Objectif

Dans cet exercice (issu de la documentation officielle de Kubernetes), nous allons utiliser un StatefulSet pour mettre en place un cluster Mysql. Celui-ci permettra de créer 3 Pods, un master et deux slaves:

- le master sera configuré pour permettre l'écriture
- les slaves seront configurés pour permettre la lecture

Pré-requis

Les seul pré-requis sont d'avoir un cluster Kubernetes disponible ainsi que le binaire *kubectl* installé. Cet exercice peut s'effectuer sur un cluster déployé chez un cloud provider ou bien simplement sur un cluster Minikube local.

Note: si vous utilisez Minikube, assurez-vous cependant de le démarrer avec assez de ressources, par exemple:

```
$ minikube start --memory=4096 --cpus=4
```

Configuration du Master et des Slaves

Nous allons commencer par créer la ressource *ConfigMap* ayant la spécification suivante:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  master.cnf: |
    # Apply this config only on the master.
    [mysqld]
    log-bin
  slave.cnf: |
    # Apply this config only on slaves.
    [mysqld]
    super-read-only
```

Celle-ci contient:

- la configuration à utiliser pour le master MySQL (clé *master.cnf*)
- la configuration à utiliser pour les slave MySQL (clé *slave.cnf*)

Sauvegardez cette spécification dans le fichier *cm.yaml* puis créez la ressource avec la commande suivante:

```
$ kubectl apply –f cm.yaml
```

Mise en place des Services

Nous allons maintenant créer 2 services:

- le premier, dit service *Headless*, sera utilisé pour donner une identité réseau stable à chaque Pods qui sera créé par le StatefulSet
- le second de type *ClusterIP* permettra de faire du load balancing entre les Pods de type slaves

Création du Service Headless

La spécification suivante définie le Service *mysql* permettant de donner une identité à chaque Pod. La propriété *.spec.clusterIP* ayant pour valeur *None*, ce Service ne permettra pas de faire du load balancing entre les Pods

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  ports:
  – name: mysql
    port: 3306
  clusterIP: None
  selector:
    app: mysql
```

Copiez cette spécification dans le fichier *headless-svc.yaml* puis créez ce Service avec la commande suivante:

```
$ kubectl apply -f headless-svc.yaml
```

Création du Service clusterIP

La spécification suivante définie le Service qui assurera le load balancing entre les slaves pour les opération de lecture.

Note: les opérations d'écriture devront être réalisées en se connectant au master

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

Copiez cette spécification dans le fichier *mysql-svc.yaml* puis créez ce Service avec la commande suivante:

```
$ kubectl apply -f mysql-svc.yaml
```

Création du cluster MySQL

Nous allons à présent créer le cluster MySQL à partir du StatefulSet défini dans la spécification suivante:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
```

```
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      initContainers:
      - name: init-mysql
        image: mysql:5.7
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Generate mysql server-id from pod ordinal index.
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          echo [mysqld] > /mnt/conf.d/server-id.cnf
          # Add an offset to avoid reserved server-id=0 value.
          echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.cnf
          # Copy appropriate conf.d files from config-map to emptyDir.
          if [[ $ordinal -eq 0 ]]; then
            cp /mnt/config-map/master.cnf /mnt/conf.d/
          else
            cp /mnt/config-map/slave.cnf /mnt/conf.d/
          fi
        volumeMounts:
        - name: conf
          mountPath: /mnt/conf.d
        - name: config-map
          mountPath: /mnt/config-map
      - name: clone-mysql
        image: gcr.io/google-samples/xtrabackup:1.0
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Skip the clone if data already exists.
          [[ -d /var/lib/mysql/mysql ]] && exit 0
          # Skip the clone on master (ordinal index 0).
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          [[ $ordinal -eq 0 ]] && exit 0
          # Clone data from previous peer.
          ncat --recv-only mysql-$(($ordinal-1)).mysql 3307 | xbstream -x -C
/var/lib/mysql
          # Prepare the backup.
          xtrabackup --prepare --target-dir=/var/lib/mysql
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
```

```
      containers:
      - name: mysql
        image: mysql:5.7
        env:
        - name: MYSQL_ALLOW_EMPTY_PASSWORD
          value: "1"
        ports:
        - name: mysql
          containerPort: 3306
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 500m
            memory: 1Gi
        livenessProbe:
          exec:
            command: ["mysqladmin", "ping"]
          initialDelaySeconds: 30
          periodSeconds: 10
          timeoutSeconds: 5
        readinessProbe:
          exec:
            # Check we can execute queries over TCP (skip-networking is off).
            command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
          initialDelaySeconds: 5
          periodSeconds: 2
          timeoutSeconds: 1
      - name: xtrabackup
        image: gcr.io/google-samples/xtrabackup:1.0
        ports:
        - name: xtrabackup
          containerPort: 3307
        command:
        - bash
        - "-c"
        - |
          set -ex
          cd /var/lib/mysql

