Lab: IBM Kubernetes Service Essentials

This lab is an introduction to using containers in Kubernetes on IBM Cloud. By the end of the course, you'll achieve these objectives:

- Understand core concepts of Kubernetes
- Build a Docker image and deploy an application to Kubernetes run by the IBM Kubernetes Service (IKS)
- Control application deployments, while minimizing your time with infrastructure management
- Add AI services to extend your app
- Secure and monitor your cluster and app

Exercise 1: Deploy Container Image

In this exercise, we will deploy an application called `guestbook`. A container image for this app is already built and uploaded to DockerHub under the name `ibmcom/guestbook:v1`.

When deploying this app, we will be using Kubernetes artifacts such as `deployments`, `pods` and `services`. To learn how these work, you can check out this 10 min video overviewing Kubernetes.

To begin, we need to start a container with our running application. The container itself is a set of metadata, bins, libraries and application code. We've already packaged this container and placed it on DockerHub, a repository for containers. At the simpliest level, a Kubernetes `deployment` specifies where to download the container image from, and how many copies of the container to start. These running containers are called "pods". Let's create a `deployment` now.

Step 1. Start by creating a `guestbook` deployment:

```
kubectl create deployment guestbook --image=ibmcom/guestbook:v1
```

The command comes back immediately, but it takes sometime for the pods in the deployment to start. To check the status of the running application, you can use:

```
kubectl get pods
```

If you were very fast, you should see output similar to the following:

```
$ kubectl get pods
NAME                          READY     STATUS              RESTARTS   AGE
guestbook-59bd679fdc-bxdg7    0/1       ContainerCreating   0          1m
```

Eventually, the status should show up as `Running`:

```
$ kubectl get pods
NAME                          READY     STATUS              RESTARTS   AGE
guestbook-59bd679fdc-bxdg7    1/1       Running             0          1m
```

The end result of the `create deployment` command is not just the pod containing our application containers, but a Deployment resource that manages the lifecycle of those pods.

Step 2. Once the status reads `Running`, we need to expose that `deployment` as a `service` so we can access it. The `guestbook` application listens on port 3000. Run:

```
kubectl expose deployment guestbook --type="NodePort" --port=3000
```

Output:

```
$ kubectl expose deployment guestbook --type="NodePort" --port=3000
service "guestbook" exposed
```

This created a new resource in Kubernetes -- a `service`. A `service` and `deployment` are two essential building blocks for creating an application and accessing it.

Step 3. To find the port used on that worker node, examine your new `service`:

```
kubectl get service guestbook
```

Output:

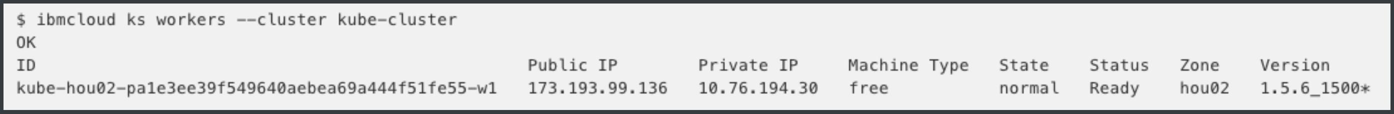
```
$ kubectl get service guestbook
NAME        TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
guestbook   NodePort   10.10.10.253   .none.        3000:31208/TCP   1m
```

In column PORT(S), you can see that internal port 3000 is mapped to the external port 31208. This means that our `<nodeport>` is `31208`. The nodeport is always in the 30000 range, it is automatically created, and could be different in your case. Remember this commmand, you will need it later in order to find out ports of your other services.

