Uruchomienie aplikacji Guestbook z Redis

Aplikacja Guesbook używa Redis do przechowywania danych. Zapisuje swoje dane do głównej instancji Redis i odczytuje je z wielu instancji podrzędnych.

W pierwszej kolejności uruchomimy Redis Master. Musimy utworzyć plik manifestu wykonując komendę:

```
vi redis-master-deployment.yaml
```

Zawartość pliku będzie wyglądała następująco:

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: master
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: k8s.gcr.io/redis:e2e  # or just image: redis
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
```

Po utworzeniu pliku należy zastosować deployment Redis Master używając polecenia:

```
kubectl apply -f redis-master-deployment.yaml
```

Sprawdź czy pod został poprawnie utworzony:

```
kubectl get pod
```

Output powinien wyglądać następująco:

```
NAME                            READY     STATUS    RESTARTS   AGE
redis-master-1068406935-3lswp   1/1       Running   0          28s
```

Uruchom poniższe polecenie aby wyświetlić logi z poda Redis Master:

```
kubectl logs -f <nazwa_poda>
```

Następnie musimy utworzyć usługę Redis Master. Musimy utworzyć plik manifestu wykonując komendę:
`vi redis-master-service.yaml`
Zawartość pliku będzie wyglądała następująco::

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
    tier: backend
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: master
    tier: backend
```

Tak jak poprzednio należy zastosować utworzenie usługi z pliku yaml:

```
kubectl apply -f redis-master-service.yaml
```

Sprawdź czy usługa została utworzona poprawnie:

```
kubectl get service
```
albo krócej:
```
kubectl get svc
```

Output powinien wyglądać podobnie do tego:

```
NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
redis-master   ClusterIP   10.0.0.151   <none>        6379/TCP
8s
```

Uruchomienie redis slave.

Deployment ten określa dwie repliki. Oznacza to, że jeśli nie będzie żadnych replik deployment będzie skalował pody do dwóch i odwrotnie, jeśli będzie więcej niż dwie repliki deployment przeskaluje je na dwie.
W celu utworzenia deploymentu dla redis slave Musimy utworzyć plik manifestu wykonując komendę:

```
vi redis-slave-deployment.yaml
```

Zawartość pliku będzie wyglądała następująco:

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: redis-slave
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: slave
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v1
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your cluster to
          # provide a dns service. As of Kubernetes 1.3, DNS is a
built-in
          # service launched automatically. However, if the
cluster you are using
          # does not have a built-in DNS service, you can instead
          # access an environment variable to find the master
          # service's host. To do so, comment out the 'value: dns'
line above, and
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 6379
```

Zastosujmy teraz deployment bazujący na pliku yaml:

```
kubectl apply -f redis-slave-deployment.yaml
```

Sprawdź nowo utworzone pody:

```
kubectl get pod
```

Powinieneś zobaczyć:

```
NAME                               READY     STATUS
RESTARTS   AGE
```

```
redis-master-1068406935-3lswp   1/1       Running
0          1m
redis-slave-2005841000-fpvqc    0/1       ContainerCreating
0          6s
redis-slave-2005841000-phfv9    0/1       ContainerCreating
0          6s
```

Tworzenie usługi dla redis slave.

Aplikacja Guestbook musi komunikować się z podami Redis Slave w celu odczytania danych.
Musimy utworzyć plik manifestu wykonując komendę:

```
vi  redis-slave-service.yaml
```

Zawartość pliku będzie wyglądała następująco:

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
    tier: backend
spec:
  ports:
  - port: 6379
  selector:
    app: redis
    role: slave
    tier: backend
```

Zastosujmy usługę:

```
kubectl apply -f redis-slave-service.yaml
```

Sprawdźmy, czy usługa działa poprawnie:

```
kubectl get svc
```

Przykładowy output:

```
NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
redis-master   ClusterIP   10.0.0.151   <none>        6379/TCP
1m
redis-slave    ClusterIP   10.0.0.223   <none>        6379/TCP
6s
```

Teraz musimy zainstalować aplikację Guestbook. Musimy utworzyć plik manifestu wykonując komendę:

```
vi frontend-deployment.yaml
```

Zawartość pliku będzie wyglądała następująco:

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your cluster to
          # provide a dns service. As of Kubernetes 1.3, DNS is a
built-in
          # service launched automatically. However, if the
cluster you are using
          # does not have a built-in DNS service, you can instead
          # access an environment variable to find the master
          # service's host. To do so, comment out the 'value: dns'
line above, and
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 80
```

Zastosuj deployment:

```
kubectl apply -f frontend-deployment.yaml
```

Sprawdź listę podów w celu weryfikacji czy faktycznie utworzyły się 3 repliki, zgodnie z tym co podaliśmy w pliku deploymentu.

