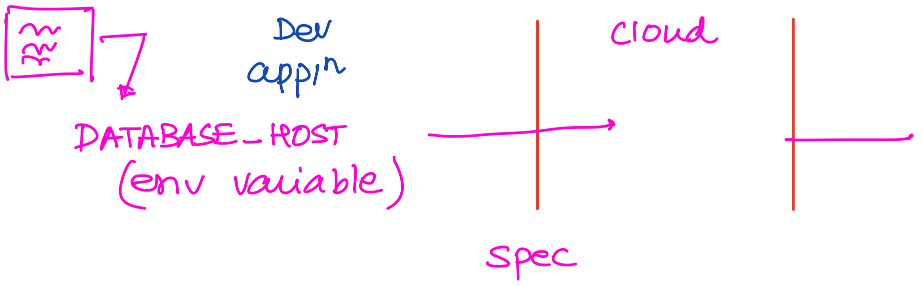
$$\begin{tikzcd}
\text{Service} \arrow[r] \arrow[dr] \arrow[d] & \text{Network abstraction} \\
\text{LB} & \text{TCP}
\end{tikzcd}$$

→ ConfigMap — API object used for storing non-confidential data in key-value pairs.

Reason
Decouple env-specific config from container images.

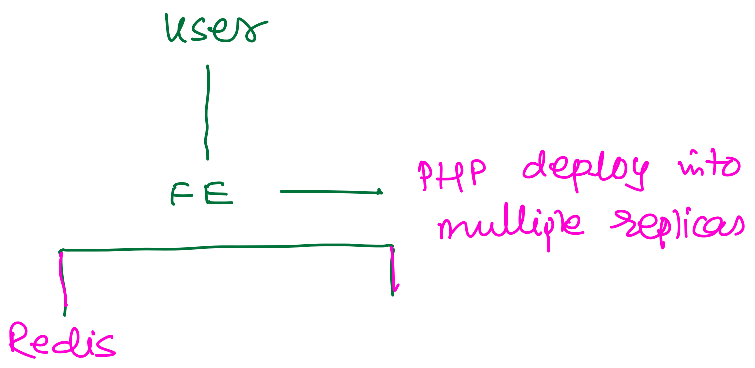
Pod — consume — env vars,
command line arguments
config file in a volume

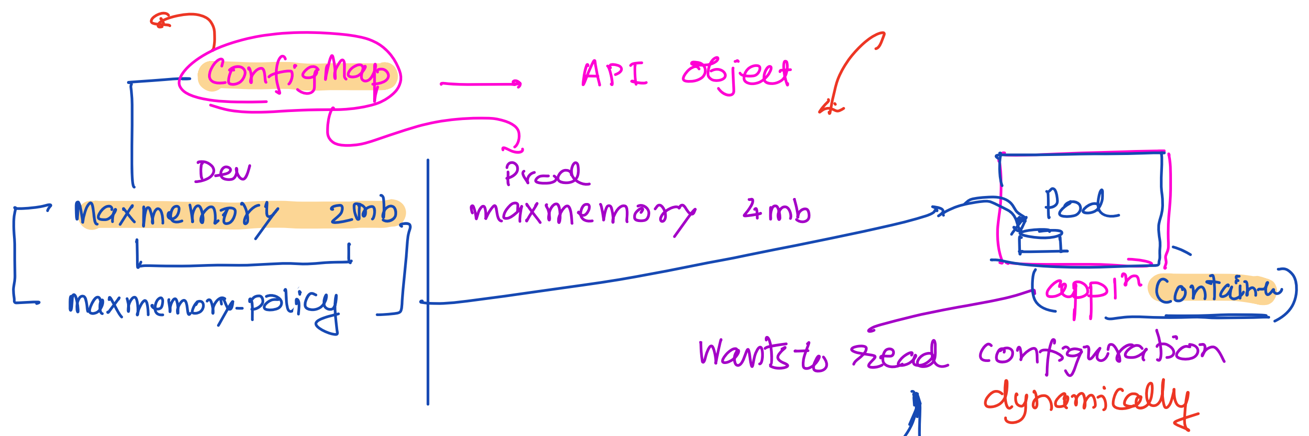
" ConfigMap does not provide secrecy or encryption."