Step 4. `guestbook` is now running on your cluster, and exposed to the internet. We need to find out where it is accessible. The worker nodes running in the container service get external IP addresses. Run the following command and **replace `<your-cluster>`** with the name of your IKS cluster, for example *mycluster-free* (if you used the default name during the cluster creation).

```
ibmcloud ks workers --cluster <your-cluster>
```

You will see output like below but the result value will be different. Note the public IP listed in the `<public-IP>` column:



```
$ ibmcloud ks workers --cluster kube-cluster
OK
ID                                                Public IP        Private IP     Machine Type   State    Status   Zone    Version
kube-hou02-pa1e3ee39f549640aebea69a444f51fe55-w1   173.193.99.136   10.76.194.30   free           normal   Ready    hou02   1.5.6_1500*
```

In this example, our `<public-IP>` is `173.193.99.136`. Yours will be different. Remember this commmand, you will need it several times later in order to get public IP addresses of your nodes.

Step 5. Now that you have both the address and the port, you can access the application in the web browser at `<public-IP>:<nodeport>`. In the example case this is `173.193.99.136:31208`.

Congratulations, you've now deployed an application to Kubernetes!

Exercise 2: Using Replicas, Update and Rollback Apps

In this section, you'll learn how to update the number of instances a deployment has and how to safely roll out an update of your application on Kubernetes.

Scale apps with replicas

A *replica* is a copy of a pod that contains a running service. By having multiple replicas of a pod, you can ensure your deployment has the available resources to handle increasing load on your application. Multiple replicase also increase availability of your app.

Step 1. `kubectl` provides a `scale` subcommand to change the size of an existing deployment. Let's increase our capacity from a single running instance of `guestbook` up to 10 instances:

```
kubectl scale --replicas=10 deployment guestbook
```

Kubernetes will now try to make reality match the desired state of 10 replicas by starting 9 new pods with the same configuration as the first.

Step 2. To see your changes being rolled out, you can run:

```
kubectl rollout status deployment guestbook
```

The rollout might occur so quickly that the following messages might *not* display:

```
$ kubectl rollout status deployment guestbook
Waiting for rollout to finish: 1 of 10 updated replicas are available...
Waiting for rollout to finish: 2 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 4 of 10 updated replicas are available...
Waiting for rollout to finish: 5 of 10 updated replicas are available...
Waiting for rollout to finish: 6 of 10 updated replicas are available...
Waiting for rollout to finish: 7 of 10 updated replicas are available...
Waiting for rollout to finish: 8 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
deployment "guestbook" successfully rolled out
```

