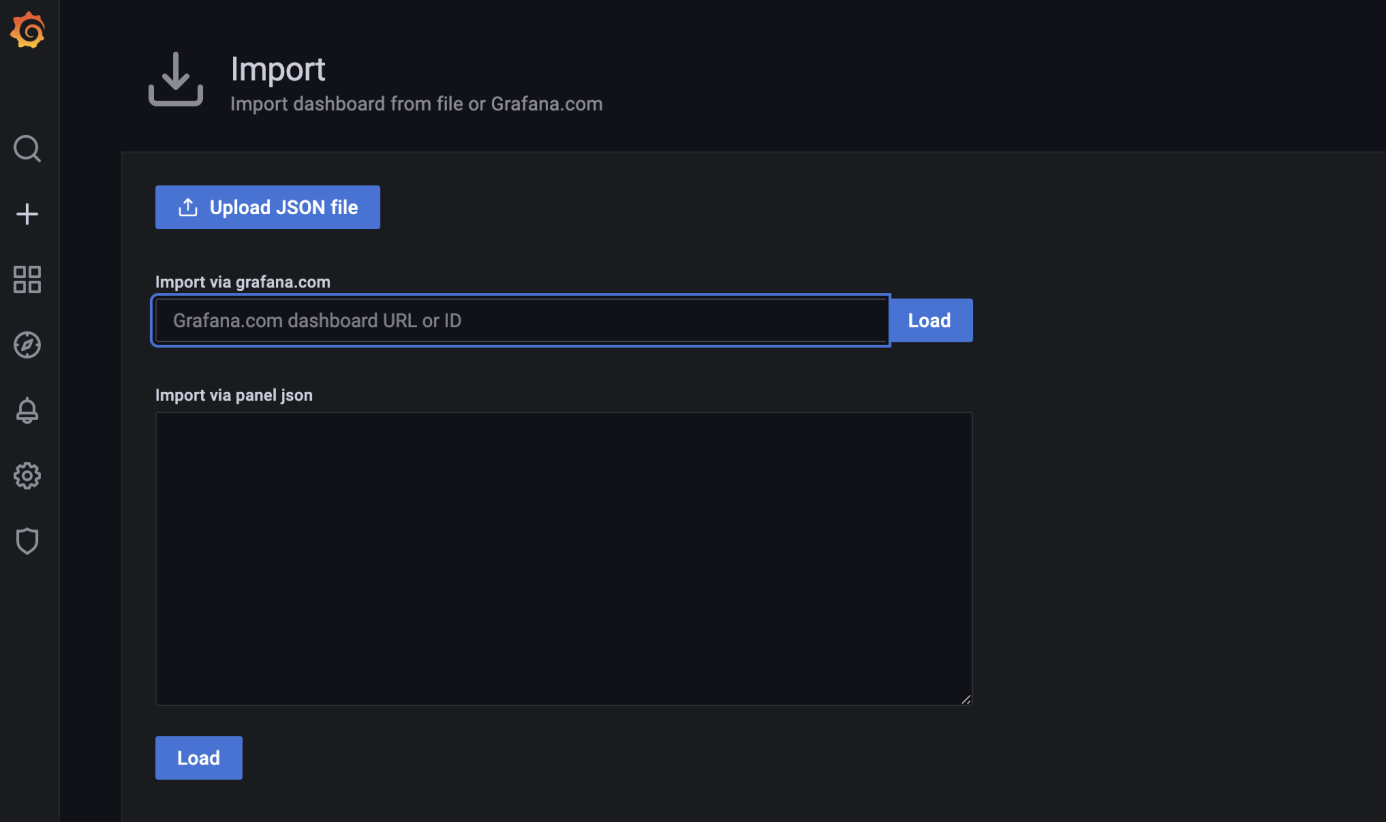
Nous cliquons sur "Enregistrer et tester" en bas de l'écran. Maintenant, Prometheus est ajouté en tant que source de données pour Grafana.

Nouns importons ensuite le tableau de bord MongoDB Overview pour Grafana. Nous pouvons importer le tableau de bord en téléchargeant un fichier JSON ou en important un ID de tableau de bord, qu’on peut trouver dans la documentation de Grafana pour les tableaux de bord. Ici, nous utiliserons l’ID du tableau de bord pour l'importer.

