3. Watch the `service` object until the public IP becomes available. To do this, type the following:

```
kubectl get service -w
```

> **Note**
>
> You can also get services in Kubernetes by using `kubectl get svc` rather than the full `kubectl get service`.

4. This will take a couple of seconds to show you the updated external IP. *Figure* 5.3 shows the service's public IP. Once you see the public IP appear (**20.72.244.113** in this case), you can exit the watch command by hitting *Ctrl* + C:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get service -w
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
frontend        LoadBalancer   10.0.23.47     <pending>       80:30619/TCP   4s
kubernetes      ClusterIP      10.0.0.1       <none>          443/TCP        27h
redis-master    ClusterIP      10.0.184.142   <none>          6379/TCP       4s
redis-replica   ClusterIP      10.0.218.85    <none>          6379/TCP       4s
frontend        LoadBalancer   10.0.23.47     20.72.244.113   80:30619/TCP   5s
```

Figure 5.3: The external IP of the frontend service changes from <pending> to an actual IP address

5. Go to `http://<EXTERNAL-IP>` (`http://20.72.244.113` in this case) as shown in *Figure* 5.4:

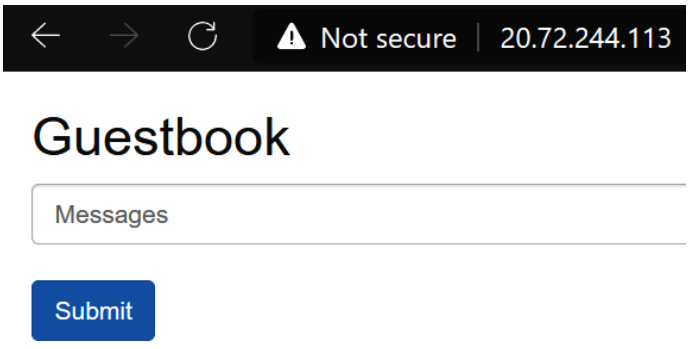


Figure 5.4: Browsing to the guestbook application

6. Let's see where the pods are currently running using the following command:

```
kubectl get pods -o wide
```

This will generate an output as shown in *Figure* 5.5:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE   IP            NODE
frontend-766d4f77cb-9w9t2         1/1     Running   0          42s   10.244.0.54   aks-agentpool-39838025-vmss000002
frontend-766d4f77cb-vkc2l         1/1     Running   0          42s   10.244.0.53   aks-agentpool-39838025-vmss000002
frontend-766d4f77cb-z7s54         1/1     Running   0          42s   10.244.1.79   aks-agentpool-39838025-vmss000000
redis-master-f46ff57fd-hmwr4      1/1     Running   0          42s   10.244.0.55   aks-agentpool-39838025-vmss000002
redis-replica-786bd64556-hf4kd    1/1     Running   0          42s   10.244.0.56   aks-agentpool-39838025-vmss000002
redis-replica-786bd64556-l72z7    1/1     Running   0          42s   10.244.1.80   aks-agentpool-39838025-vmss000000
```

Figure 5.5: The pods are spread between node 0 and node 2

This shows you that you should have the workload spread between node 0 and node 2.

> **Note**
>
> In the example shown in *Figure 5.5*, the workload is spread between nodes 0 and 2. You might notice that node 1 is missing here. If you followed the example in *Chapter 4, Building scalable applications*, your cluster should be in a similar state. The reason for this is that as Azure removes old nodes and adds new nodes to a cluster (as you did in *Chapter 4, Building scalable applications*), it keeps incrementing the node counter.

7. Before introducing the node failures, there are two optional steps you can take to verify whether your application can continue to run. You can run the following command to hit the guestbook front end every 5 seconds and get the HTML. It's recommended to open this in a new Cloud Shell window:

```
while true; do
  curl -m 1 http://<EXTERNAl-IP>/;
  sleep 5;
done
```

> **Note**
>
> The preceding command will keep calling your application till you press *Ctrl* + *C*. There might be intermittent times where you don't get a reply, which is to be expected as Kubernetes takes a couple of minutes to rebalance the system.

You can also add some guestbook entries to see what happens to them when you cause the node to shut down. This will display an output as shown in *Figure* 5.6:

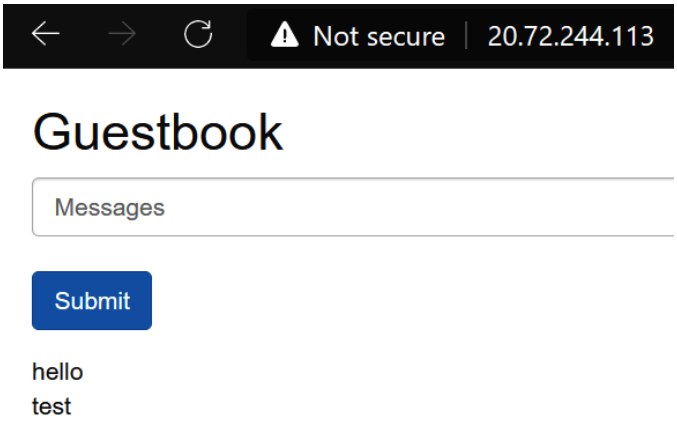


Figure 5.6: Writing a couple of messages in the guestbook

8. In this example, you are exploring how Kubernetes handles a node failure. To demonstrate this, shut down a node in the cluster. You can shut down either node, although for maximum impact it is recommended you shut down the node from *step* 6 that hosted the most pods. In the case of the example shown, node 2 will be shut down.

   To shut down this node, look for **VMSS** (**virtual machine scale sets**) in the Azure search bar, and select the scale set used by your cluster, as shown in *Figure* 5.7. If you have multiple scale sets in your subscription, select the one whose name corresponds to the node names shown in *Figure* 5.5:

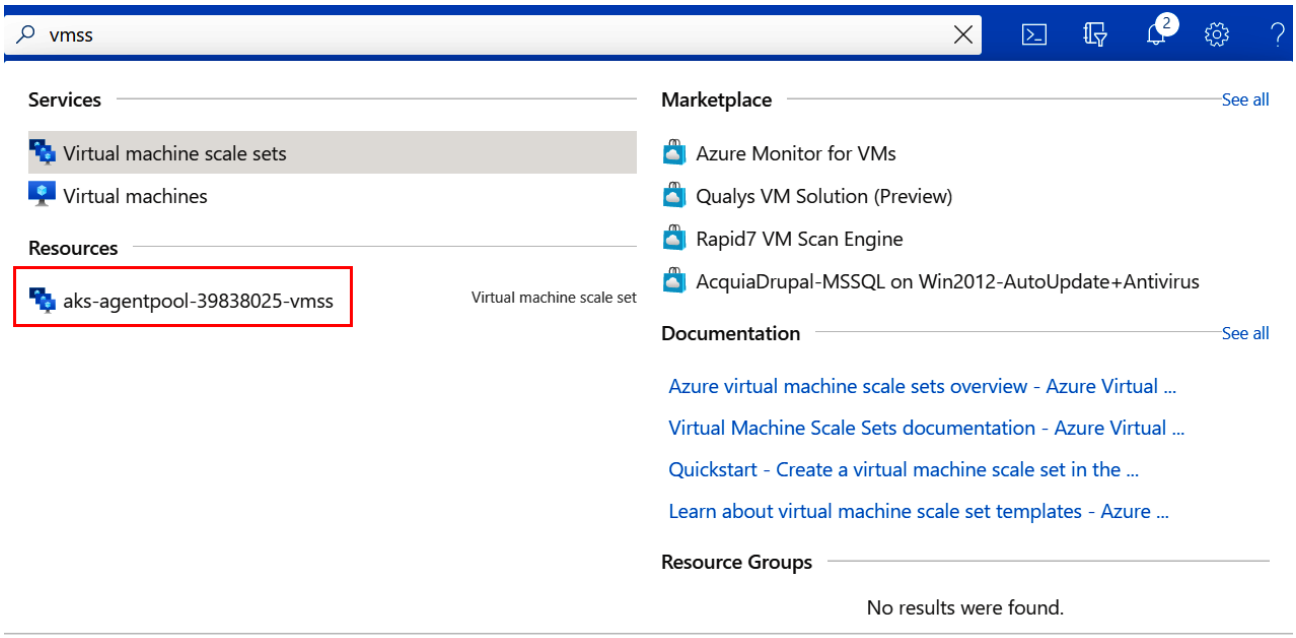


Figure 5.7: Looking for the scale set hosting your cluster

After navigating to the pane of the scale set, go to the **Instances** view, select the instance you want to shut down, and then hit the **Stop** button, as shown in *Figure* 5.8:

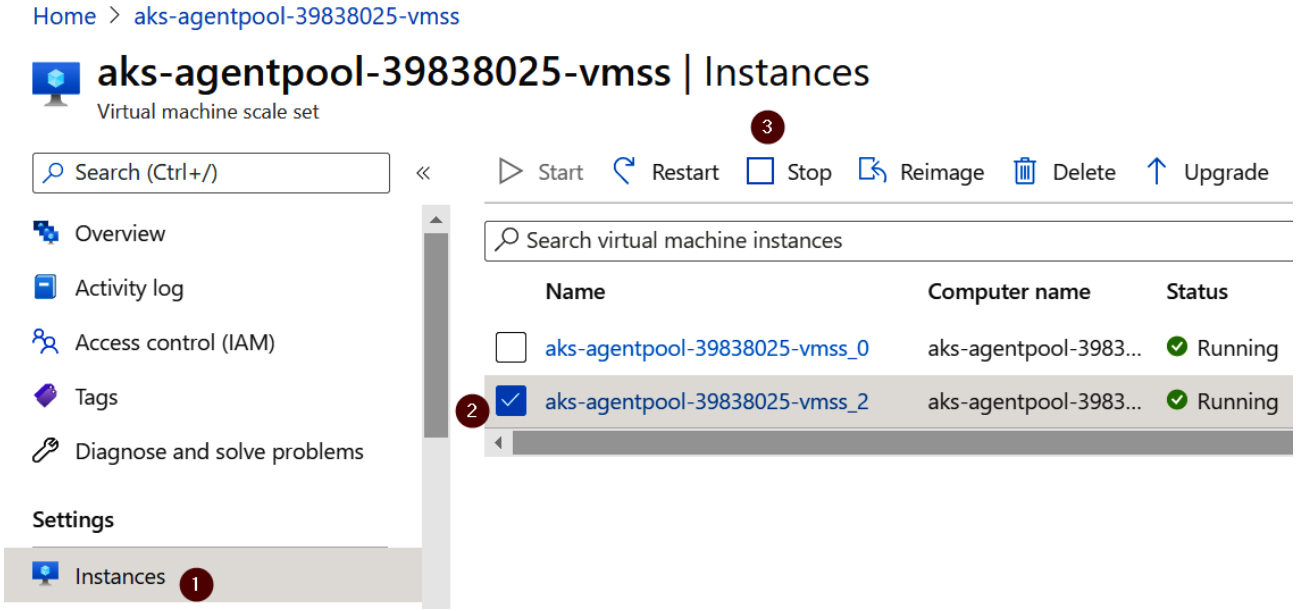


Figure 5.8: Shutting down node 2

This will shut down the node. To see how Kubernetes will react with your pods, you can watch the pods in your cluster via the following command:

```
kubectl get pods -o wide -w
```