```
kubectl get pods -l app=guestbook -l tier=frontend
```

Wynik powinien wyglądać podobnie do tego:

```
NAME                        READY     STATUS    RESTARTS   AGE
frontend-3823415956-dsvc5   1/1       Running   0          54s
frontend-3823415956-k22zn   1/1       Running   0          54s
frontend-3823415956-w9gbt   1/1       Running   0          54s
```

Teraz musimy utworzyć usługę frontend.
Zastosowane usługi redis-slave i redis-master są dostępne tylko w klastrze kontenera, ponieważ domyślnym typem usługi jest ClusterIP. ClusterIP zapewnia pojedynczy adres IP dla zestawu slave-wów, na które wskazuje Usługa. Ten adres IP jest dostępny tylko w klastrze.
Aby umożliwić gościom dostęp do księgi gości, należy skonfigurować usługę Frontend Service tak, aby była widoczna zewnętrznie, aby klient mógł zażądać usługi poza klastrem kontenerów.

Musimy utworzyć plik manifestu wykonując komendę:

```
vi frontend-service.yaml
```

Zawartość pliku będzie wyglądała następująco:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # comment or delete the following line if you want to use a
LoadBalancer
  #type: NodePort
  # if your cluster supports it, uncomment the following to
automatically create
  # an external load-balanced IP for the frontend service.
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: guestbook
    tier: frontend
```

Zastosuj usługę:

```
kubectl apply -f frontend-service.yaml
```

Sprawdź dostępne usługi:

```
  kubectl get services
```

Output powinien wyglądać podobnie do tego:

```
 NAME            TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)         AGE
frontend        ClusterIP   10.0.0.112    <none>
80:31323/TCP    6s
redis-master    ClusterIP   10.0.0.151    <none>
6379/TCP        2m
redis-slave     ClusterIP   10.0.0.223    <none>
6379/TCP        1m
```

INGRESS

W celu uzyskania dostępu do swojej aplikacji możemy skorzystać z ingressa, zamiast uruchamiać każdą z usług z zewnętrznym adresem IP. W tym celu musimy najpierw zainstalować ingressa a później dostosować naszą aplikację aby z niego korzystała.

INSTALACJA INGRESSA

Aby zainstalować ingressa i wszystkie potrzebne do jego poprawnego działania aplikacje wykonaj po kolei instrukcje z plików umieszczonych w folderze „INFO”:

1. Instalacja Helm’a
2. Instalacja traefik
3. Konfiguracja dostępu do dashboardu
4. Instalacja ingress-a

Zanim dodamy ingressa do naszej Księgi Gości musimy zmienić w poprzednim pliku, który wykonywaliśmy (`frontend-service.yaml`) type z LoadBalancer na NodePort i zastosować zmiany.

```
kubectl apply -f frontend-service.yaml
```

UWAGA DO PONIŻSZEJ KONFIGURACJI:
W miejscu, w którym podajemy - host: gbook.devopsseries.pl będzie to url, na którym będziemy się dostawać do aplikacji.
Ponieważ nie ma ustawionych dns-ów, musimy dodać do pliku hosts (w Windowsie: C:\Windows\System32\drivers\etc\hosts, na MacOS /etc/hosts) adres IP dla naszego url. Wpis powinien wyglądać następująco, (gdzie EXTERNAL_IP to adres IP naszego LoadBalancera):

```
EXTERNAL_IP gbook.devopsseries.pl
```

Teraz pozostało jeszcze skonfigurowanie ingresa dla GuestBook-a.
Musimy utworzyć plik manifestu wykonując komendę:

```
vi frontend-ingress.yaml
```

Zawartość pliku będzie wyglądała następująco:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend
  namespace: ns-user01
```

```
  annotations:
    kubernetes.io/ingress.class: "traefik"
spec:
  rules:
  - host: gbook.devopsseries.pl
    http:
      paths:
      - backend:
          serviceName: frontend
          servicePort: 80
```

Zastosujmy ingressa:

```
kubectl apply -f frontend-ingress.yaml
```

Po utworzeniu ingressa możemy przejść do przeglądarki internetowej i sprawdzić czy nasza księga gości wyświetla się poprawnie (http://gbook.devopsseries.pl)

Możemy teraz spróbować zeskalować aplikację ręcznie:

w górę:

```
kubectl scale deployment frontend --replicas=5
```

Sprawdźmy czy faktycznie zwiększyła się liczba replik do 5.

```
kubectl get pod
```

Przykładowy output:

```
NAME                              READY     STATUS    RESTARTS   AGE
  frontend-3823415956-70qj5       1/1       Running   0
5s
  frontend-3823415956-dsvc5       1/1       Running   0
54m
  frontend-3823415956-k22zn       1/1       Running   0
54m
  frontend-3823415956-w9gbt       1/1       Running   0
54m
  frontend-3823415956-x2pld       1/1       Running   0
5s
  redis-master-1068406935-3lswp   1/1       Running   0
56m
  redis-slave-2005841000-fpvqc    1/1       Running   0
55m
  redis-slave-2005841000-phfv9    1/1       Running   0
55m
```

Teraz zmniejszmy ilość podów.

```
kubectl scale deployment frontend --replicas=2
```

Sprawdźmy czy faktycznie liczba podów zmniejszyła się do 2.

```
kubectl get pod
```

Przykładowy output:

```
NAME                              READY     STATUS    RESTARTS   AGE
  frontend-3823415956-k22zn       1/1       Running   0
1h
  frontend-3823415956-w9gbt       1/1       Running   0
1h
  redis-master-1068406935-3lswp   1/1       Running   0
1h
  redis-slave-2005841000-fpvqc    1/1       Running   0
1h
  redis-slave-2005841000-phfv9    1/1       Running   0
1h
```