Dans le menu de gauche, nous cliquons sur l'icône «plus» pour créer et nous sélectionnons «Importer». De là, nous sommes dirigés vers la page d'importation:



Ici, nous pouvons télécharger le fichier JSON du tableau de bord ou coller l'identifiant du tableau de bord Grafana :



Nous ajoutons l'identifiant du tableau de bord Grafana, qu’on trouvera sur la [page Grafana](https://grafana.com/grafana/dashboards/7353-mongodb-overview/) pour le tableau de bord MongoDB Overview :

- <https://grafana.com/grafana/dashboards/7353>

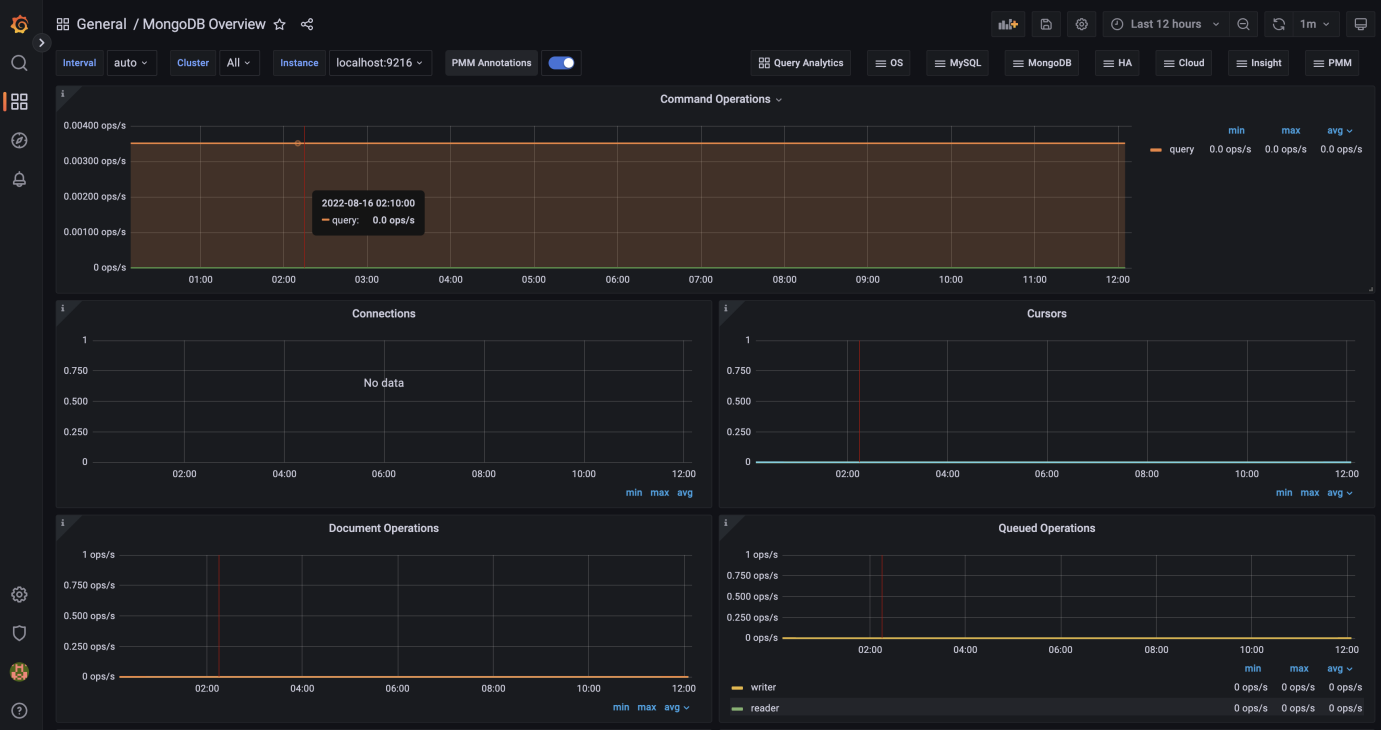
De nombreux tableaux de bord sont disponibles en visitant la [page Grafana](https://grafana.com/grafana/dashboards/) sur les tableaux de bord.

Après avoir ajouté l'identifiant du tableau de bord, nous cliquons sur Charger (Load).

Maintenant, une page Options s'ouvre, où nous pouvons fournir un nom pour le tableau de bord, sélectionner le dossier pour le tableau de bord et choisir une source de données. On peut laisser les noms du tableau de bord et du dossier par défaut.

Pour la source de données, on choisit biensur Prometheus. Une fois les options remplies, on clique sur «Importer».

Le dashboard est alors créé:



Le tableau de bord est en mesure d’afficher des mises à jour en temps réel de la base de données MongoDB.

**Conclusion**

Nous avons désormais un pipeline de surveillance entièrement opérationnel pour MongoDB grâce à Prometheus et Grafana pour visualiser nos données récoltées en temps réel!