After a while, you should notice additional output, showing you that the pods got rescheduled on the healthy host, as shown in *Figure* 5.9:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -o wide -w
NAME                              READY   STATUS              RESTARTS   AGE     IP            NODE                                NOMINATED NODE
frontend-766d4f77cb-9w9t2         1/1     Running             0          3m11s   10.244.0.54   aks-agentpool-39838025-vmss000002   <none>
frontend-766d4f77cb-vkc2l         1/1     Running             0          3m11s   10.244.0.53   aks-agentpool-39838025-vmss000002   <none>
frontend-766d4f77cb-z7s54         1/1     Running             0          3m11s   10.244.1.79   aks-agentpool-39838025-vmss000000   <none>
redis-master-f46ff57fd-hmwr4      1/1     Running             0          3m11s   10.244.0.55   aks-agentpool-39838025-vmss000002   <none>
redis-replica-786bd64556-hf4kd    1/1     Running             0          3m11s   10.244.0.56   aks-agentpool-39838025-vmss000002   <none>
redis-replica-786bd64556-l72z7    1/1     Running             0          3m11s   10.244.1.80   aks-agentpool-39838025-vmss000000   <none>
frontend-766d4f77cb-9w9t2         1/1     Terminating         0          7m31s   10.244.0.54   aks-agentpool-39838025-vmss000002   <none>
redis-replica-786bd64556-hf4kd    1/1     Terminating         0          7m31s   10.244.0.56   aks-agentpool-39838025-vmss000002   <none>
redis-master-f46ff57fd-hmwr4      1/1     Terminating         0          7m31s   10.244.0.55   aks-agentpool-39838025-vmss000002   <none>
frontend-766d4f77cb-vkc2l         1/1     Terminating         0          7m31s   10.244.0.53   aks-agentpool-39838025-vmss000002   <none>
redis-master-f46ff57fd-wpsf4      0/1     Pending             0          0s      <none>        <none>                              <none>
frontend-766d4f77cb-hv8sr         0/1     Pending             0          0s      <none>        <none>                              <none>
redis-replica-786bd64556-qwr9v    0/1     Pending             0          0s      <none>        <none>                              <none>
redis-master-f46ff57fd-wpsf4      0/1     Pending             0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
frontend-766d4f77cb-hv8sr         0/1     Pending             0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
redis-replica-786bd64556-qwr9v    0/1     Pending             0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
frontend-766d4f77cb-qbvtq         0/1     Pending             0          0s      <none>        <none>                              <none>
frontend-766d4f77cb-qbvtq         0/1     Pending             0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
redis-master-f46ff57fd-wpsf4      0/1     ContainerCreating   0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
frontend-766d4f77cb-hv8sr         0/1     ContainerCreating   0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
redis-replica-786bd64556-qwr9v    0/1     ContainerCreating   0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
frontend-766d4f77cb-qbvtq         0/1     ContainerCreating   0          0s      <none>        aks-agentpool-39838025-vmss000000   <none>
frontend-766d4f77cb-hv8sr         1/1     Running             0          1s      10.244.1.82   aks-agentpool-39838025-vmss000000   <none>
redis-replica-786bd64556-qwr9v    1/1     Running             0          2s      10.244.1.84   aks-agentpool-39838025-vmss000000   <none>
frontend-766d4f77cb-qbvtq         1/1     Running             0          3s      10.244.1.83   aks-agentpool-39838025-vmss000000   <none>
redis-master-f46ff57fd-wpsf4      1/1     Running             0          3s      10.244.1.81   aks-agentpool-39838025-vmss000000   <none>
```

Figure 5.9: The pods from the failed node getting recreated on a healthy node

What you see here is the following:

- The Redis master pod running on **node 2** got terminated as the host became unhealthy.
- A new Redis master pod got created, on host **0**. This went through the stages **Pending**, **ContainerCreating**, and then **Running**.

> **Note**
>
> In the preceding example, Kubernetes picked up that the host was unhealthy before it rescheduled the pods. If you were to do `kubectl get nodes`, you would see node **2** is in a **NotReady** state. There is a configuration in Kubernetes called `pod-eviction-timeout` that defines how long the system will wait to reschedule pods on a healthy host. The default is 5 minutes.

9. If you recorded a number of messages in the guestbook during *step* 7, browse back to the guestbook application on its public IP. What you can see is that all your precious messages are gone! This shows the importance of having **PersistentVolumeClaims** (**PVCs**) for any data that you want to survive in the case of a node failure, which is not the case in our application here. You will see an example of this in the last section of this chapter.

In this section, you learned how Kubernetes automatically handles node failures by recreating pods on healthy nodes. In the next section, you will learn how you can diagnose and solve out-of-resource issues.

Solving out-of-resource failures

Another common issue that can come up with Kubernetes clusters is the cluster running out of resources. When the cluster doesn't have enough CPU power or memory to schedule additional pods, pods will become stuck in a `Pending` state. You have seen this behavior in *Chapter* 4, *Building scalable applications*, as well.

Kubernetes uses requests to calculate how much CPU power or memory a certain pod requires. The guestbook application has requests defined for all the deployments. If you open the `guestbook-all-in-one.yaml` file in the folder `Chapter05`, you'll see the following for the `redis-replica` deployment:

```
63  kind: Deployment
64  metadata:
65    name: redis-replica
...
83          resources:
84            requests:
85              cpu: 200m
86              memory: 100Mi
```

This section explains that every pod for the `redis-replica` deployment requires `200m` of a CPU core (`200` milli or `20%`) and `100MiB` (Mebibyte) of memory. In your 2 CPU clusters (with node 1 shut down), scaling this to 10 pods will cause issues with the available resources. Let's look into this:

> **Note**
>
> In Kubernetes, you can use either the binary prefix notation or the base 10 notation to specify memory and storage. Binary prefix notation means using KiB (kibibyte) to represent 1,024 bytes, MiB (mebibyte) to represent 1,024 KiB, and Gib (gibibyte) to represent 1,024 MiB. Base 10 notation means using kB (kilobyte) to represent 1,000 bytes, MB (megabyte) to represent 1,000 kB, and GB (gigabyte) represents 1,000 MB.

1. Let's start by scaling the `redis-replica` deployment to 10 pods:

   ```
   kubectl scale deployment/redis-replica --replicas=10
   ```

2. This will cause a couple of new pods to be created. We can check our pods using the following:

   ```
   kubectl get pods
   ```

   This will generate an output as shown in *Figure 5.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl scale deployment/redis-replica --replicas=10
deployment.apps/redis-replica scaled
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods
NAME                             READY   STATUS        RESTARTS   AGE
frontend-766d4f77cb-9w9t2        1/1     Terminating   0          9m55s
frontend-766d4f77cb-hv8sr        1/1     Running       0          2m24s
frontend-766d4f77cb-qbvtq        1/1     Running       0          2m24s
frontend-766d4f77cb-vkc2l        1/1     Terminating   0          9m55s
frontend-766d4f77cb-z7s54        1/1     Running       0          9m55s
redis-master-f46ff57fd-hmwr4     1/1     Terminating   0          9m55s
redis-master-f46ff57fd-wpsf4     1/1     Running       0          2m24s
redis-replica-786bd64556-2t8ms   1/1     Running       0          5s
redis-replica-786bd64556-7k7cn   0/1     Pending       0          5s
redis-replica-786bd64556-7shvm   0/1     Pending       0          4s
redis-replica-786bd64556-dv7qv   0/1     Pending       0          5s
redis-replica-786bd64556-hf4kd   1/1     Terminating   0          9m55s
redis-replica-786bd64556-jdfxj   0/1     Pending       0          5s
redis-replica-786bd64556-l72z7   1/1     Running       0          9m55s
redis-replica-786bd64556-qwr9v   1/1     Running       0          2m24s
redis-replica-786bd64556-r8j6b   1/1     Running       0          5s
redis-replica-786bd64556-xk82s   1/1     Running       0          5s
redis-replica-786bd64556-zgfjq   0/1     Pending       0          5s
```

Figure 5.10: Some pods are in the Pending state

Highlighted here is one of the pods that are in the **Pending** state. This occurs if the cluster is out of resources.

3. We can get more information about these pending pods using the following command:

```
kubectl describe pod redis-replica-<pod-id>
```

   This will show you more details. At the bottom of the `describe` command, you should see something like what's shown in *Figure 5.11*:

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  104s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu
, 1 node(s) had taint {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
  Warning  FailedScheduling  104s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu
, 1 node(s) had taint {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
```

Figure 5.11: Kubernetes is unable to schedule this pod

   It explains two things:

   - One of the nodes is out of CPU resources.
   - One of the nodes has a taint (node.kubernetes.io/unreachable) that the pod didn't tolerate. This means that the node that is `NotReady` can't accept pods.

4. We can solve this capacity issue by starting up node **2** as shown in *Figure 5.12*. This can be done in a way similar to the shutdown process:

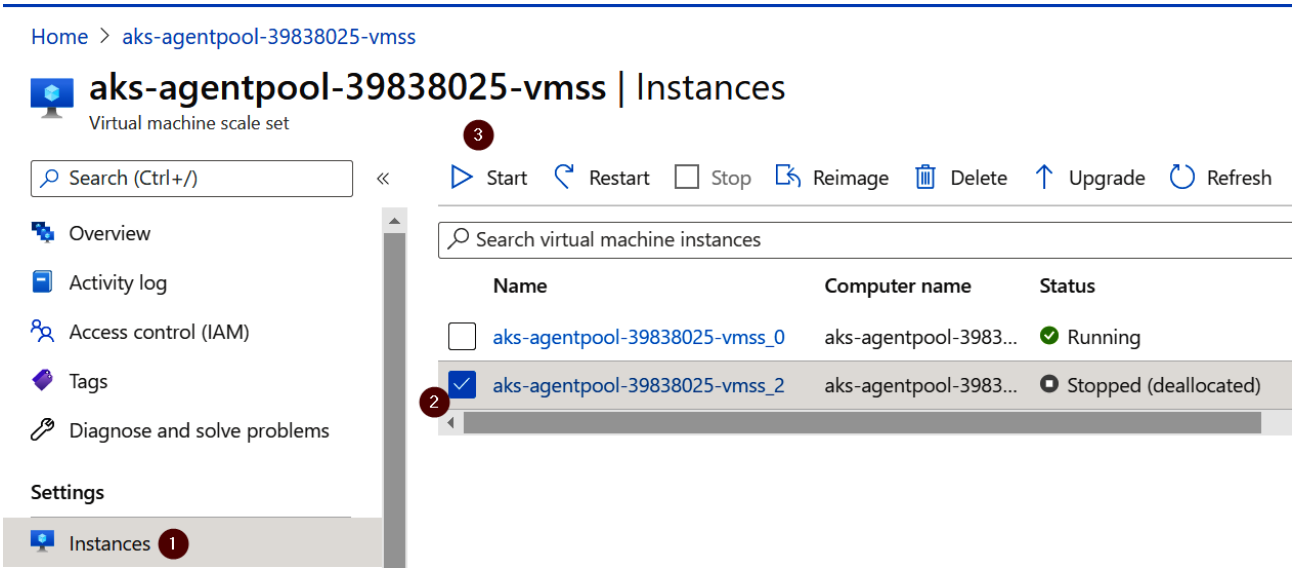


Figure 5.12: Start node 2 again

5. It will take a couple of minutes for the other node to become available again in Kubernetes. You can monitor the progress on the pods by executing the following command:

```
kubectl get pods -w
```

This will show you an output after a couple of minutes similar to *Figure 5.13*:

```
redis-replica-786bd64556-7k7cn   0/1     Pending             0          2m29s
redis-replica-786bd64556-dv7qv   0/1     Pending             0          2m29s
redis-replica-786bd64556-jdfxj   0/1     Pending             0          2m29s
redis-replica-786bd64556-7shvm   0/1     Pending             0          2m28s
redis-replica-786bd64556-zgfjq   0/1     Pending             0          2m29s
redis-replica-786bd64556-7k7cn   0/1     ContainerCreating   0          2m29s
redis-replica-786bd64556-dv7qv   0/1     ContainerCreating   0          2m29s
redis-replica-786bd64556-jdfxj   0/1     ContainerCreating   0          2m29s
redis-replica-786bd64556-7shvm   0/1     ContainerCreating   0          2m28s
redis-replica-786bd64556-zgfjq   0/1     ContainerCreating   0          2m30s
redis-replica-786bd64556-7k7cn   1/1     Running             0          2m30s
redis-replica-786bd64556-zgfjq   1/1     Running             0          2m31s
redis-replica-786bd64556-dv7qv   1/1     Running             0          2m31s
redis-replica-786bd64556-jdfxj   1/1     Running             0          2m31s
redis-replica-786bd64556-7shvm   1/1     Running             0          2m31s
```

Figure 5.13: Pods move from a Pending state to ContainerCreating to Running

Here again, you see the container status change from **Pending**, to **ContainerCreating**, to finally **Running**.

6. If you re-execute the `describe` command on the previous pod, you'll see an output like what's shown in *Figure 5.14*:

```
Events:
  Type     Reason            Age    From               Message
  ----     ------            ----   ----               -------
  Warning  FailedScheduling  4m48s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu, 1 node(s) had taint
 {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
  Warning  FailedScheduling  4m48s  default-scheduler  0/2 nodes are available: 1 Insufficient cpu, 1 node(s) had taint
 {node.kubernetes.io/unreachable: }, that the pod didn't tolerate.
  Normal   Scheduled         2m20s  default-scheduler  Successfully assigned default/redis-replica-786bd64556-7k7cn to
aks-agentpool-39838025-vmss000002
  Normal   Pulled            2m20s  kubelet            Container image "gcr.io/google_samples/gb-redis-follower:v1" alr
eady present on machine
  Normal   Created           2m19s  kubelet            Created container replica
  Normal   Started           2m19s  kubelet            Started container replica
```

Figure 5.14: When the node is available again, the Pending pods are assigned to that node

This shows that after node 2 became available, Kubernetes scheduled the pod on that node, and then started the container.

In this section, you learned how to diagnose out-of-resource errors. You were able to solve the error by adding another node to the cluster. Before moving on to the final failure mode, clean up the guestbook deployment.

> **Note**
>
> In *Chapter 4, Building scalable applications*, the **cluster autoscaler** was introduced. The cluster autoscaler will monitor out-of-resource errors and add new nodes to the cluster automatically.

Let's clean up the guestbook deployment by running the following `delete` command:

```
kubectl delete -f guestbook-all-in-one.yaml
```

It is now also safe to close the other Cloud Shell window you opened earlier.

So far, you have learned how to recover from two failure modes for nodes in a Kubernetes cluster. First, you saw how Kubernetes handles a node going offline and how the system reschedules pods to a working node. After that, you saw how Kubernetes uses requests to manage the scheduling of pods on a node, and what happens when a cluster is out of resources. In the next section, you'll learn about another failure mode in Kubernetes, namely what happens when Kubernetes encounters storage mounting issues.

Fixing storage mount issues

Earlier in this chapter, you noticed how the guestbook application lost data when the Redis master was moved to another node. This happened because that sample application didn't use any persistent storage. In this section, you'll see an example of how PVCs can be used to prevent data loss when Kubernetes moves a pod to another node. You will see a common error that occurs when Kubernetes moves pods with PVCs attached, and you'll learn how to fix this.

For this, you will reuse the WordPress example from the previous chapter. Before starting, let's make sure that the cluster is in a clean state:

```
kubectl get all
```

This should show you just the one Kubernetes service, as in *Figure 5.15*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.0.0.1     <none>        443/TCP   87m
```

Figure 5.15: You should only have the one Kubernetes service running for now

Let's also ensure that both nodes are running and **Ready**:

```
kubectl get nodes
```

This should show us both nodes in a **Ready** state, as in *Figure 5.16*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-agentpool-39838025-vmss000000   Ready    agent   86m   v1.19.6
aks-agentpool-39838025-vmss000002   Ready    agent   26m   v1.19.6
```

Figure 5.16: You should have two nodes available in your cluster

In the previous example, under the *Handling node failures* section, you saw that the messages stored in `redis-master` are lost if the pod gets restarted. The reason for this is that `redis-master` stores all data in its container, and whenever it is restarted, it uses the clean image without the data. In order to survive reboots, the data has to be stored outside. Kubernetes uses PVCs to abstract the underlying storage provider to provide this external storage.

To start this example, set up the WordPress installation.

Starting the WordPress installation

Let's start by installing WordPress. We will demonstrate how it works and then verify that storage is still present after a reboot.

If you have not done so yet in a previous chapter, add the Helm repository for Bitnami:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Begin reinstallation by using the following command:

```
helm install wp bitnami/wordpress
```

This will take a couple of minutes to process. You can follow the status of this installation by executing the following command:

```
kubectl get pods –w
```

After a couple of minutes, this should show you two pods with a status of **Running** and with a ready status of **1/1** for both pods, as shown in *Figure 5.17*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -w
NAME                              READY   STATUS              RESTARTS   AGE
wp-mariadb-0                      0/1     Pending             0          3s
wp-wordpress-6f7c4f85b5-4p264     0/1     Pending             0          3s
wp-wordpress-6f7c4f85b5-4p264     0/1     Pending             0          15s
wp-wordpress-6f7c4f85b5-4p264     0/1     ContainerCreating   0          15s
wp-mariadb-0                      0/1     Pending             0          19s
wp-mariadb-0                      0/1     ContainerCreating   0          19s
wp-wordpress-6f7c4f85b5-4p264     0/1     Running             0          95s
wp-mariadb-0                      0/1     Running             0          110s
wp-wordpress-6f7c4f85b5-4p264     0/1     Error               0          2m27s
wp-wordpress-6f7c4f85b5-4p264     0/1     Running             1          2m28s
wp-mariadb-0                      1/1     Running             0          2m29s
wp-wordpress-6f7c4f85b5-4p264     1/1     Running             1          3m4s
```

Figure 5.17: All pods will have the status of Running after a couple of minutes

You might notice that the `wp-wordpress` pod went through an Error status and was restarted afterward. This is because the `wp-mariadb` pod was not ready in time, and `wp-wordpress` went through a restart. You will learn more about readiness and how this can influence pod restarts in *Chapter 7, Monitoring the AKS cluster and the application*.

In this section, you saw how to install WordPress. Now, you will see how to avoid data loss using persistent volumes.

Using persistent volumes to avoid data loss

A **persistent volume** (**PV**) is the way to store persistent data in the cluster with Kubernetes. PVs were discussed in more detail in *Chapter 3*, *Application deployment on AKS*. Let's explore the PVs created for the WordPress deployment:

1. You can get the PersistentVolumeClaims using the following command:

   ```
   kubectl get pvc
   ```

   This will generate an output as shown in *Figure 5.18*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pvc
NAME              STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-wp-mariadb-0 Bound    pvc-50507406-5dfe-46d5-88e5-a6e3f477b040   8Gi        RWO            default        2m46s
wp-wordpress      Bound    pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997   10Gi       RWO            default        2m46s
```

Figure 5.18: Two PVCs are created by the WordPress deployment

   A PersistentVolumeClaim will result in the creation of a PersistentVolume. The PersistentVolume is the link to the physical resource created, which is an Azure disk in this case. The following command shows the actual PVs that are created:

   ```
   kubectl get pv
   ```

   This will show you the two PersistentVolumes:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
pvc-50507406-5dfe-46d5-88e5-a6e3f477b040   8Gi        RWO            Delete           Bound    default/data-wp-mariadb-0
pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997   10Gi       RWO            Delete           Bound    default/wp-wordpress
```

Figure 5.19: Two PVs are created to store the data of the PVCs

   You can get more details about the specific PersistentVolumes that were created. Copy the name of one of the PVs, and run the following command:

   ```
   kubectl describe pv <pv name>
   ```

This will show you the details of that volume, as in *Figure* 5.20:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl describe pv pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
Name:              pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
Labels:            failure-domain.beta.kubernetes.io/region=westus2
Annotations:       pv.kubernetes.io/bound-by-controller: yes
                   pv.kubernetes.io/provisioned-by: kubernetes.io/azure-disk
                   volumehelper.VolumeDynamicallyCreatedByKey: azure-disk-dynamic-provisioner
Finalizers:        [kubernetes.io/pv-protection]
StorageClass:      default
Status:            Bound
Claim:             default/wp-wordpress
Reclaim Policy:    Delete
Access Modes:      RWO
VolumeMode:        Filesystem
Capacity:          10Gi
Node Affinity:
  Required Terms:
    Term 0:        failure-domain.beta.kubernetes.io/region in [westus2]
Message:
Source:
    Type:         AzureDisk (an Azure Data Disk mount on the host and bind mount to the pod)
    DiskName:     kubernetes-dynamic-pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
    DiskURI:      /subscriptions/ede7a1e5-4121-427f-876e-e100eba989a0/resourceGroups/mc_rg-handsonaks_handsonaks_
westus2/providers/Microsoft.Compute/disks/kubernetes-dynamic-pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997
    Kind:         Managed
    FSType:
    CachingMode:  ReadOnly
    ReadOnly:     false
Events:           <none>
```

Figure 5.20: The details of one of the PVs

Here, you can see which PVC has claimed this volume and what the **DiskName** is in Azure.

2. Verify that your site is working:

```
kubectl get service
```

This will show us the public IP of our WordPress site, as seen in *Figure* 5.21:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get service
NAME           TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)                      AGE
kubernetes     ClusterIP      10.0.0.1      <none>         443/TCP                      7d14h
wp-mariadb     ClusterIP      10.0.204.3    <none>         3306/TCP                     17m
wp-wordpress   LoadBalancer   10.0.189.10   20.72.222.87   80:32239/TCP,443:30828/TCP   17m
```

Figure 5.21: Public IP of the WordPress site

3. If you remember from *Chapter 3, Application deployment of AKS*, Helm showed you the commands you need to get the admin credentials for our WordPress site. Let's grab those commands and execute them to log on to the site as follows:

```
helm status wp
echo Username: user
echo Password: $(kubectl get secret --namespace default wp-wordpress
-o jsonpath="{.data.wordpress-password}" | base64 -d)
```

