Containers and Kubernetes Essentials

Contents

Getting Started

Objectives

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

- Understand core concepts of Kubernetes
- Build a Docker image and deploy an application on Kubernetes in the IBM Cloud Kubernetes Service
- 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: Configure Cluster Access

Login to IBM Cloud

IBM Cloud CLI

IBM Cloud provides a Command Line Interface (CLI) used to interact with the platform and cloud services. For all Exercises in this lab, the IBM Cloud and Kubernetes CLIs are already installed and accessible in the terminal to the right. For future reference, the CLI is installed with a single command on Mac, Linux or Windows using the steps outlined in in this link.

Throughout this lab you will be presented with commands to run, as shown below. To run each command, click the command text and observe when it runs in the terminal to the right. Also, when you run some commands that return information, it will be displayed in the terminal window.

Log in to IBM Cloud

Step 1. Click the command below to log into IBM Cloud in your terminal

```
ibmcloud login --sso -a cloud.ibm.com -r us-south --apikey ##KUBE.apikey##
```

The CLI should return something like this:

```
Authenticating...
OK

Targeted account Cloud Lab Tutorials (c098d833f9b04883942bcec5c7b6a37d) <-> 2030430

Targeted resource group (resource group name here)

Targeted region (region here)


API endpoint:      https://cloud.ibm.com
Region:            (region here)
User:              (your IBM id here)
Account:           Cloud Lab Tutorials (c098d833f9b04883942bcec5c7b6a37d) <-> 2030430
Resource group:    No resource group targeted, use 'ibmcloud target -g RESOURCE_GROUP'
CF API endpoint:
Org:
Space:

Do you want to send usage statistics to IBM? [y/n]>
```

Step 2. If you're prompted to send usage statistics to IBM, enter **Y** or **N** and hit enter

You are now logged into the lab account!

Setup CLI to Access IKS

In this section, you will setup CLI access to the IBM Kubernetes cluster.

Run the following command. In order for the CLI to be able to run commands against the cluster, we will need to download the Kubernetes configuration file. Run the following command which will download the config from IBM Cloud for the cluster: ##KUBE.id##. This enables the `kubectl` CLI to work.

```
ibmcloud ks cluster config --cluster ##KUBE.id##
```

Once your client is configured, you are ready to deploy your first application, `guestbook`.

End of Exercise 1

Exercise 2: Deploy an Application

Deploy Container Image

In this part of the lab we will deploy an application called `guestbook` that has already been built and uploaded to DockerHub under the name `ibmcom/guestbook:v1`.

Step 1. Start by running `guestbook`:

```
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
```

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 through the IP of the worker nodes. 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
```

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
```

We can see that our `<nodeport>` is `31208`. In the output, the port indicates that the internal port 3000 is mapped to the external port 31208.) This port in the 30000 range is automatically chosen, and could be different for you.

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 and note the public IP listed on the `<public-IP>` line:

```
ibmcloud ks workers --cluster ##KUBE.name##
```

You will see output like below but the result value will be different:

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

In this example, we see that our `<public-IP>` is `173.193.99.136`.

Step 5. Now that you have both the address and the port, you can now 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!

End of Exercise 2

Exercise 3: Scale, Update and Rollback

Using Replicas

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.

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:

```execute
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:

```execute
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:

~~~
```execute
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. This allows you to easily update the running image and also allows you to 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.

