Lab 8 – Multiple-tier applications Containers

Introduction

In this lab, you will learn how to build and deploy a simple, multi-tier web application using Kubernetes and Docker.

Create the Redis master pod

The guestbook application uses Redis to store its data. It writes its data to a Redis master instance and reads data from multiple Redis slave instances

Creating the Redis Master Deployment

The manifest file, included below, specifies a Deployment controller that runs a single replica Redis master Pod.

- Launch a terminal window in the directory you downloaded the manifest files.
- Apply the Redis Master Deployment from the yaml file

1. Deploying an Multi-tier web application

1.1 Create the Redis Master Deployment from the **redis-master-deployment.yaml** file:

Copy

```
cat > redis-master-deployment.yaml <<EOF

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: redis-master

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
```

```
EOF
```

1.2 Create the **redis-master-deployment.yaml** file

Copy

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

1.3 Query the list of pods to verify that the redis master pod is running:

Copy

```
kubectl get pods
```

Sample output:

```
NAME                            READY     STATUS    RESTARTS   AGE

redis-master-1068406935-3lswp   1/1       Running   0          28s
```

Note: Wait till pod changes to **Running** status, before proceeding to next step .

1.4 Run the following command to view the logs from the Redis Master Pod:

Capture the redis_pod name from the pod status

Copy

```
redis_pod=`kubectl get pods | grep redis-master | awk '{print $1}'`

echo $redis_pod
```

Sample output:

```
redis-master-55db5f7567-qmq4v
```

1.5 Verify the logs from assigned variable redis_pod

Copy

```
kubectl logs -f $redis_pod
```

Output:

```
                _._

           _.-``__ ''-._

      _.-``    `.  `_.  ''-._           Redis 2.8.19 (00000000/0) 64 b
it

  .-`` .-```.  ```\/    _.,_ ''-._

 (    '      ,       .-`  | `,    )     Running in stand alone mode

 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379

 |    `-._   `._    /     _.-'    |     PID: 1

  `-._    `-._  `-./  _.-'    _.-'

 |`-._`-._    `-.__.-'    _.-'_.-'|

 |    `-._`-._        _.-'_.-'    |           http://redis.io

  `-._    `-._`-.__.-'_.-'    _.-'

 |`-._`-._    `-.__.-'    _.-'_.-'|

 |    `-._`-._        _.-'_.-'    |

  `-._    `-._`-.__.-'_.-'    _.-'

      `-._    `-.__.-'    _.-'

          `-._        _.-'

              `-.__.-'
```

```
[1] 22 Apr 11:45:17.967 # Server started, Redis version 2.8.19

[1] 22 Apr 11:45:17.967 # WARNING you have Transparent Huge Pages (THP
) support enabled in your kernel. This will create latency and memory
usage issues with Redis. To fix this issue run the command 'echo never
> /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to
your /etc/rc.local in order to retain the setting after a reboot. Redi
s must be restarted after THP is disabled.

[1] 22 Apr 11:45:17.967 # WARNING: The TCP backlog setting of 511 cann
ot be enforced because /proc/sys/net/core/somaxconn is set to the lowe
r value of 128.

[1] 22 Apr 11:45:17.967 * The server is now ready to accept connection
s on port 6379
```

Note: Press <ctrl+c> to exit.

2. Creating the Redis Master Service

2.1 The guestbook applications needs to communicate to the Redis master to write its data. You need to apply a Service to proxy the traffic to the Redis master Pod. A Service defines a policy to access the Pods.

Copy

```
cat > redis-master-service.yaml <<EOF

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
EOF
```

Note: This manifest file creates a Service named redis-master with a set of labels that match the labels previously defined, so the Service routes network traffic to the Redis master Pod.

2.2 Create the **redis-master-service.yaml** file

Copy

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

Output:

```
service "redis-master" created
```

2.3 Query the list of Services to verify that the Redis Master Service is running:

Copy

```
kubectl get service
```

Output:

```
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
```

```
kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP
2h

redis-master   ClusterIP   10.108.95.60     <none>        6379/TCP
4s
```

Start up the Redis Slaves

Although the Redis master is a single pod, you can make it highly available to meet traffic demands by adding replica Redis slaves.

3. Creating the Redis Slave Deployment

Deployments scale based off of the configurations set in the manifest file. In this case, the Deployment object specifies two replicas.

If there are not any replicas running, this Deployment would start the two replicas on your container cluster. Conversely, if there are more than two replicas are running, it would scale down until two replicas are running.

3.1 Create the slave deploy, from **redis-slave-deployment.yam**l file

Copy

```
cat > redis-slave-deployment.yaml <<EOF

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: redis-slave

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

EOF
```

3.2 Apply the Redis Slave Deployment from the **redis-slave-deployment.yaml** file:

Copy

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

Output:

```
deployment.apps "redis-slave" created
```

3.3 Query the list of Pods to verify that the Redis Slave Pods are running:

Copy

```
watch kubectl get pods
```

Sample output:

```
NAME                            READY     STATUS              RESTARTS
AGE

redis-master-1068406935-3lswp   1/1       Running             0
1m

redis-slave-2005841000-fpvqc    1/1       Running             0
6s

redis-slave-2005841000-phfv9    1/1       Running             0
6s
```

Note: Wait for couple of minutes to get pod status as **ready** and then press **<ctrl+c>** to interrupt.

4. Creating the Redis Slave Service

The guestbook application needs to communicate to Redis slaves to read data. To make the Redis slaves discoverable, you need to set up a Service. A Service provides transparent load balancing to a set of Pods.