          # Determine binlog position of cloned data, if any.
          if [[ -f xtrabackup_slave_info && "x$(<xtrabackup_slave_info)" != "x"
]]; then
            # XtraBackup already generated a partial "CHANGE MASTER TO" query
            # because we're cloning from an existing slave. (Need to remove the
tailing semicolon!)
            cat xtrabackup_slave_info | sed -E 's/;$//g' >
change_master_to.sql.in
            # Ignore xtrabackup_binlog_info in this case (it's useless).
            rm -f xtrabackup_slave_info xtrabackup_binlog_info
          elif [[ -f xtrabackup_binlog_info ]]; then
            # We're cloning directly from master. Parse binlog position.
```

```
            [[ `cat xtrabackup_binlog_info` =~ ^(.*?)[[:space:]]+(.*?)$ ]] ||
exit 1
            rm -f xtrabackup_binlog_info xtrabackup_slave_info
            echo "CHANGE MASTER TO MASTER_LOG_FILE='${BASH_REMATCH[1]}',\
                  MASTER_LOG_POS=${BASH_REMATCH[2]}" > change_master_to.sql.in
          fi

          # Check if we need to complete a clone by starting replication.
          if [[ -f change_master_to.sql.in ]]; then
            echo "Waiting for mysqld to be ready (accepting connections)"
            until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

            echo "Initializing replication from clone position"
            mysql -h 127.0.0.1 \
                  -e "$(<change_master_to.sql.in), \
                          MASTER_HOST='mysql-0.mysql', \
                          MASTER_USER='root', \
                          MASTER_PASSWORD='', \
                          MASTER_CONNECT_RETRY=10; \
                        START SLAVE;" || exit 1
            # In case of container restart, attempt this at-most-once.
            mv change_master_to.sql.in change_master_to.sql.orig
          fi

          # Start a server to send backups when requested by peers.
          exec ncat --listen --keep-open --send-only --max-conns=1 3307 -c \
            "xtrabackup --backup --slave-info --stream=xbstream --host=127.0.0.1
--user=root"
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
      volumes:
      - name: conf
        emptyDir: {}
      - name: config-map
        configMap:
          name: mysql
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 2Gi
```

Cette spécification est assez impressionnante, c'est principalement dû à la présence de

quelques scripts shell un peu opaque à première vue. Nous allons expliquer, dans les grandes lignes, les éléments qu'elle contient:

- la valeur de la clé *.spec.replicas* indique que 3 Pods seront lancés. De part la nature du StatefulSet, ces Pods seront lancés de manière séquentielle, un Pod étant lancé lorsque le Pod précédent est actif. Chacun de ces Pods aura un nom constitué du nom du StatefulSet auquel est ajouté un entier incrémenté pour chaque nouveau Pod:
    - le premier Pod *mysql-0* sera le Pod master
    - les suivants, *mysql-1* et *mysql-2*, seront des Pods slave
- la cléc *.spec.volumeClaimTemplates* défini le stockage demandé pour chaque Pod, ici 2Gi, qui sera provisionné en utilisant la *StorageClass* par défaut
- la spécification des Pods est définie sous la clé *.spec.template.spec*
- 2 volumes sont définis pour chaque Pod:
    - le premier est basé sur la ConfigMap créée précédemment, celle-ci contient la configuration du master ainsi que celle des slaves
    - le second est un répertoire vide créé sur la machine hôte dans lequel sera copiée la configuration du Pod (master / slave) et qui sera lu au lancement du processus MySQL
- la spécification des Pods, contient 2 listes de containers:
    - *.spec.template.spec.initContainers*
    - *.spec.template.spec.containers*
- les *initContainers* sont des containers en charge de préparer l'environnement. Cette liste contient 2 containers:
    - *init-mysql* détermine, en fonction de l'identifiant du Pod (0, 1 ou 2) la configuration qui doit être utilisée depuis la ConfigMap et copie celle-ci dans le volume de type *emtpyDir*. Si l'identifiant est 0 (cas du master), la config *master.cnf* est copiée, si l'identifiant est 1 ou 2 (cas d'un slave) c'est la config *slave.cnf* qui est copiée
    - *clone-mysql* clone les données du Pod qui a été crée avant le Pod actuel. Cette action ne sera effectuée que dans le cas de Pods de type slave
- *containers* contient la liste de containers qui sont lancés une fois que les *initContainers* sont terminés. Cette liste contient 2 containers:

- *mysql* lance un des process MySQL du cluster. Il utilise la configuration disponible dans la *ConfigMap* nommé *conf* mise en place par le container *init-mysql*

- *xtrabackup* permet de mette en place la stratégie de backup

Copiez cette spécification dans le fichier *mysql-statefulset.yaml*, puis créez la ressource avec la commande suivante:

```
$ kubectl apply -f mysql-statefulset.yaml
```

Vérification

On peut voir qu'en quelques dizaines de secondes les 3 Pods, *mysql-0*, *mysql-1* et *mysql-2* sont créés:

Note: l'option *--watch* permet d'observer la création de ces Pods de manière intéractive:

