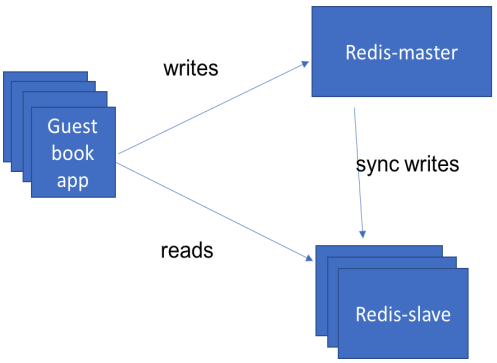
Kubernetes Guestbook Microservice



This example shows how to build a simple, multi-tier web application using Kubernetes and Docker.

The example consists of:

- A web frontend
- A redis master (for storage),
- and a replicated set of redis 'slaves'.

The web frontend interacts with the Redis master via javascript redis API calls.

Detailed step-by-step section

This section does the same thing as section 1 but is more detailed. Ensure you clean up section 1 if you want to do this section as well as section one.

Step One: Create Redis master Service and Deployment

Create the service

To create the Redis master service, use the `redis-master-service.yaml`, which describes the service pointing to the backend pods

The file `redis-master-service.yaml` defines the Redis master Service:

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

```
    # the port that this service should serve on
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: master
    tier: backend
```

Create and check the list of services, which should include the redis-master:

```
$ kubectl create -f redis-master-service.yaml
service "redis-master" created
```

```
$ kubectl get services
NAME              CLUSTER-IP       EXTERNAL-IP       PORT(S)       AGE
redis-master      10.0.76.248      <none>            6379/TCP      1s
```

Create the Deployment

To create the Redis master deployment, use the `redis-master-deployment.yaml`, which describes a single pod running a Redis key-value server in a container.

The file `redis-master-deployment.yaml` defines the Redis master Deployment:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis-master
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
  #   app: redis
  #   role: master
  #   tier: backend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 1
  # selector can be applied automatically
  # from the labels in the pod template if not set
  # selector:
  #   matchLabels:
  #     app: guestbook
  #     role: master
  #     tier: backend
  template:
    metadata:
      labels:
        app: redis
        role: master
```

```
        tier: backend
    spec:
      containers:
      - name: master
        image: gcrio/redis:e2e  # or just image: redis
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
```

```
$ kubectl create -f redis-master-deployment.yaml
deployment "redis-master" created
```

You can see the Deployment for your cluster by running:

```
$ kubectl get deployments
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
redis-master   1         1         1            1           27s
```

Then, you can list the pods in the cluster, to verify that the master is running:

```
$ kubectl get pods
```

You'll see all pods in the cluster, including the redis master pod, and the status of each pod. The name of the redis master will look similar to that in the following list:

```
NAME                            READY     STATUS    RESTARTS   AGE
redis-master-2353460263-1ecey   1/1       Running   0          1m
...
```

Step Two: Create Redis Slave Deployment and Service

Now that the redis master is running, we can start up its 'read slaves' deployment and service objects..

Create the Service:

The specification for the slaves is in the file `redis-slave-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
```

```
    tier: backend
spec:
  ports:
    # the port that this service should serve on
  - port: 6379
  selector:
    app: redis
    role: slave
    tier: backend
```

This time the selector for the Service is `app=redis,role=slave,tier=backend`, because that identifies the pods running redis slaves. It is generally helpful to set labels on your Service itself as we've done here to make it easy to locate them with the `kubectl get services -l "app=redis,role=slave,tier=backend"` command.

Create the Service in your cluster by running:

```
$ kubectl create -f redis-slave.yaml

deployment "redis-slave" created

$ kubectl get services
NAME           CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
redis-master   10.0.76.248    <none>        6379/TCP   20m
redis-slave    10.0.112.188   <none>        6379/TCP   16s
```

Create the Deployment:

The specification for the slaves is in the file `redis-slave-deployment.yaml`:

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis-slave
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
  #   app: redis
  #   role: slave
  #   tier: backend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 2
  # selector can be applied automatically
  # from the labels in the pod template if not set
  # selector:
  #   matchLabels:
  #     app: guestbook
```

```
#     role: slave
#     tier: backend
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
        # If your cluster config does not include a dns service, then to
        # instead access an environment variable to find the master
        # service's host, comment out the 'value: dns' line above, and
        # uncomment the line below.
        # value: env
      ports:
      - containerPort: 6379
```