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

`execute   kubectl rollout status deployment/guestbook`

Output: `console $ 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`

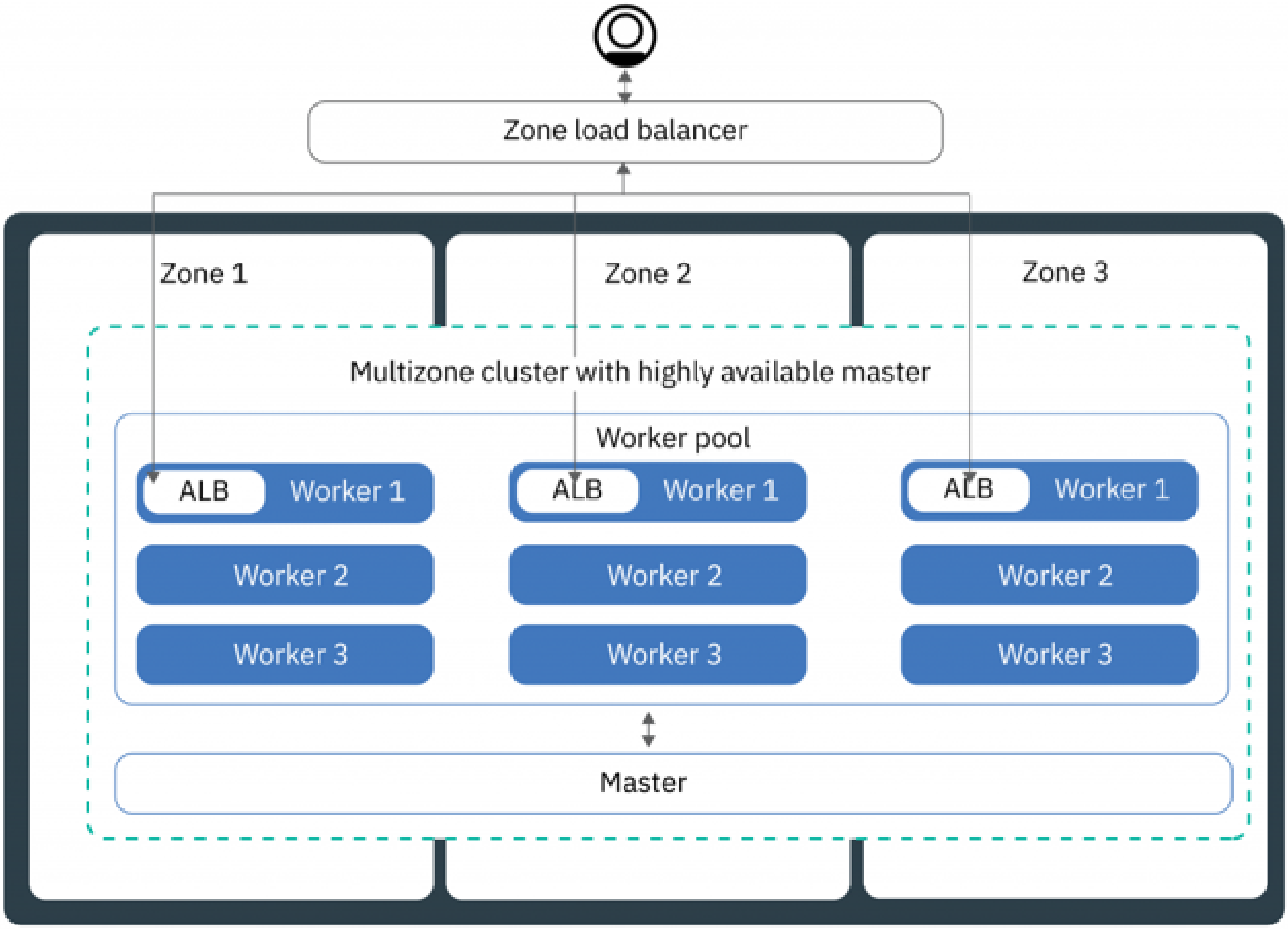

Figure 1: HA with more clusters and regions

```
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` or `Cmd+Shift+R`.

> **Tip:** Use `kubectl get svc` and `kubectl ks workers --cluster clusterName`

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

```
execute    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:

```
execute    kubectl get replicaset
```

Output:

```
        console    $ 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
```

End of Exercise 3

Exercise 4: Deploy App with Config YAML

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 && ls
```

```
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 should provide two nested object fields that govern the object's configuration: the object `spec` and the object `status`.