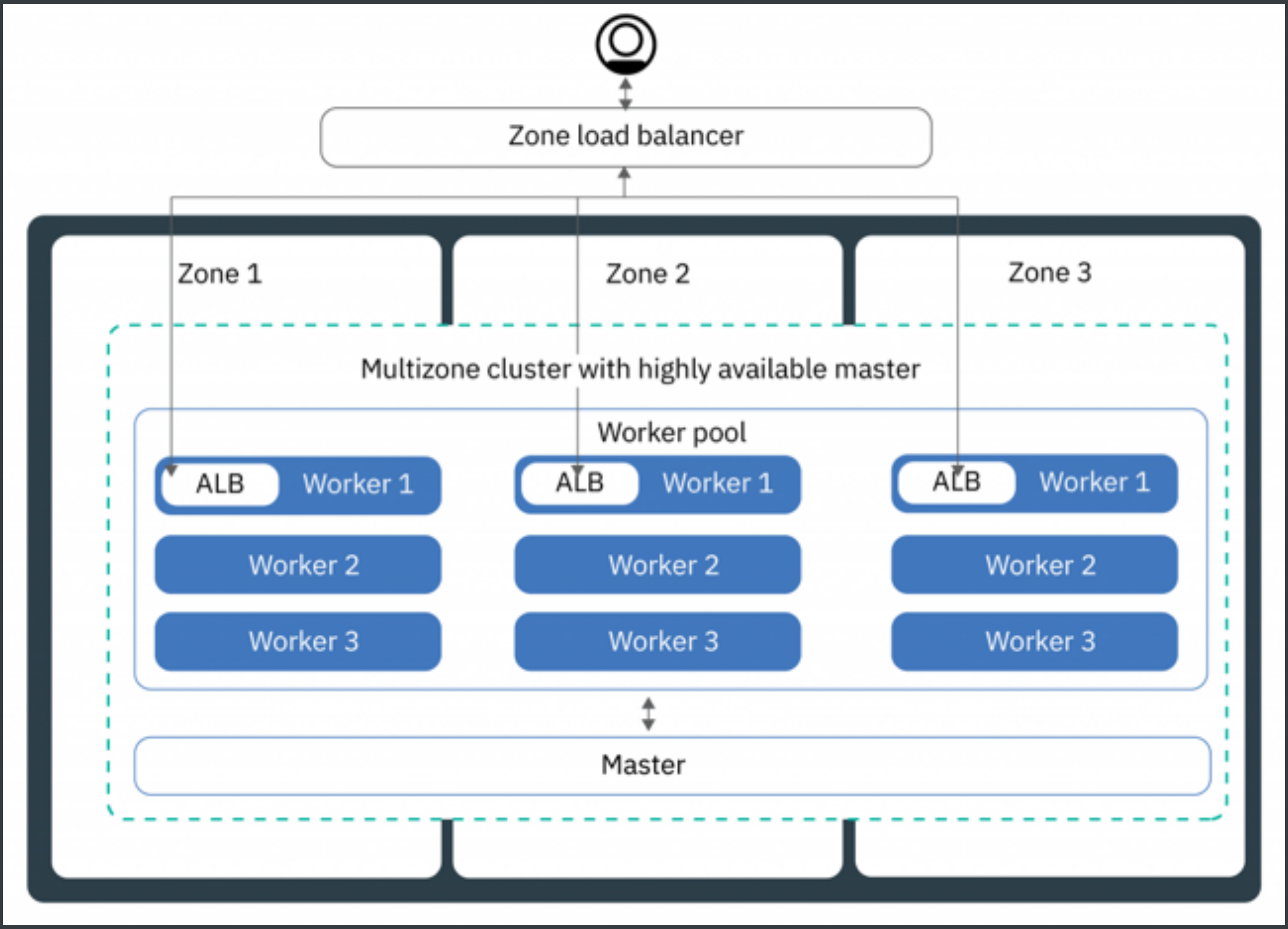
Step 3. Once the rollout has finished, ensure your pods are running by using:

```
kubectl get pods
```

You should see output listing 10 replicas of your deployment:

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
guestbook-562211614-1tqm7     1/1       Running   0          1d
guestbook-562211614-1zqn4     1/1       Running   0          2m
guestbook-562211614-5htdz     1/1       Running   0          2m
guestbook-562211614-6h04h     1/1       Running   0          2m
guestbook-562211614-ds9hb     1/1       Running   0          2m
guestbook-562211614-nb5qp     1/1       Running   0          2m
guestbook-562211614-vtfp2     1/1       Running   0          2m
guestbook-562211614-vz5qw     1/1       Running   0          2m
guestbook-562211614-zksw3     1/1       Running   0          2m
guestbook-562211614-zsp0j     1/1       Running   0          2m
```

> **Tip:** Another way to improve availability is to use multizone clusters, spreading your application over multiple datacenters in the same region, as shown in the following diagram:



Update and Roll Back Apps

Kubernetes allows you to do a rolling upgrade of your application to a new container image. With this, you can easily update the running image and also easily undo a rollout if a problem is discovered during or after deployment.

In the previous lab, we used an image with a `v1` tag. For our upgrade we'll use the image with the `v2` tag.

To update and roll back:

Step 1. Using `kubectl`, you can now update your deployment to use the `v2` image. `kubectl` allows you to change details about existing resources with the `set` subcommand. We can use it to change the image being used.

```
kubectl set image deployment/guestbook guestbook=ibmcom/guestbook:v2
```

Note that a pod could have multiple containers, each with its own name. Each image can be changed individually or all at once by referring to the name. In the case of our `guestbook` Deployment, the container name is also `guestbook`.

Step 2. Run the following to check the status of the rollout.