   This will show you the **username** and **password**, as displayed in *Figure 5.22*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ helm status wp
NAME: wp
LAST DEPLOYED: Sat Jan 23 16:33:41 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:

    wp-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 wp-wordpress'

   export SERVICE_IP=$(kubectl get svc --namespace default wp-wordpress --template "{{ range (index .status.loadB
alancer.ingress 0) }}{{.}}{{ end }}")
   echo "WordPress URL: http://$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: $(kubectl get secret --namespace default wp-wordpress -o jsonpath="{.data.wordpress-password}" |
 base64 --decode)
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ echo Username: user
Username: user
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ echo Password: $(kubectl get secret --namespace default wp-w
ordpress -o jsonpath="{.data.wordpress-password}" | base64 --decode)
Password: 1oV0FVacNF
```

Figure 5.22: Getting the username and password for the WordPress application

You can log in to our site via the following address: `http://<external-ip>/admin`. Log in here with the credentials from the previous step. Then you can go ahead and add a post to your website. Click the **Write your first blog post** button, and then create a short post, as shown in *Figure* 5.23:

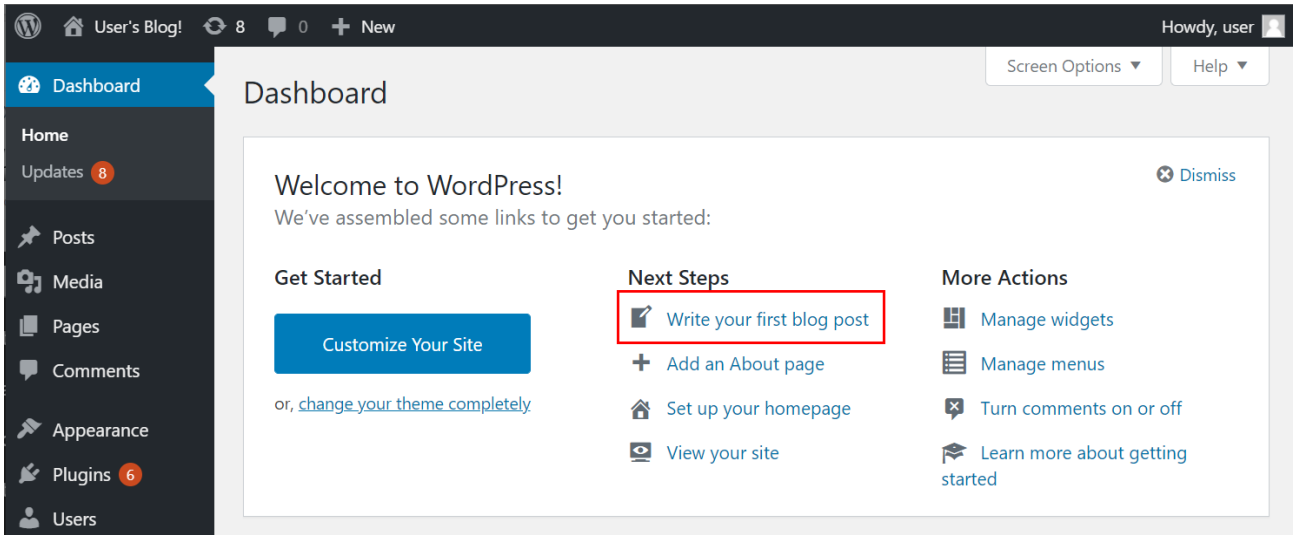


Figure 5.23: Writing your first blog post

Type some text now and hit the **Publish** button, as shown in *Figure* 5.24. The text itself isn't important; you are writing this to verify that data is indeed persisted to disk:

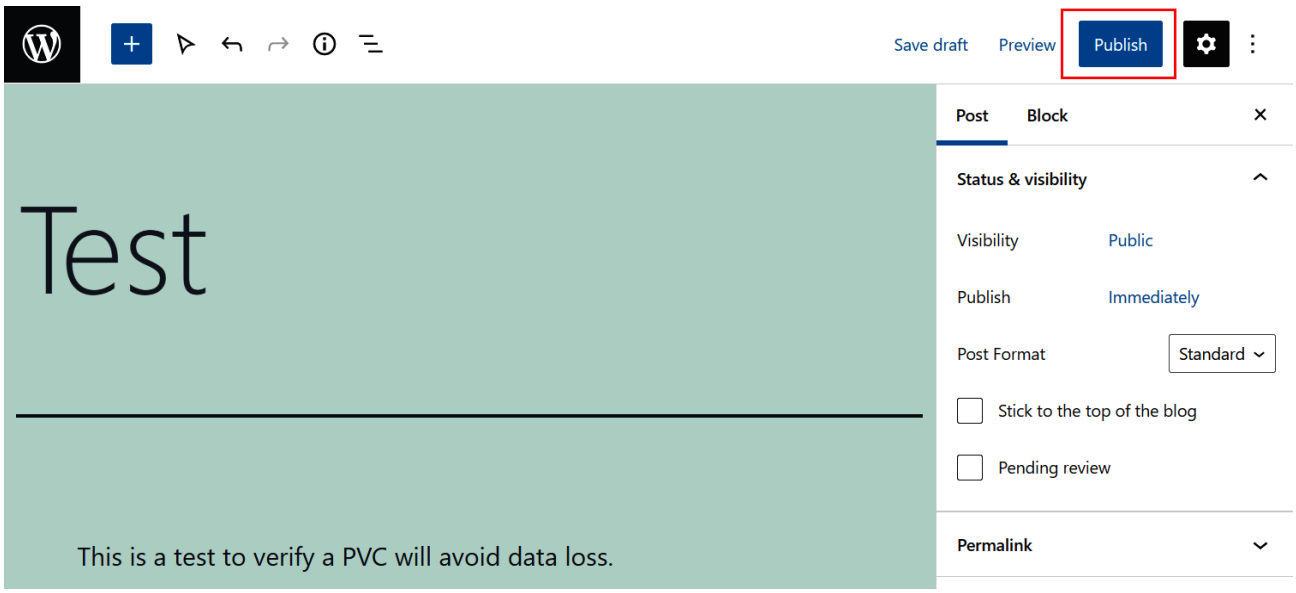


Figure 5.24: Publishing a post with random text

If you now head over to the main page of your website at `http://<external-ip>`, you'll see your test post as shown in *Figure 5.25*:

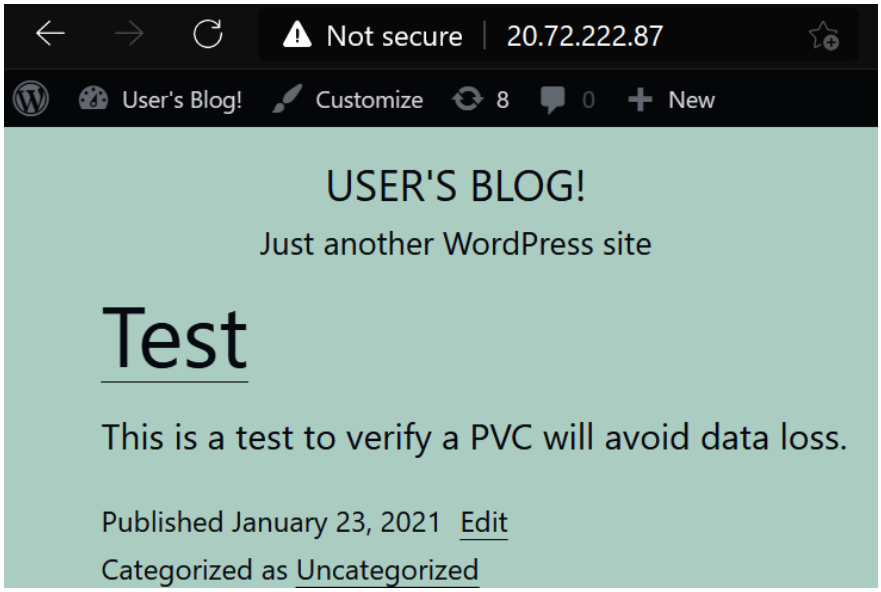


Figure 5.25: The published blog post appears on the home page

In this section, you deployed a WordPress site, you logged in to your WordPress site, and you created a post. You will verify whether this post survives a node failure in the next section.

Handling pod failure with PVC involvement

The first test you'll do with the PVCs is to kill the pods and verify whether the data has indeed persisted. To do this, let's do two things:

1. **Watch the pods in your application**: To do this, use the current Cloud Shell and execute the following command:

   ```
   kubectl get pods -w
   ```

2. **Kill the two pods that have the PVC mounted**: To do this, create a new Cloud Shell window by clicking on the icon shown in *Figure 5.26*:

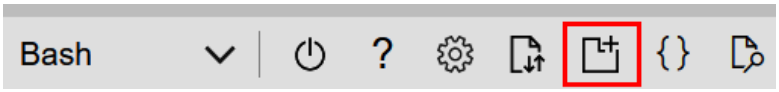


Figure 5.26: Opening a new Cloud Shell instance

Once you open a new Cloud Shell, execute the following command:

```
kubectl delete pod --all
```

In the original Cloud Shell, follow along with the `watch` command that you executed earlier. You should see an output like what's shown in *Figure 5.27*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -w
NAME                          READY   STATUS    RESTARTS   AGE
wp-mariadb-0                  1/1     Running   0          2m27s
wp-wordpress-6f7c4f85b5-gq27t   1/1     Running   0          2m40s
wp-mariadb-0                  1/1     Terminating   0          2m29s
wp-wordpress-6f7c4f85b5-gq27t   1/1     Terminating   0          2m42s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     Pending       0          0s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     Pending       0          0s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     ContainerCreating   0          0s
wp-mariadb-0                  0/1     Terminating         0          2m30s
wp-wordpress-6f7c4f85b5-gq27t   0/1     Terminating         0          2m43s
wp-wordpress-6f7c4f85b5-gq27t   0/1     Terminating         0          2m48s
wp-wordpress-6f7c4f85b5-gq27t   0/1     Terminating         0          2m48s
wp-mariadb-0                  0/1     Terminating         0          2m40s
wp-mariadb-0                  0/1     Terminating         0          2m40s
wp-mariadb-0                  0/1     Pending             0          0s
wp-mariadb-0                  0/1     Pending             0          0s
wp-mariadb-0                  0/1     ContainerCreating   0          0s
wp-wordpress-6f7c4f85b5-xm5bb   0/1     Running             0          66s
wp-mariadb-0                  0/1     Running             0          66s
wp-mariadb-0                  1/1     Running             0          98s
wp-wordpress-6f7c4f85b5-xm5bb   1/1     Running             0          111s
```

Figure 5.27: After deleting the pods, Kubernetes will automatically recreate both pods

As you can see, the two original pods went into a **Terminating** state. Kubernetes quickly started creating new pods to recover from the pod outage. The pods went through a similar life cycle as the original ones, going from **Pending** to **ContainerCreating** to **Running**.

3. If you head on over to your website, you should see that your demo post has been persisted. This is how PVCs can help you prevent data loss, as they persist data that would not have been persisted in the pod itself.

In this section, you've learned how PVCs can help when pods get recreated on the same node. In the next section, you'll see how PVCs are used when a node has a failure.

Handling node failure with PVC involvement

In the previous example, you saw how Kubernetes can handle pod failures when those pods have a PV attached. In this example, you'll learn how Kubernetes handles node failures when a volume is attached:

1. Let's first check which node is hosting your application, using the following command:

   ```
   kubectl get pods -o wide
   ```

   In the example shown in *Figure* 5.28, node **2** was hosting **MariaDB**, and node **0** was hosting the **WordPress** site:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP            NODE
wp-mariadb-0                  1/1     Running   0          9m17s   10.244.2.11   aks-agentpool-39838025-vmss000002
wp-wordpress-6f7c4f85b5-wp7qt   1/1     Running   0          2m13s   10.244.0.25   aks-agentpool-39838025-vmss000000
```

Figure 5.28: Check which node hosts the WordPress site

2. Introduce a failure and stop the node that is hosting the **WordPress** pod using the Azure portal. You can do this in the same way as in the earlier example. First, look for the scale set backing your cluster, as shown in *Figure* 5.29:

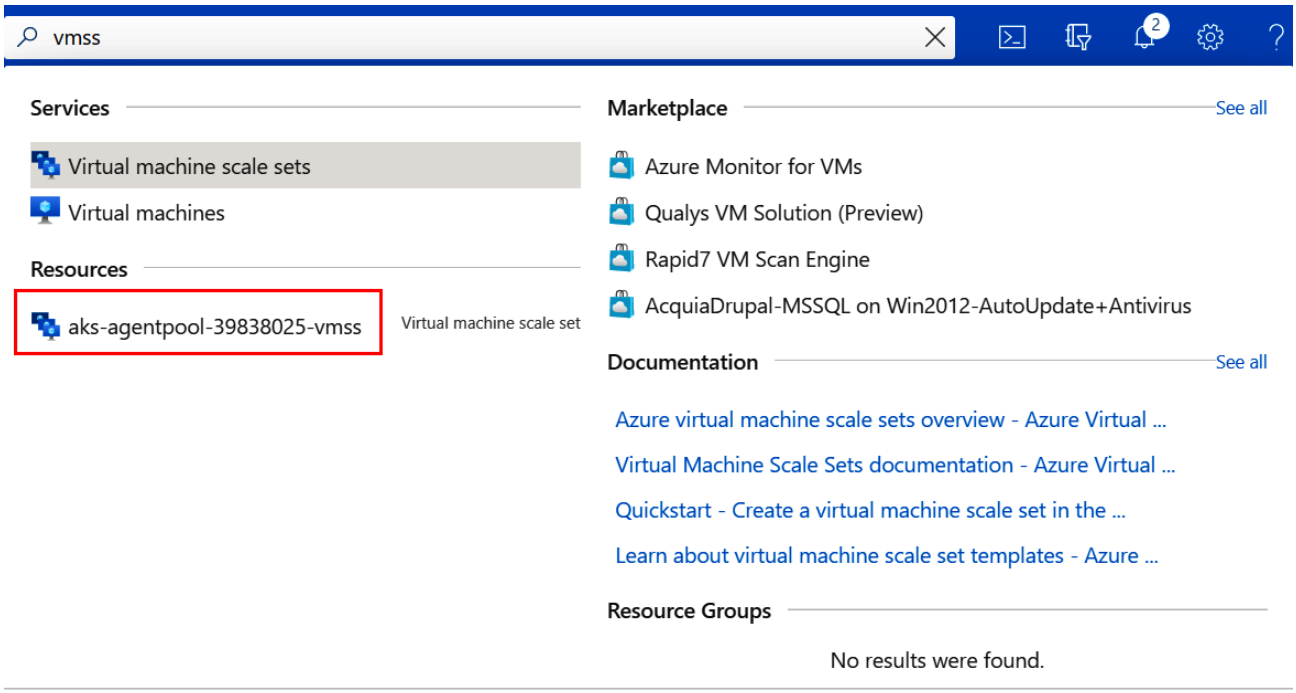


Figure 5.29: Looking for the scale set hosting your cluster

3. Then shut down the node, by clicking on **Instances** in the left-hand menu, then selecting the node you need to shut down and clicking the **Stop** button, as shown in *Figure* 5.30:

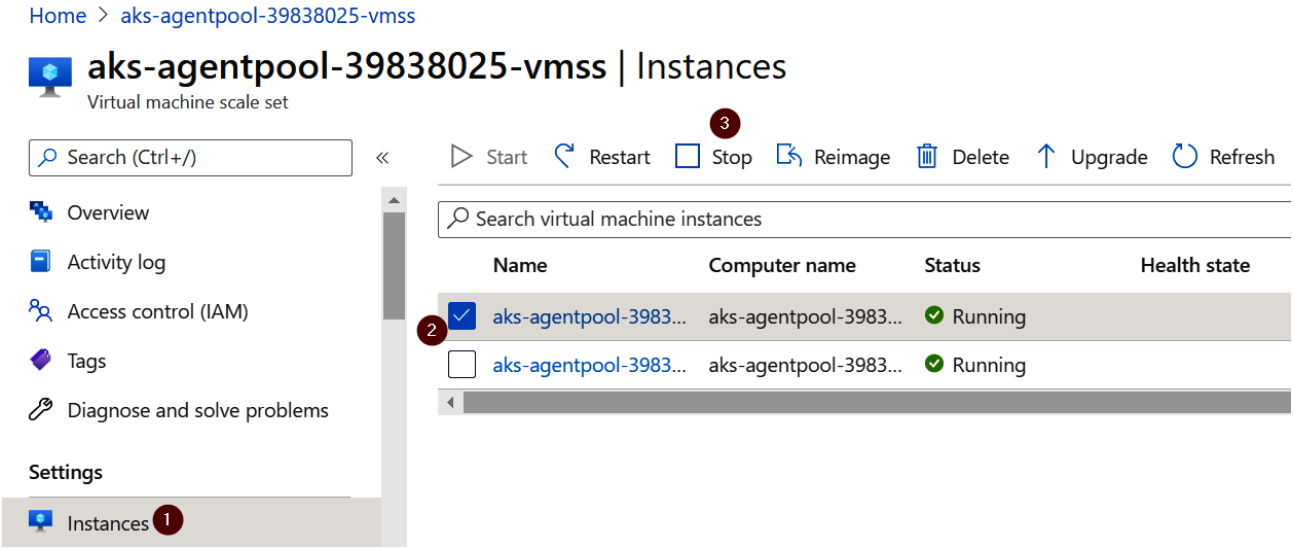


Figure 5.30: Shutting down the node

4. After this action, once again, watch the pods to see what is happening in the cluster:

```
kubectl get pods -o wide -w
```

As in the previous example, it is going to take 5 minutes before Kubernetes will start taking action against the failed node. You can see that happening in *Figure* 5.31:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -o wide -w
NAME                        READY   STATUS    RESTARTS   AGE     IP           NODE                                NOMINATED NODE
wp-mariadb-0                1/1     Running   0          10m     10.244.2.11  aks-agentpool-39838025-vmss000002   <none>
wp-wordpress-6f7c4f85b5-wp7qt   1/1     Running   0          3m41s   10.244.0.25  aks-agentpool-39838025-vmss000000   <none>
wp-wordpress-6f7c4f85b5-wp7qt   1/1     Running   0          4m32s   10.244.0.25  aks-agentpool-39838025-vmss000000   <none>
wp-wordpress-6f7c4f85b5-wp7qt   1/1     Terminating   0          9m37s   10.244.0.25  aks-agentpool-39838025-vmss000000   <none>
wp-wordpress-6f7c4f85b5-lczvx   0/1     Pending       0          0s      <none>       <none>                              <none>
wp-wordpress-6f7c4f85b5-lczvx   0/1     Pending       0          0s      <none>       aks-agentpool-39838025-vmss000002   <none>
wp-wordpress-6f7c4f85b5-lczvx   0/1     ContainerCreating   0          0s      <none>       aks-agentpool-39838025-vmss000002   <none>
```

Figure 5.31: A pod in a ContainerCreating state

5. You are seeing a new issue here. The new pod is stuck in a **ContainerCreating** state. Let's figure out what is happening here. First, describe that pod:

```
kubectl describe pods/wp-wordpress-<pod-id>
```

You will get an output as shown in *Figure* 5.32:

```
Events:
  Type     Reason              Age    From                     Message
  ----     ------              ----   ----                     -------
  Normal   Scheduled           3m31s  default-scheduler        Successfully assigned default/wp-wordpress-6f7c4f85b5-lczvx
to aks-agentpool-39838025-vmss000002
  Warning  FailedAttachVolume  3m31s  attachdetach-controller  Multi-Attach error for volume "pvc-848e0fc5-ed4b-4765-aa0d-5
44f014d6997" Volume is already used by pod(s) wp-wordpress-6f7c4f85b5-wp7qt
  Warning  FailedMount         88s    kubelet                  Unable to attach or mount volumes: unmounted volumes=[wordpr
ess-data], unattached volumes=[wordpress-data default-token-ktl66]: timed out waiting for the condition
```

Figure 5.32: Output explaining why the pod is in a ContainerCreating state

This tells you that there is a problem with the volume. You see two errors related to that volume: the `FailedAttachVolume` error explains that the volume is already used by another pod, and `FailedMount` explains that the current pod cannot mount the volume. You can solve this by manually forcefully removing the old pod stuck in the `Terminating` state.

> **Note**
>
> The behavior of the pod stuck in the `Terminating` state is not a bug. This is default Kubernetes behavior. The Kubernetes documentation states the following: *"Kubernetes (versions 1.5 or newer) will not delete pods just because a Node is unreachable. The pods running on an unreachable Node enter the Terminating or Unknown state after a timeout. Pods may also enter these states when the user attempts the graceful deletion of a pod on an unreachable Node."* You can read more at https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/.

6. To forcefully remove the terminating pod from the cluster, get the full pod name using the following command:

```
kubectl get pods
```

This will show you an output similar to *Figure* 5.33:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods
NAME                              READY   STATUS              RESTARTS   AGE
wp-mariadb-0                      1/1     Running             0          23m
wp-wordpress-6f7c4f85b5-lczvx     0/1     ContainerCreating   0          6m43s
wp-wordpress-6f7c4f85b5-wp7qt     1/1     Terminating         0          16m
```

Figure 5.33: Getting the name of the pod stuck in the Terminating state

7. Use the pod's name to force the deletion of this pod:

```
kubectl delete pod wordpress-wp-<pod-id> --force
```

8. After the pod has been deleted, it will take a couple of minutes for the other pod to enter a Running state. You can monitor the state of the pod using the following command:

```
kubectl get pods -w
```

This will return an output similar to *Figure* 5.34:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl get pods -w
NAME                              READY   STATUS              RESTARTS   AGE
wp-mariadb-0                      1/1     Running             0          30m
wp-wordpress-6f7c4f85b5-lczvx     0/1     ContainerCreating   0          14m
wp-wordpress-6f7c4f85b5-lczvx     0/1     Running             0          18m
wp-wordpress-6f7c4f85b5-lczvx     1/1     Running             0          18m
```

Figure 5.34: The new WordPress pod returning to a Running state

9. As you can see, this brought the new pod to a healthy state. It did take a couple of minutes for the system to pick up the changes and then mount the volume to the new pod. Let's get the details of the pod again using the following command:

```
kubectl describe pod wp-wordpress-<pod-id>
```

This will generate an output as follows:

```
Events:
  Type     Reason                  Age                  From                     Message
  ----     ------                  ----                 ----                     -------
  Normal   Scheduled               21m                  default-scheduler        Successfully assigned default/wp-wordpress
-6f7c4f85b5-lczvx to aks-agentpool-39838025-vmss000002
  Warning  FailedAttachVolume      21m                  attachdetach-controller  Multi-Attach error for volume "pvc-848e0fc
5-ed4b-4765-aa0d-544f014d6997" Volume is already used by pod(s) wp-wordpress-6f7c4f85b5-wp7qt
  Warning  FailedMount             6m2s (x3 over 15m)   kubelet                  Unable to attach or mount volumes: unmount
ed volumes=[wordpress-data], unattached volumes=[default-token-ktl66 wordpress-data]: timed out waiting for the condition
  Normal   SuccessfulAttachVolume  5m2s                 attachdetach-controller  AttachVolume.Attach succeeded for volume "
pvc-848e0fc5-ed4b-4765-aa0d-544f014d6997"
  Warning  FailedMount             3m46s (x5 over 19m)  kubelet                  Unable to attach or mount volumes: unmount
ed volumes=[wordpress-data], unattached volumes=[wordpress-data default-token-ktl66]: timed out waiting for the condition
  Normal   Pulled                  3m11s                kubelet                  Container image "docker.io/bitnami/wordpre
ss:5.6.0-debian-10-r30" already present on machine
  Normal   Created                 3m11s                kubelet                  Created container wordpress
  Normal   Started                 3m11s                kubelet                  Started container wordpress
```

Figure 5.35: The new pod is now attaching the volume and pulling the container image

10. This shows you that the new pod successfully got the volume attached and that the container image got pulled. This also made your WordPress website available again, which you can verify by browsing to the public IP. Before continuing to the next chapter, clean up the application using the following command:

```
helm delete wp
kubectl delete pvc --all
kubectl delete pv --all
```

11. Let's also start the node that was shut down: go back to the scale set pane in the Azure portal, click **Instances** in the left-hand menu, select the node you need to start, and click on the **Start** button, as shown in *Figure* 5.36:

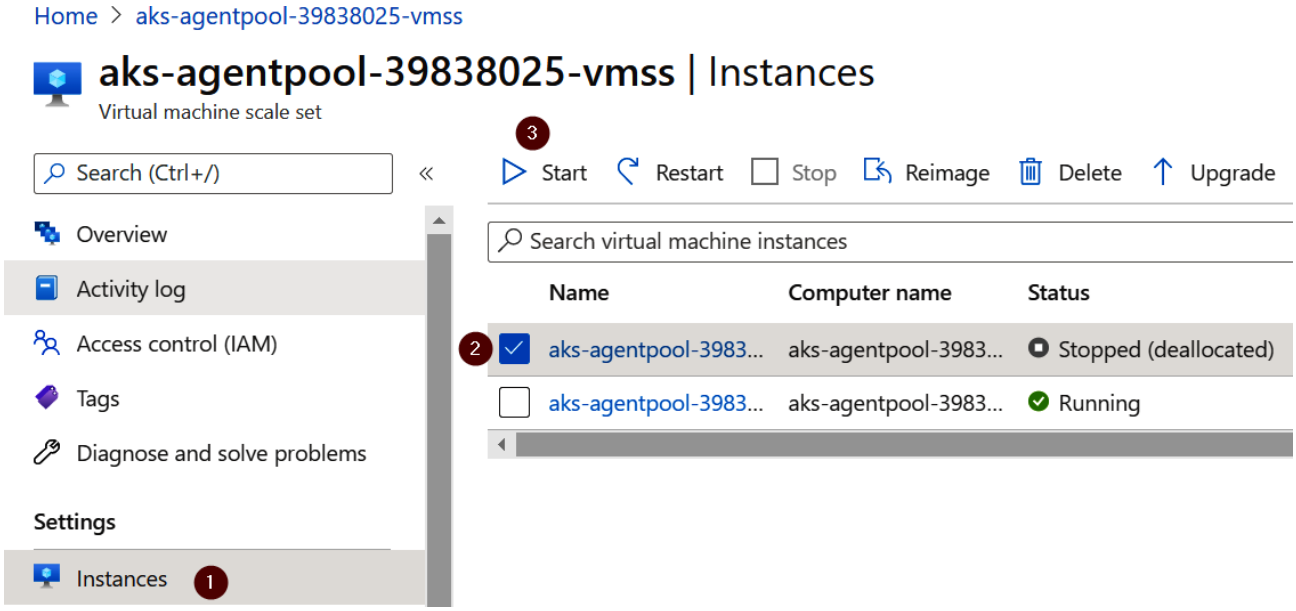


Figure 5.36: Starting node 0 again

In this section, you learned how you can recover from a node failure when PVCs aren't mounting to new pods. All you needed to do was forcefully delete the pod that was stuck in the `Terminating` state.

Summary

In this chapter, you learned about common Kubernetes failure modes and how you can recover from them. This chapter started with an example of how Kubernetes automatically detects node failures and how it will start new pods to recover the workload. After that, you scaled out your workload and had your cluster run out of resources. You recovered from that situation by starting the failed node again to add new resources to the cluster.

Next, you saw how PVs are useful to store data outside of a pod. You deleted all pods on the cluster and saw how the PV ensured that no data was lost in your application. In the final example in this chapter, you saw how you can recover from a node failure when PVs are attached. You were able to recover the workload by forcefully deleting the terminating pod. This brought your workload back to a healthy state.

This chapter has explained common failure modes in Kubernetes. In the next chapter, we will introduce HTTPS support to our services and introduce authentication with Azure Active Directory.

6

Securing your application with HTTPS

HTTPS has become a necessity for any public-facing website. Not only does it improve the security of your website, but it is also becoming a requirement for new browser functionalities. HTTPS is a secure version of the HTTP protocol. HTTPS makes use of **Transport Layer Security** (**TLS**) certificates to encrypt traffic between an end user and a server, or between two servers. TLS is the successor to the **Secure Sockets Layer** (**SSL**). The terms *TLS* and *SSL* are often used interchangeably.

In the past, you needed to buy certificates from a **certificate authority** (**CA**), then set them up on your web server and renew them periodically. While that is still possible today, the **Let's Encrypt** service and helpers in Kubernetes make it very easy to set up verified TLS certificates in your cluster. Let's Encrypt is a non-profit organization run by the **Internet Security Research Group** and backed by multiple companies. It is a free service that offers verified TLS certificates in an automated manner. Automation is a key benefit of the Let's Encrypt service.

In terms of Kubernetes helpers, you will learn about a new object called an **Ingress** and use a Kubernetes add-on called **cert-manager**. An ingress is an object within Kubernetes that manages external access to services, commonly used for HTTP services. An ingress adds additional functionality on top of the service object we explained in *Chapter 3*, *Application deployment on* AKS. It can be configured to handle HTTPS traffic. It can also be configured to route traffic to different back-end services based on the hostname, which is assigned by the **Domain Name System** (**DNS**) that is used to connect.

`cert-manager` is a Kubernetes add-on that helps in automating the creation of TLS certificates. It also helps in the rotation of certificates when they are close to expiring. `cert-manager` can interface with Let's Encrypt to request certificates automatically.

In this chapter, you will see how to set up Azure Application Gateway as a Kubernetes ingress, and `cert-manager` to interface with Let's Encrypt.

The following topics will be covered in this chapter:

- Setting up Azure Application Gateway as a Kubernetes ingress
- Setting up an ingress in front of a service
- Adding TLS support to an ingress

Let's start with setting up Azure Application Gateway as an ingress for AKS.

Setting up Azure Application Gateway as a Kubernetes ingress

An ingress in Kubernetes is an object that is used to route HTTP and HTTPS traffic from outside the cluster to services in a cluster. Exposing services using an ingress rather than exposing them directly, as you've done up to this point—has a number of advantages. These advantages include the ability to route multiple hostnames to the same public IP address and offloading TLS termination from the actual application to the ingress.

To create an ingress in Kubernetes, you need to install an ingress controller. An ingress controller is software that can create, configure, and manage ingresses in Kubernetes. Kubernetes does not come with a preinstalled ingress controller. There are multiple implementations of ingress controllers, and a full list is available at this URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

In Azure, application gateway is a Layer 7 load balancer, which can be used as an ingress for Kubernetes by using the **Application Gateway Ingress Controller (AGIC)**. A layer 7 load balancer is a load balancer that works at the application layer, which is the seventh and highest layer in the OSI networking reference model. Azure Application Gateway has a number of advanced features such as autoscaling and **Web Application Firewall (WAF)**.

There are two ways of configuring the AGIC, either using Helm or as an **Azure Kubernetes Service** (**AKS**) add-on. Installing AGIC using the AKS add-on functionality will result in a Microsoft-supported configuration. Additionally, the add-on method of deployment will be automatically updated by Microsoft, ensuring that your environment is always up to date.

In this section, you will create a new application gateway instance, set up AGIC using the add-on method, and finally, deploy an ingress resource to expose an application. Later in this chapter, you will extend this setup to also include TSL using a Let's Encrypt certificate.

Creating a new application gateway

In this section, you will use the Azure CLI to create a new application gateway. You will then use this application gateway in the next section to integrate with AGIC. The different steps in this section are summarized in the code samples for this chapter in the `setup-appgw.sh` file that is part of the code samples that come with this book.

1. To organize the resources created in this chapter, it is recommended that you create a new resource group. Make sure to create the new resource group in the same location you deployed your AKS cluster in. You can do this using the following command in the Azure CLI:

```
az group create -n agic -l westus2
```

2. Next, you will need to create the networking components required for your application gateway. These are a public IP with a DNS name and a new virtual network. You can do this using the following commands:

```
az network public-ip create -n agic-pip \
   -g agic --allocation-method Static --sku Standard \
   --dns-name "<your unique DNS name>"
az network vnet create -n agic-vnet -g agic \
  --address-prefix 192.168.0.0/24 --subnet-name agic-subnet \
  --subnet-prefix 192.168.0.0/24
```

> **Note**
>
> The az network public-ip create command might show you a warning message `[Coming breaking change] In the coming release, the default behavior will be changed as follows when sku is Standard and zone is not provided: For zonal regions, you will get a zone-redundant IP indicated by zones:["1","2","3"]; For non-zonal regions, you will get a non zone-redundant IP indicated by zones:[].`

3. Finally, you can create the application gateway. This command will take a few minutes to execute

```
az network application-gateway create -n agic -l westus2 \
  -g agic --sku Standard_v2 --public-ip-address agic-pip \
  --vnet-name agic-vnet --subnet agic-subnet
```

4. It will take a couple of minutes for the application gateway to deploy. Once it is created, you can see the resource in the Azure portal. To find this, look for `agic` (or the name you gave your application gateway) in the Azure search bar, and select your application gateway.

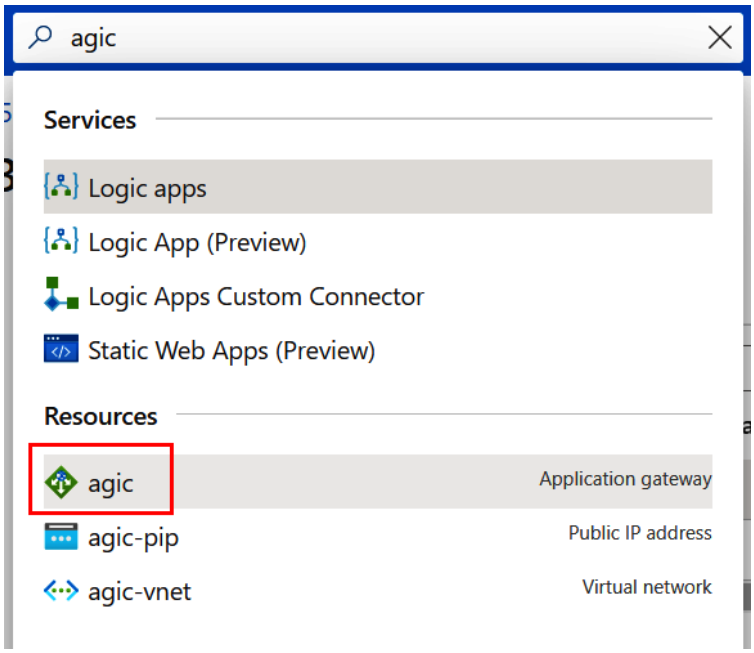


Figure 6.1: Looking for the application gateway in the Azure search bar

5. This will show you your application gateway in the Azure portal, as shown in *Figure 6.2*:

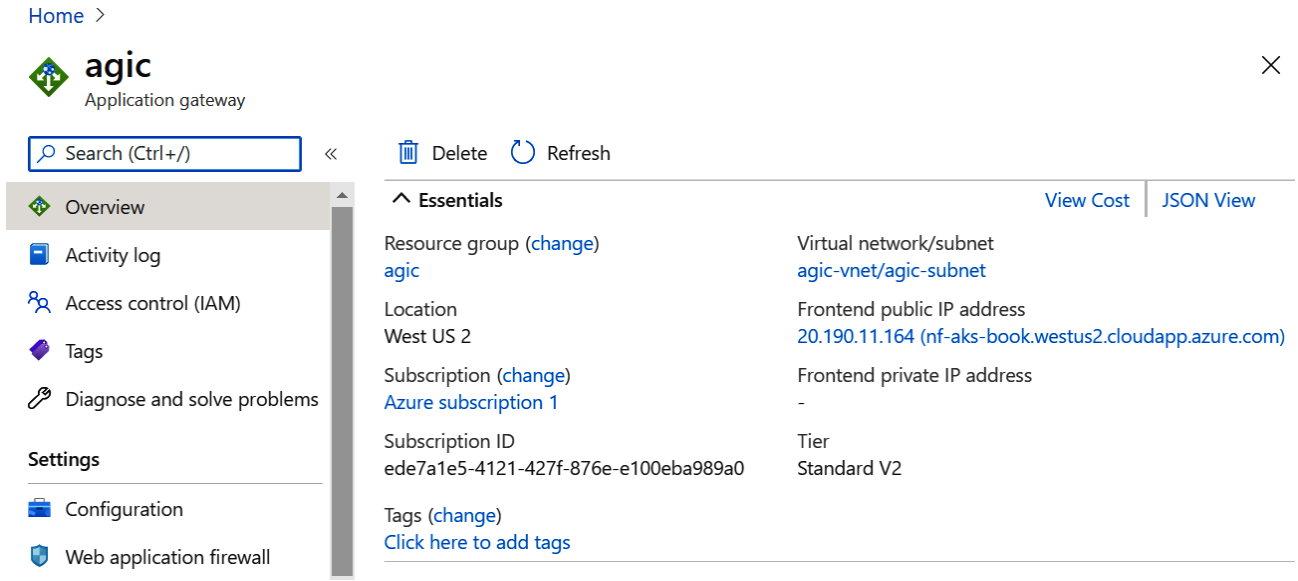


Figure 6.2: The application gateway in the Azure portal

6. To verify that it has been created successfully, browse to the DNS name you configured for the public IP address. This will show you an output similar to *Figure 6.3*. Note that the error message shown is expected since you haven't configured any applications yet behind the application gateway. You will configure applications behind the application gateway using AGIC in the *Adding an ingress rule for the guestbook application* section.