ConfigMap
data — UTF-8
binaryData → base64-encoded string

```
kind: ConfigMap
 metadata:
       name: game-demo
data:
       player_level: "3"
       ui_prop_file_name: " ui-prop.properties"
       ui-prop: |
            color.good = purple
```





volumeMounts

Read the config from mounted volume

```
name: config
  - redis-server
  - "/redis-master/redis.conf"
```

configmap object in AKS.

```
Volumes:
  - name : config
    configMap:
      name : example-redis-config
      item:
        - key : redis-config
          path : redis.conf.
```

Deployment ┬ Read configMap

written configMap key/values into a redis.conf. readable from a volume

Deployment is complete

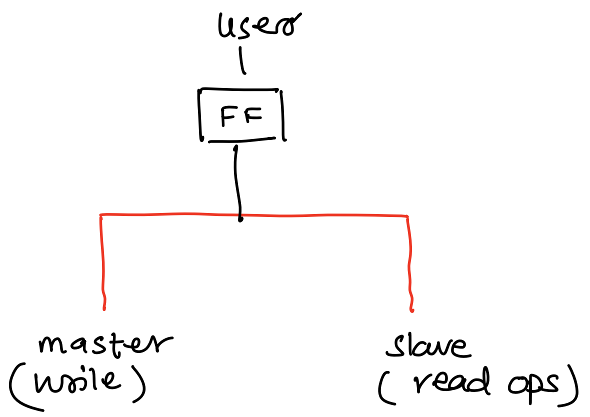
Exposed redis-master to applⁿ — Service

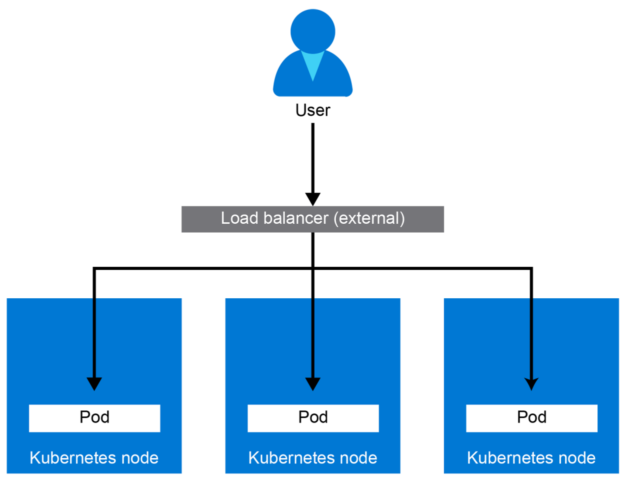
solve

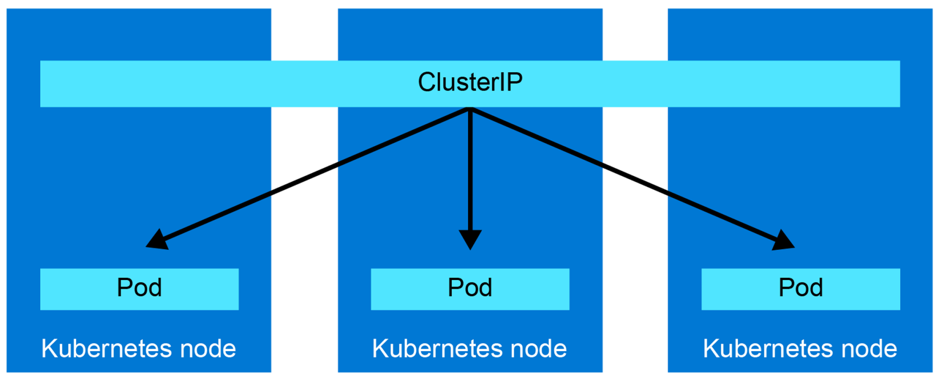
ClusterIP

kubectl exec -it ______ pod — [emphemeral]