```
kubectl rollout status deployment/guestbook
```

The rollout might occur so quickly that the following messages might *not* display::

```
$ kubectl rollout status deployment/guestbook
Waiting for rollout to finish: 2 out of 10 new replicas have been updated...
Waiting for rollout to finish: 3 out of 10 new replicas have been updated...
Waiting for rollout to finish: 3 out of 10 new replicas have been updated...
Waiting for rollout to finish: 3 out of 10 new replicas have been updated...
Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
Waiting for rollout to finish: 4 out of 10 new replicas have been updated...
Waiting for rollout to finish: 5 out of 10 new replicas have been updated...
Waiting for rollout to finish: 5 out of 10 new replicas have been updated...
Waiting for rollout to finish: 5 out of 10 new replicas have been updated...
Waiting for rollout to finish: 6 out of 10 new replicas have been updated...
Waiting for rollout to finish: 6 out of 10 new replicas have been updated...
Waiting for rollout to finish: 6 out of 10 new replicas have been updated...
Waiting for rollout to finish: 7 out of 10 new replicas have been updated...
Waiting for rollout to finish: 7 out of 10 new replicas have been updated...
Waiting for rollout to finish: 7 out of 10 new replicas have been updated...
Waiting for rollout to finish: 7 out of 10 new replicas have been updated...
Waiting for rollout to finish: 8 out of 10 new replicas have been updated...
Waiting for rollout to finish: 8 out of 10 new replicas have been updated...
Waiting for rollout to finish: 8 out of 10 new replicas have been updated...
Waiting for rollout to finish: 8 out of 10 new replicas have been updated...
Waiting for rollout to finish: 9 out of 10 new replicas have been updated...
Waiting for rollout to finish: 9 out of 10 new replicas have been updated...
Waiting for rollout to finish: 9 out of 10 new replicas have been updated...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
deployment "guestbook" successfully rolled out
```

Step 3. Test the application as before, by accessing `<public-IP>:<nodeport>` in the browser to confirm your new code is active. To verify that you're running "v2" of guestbook, look at the title of the page, it should now be `Guestbook - v2`. You may need to do a "cache-less" reload of the web-page to refresh the cache -- `Ctrl+Shift+R` (on Windows) or `Cmd+Shift+R` (on Mac).

> **Tip:** If you don' t remember your IP address and port, you can get them with commands:
>
> `kubectl get svc`
>
> `ibmcloud ks workers --cluster` `<your-cluster>`

Step 4. If you want to undo your latest rollout, use:

```
kubectl rollout undo deployment guestbook
```

You can then use `kubectl rollout status deployment/guestbook` to see the status.

Step 5. When doing a rollout, you see references to *old* replicas and *new* replicas.

- The **old** replicas are the original 10 pods deployed when we scaled the application.
- The **new** replicas come from the newly created pods with the different image.

All of these pods are owned by the Deployment. The deployment manages these two sets of pods with a resource called a ReplicaSet. We can see the guestbook ReplicaSets with:

```
kubectl get replicaset
```

Output:

```
$ kubectl get replicaset
NAME                    DESIRED   CURRENT   READY     AGE
guestbook-5f5548d4f     10        10        10        21m
guestbook-768cc55c78    0         0         0         3h
```

Congratulations! You deployed the second version of the app.

Before we continue, let's delete the application so we can learn about a different way to achieve the same results.

Remove deployment :

```
kubectl delete deployment guestbook
```

Remove service:

```
kubectl delete service guestbook
```

Exercise 3: Using Configuration Files

In this lab you'll learn how to deploy the same guestbook application we deployed in the previous labs, however, instead of using the `kubectl` command line helper functions we'll be deploying the application using configuration files. The configuration file mechanism allows you to have more fine-grained control over all of resources being created within the Kubernetes cluster.

Before we work with the application we need to clone a github repo:

```
git clone https://github.com/IBM/guestbook.git
```

This repo contains multiple versions of the guestbook application as well as the configuration files we'll use to deploy the pieces of the application.

Change directory by running the command `cd guestbook`. You will find all the configurations files for this exercise under the directory `v1`.