4.1 Create the slave service deployment, from the **redis-slave-service.yaml** file

Copy

```
cat > redis-slave-service.yaml <<EOF

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

EOF
```

4.2 Apply the Redis Slave Service from the following redis-slave-service.yaml file:

Copy

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

Output:

```
service "redis-slave" created
```

4.3 Query the list of Services to verify that the Redis Slave Service is running:

Copy

```
kubectl get services
```

Output:

```
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE

kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP
2h
```

```
redis-master   ClusterIP   10.108.95.60     <none>        6379/TCP
5m

redis-slave    ClusterIP   10.97.140.193    <none>        6379/TCP
1m
```

5. Set up and Expose the Guestbook Frontend

The guestbook application has a web frontend serving the HTTP requests written in PHP. It is configured to connect to the **redis-master** Service for write requests and the **redis-slave** service for Read requests.

5.1 Creating the Guestbook Frontend Deployment

Copy

```
cat > frontend-deployment.yaml <<EOF

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: frontend

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

EOF
```

5.2 Apply the frontend Deployment from the following frontend-deployment.yaml file:

Copy

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

Output:

```
deployment.apps "frontend" created
```

5.3 Query the list of Pods to verify that the three frontend replicas are running:

Copy

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

Sample output:

```
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
```

Note: Wait for couple of minutes to get pods to create and running on **READY** state and then press **<ctrl+c>** to interrupt.

a. Creating the Frontend Service

The **redis-slave** and **redis-master** Services you applied are only accessible within the container cluster because the default type for a Service is ClusterIP. **ClusterIP** provides a single IP address for the set of Pods the Service is pointing to. This IP address is accessible only within the cluster.

If you want guests to be able to access your guestbook, you must configure the frontend Service to be externally visible, so a client can request the Service from outside the container cluster. Minikube can only expose Services through **NodePort**.

5.3 Create frontend-service.yaml

Copy

```
cat > frontend-service.yaml <<EOF

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

EOF
```

5.4 Apply the frontend Service from the following frontend-service.yaml file

Copy

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

Output:

```
service "frontend" created
```

5.5 Query the list of Services to verify that the frontend Service is running:

Copy

```
kubectl get services
```

Output

```
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE

frontend       NodePort    10.97.74.175     <none>        80:31288/TCP
4s

kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP
2h

redis-master   ClusterIP   10.108.95.60     <none>        6379/TCP
10m
```

```
redis-slave    ClusterIP   10.97.140.193    <none>        6379/TCP
6m
```

b. Viewing the Frontend Service via NodePort

5.6 Capture and take a note of node_port from the frontend service

Copy

```
node_port=`kubectl get services | grep frontend | awk '{print $5}' | cut -d ":" -f 2 | cut -d "/" -f 1`

echo $node_port
```

Sample output:

```
30195
```

5.7 Copy the below URL, and load the page in your browser to view your guestbook.

Copy

```
http://localhost:node_port
```

Note: Make sure the port-forwarding is configured, Ex: **30195**.

6. Scale the Web Frontend

Scaling up or down is easy because your servers are defined as a Service that uses a Deployment controller.

6.1 Run the following command to scale up the number of frontend Pods:

Copy

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

Output:

```
deployment.extensions "frontend" scaled
```

6.2 Query the list of Pods to verify the number of frontend Pods running:

Copy

```
kubectl get pods
```

Sample output:

```
NAME                                READY     STATUS    RESTARTS   AGE

frontend-5c548f4769-4lv9g               1/1       Running   0          37s

frontend-5c548f4769-7zdcb               1/1       Running   0          6m

frontend-5c548f4769-cqngw               1/1       Running   0          6m

frontend-5c548f4769-wb2st               1/1       Running   0          6m

frontend-5c548f4769-z8r2n               1/1       Running   0          37s

nginx                                   1/1       Running   0          35m

redis-master-55db5f7567-qmq4v           1/1       Running   0          16m

redis-slave-584c66c5b5-fm5cd            1/1       Running   0          11m

redis-slave-584c66c5b5-jj7p9            1/1       Running   0          11m
```

6.3 Run the following command to scale down the number of frontend Pods:

Copy

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

Output:

```
deployment.extensions "frontend" scaled
```

6.4 Query the list of Pods to verify the number of frontend Pods running:

Copy

```
kubectl get pods
```

Sample output:

```
NAME                            READY     STATUS    RESTARTS   AGE

frontend-5c548f4769-cqngw           1/1       Running   0          7m

frontend-5c548f4769-wb2st           1/1       Running   0          7m

nginx                               1/1       Running   0          37m

redis-master-55db5f7567-qmq4v       1/1       Running   0          17m

redis-slave-584c66c5b5-fm5cd        1/1       Running   0          12m

redis-slave-584c66c5b5-jj7p9        1/1       Running   0          12m
```

7. Cleanup

Deleting the Deployments and Services also deletes any running Pods. Use labels to delete multiple resources with one command.

7.1 Run the following commands to delete all Pods, Deployments, and Services.

Copy

```
kubectl delete service -l app=redis

kubectl delete service -l app=guestbook

kubectl delete deploy frontend redis-master redis-slave
```

Output:

```
deployment "redis-master" deleted

deployment "redis-slave" deleted

service "redis-master" deleted

service "redis-slave" deleted

deployment "frontend" deleted
```

```
service "frontend" deleted
```

7.2 Query the list of Pods to verify that no Pods are running:

Copy

```
kubectl get pods
```

Output:

```
No resources found.
```