```
$ kubectl get deployments
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
redis-master   1         1         1            1           22m
redis-slave    2         2         2            2           2m
```

Once the Deployment is up, you can list the pods in the cluster, to verify that the master and slaves are running. You should see a list that includes something like the following:

```
$ kubectl get pods
NAME                            READY     STATUS    RESTARTS   AGE
redis-master-2353460263-1ecey   1/1       Running   0          35m
redis-slave-1691881626-dlf5f    1/1       Running   0          15m
redis-slave-1691881626-sfn8t    1/1       Running   0          15m
```

You should see a single redis master pod and two redis slave pods. As mentioned above, you can get more information about any pod with: `kubectl describe pods/<POD_NAME>`.

Step Three: Start up the guestbook frontend Deployment and Services

A frontend pod is a simple PHP server that is configured to talk to either the slave or master services, depending on whether the client request is a read or a write. It exposes a simple AJAX interface, and serves an Angular-based UX. Again we'll create a set of replicated frontend pods instantiated by a Deployment — this time, with three replicas.

Create the service

To create the frontend service, use the `frontend-service.yaml`, which describes the service pointing to the backend pods

The file `frontend-service.yaml` defines the frontend Service:

```
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
```

Create and check the list of services, which should include the redis-master:

```
$ kubectl create -f frontend-service.yaml
service "frontend" created
```

```
$ kubectl get services
NAME              CLUSTER-IP       EXTERNAL-IP       PORT(S)       AGE
front-end      10.0.76.248      <none>            6379/TCP      1s
```

As with the other pods, we now want to create a frontend-deployment.yaml. The

Deployment file `frontend.yaml`:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
  #   app: guestbook
```

```
  #   tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  # selector can be applied automatically
  # from the labels in the pod template if not set
  # selector:
  #   matchLabels:
  #     app: guestbook
  #     tier: frontend
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:v5
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

Create the Deployment like this:

```
$ kubectl create -f frontend.yaml

deployment "frontend" created
```

Then, list all your services again:

```
$ kubectl get services
NAME           CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
frontend       10.0.63.63     <none>        80/TCP     1m
redis-master   10.0.76.248    <none>        6379/TCP   39m
redis-slave    10.0.112.188   <none>        6379/TCP   19m
```

Also list all your Deployments:

```
$ kubectl get deployments
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
```

```
frontend       3         3         3           3           2m
redis-master   1         1         1           1           39m
redis-slave    2         2         2           2           20m
```

Once it's up, i.e. when desired replicas match current replicas (again, it may take up to thirty seconds to create the pods), you can list the pods with specified labels in the cluster, to verify that the master, slaves and frontends are all running. You should see a list containing pods with label 'tier' like the following:

```
$ kubectl get pods -L tier
NAME                            READY     STATUS    RESTARTS   AGE       TIER
frontend-1211764471-4e1j2       1/1       Running   0          4m        frontend
frontend-1211764471-gkbkv       1/1       Running   0          4m        frontend
frontend-1211764471-rk1cf       1/1       Running   0          4m        frontend
redis-master-2353460263-1ecey   1/1       Running   0          42m       backend
redis-slave-1691881626-dlf5f    1/1       Running   0          22m       backend
redis-slave-1691881626-sfn8t    1/1       Running   0          22m       backend
```

You should see a single redis master pod, two redis slaves, and three frontend pods.

Accessing the Frontend Application:

There are several ways for you to access the guestbook.

ClusterIP: You can access the guestbook from the web browser with frontend Service objects like:

`<Cluster-IP>:<PORT>`

e.g. `10.0.0.117:80`

`<Cluster-IP>` is a cluster-internal IP.

NodePort: If you want to access the guestbook from outside the cluster, modify the service file to have:

`type: NodePort` to the frontend Service `spec` field. Then you can access the guestbook with

`<NodeIP>:NodePort`

from outside the cluster.

On cloud providers that support external load balancers, adding

`type: LoadBalancer` to the frontend Service `spec` field will provision a load balancer for your Service.

Depending on the type of service object used, you can visit the frontend microservice using the service object. This is an example using the LoadBalancer service type.

i.e. `http://<EXTERNAL-IP>:<PORT>`.

You should see a web page that looks something like this (without the messages). Try adding some entries to it!

Step Four: Cleanup

```
$ kubectl delete deployments,services -l "app in (redis, guestbook)"
```

or

```
$ kubectl delete -f .
```