```
cd guestbook
cd v1
```

Scale apps natively

Kubernetes can deploy an individual pod to run an application, but when you need to scale it to handle a large number of requests, a `Deployment` is the resource you want to use. A Deployment manages a collection of similar pods. When you ask for a specific number of replicas the Kubernetes Deployment Controller will attempt to maintain that number of replicas at all times.

Every Kubernetes object we create contains two nested object fields that govern the object's configuration: the field `spec` and the field `status`.

Field `spec` defines the desired state of an object (what we want), while field `status` shows the current state (what is now). Field `status` is provided by the Kubernetes system, we don't define it by ourselves. As described before, Kubernetes will attempt to reconcile your desired state with the actual state of the system.

A configuration file for every object that we create, must contain these four fields : `apiVersion`, `kind`, `metadata`, and `spec`.

Have a look at the following deployment configuration for our `guestbook` application.

guestbook-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook-v1
  labels:
    app: guestbook
    version: "1.0"
```

```yaml
spec:
  replicas: 3
  selector:
    matchLabels:
      app: guestbook
  template:
    metadata:
      labels:
        app: guestbook
        version: "1.0"
    spec:
      containers:
      - name: guestbook
        image: ibmcom/guestbook:v1
        ports:
        - name: http-server
```

The above configuration file is used to create a deployment object named 'guestbook' with a pod containing a single container running the image `ibmcom/guestbook:v1`. Also this configuration specifies replicas set to 3 so Kubernetes always tries to maintain exactly three active pods at all times.

Step 1. Create guestbook deployment

```
kubectl create -f guestbook-deployment.yaml
```

Step 2. List the pod with label app=guestbook

We can now list all pods created with this deployment, by listing all pods that have a label "app" with a value of "guestbook". This matches the labels defined in the above yaml file in the `spec.template.metadata.labels` section.

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

Step 3. Editing a deployment

When you change the number of replicas in the configuration, Kubernetes will try to add, or remove, pods from the system to match your request. You can make these modifications by using the following command -- you don't have to run this command now.

```
kubectl edit deployment guestbook-v1
```

> **Tip:** Above command will open up your default text editor, you can make changes and then save the file.

This will retrieve the latest configuration for the Deployment from the Kubernetes server and then load it into an editor for you. You'll notice that there are a lot more fields than in the original yaml file we used. This is because the file contains all of the properties about the deployment object that Kubernetes knows about, not just the ones we chose to specify when we create it. Also notice that it now contains the `status` section mentioned previously.

You can also edit the original deployment file we used to create the Deployment to make changes. You should use the following command to make the change effective when you edit the deployment locally.

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

This will ask Kubernetes to "diff" our yaml file with the current state of the Deployment and apply just those changes.

Step 4. Create Service object to expose the deployment to external clients. We will use the below configuration file to create a Service resource named guestbook.

guestbook-service.yaml

```
apiVersion: v1
kind: Service
  metadata:
    name: guestbook
    labels:
      app: guestbook
  spec:
    ports:
    - port: 3000
      targetPort: http-server
    selector:
      app: guestbook
    type: LoadBalancer
```

A Service can be used to create a network path for incoming traffic to your running application. In this case, we are setting up a route from port 3000 on the cluster to the "http-server" port on our app, which is port 3000 per the Deployment container spec.

Step 5. Create guestbook service using the same type of command we used when we created the Deployment:

```
kubectl create -f guestbook-service.yaml
```

Step 6. Test guestbook app in a browser of your choice using the url `<your-public-ip>:<node-port>`.