✓ Deploy mulliple replicas of master
slave







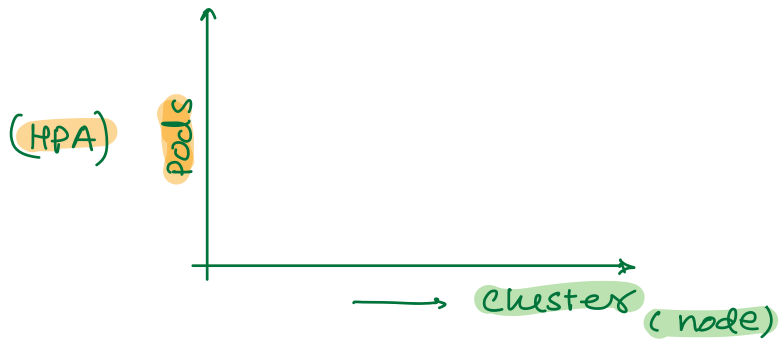
ClusterIP

only within cluster

Build Scalable Application

Cluster Scaling — Autoscaler

AutoScaling → HPA Horizontal Pod Autoscalor.



Upgrading Applications

upgrade by changing YAML file
(It needs access to full YAML file)

Kubectl edit
minor changes

Kubectl patch
when you need to make small update BUT NO ACCESS TO FULL YAML

Upgrading using Helm

PersistentVolumeClaims

A typical process requires compute, memory, network, and storage. In the guestbook example, we saw how Kubernetes helps us abstract the compute, memory, and network. The same YAML files work across all cloud providers, including a cloud-specific setup of public-facing load balancers. The WordPress example shows how the last piece, namely storage, is abstracted from the underlying cloud provider.

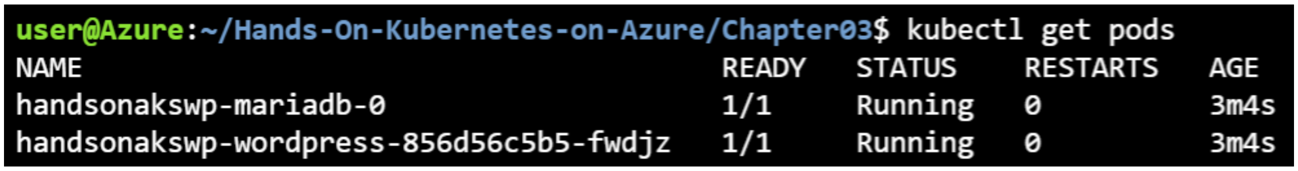
In this case, the WordPress Helm Chart depends on the MariaDB helm chart (https://github.com/bitnami/charts/tree/master/bitnami/mariadb) for its database installation.

Unlike stateless applications, such as our front ends, MariaDB requires careful handling of storage. To make Kubernetes handle stateful workloads, it has a specific object called a **StatefulSet**. A StatefulSet (https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/) is like a deployment with the additional capability of ordering, and the uniqueness of the pods. This means that Kubernetes will ensure that the pod and its storage are kept together. Another way that StatefulSets help is with the consistent naming of pods in a StatefulSet. The pods are named `<pod-name>-#`, where # starts from `0` for the first pod, and 1 for the second pod.

Running the following command, you can see that MariaDB has a predictable number attached to it, whereas the WordPress deployment has a random number attached to the end:

```
kubectl get pods
```