Object `spec` defines the desired state, and object `status` contains Kubernetes system provided information about the actual state of the resource. As described before, Kubernetes will attempt to reconcile your desired state with the actual state of the system.

For an Object that we create, we need to provide the `apiVersion`, `kind`, and `metadata` about the object.

Consider the following deployment configuration for `guestbook` application.

guestbook-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook
  labels:
    app: guestbook
spec:
  replicas: 3
  selector:
    matchLabels:
      app: guestbook
  template:
    metadata:
      labels:
        app: guestbook
    spec:
      containers:
      - name: guestbook
        image: ibmcom/guestbook:v1
        ports:
        - name: http-server
          containerPort: 3000
```

The above configuration file create a deployment object named ‘guestbook’ with a pod containing a single container running the image `ibmcom/guestbook:v1`. Also the configuration specifies replicas set to 3 and Kubernetes tries to make sure that exactly three active pods are running at all times.

Step 1. Create guestbook deployment

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

Step 2. List the pod with label app=guestbook

We can then list the pods it created by listing all pods that have a label of “app” with a value of “guestbook”. This matches the labels defined above in the yaml file in the `spec.template.metadata.labels` section.

```
execute     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 the vi editor, you can follow the vi syntax , make changes and then save it using :wq or quit using :q!

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 in this version than the original yaml file we used. This is because it contains all of the properties about the Deployment 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 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.

```
execute    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.

guestbook-service.yaml

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

The above configuration creates a Service resource named guestbook. 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:

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

Step 6. Test guestbook app using a browser of your choice using the url `<your-cluster-ip>:<node-port>`. If you forgot what the IP and port area, use `kubectl get svc` and `ibmcloud ks workers --cluster ##KUBE.name##`.

End of Exercise 4

Exercise 5: Connect with Redis Storage

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.

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
```

This yaml creates a redis database in a Deployment named `redis-master`. It will create a single instance, with replicas set to 1, and 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' and exposes the standard redis port 6379.

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

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

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

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

Output: `console    $ 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. The following command will open a shell into the pod and run the `redis-cli` tool.

```
copyCommand    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" command to make sure the redis database is running properly, or to configure it if needed.

```
console    redis-cli> ping    PONG    redis-cli> 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.

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 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

```
execute    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.

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

Step 7. Test guestbook app and input some sample messages into the application.

Using a browser of your choice using the url: `<your-cluster-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 need to scale the application if traffic increases. Our main bottleneck is that we only have one database server to process each request coming though guestbook. One simple solution is to separate the reads and writes such that they go to different databases that are replicated properly to achieve data consistency.

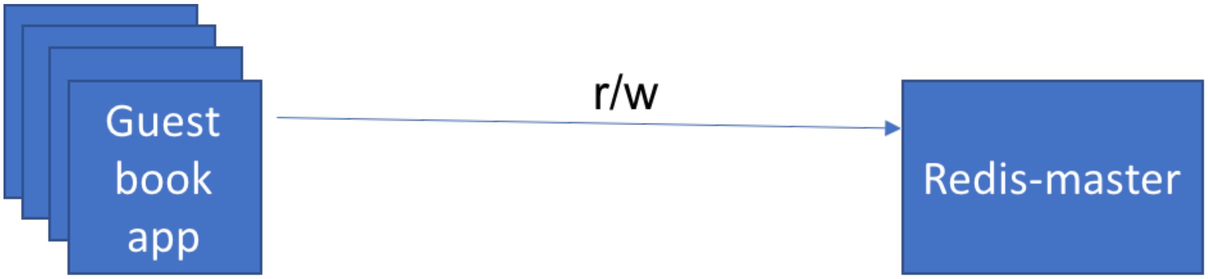

Figure 2: rw_to_master

Next, we'll create a deployment named 'redis-slave' that can talk to redis database to manage data reads. 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.

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
```