Note that your port probably changed since we created a new service. If you need to find the IP and port again, try `kubectl get svc` and `ibmcloud ks workers --cluster <your-cluster>`.

Note: If you are using an IKS Free plan, the external IP address of this particular service will be indefinitely in status *pending*. This is because the service type LoadBalancer is not supported in Free plan, it only works properly in the Standard plan. But our lab will work anyway as we won't use the external IP address of this service now.

Exercise 4: Connect to a Back-end Service

If you look at the guestbook source code, under the `guestbook/v1/guestbook` directory, you'll notice that it is written to support a variety of data stores.

By default it will keep the log of guestbook entries in memory. That's ok for testing purposes, but as you get into a more "real" environment where you scale your application that model will not work because based on which instance of the application the user is routed to they'll see very different results.

To solve this we need to have all instances of our app share the same data store - in this case we're going to use a redis database that we deploy to our cluster. This instance of redis will be defined in a similar manner to the guestbook as shown in the below file.

redis-master-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
      role: master
  template:
    metadata:
      labels:
        app: redis
        role: master
    spec:
      containers:
      - name: redis-master
        image: redis:2.8.23
        ports:
        - name: redis-server
          containerPort: 6379
```

We will use this yaml to create a redis database in a Deployment named `redis-master`. It will create a single instance, with replicas set to 1. The guestbook app instances will connect to it to persist data, as well as read the persisted data back.

The image running in the container is 'redis:2.8.23' . This deployment will be available on the standard redis port 6379.

Step 1. Create a redis Deployment, like we did for guestbook:

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

Step 2. Check to see that redis server pod is running:

```
kubectl get pods -l app=redis,role=master
```

Output:

```
$ kubectl get pods -l app=redis,role=master
NAME                 READY     STATUS    RESTARTS   AGE
redis-master-q9zg7   1/1       Running   0          2d
```

Step 3. Test the redis standalone.

Edit the pod name in the below command to the one you got from previous command - replace XXXX with relevant characters. The following command will open a shell into the pod and run the `redis-cli` tool.

```
kubectl exec -it redis-master-XXXX redis-cli
```

The kubectl exec command will start a secondary process in the specified container. In this case we're asking for the "redis-cli" command to be executed in the container named "redis-master-q9zg7". When this process ends the "kubectl exec" command will also exit but the other processes in the container will not be impacted.

Once in the container, we can use the redis-cli commands to check if the redis database is running properly, or to configure it if needed. Enter the ping command, observe the response, then exit the container as shwon below:

```
127.0.01:6379> ping
PONG
127.0.01:6379> exit
```

Step 4. Expose `redis-master` Deployment

Now we need to expose the `redis-master` Deployment as a Service so that the guestbook application can connect to it through DNS lookup. Below is our Service's configuration file.

redis-master-service.yaml

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

This YAML creates a Service object named 'redis-master' and configures it to target port 6379 on the pods selected by the selectors "app=redis" and "role=master".

Step 5. Create the service to access redis master

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

Step 6. Restart guestbook

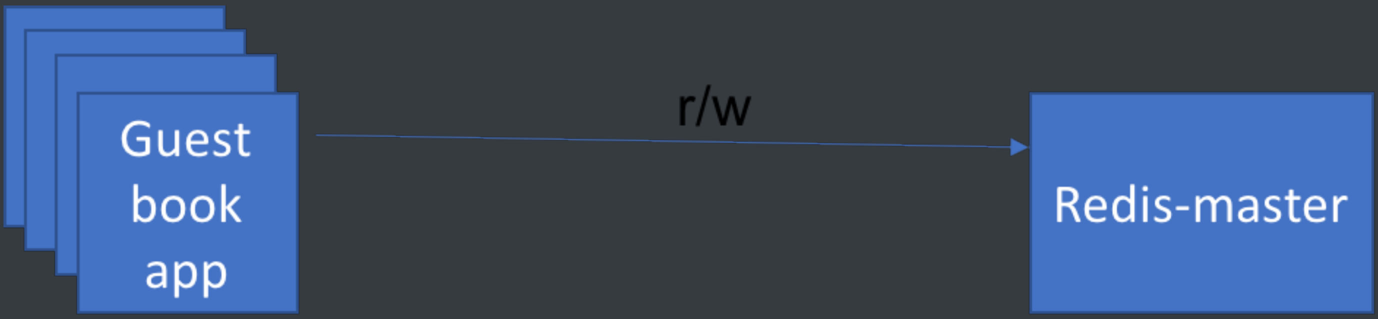
Let's restart the guestbook so that it will find the redis service to use database.