This will generate the output shown in *Figure 3.20*:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
handsonakswp-mariadb-0                   1/1     Running   0          3m4s
handsonakswp-wordpress-856d56c5b5-fwdjz  1/1     Running   0          3m4s
```

Figure 3.20: Numbers attached to MariaDB and WordPress pods

The numbering reinforces the ephemeral nature of the deployment pods versus the StatefulSet pods.

Another difference is how pod deletion is handled. When a deployment pod is deleted, Kubernetes will launch it again anywhere it can, whereas when a StatefulSet pod is deleted, Kubernetes will relaunch it only on the node it was running on. It will relocate the pod only if the node is removed from the Kubernetes cluster.

Often, you will want to attach storage to a StatefulSet. To achieve this, a StatefulSet requires a **PersistentVolume** (**PV**). This volume can be backed by many mechanisms (including blocks, such as Azure Blob, EBS, and iSCSI, and network filesystems, such as AFS, NFS, and GlusterFS). StatefulSets require either a pre-provisioned volume or a dynamically provisioned volume handled by a **PersistentVolumeClaim** (**PVC**). A PVC allows a user to dynamically request storage, which will result in a PV being created.

Please refer to https://kubernetes.io/docs/concepts/storage/persistent-volumes/ for more detailed information.

In this WordPress example, you are using a PVC. A PVC provides an abstraction over the underlying storage mechanism. Let's look at what the MariaDB Helm Chart did by running the following:

```
kubectl get statefulset -o yaml > mariadbss.yaml
code mariadbss.yaml
```

In the preceding command, you got the YAML definition of the StatefulSet that was created and stored it in a file called `mariadbss.yaml`. Let's look at the most relevant parts of that YAML file. The code has been truncated to only show the most relevant parts:

```
apiVersion: v1
items:
- apiVersion: apps/v1
  kind: StatefulSet
...
          volumeMounts:
          - mountPath: /bitnami/mariadb
            name: data
...
volumeClaimTemplates:
- apiVersion: v1
  kind: PersistentVolumeClaim
```

```
309   metadata:
310     creationTimestamp: null
311     labels:
312       app.kubernetes.io/component: primary
313       app.kubernetes.io/instance: handsonakswp
314       app.kubernetes.io/name: mariadb
315     name: data
316   spec:
317     accessModes:
318     - ReadWriteOnce
319     resources:
320       requests:
321         storage: 8Gi
322     volumeMode: Filesystem
...
```

Most of the elements of the preceding code have been covered earlier in the deployment. In the following points, we will highlight the key differences, to take a look at just the PVC:

> **Note**
>
> PVC can be used by any pod, not just StatefulSet pods.

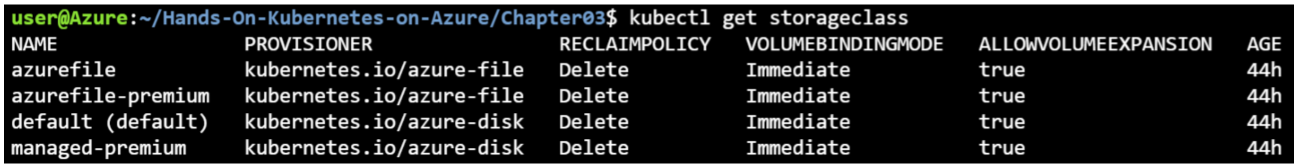
Let's discuss the different elements of the preceding code in detail:

- **Line 4**: This line indicates the `StatefulSet` declaration.
- **Lines 285-287**: These lines mount the volume defined as `data` and mount it under the `/bitnami/mariadb` path.
- **Lines 306-322**: These lines declare the PVC. Note specifically:
    - **Line 315**: This line gives it the name `data`, which is reused at *line* 285.
    - **Line 318**: This line gives the access mode `ReadWriteOnce`, which will create block storage, which on Azure is a disk. There are other access modes as well, namely `ReadOnlyMany` and `ReadWriteMany`. As the name suggests, a `ReadWriteOnce` volume can only be attached to a single pod, while a `ReadOnlyMany` or `ReadWriteMany` volume can be attached to multiple pods at the same time. These last two types require a different underlying storage mechanism such as Azure Files or Azure Blob.
    - **Line 321**: This line defines the size of the disk.

Based on the preceding information, Kubernetes dynamically requests and binds an 8 GiB volume to this pod. In this case, the default dynamic-storage provisioner backed by the Azure disk is used. The dynamic provisioner was set up by Azure when you created the cluster. To see the storage classes available on your cluster, you can run the following command:

```
kubectl get storageclass
```

This will show you an output similar to *Figure 3.21*:



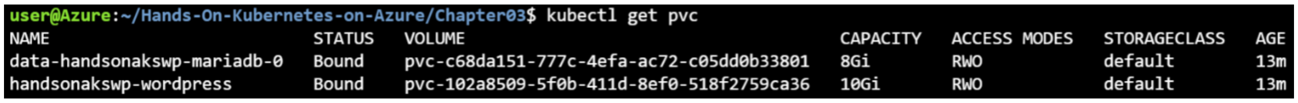
```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get storageclass
NAME                PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
azurefile           kubernetes.io/azure-file   Delete          Immediate           true                   44h
azurefile-premium   kubernetes.io/azure-file   Delete          Immediate           true                   44h
default (default)   kubernetes.io/azure-disk   Delete          Immediate           true                   44h
managed-premium     kubernetes.io/azure-disk   Delete          Immediate           true                   44h
```

Figure 3.21: Different storage classes in your cluster

We can get more details about the PVC by running the following:

```
kubectl get pvc
```

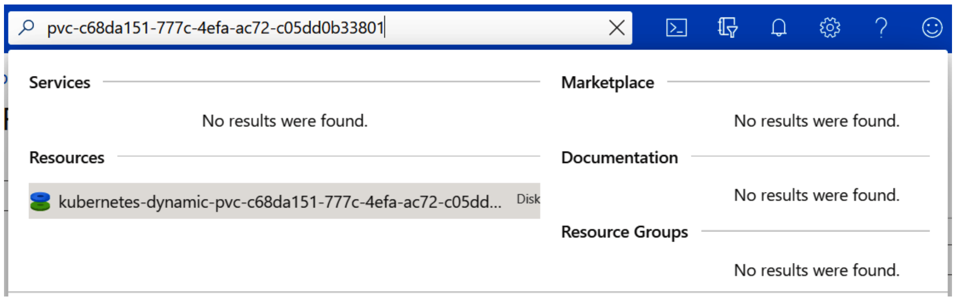
The output generated is displayed in *Figure 3.22*:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pvc
NAME                        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-handsonakswp-mariadb-0 Bound    pvc-c68da151-777c-4efa-ac72-c05dd0b33801   8Gi        RWO            default        13m
handsonakswp-wordpress      Bound    pvc-102a8509-5f0b-411d-8ef0-518f2759ca36   10Gi       RWO            default        13m
```

Figure 3.22: Different PVCs in the cluster

When we asked for storage in the StatefulSet description (*lines* 128-143), Kubernetes performed Azure-disk-specific operations to get the Azure disk with 8 GiB of storage. If you copy the name of the PVC and paste that in the Azure search bar, you should find the disk that was created:



pvc-c68da151-777c-4efa-ac72-c05dd0b33801

Services

No results were found.

Marketplace

No results were found.

Resources

kubernetes-dynamic-pvc-c68da151-777c-4efa-ac72-c05dd... Disk

Documentation

No results were found.

Resource Groups

No results were found.

Figure 3.23: Getting the disk linked to a PVC

The concept of a PVC abstracts cloud provider storage specifics. This allows the same Helm template to work across Azure, AWS, or GCP. On AWS, it will be backed by **Elastic Block Store** (**EBS**), and on GCP it will be backed by Persistent Disk.

Also, note that PVCs can be deployed without using Helm.

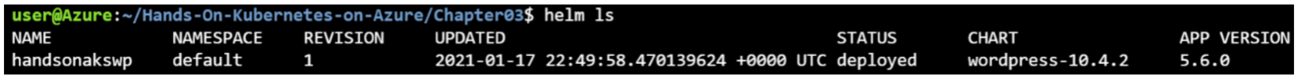
In this section, the concept of storage in Kubernetes using **PersistentVolumeClaim** (**PVC**) was introduced. You saw how they were created by the WordPress Helm deployment, and how Kubernetes created an Azure disk to support the PVC used by MariaDB. In the next section, you will explore the WordPress application on Kubernetes in more detail.