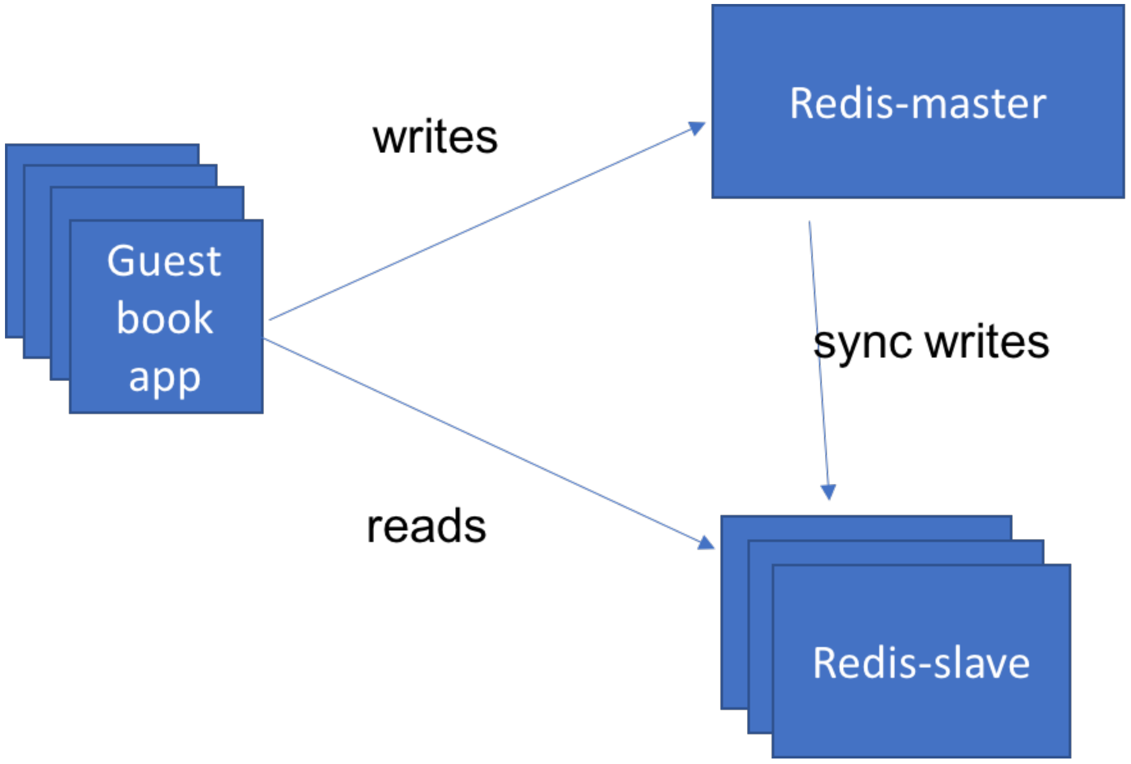

Figure 3: w_to_master-r_to_slave

```
        role: slave
    spec:
      containers:
      - name: redis-slave
        image: kubernetes/redis-slave:v2
        ports:
        - name: redis-server
          containerPort: 6379
```

Step 1. Create the pod running redis slave deployment.

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

Step 2. Check if all the slave replicas are running

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

Output: `console    $ 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 the pod name in the below command to go into one of those pods and look at the database to see that everything looks right:

```
copyCommand    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.

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

```
execute    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.

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

Step 7. Test guestbook app

Using a browser of your choice using the url `<your-cluster-ip>:<node-port>`.

Congratluations! 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 Exercise 5

Next Steps

Next Steps

Next Steps

You have completed the **Containers and Kubernetes Essentials** Hands-on-Lab.

In this lab, you learned how to:

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

Don't stop! In the next lab learn how to **Scale Web Applications on Kubernetes**.

We would also like to get your feedback. Please take a minute to complete the survey about this lab.

`IKSLab1`

End of Lab