Figure 6.3: Verify that you can connect to the application gateway

Now that you've created a new application gateway and were able to connect to it, we will move on to integrating this application gateway with your existing Kubernetes cluster.

Setting up the AGIC

In this section, you will integrate the application gateway with your Kubernetes cluster using the AGIC AKS add-on. You will also set up virtual network peering so the application gateway can send traffic to your Kubernetes cluster.

1. To enable integration between your cluster and your application gateway, use the following command:

```
appgwId=$(az network application-gateway \
  show -n agic -g agic -o tsv --query "id")
az aks enable-addons -n handsonaks \
  -g rg-handsonaks -a ingress-appgw \
  --appgw-id $appgwId
```

2. Next, you will need to peer the application gateway network with the AKS network. To peer both networks, you can use the following code:

```
nodeResourceGroup=$(az aks show -n handsonaks \
  -g rg-handsonaks -o tsv --query "nodeResourceGroup")
aksVnetName=$(az network vnet list \
  -g $nodeResourceGroup -o tsv --query "[0].name")

aksVnetId=$(az network vnet show -n $aksVnetName \
  -g $nodeResourceGroup -o tsv --query "id")
az network vnet peering create \
  -n AppGWtoAKSVnetPeering -g agic \
  --vnet-name agic-vnet --remote-vnet $aksVnetId \
  --allow-vnet-access

appGWVnetId=$(az network vnet show -n agic-vnet \
  -g agic -o tsv --query "id")
az network vnet peering create \
  -n AKStoAppGWVnetPeering -g $nodeResourceGroup \
  --vnet-name $aksVnetName --remote-vnet $appGWVnetId --allow-vnet-
access
```

This concludes the integration between the application gateway and your AKS cluster. You've enabled the AGIC add-on, and connected both the networks together. In the next section, you will use this AGIC integration to create an ingress for a demo application.

Adding an ingress rule for the guestbook application

Up to this point, you have created a new application gateway and integrated it with your Kubernetes cluster. In this section, you will deploy the guestbook application and then expose it using an ingress.

1. To launch the guestbook application, type in the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

This will create the guestbook application you've used in the previous chapters. You should see the objects being created as shown in *Figure 6.4*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl create -f guestbook-all-in-one.yaml
service/redis-master created
deployment.apps/redis-master created
service/redis-replica created
deployment.apps/redis-replica created
service/frontend created
deployment.apps/frontend created
```

Figure 6.4: Creating the guestbook application

2. You can then use the following YAML file to expose the front-end service via the ingress. This is provided as `simple-frontend-ingress.yaml` in the source code for this chapter:

```
1   apiVersion: networking.k8s.io/v1
2   kind: Ingress
3   metadata:
4     name: simple-frontend-ingress
5     annotations:
6       kubernetes.io/ingress.class: azure/application-gateway
7   spec:
8     rules:
9     - http:
10        paths:
11        - path: /
12          pathType: Prefix
13          backend:
14            service:
15              name: frontend
16              port:
17                number: 80
```

Let's have a look at what is defined in this YAML file:

- **Line 1**: You specify the Kubernetes API version for the object you are creating.
- **Line 2**: You define that you are creating an `Ingress` object.
- **Lines 5-6**: Here, you're telling Kubernetes that you want to create an ingress of the class `azure/application-gateway`.

The following lines define the actual ingress:

- **Lines 8-12**: Here, you define the path this ingress is listening on. In our case, this is the top-level path. In more advanced cases, you can have different paths pointing to different services.
- **Lines 13-17**: These lines define the actual service this traffic should be pointed to.

You can use the following command to create this ingress:

```
kubectl apply -f simple-frontend-ingress.yaml
```

3. If you now go to http://dns-name/, which you created in the *Creating a new application gateway* section, you should get an output as shown in *Figure 6.5*:

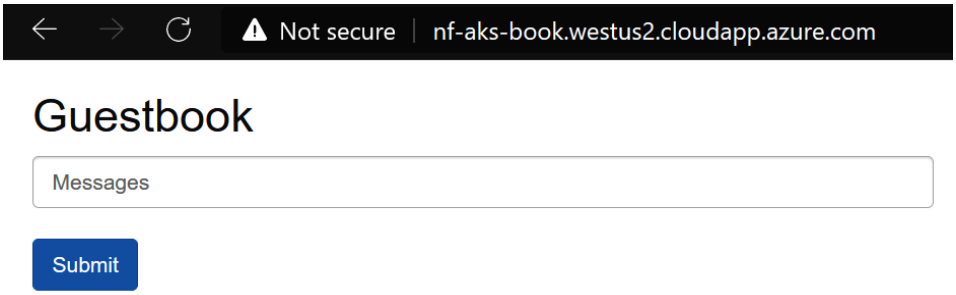


Figure 6.5: Accessing the guestbook application via the ingress

> **Note**
>
> You didn't have to publicly expose the front-end service as you have done in the preceding chapters. You have added the ingress as the exposed service, and the front-end service remains private to the cluster.

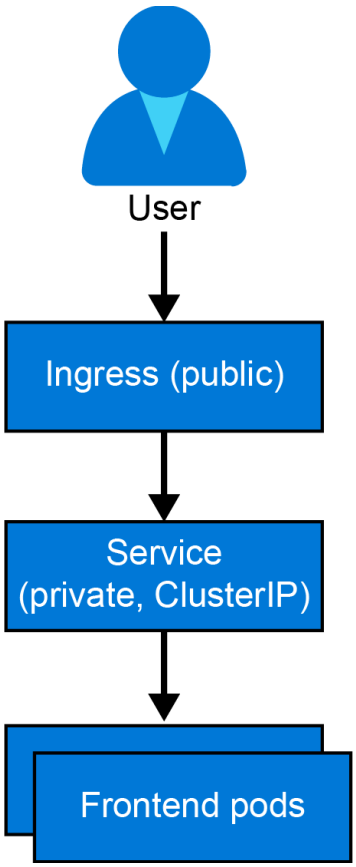


Figure 6.6: Flowchart displaying publicly accessible ingress

4. You can verify this by running the following command:

```
kubectl get service
```

5. This should show you that you have no public services, as seen by the lack of `EXTERNAL-IP` in *Figure 6.7*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get svc
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
frontend        ClusterIP   10.0.42.112    <none>        80/TCP     11m
kubernetes      ClusterIP   10.0.0.1       <none>        443/TCP    5h57m
redis-master    ClusterIP   10.0.36.111    <none>        6379/TCP   11m
redis-replica   ClusterIP   10.0.230.112   <none>        6379/TCP   11m
```

Figure 6.7: Output shows that you have no public services

In this section, you launched an instance of the guestbook application. You then exposed it publicly by creating an ingress, which in turn configured the application gateway that you created earlier. Only the ingress was publicly accessible.

Next, you'll extend the functionality of AGIC and learn how to secure traffic using a Certificate from Let's Encrypt.

Adding TLS to an ingress

You will now add HTTPS support to your application. To do this, you need a TLS certificate. You will be using the `cert-manager` Kubernetes add-on to request a certificate from Let's Encrypt.

> **Note**
>
> Although this section focuses on using an automated service such as Let's Encrypt, you can still pursue the traditional path of buying a certificate from an existing CA and importing it into Kubernetes. Please refer to the Kubernetes documentation for more information on how to do this: https://kubernetes.io/docs/concepts/services-networking/ingress/#tls

There are a couple of steps involved. The process of adding HTTPS to the application involves the following:

1. Install `cert-manager`, which interfaces with the Let's Encrypt API to request a certificate for the domain name you specify.
2. Install the certificate issuer, which will get the certificate from Let's Encrypt.
3. Create an SSL certificate for a given **Fully Qualified Domain Name** (**FQDN**). An FQDN is a fully qualified DNS record that includes the top-level domain name (such as .org or .com). You created an FQDN linked to your public IP in *step* 2 in the section *Creating a new application gateway*.
4. Secure the front-end service by creating an ingress to the service with the certificate created in *step* 3. In the example in this section, you will not be executing this step as an individual step. You will, however, reconfigure the ingress to automatically pick up the certificate created in *step* 3.

Let's start with the first step by installing `cert-manager` in the cluster.

Installing cert-manager

`cert-manager` (https://github.com/jetstack/cert-manager) is a Kubernetes add-on that automates the management and issuance of TLS certificates from various issuing sources. It is responsible for renewing certificates and ensuring they are updated periodically.

> **Note**
>
> The `cert-manager` project is not managed or maintained by Microsoft. It is an open-source solution previously managed by the company **Jetstack**, which recently donated it to the Cloud Native Computing Foundation.

The following commands install `cert-manager` in your cluster:

```
kubectl apply -f https://github.com/jetstack/cert-manager/releases/
download/v1.2.0/cert-manager.yaml
```

This will install a number of components in your cluster as shown in *Figure 6.8*. A detailed explanation of these components can be found in the `cert-manager` documentation at https://cert-manager.io/docs/installation/kubernetes/.