Checking the WordPress deployment

After our analysis of the PVCs, let's check back in with the Helm deployment. You can check the status of the deployment using:

```
helm ls
```

This should return the output shown in *Figure* 3.24:



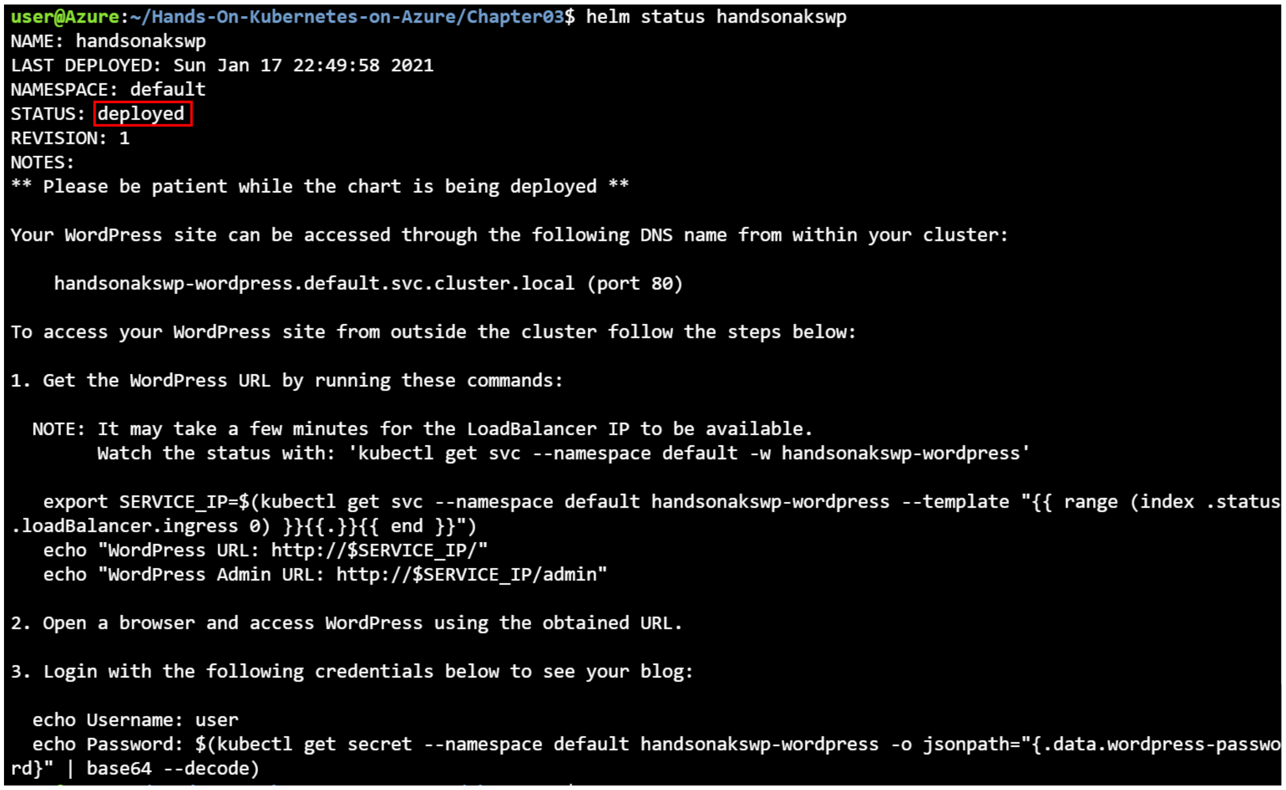
```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ helm ls
NAME           NAMESPACE    REVISION    UPDATED                                  STATUS      CHART                 APP VERSION
handsonakswp   default      1           2021-01-17 22:49:58.470139624 +0000 UTC deployed    wordpress-10.4.2      5.6.0
```

Figure 3.24: WordPress application deployment status

We can get more info from our deployment in Helm using the following command:

```
helm status handsonakswp
```

This will return the output shown in *Figure* 3.25:



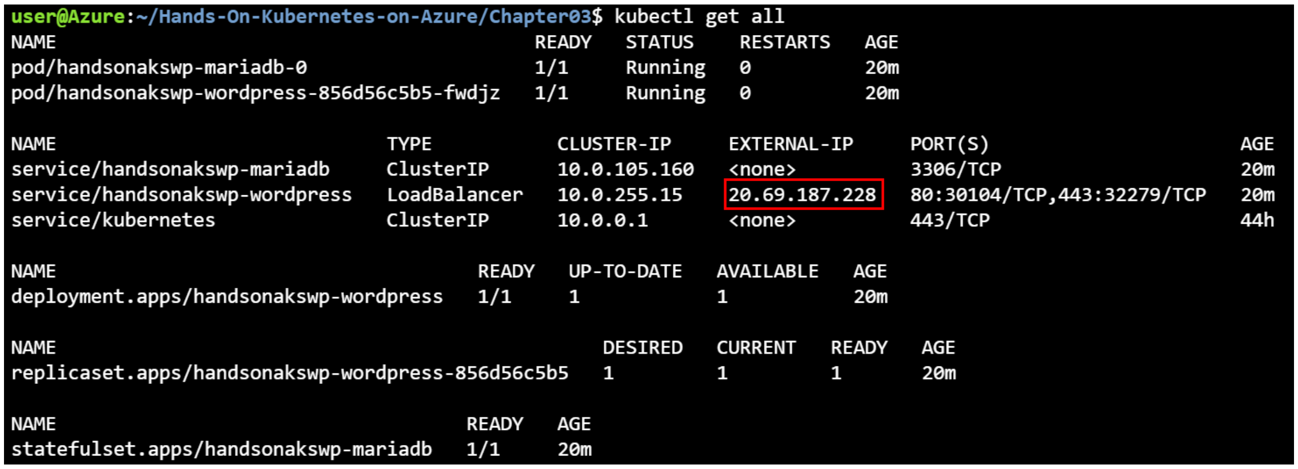
```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ helm status handsonakswp
NAME: handsonakswp
LAST DEPLOYED: Sun Jan 17 22:49:58 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
** Please be patient while the chart is being deployed **

Your WordPress site can be accessed through the following DNS name from within your cluster:

    handsonakswp-wordpress.default.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
        Watch the status with: 'kubectl get svc --namespace default -w handsonakswp-wordpress'

   export SERVICE_IP=$(kubectl get svc --namespace default handsonakswp-wordpress --template "{{ range (index .status
.loadBalancer.ingress 0) }}{{.}}{{ end }}")
   echo "WordPress URL: http://$SERVICE_IP/"
   echo "WordPress Admin URL: http://$SERVICE_IP/admin"

2. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

  echo Username: user
  echo Password: $(kubectl get secret --namespace default handsonakswp-wordpress -o jsonpath="{.data.wordpress-passwo
rd}" | base64 --decode)
```

Figure 3.25: Getting more details about the deployment

This shows you that your chart was deployed successfully. It also shows more info on how you can connect to your site. You won't be using these steps for now; you will revisit these steps in *Chapter* 5, *Handling common failures in* AKS, in the section where we cover fixing storage mount issues. For now, let's look into everything that Helm created for you:

```
kubectl get all
```

This will generate an output similar to *Figure* 3.26:

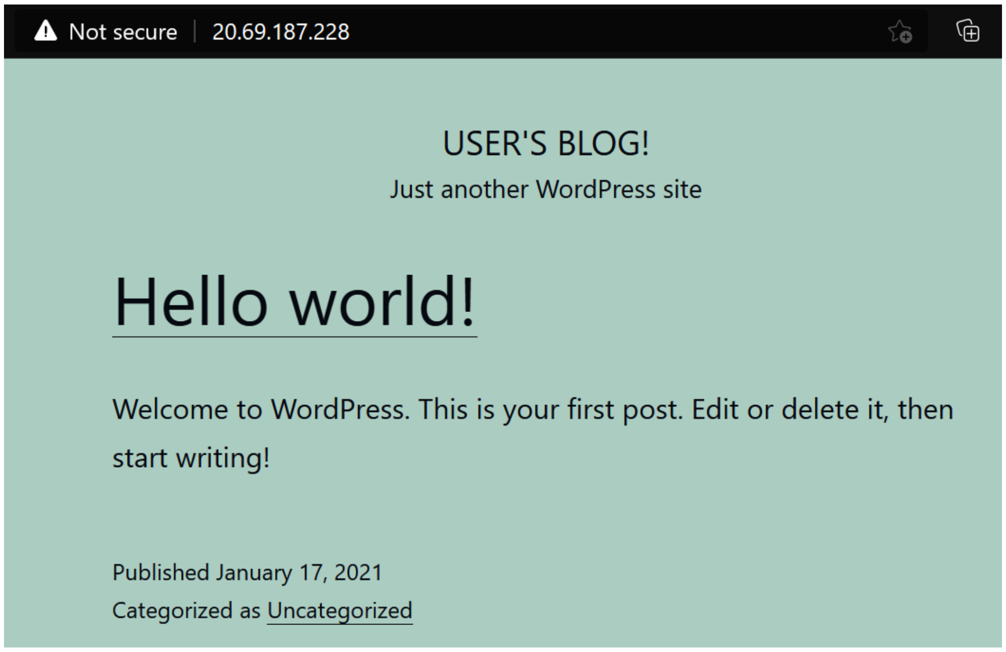


```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get all
NAME                                        READY   STATUS    RESTARTS   AGE
pod/handsonakswp-mariadb-0                  1/1     Running   0          20m
pod/handsonakswp-wordpress-856d56c5b5-fwdjz 1/1     Running   0          20m

NAME                             TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                      AGE
service/handsonakswp-mariadb     ClusterIP      10.0.105.160   <none>          3306/TCP                     20m
service/handsonakswp-wordpress   LoadBalancer   10.0.255.15    20.69.187.228   80:30104/TCP,443:32279/TCP   20m
service/kubernetes               ClusterIP      10.0.0.1       <none>          443/TCP                      44h

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/handsonakswp-wordpress   1/1     1            1           20m

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/handsonakswp-wordpress-856d56c5b5   1         1         1       20m

NAME                                    READY   AGE
statefulset.apps/handsonakswp-mariadb   1/1     20m
```

Figure 3.26: List of objects created by Helm

If you don't have an external IP yet, wait for a couple of minutes and retry the command.

You can then go ahead and connect to your external IP and access your WordPress site. *Figure* 3.27 is the resulting output:



Not secure | 20.69.187.228

USER'S BLOG!

Just another WordPress site

Hello world!

Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

Published January 17, 2021

Categorized as Uncategorized

Figure 3.27: WordPress site being displayed on connection with the external IP

To make sure you don't run into issues in the following chapters, let's delete the WordPress site. This can be done in the following way:

```
helm delete handsonakswp
```

By design, the PVCs won't be deleted. This ensures persistent data is kept. As you don't have any persistent data, you can safely delete the PVCs as well:

```
kubectl delete pvc --all
```

> **Note**
>
> Be very careful when executing `kubectl delete <object> --all` as it will delete all the objects in a namespace. This is not recommended on a production cluster.

In this section, you have deployed a full WordPress site using Helm. You also learned how Kubernetes handles persistent storage using PVCs.

Summary

In this chapter, you deployed two applications. You started the chapter by deploying the guestbook application. During that deployment, the details of pods, ReplicaSets, and deployments were explored. You also used dynamic configuration using ConfigMaps. Finally, you looked into how services are used to route traffic to the deployed applications.

The second application you deployed was a WordPress application. You deployed it via the Helm package manager. As part of this deployment, PVCs were used, and you explored how they were used in the system and how they were linked to disks on Azure.

In *Chapter 4*, *Building scalable applications*, you will look into scaling applications and the cluster itself. You will first learn about the manual and automatic scaling of the application, and afterward, you'll learn about the manual and automatic scaling of the cluster itself. Finally, different ways in which applications can be updated on Kubernetes will be explained.