```
$ kubectl get pods -l app=mysql --watch
NAME      READY   STATUS            RESTARTS   AGE
mysql-0   1/2     Running           0          59s
mysql-0   2/2     Running           0          76s
mysql-1   0/2     Pending           0          0s
mysql-1   0/2     Pending           0          0s
mysql-1   0/2     Pending           0          5s
mysql-1   0/2     Init:0/2          0          5s
mysql-1   0/2     Init:1/2          0          29s
mysql-1   0/2     Init:1/2          0          44s
mysql-1   0/2     PodInitializing   0          62s
mysql-1   1/2     Running           0          63s
mysql-1   2/2     Running           0          67s
mysql-2   0/2     Pending           0          0s
mysql-2   0/2     Pending           0          0s
mysql-2   0/2     Pending           0          4s
mysql-2   0/2     Init:0/2          0          4s
mysql-2   0/2     Init:1/2          0          28s
mysql-2   0/2     Init:1/2          0          38s
mysql-2   0/2     PodInitializing   0          58s
mysql-2   1/2     Running           0          59s
mysql-2   2/2     Running           0          64s
```

Test d'écriture

Nous allons maintenant lancer un Pod dans lequel tourne un container basé sur l'image *mysql*

et depuis ce Pod effectuer les actions suivantes:

- connection au Pod master
- création de la database *test*
- création de la table *message*
- ajout d'un enregistrement dans cette table

Utilisez pour cela la commande suivante:

```
$ kubectl run mysql-client --image=mysql:5.7 -i --rm --restart=Never -- \
  mysql -h mysql-0.mysql <<EOF
CREATE DATABASE test;
CREATE TABLE test.messages (message VARCHAR(250));
INSERT INTO test.messages VALUES ('hello');
EOF
```

Test de lecture

Afin de vérifier que l'enregistrement précédent à bien été créé, nous allons procéder de la même façon et lancer un Pod dans lequel tourne un container basé sur l'image *mysql* et depuis ce Pod effectuer les actions suivantes:

- connection au Service *mysql-client* (en charge de loadbalancer les requetes sur les slaves du cluster)
- récupération des éléments de la table *message* dans la database *test*

```
$ kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never --\
  mysql -h mysql-read -e "SELECT * FROM test.messages"
+---------+
| message |
+---------+
| hello   |
+---------+
```

Le résultat de la commande montre bien que les données écrites depuis le master sont bien disponible en lecture depuis un slave.

Stockage

Si nous listons les PersistentVolumeClaim et les PersitentVolume, nous pouvons voir qu'un PVC a été créé pour chaque Pod et qu'un PV a été provisionné dynamiquement et associé à

ce PVC (la StorageClass utilisée ici est nommé *standard*, c'est la StorageClass par défaut sur un cluster Minikube).

```
$ kubectl get pvc,pv
NAME                                     STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/data-mysql-0   Bound    pvc-abded038-c28d-4fe6-8d8e-
340e5cc745f7   2Gi        RWO            standard       8m11s
persistentvolumeclaim/data-mysql-1   Bound    pvc-97587c8e-dfdf-448a-af12-
75d6e1e5d770   2Gi        RWO            standard       7m23s
persistentvolumeclaim/data-mysql-2   Bound    pvc-93aa73c9-a396-41ad-bb91-
ec690bd00674   2Gi        RWO            standard       7m5s

NAME                                                        CAPACITY   ACCESS
MODES   RECLAIM POLICY   STATUS   CLAIM                  STORAGECLASS   REASON
AGE
persistentvolume/pvc-93aa73c9-a396-41ad-bb91-ec690bd00674   2Gi        RWO
Delete           Bound    default/data-mysql-2   standard                7m5s
persistentvolume/pvc-97587c8e-dfdf-448a-af12-75d6e1e5d770   2Gi        RWO
Delete           Bound    default/data-mysql-1   standard                7m23s
persistentvolume/pvc-abded038-c28d-4fe6-8d8e-340e5cc745f7   2Gi        RWO
Delete           Bound    default/data-mysql-0   standard                8m11s
```

Mise à l'échelle

Nous pouvons alors facilement augmenter ou diminuer le nombre de slaves du cluster. La commande suivante permet par exemple d'ajouter un slave:

```
$ kubectl scale --replicas=4 statefulset/mysql
statefulset.apps/mysql scaled
```

Après quelques dizaines de secondes, nous pouvons alors voir que le cluster est maintenant constitué de 4 Pods:

```
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
mysql-0   2/2     Running   0          15m
mysql-1   2/2     Running   0          14m
mysql-2   2/2     Running   0          14m
mysql-3   2/2     Running   0          3m4s
```

Note: si le nouveau Pod n'apparait pas, cela peut être du à des contraintes que le scheduler

ne pourrait pas satisfaire, notamment si les ressources du cluster ne sont pas suffisantes.

En résumé

Nous avons vu ici comment mettre en place un cluster MySQL à l'aide d'un StatefulSet. Comme vous pouvez le voir, cette ressource est plus complexe que les ressources utilisées pour la gestion des workload stateless, c'est à dire qui n'ont pas de données à gérer. Dans les faits, la gestion des clusters de base de données, ou d'autres applications complexes, est souvent réalisée à l'aide d'Operator, processus dans lequel est codé toute la logique de gestion d'une application.