```
kubectl delete deploy guestbook-v1
kubectl create -f guestbook-deployment.yaml
```

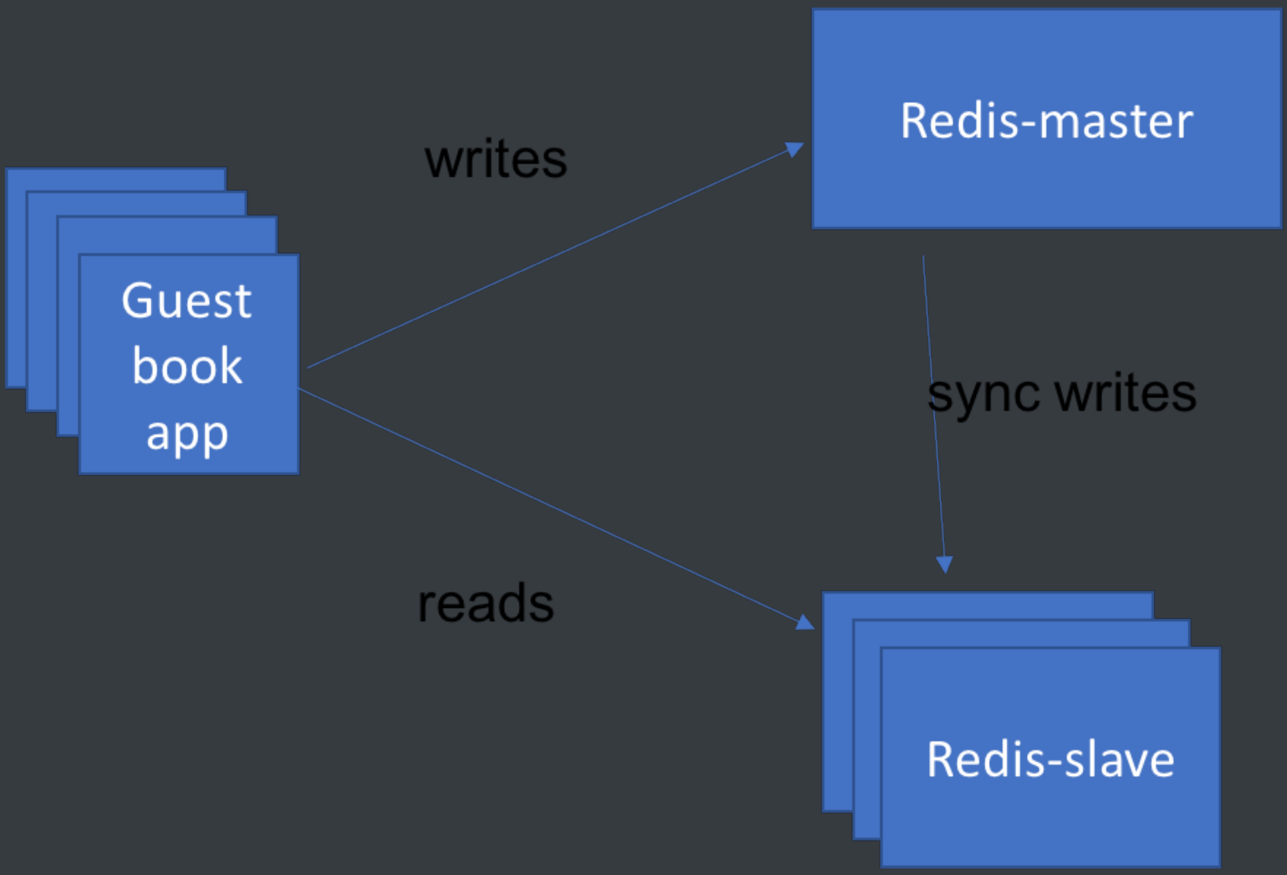
Step 7. Test guestbook app and input some sample messages into the application. As previously, use a browser of your choice with the url: `<your-public-ip>:<node-port>`

You can see now that if you open up multiple browsers and refresh the page to access the different copies of guestbook that they all have a consistent state. All instances write to the same backing persistent storage, and all instances read from that storage to display the guestbook entries that have been stored.

We have our simple 3-tier application running but we will need to scale the application if traffic increases. Our main bottleneck is that we only have one database server to process all requests coming through guestbook as shown below.



One simple solution is to separate the reads and writes so that they go to different databases. The databases are replicated properly to achieve data consistency. For this purpose, we'll create a deployment named 'Redis-slave' that can talk to the main database ('Redis-master'). 'Redis-slave' will be used to manage data reads while 'Redis-master' will be used for data writes. In order to scale the database we use the pattern where we can scale the reads using redis slave deployment which can run several instances to read. Redis slave deployments are configured to run two replicas. The new database setup is shown below:



Below is the configuration file of our new Redis-slave deployment.

redis-slave-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-slave
  labels:
    app: redis
```

```
      role: slave
spec:
  replicas: 2
  selector:
    matchLabels:
      app: redis
      role: slave
  template:
    metadata:
      labels:
        app: redis
        role: slave
    spec:
      containers:
      - name: redis-slave
        image: kubernetes/redis-slave:v2
        ports:
        - name: redis-server
          containerPort: 6379
```

Step 1. Create the object for running redis slave deployment.

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

Step 2. Check if both slave replicas are running.

```
kubectl get pods -l app=redis,role=slave
```

Output:

```
$ kubectl get pods -l app=redis,role=slave
NAME                 READY     STATUS     RESTARTS    AGE
redis-slave-kd7vx    1/1       Running    0           2d
redis-slave-wwcxw    1/1       Running    0           2d
```

Step 3. Test standalone pod

Edit one of the two pods (doesn't matter which one) using the below command to go into the pod and look at its database. You should see there the content you entered in the guestbook app. Don't forget to replace XXXX with proper character from your pod's name:

```
kubectl exec -it redis-slave-XXXX redis-cli
127.0.0.1:6379> keys *
1) "guestbook"
127.0.0.1:6379> lrange guestbook 0 10
1) "hello world"
2) "welcome to the Kube workshop"
127.0.0.1:6379> exit
```

Step 4. Deploy redis slave service

Let's deploy the redis slave service so we can access it by DNS name. Once redeployed, the application will send "read" operations to the `redis-slave` pods while "write" operations will go to the `redis-master` pods. This is the pertaining configuration file:

redis-slave-service.yaml

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

```
        role: slave
```

Step 5. Create service to access redis-slave

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

Step 6. Restart guestbook

Let's restart the guestbook application so that it will find the slave service to read from.

```
kubectl delete deploy guestbook-v1
kubectl create -f guestbook-deployment.yaml
```

Step 7. Test guestbook app in a browser of your choice using the url `<your-public-ip>:<node-port>`.

Congratulations! That's the end of the lab. Now let's clean-up our environment:

```
kubectl delete -f guestbook-deployment.yaml
kubectl delete -f guestbook-service.yaml
kubectl delete -f redis-slave-service.yaml
kubectl delete -f redis-slave-deployment.yaml
kubectl delete -f redis-master-service.yaml
kubectl delete -f redis-master-deployment.yaml
```

End of Lab