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl apply -f https://github.com/jetstack
/cert-manager/releases/download/v1.1.0/cert-manager.yaml
customresourcedefinition.apiextensions.k8s.io/certificaterequests.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/certificates.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/challenges.acme.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/clusterissuers.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/issuers.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/orders.acme.cert-manager.io created
namespace/cert-manager created
serviceaccount/cert-manager-cainjector created
serviceaccount/cert-manager created
serviceaccount/cert-manager-webhook created
clusterrole.rbac.authorization.k8s.io/cert-manager-cainjector created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-issuers created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-clusterissuers created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-certificates created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-orders created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-challenges created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-ingress-shim created
clusterrole.rbac.authorization.k8s.io/cert-manager-view created
clusterrole.rbac.authorization.k8s.io/cert-manager-edit created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-cainjector created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-issuers created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-clusterissuers created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-certificates created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-orders created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-challenges created
clusterrolebinding.rbac.authorization.k8s.io/cert-manager-controller-ingress-shim created
role.rbac.authorization.k8s.io/cert-manager-cainjector:leaderelection created
role.rbac.authorization.k8s.io/cert-manager:leaderelection created
role.rbac.authorization.k8s.io/cert-manager-webhook:dynamic-serving created
rolebinding.rbac.authorization.k8s.io/cert-manager-cainjector:leaderelection created
rolebinding.rbac.authorization.k8s.io/cert-manager:leaderelection created
rolebinding.rbac.authorization.k8s.io/cert-manager-webhook:dynamic-serving created
service/cert-manager created
service/cert-manager-webhook created
deployment.apps/cert-manager-cainjector created
deployment.apps/cert-manager created
deployment.apps/cert-manager-webhook created
mutatingwebhookconfiguration.admissionregistration.k8s.io/cert-manager-webhook created
validatingwebhookconfiguration.admissionregistration.k8s.io/cert-manager-webhook created
```

Figure 6.8: Installing cert-manager in your cluster

`cert-manager` makes use of a Kubernetes functionality called **CustomResourceDefinition** (**CRD**). CRD is a functionality used to extend the Kubernetes API server to create custom resources. In the case of `cert-manager`, there are six CRDs that are created, some of which you will use later in this chapter.

Now that you have installed `cert-manager`, you can move on to the next step: setting up a certificate issuer.

Installing the certificate issuer

In this section, you will install the Let's Encrypt staging certificate issuer. A certificate can be issued by multiple issuers. `letsencrypt-staging`, for example, is for testing purposes. As you are building tests, you'll use the staging server. The code for the certificate issuer has been provided in the source code for this chapter in the `certificate-issuer.yaml` file. As usual, use `kubectl create -f certificate-issuer.yaml`; the YAML file has the following contents:

```
1   apiVersion: cert-manager.io/v1
2   kind: Issuer
3   metadata:
4     name: letsencrypt-staging
5   spec:
6     acme:
7       server: https://acme-staging-v02.api.letsencrypt.org/directory
8       email: <your e-mail address>
9       privateKeySecretRef:
10        name: letsencrypt-staging
11      solvers:
12      - http01:
13          ingress:
14            class: azure/application-gateway
```

Let's look at what we have defined here:

- **Lines 1-2**: Here, you point to one of the CRDs that `cert-manager` created. In this case, specifically, you point to the `Issuer` object. An issuer is a link between your Kubernetes cluster and the actual certificate authority creating the certificate, which is Let's Encrypt in this case.
- **Lines 6-10**: Here you provide the configuration for Let's Encrypt and point to the staging server.
- **Lines 11-14**: This is additional configuration for the ACME client to certify domain ownership. You point Let's Encrypt to the Azure Application Gateway ingress to verify that you own the domain you will request a certificate for later.

With the certificate issuer installed, you can now move on to the next step: creating the TLS certificate on the ingress.

Creating the TLS certificate and securing the ingress

In this section, you will create a TLS certificate. There are two ways you can configure `cert-manager` to create certificates. You can either manually create a certificate and link it to the ingress, or you can configure your ingress controller, so `cert-manager` automatically creates the certificate.

In this example, you will configure your ingress using the latter method.

1. To start, edit the ingress to look like the following YAML code. This file is present in the source code on GitHub as `ingress-with-tls.yaml`:

```
1   apiVersion: networking.k8s.io/v1
2   kind: Ingress
3   metadata:
4     name: simple-frontend-ingress
5     annotations:
6       kubernetes.io/ingress.class: azure/application-gateway
7       cert-manager.io/issuer: letsencrypt-staging
8       cert-manager.io/acme-challenge-type: http01
9   spec:
10    rules:
```

```
11    - http:
12        paths:
13        - path: /
14          pathType: Prefix
15          backend:
16            service:
17              name: frontend
18              port:
19                number: 80
20      host: <your dns-name>.<your azure region>.cloudapp.azure.com
21    tls:
22      - hosts:
23        - <your dns-name>.<your azure region>.cloudapp.azure.com
24        secretName: frontend-tls
```

You should make the following changes to the original ingress:

- **Lines 7-8**: You add two additional annotations to the ingress that points to a certificate issuer and `acme-challenge` to prove domain ownership.
- **Line 20**: The domain name for the ingress is added here. This is required because Let's Encrypt only issues certificates for domains.
- **Line 21-24**: This is the TLS configuration of the ingress. It contains the hostname as well as the name of the secret that will be created to store the certificate.

2. You can update the ingress you created earlier with the following command:

```
kubectl apply -f ingress-with-tls.yaml
```

It takes `cert-manager` about a minute to request a certificate and configure the ingress to use that certificate. While you are waiting for that, let's have a look at the intermediate resources that `cert-manager` created on your behalf.

3. First off, `cert-manager` created a `certificate` object for you. You can look at the status of that object using the following:

```
kubectl get certificate
```

This command will generate an output as shown in *Figure 6.9*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get certificate
NAME           READY   SECRET         AGE
frontend-tls   False   frontend-tls   3s
```

Figure 6.9: The status of the certificate object

4. As you can see, the certificate isn't ready yet. There is another object that `cert-manager` created to actually get the certificate. This object is `certificaterequest`. You can get its status by using the following command:

```
kubectl get certificaterequest
```

This will generate the output shown in *Figure 6.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get certificaterequest
NAME                  READY   AGE
frontend-tls-p528r    False   4s
```

Figure 6.10: The status of the certificaterequest obiect

You can also get more details about the request by issuing a `describe` command against the `certificaterequest` object:

```
kubectl describe certificaterequest
```

While you're waiting for the certificate to be issued, the status will look similar to *Figure 6.11*:

```
Status:
  Conditions:
    Last Transition Time:  2021-01-24T22:57:12Z
    Message:               Waiting on certificate issuance from order default/frontend-tls-p528r-
3330258237: "pending"
    Reason:                Pending
    Status:                False
    Type:                  Ready
Events:
  Type    Reason        Age   From          Message
  ----    ------        ----  ----          -------
  Normal  OrderCreated  10s   cert-manager  Created Order resource default/frontend-tls-p528r-333
0258237
  Normal  OrderPending  10s   cert-manager  Waiting on certificate issuance from order default/fr
ontend-tls-p528r-3330258237: ""
```

Figure 6.11: Using the kubectl describe command to obtain details of the certificaterequest object

As you can see, the `certificaterequest` object shows you that the order has been created and that it is pending.

5. After a couple of additional seconds, the `describe` command should return a successful certificate creation message. Run the following command to get the updated status:

```
kubectl describe certificaterequest
```

The output of this command is shown in *Figure 6.12*:

```
Events:
  Type    Reason             Age    From          Message
  ----    ------             ----   ----          -------
  Normal  OrderCreated       6m39s  cert-manager  Created Order resource default/frontend-tls-p52
8r-3330258237
  Normal  OrderPending       6m39s  cert-manager  Waiting on certificate issuance from order defa
ult/frontend-tls-p528r-3330258237: ""
  Normal  CertificateIssued  5m54s  cert-manager  Certificate fetched from issuer successfully
```

Figure 6.12: The issued certificate

This should now enable the front-end ingress to be served over HTTPS.

6. Let's try this out in a browser by browsing to the DNS name you created in the *Creating a new application gateway* section. Depending on your browser's cache, you might need to add `https://` in front of the URL.
7. Once you reach the ingress, it will indicate an error in the browser, showing you that the certificate isn't valid, similar to *Figure 6.13*. This is to be expected since you are using the Let's Encrypt staging server:

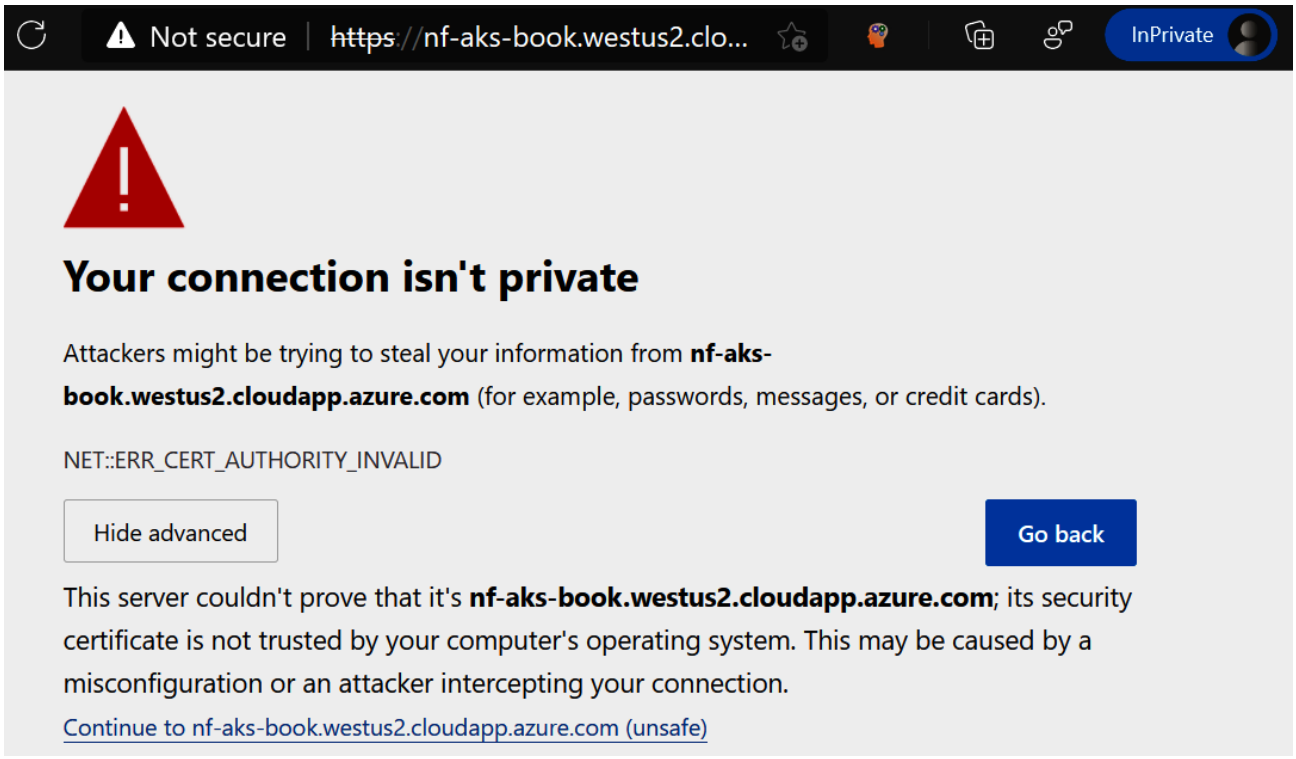


Figure 6.13: Using the Let's Encrypt staging server, the certificate isn't trusted by default

You can browse to your application by clicking **Advanced** and selecting **Continue**.

In this section, you successfully added a TLS certificate to your ingress to secure traffic to it. Since you were able to complete the test with the staging certificate, you can now move on to a production system.

Switching from staging to production

In this section, you will switch from a staging certificate to a production-level certificate. To do this, you can redo the previous exercise by creating a new issuer in your cluster, like the following (provided in `certificate-issuer-prod.yaml` as part of the code samples with this book). Don't forget to change your email address in the file. The following code is contained in that file:

```
apiVersion: cert-manager.io/v1alpha2
kind: Issuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: <your e-mail>
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
    - http01:
        ingress:
          class: azure/application-gateway
```

Then, replace the reference to the issuer in the `ingress-with-tls.yaml` file with `letsencrypt-prod` as shown (provided in the `ingress-with-tls-prod.yaml` file):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-frontend-ingress
  annotations:
    kubernetes.io/ingress.class: azure/application-gateway
    cert-manager.io/issuer: letsencrypt-prod
    cert-manager.io/acme-challenge-type: http01
spec:
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
```

```
17              name: frontend
18              port:
19                number: 80
20      host: <your dns-name>.<your azure region>.cloudapp.azure.com
21    tls:
22      - hosts:
23        - <your dns-name>.<your azure region>.cloudapp.azure.com
24        secretName: frontend-prod-tls
```

To apply these changes, execute the following commands:

```
kubectl create -f certificate-issuer-prod.yaml
kubectl apply -f ingress-with-tls-prod.yaml
```

It will again take about a minute for the certificate to become active. Once the new certificate is issued, you can browse to your DNS name again and shouldn't see any more warnings regarding invalid certificates. If you click the padlock icon in the browser, you should see that your connection is secure and uses a valid certificate:

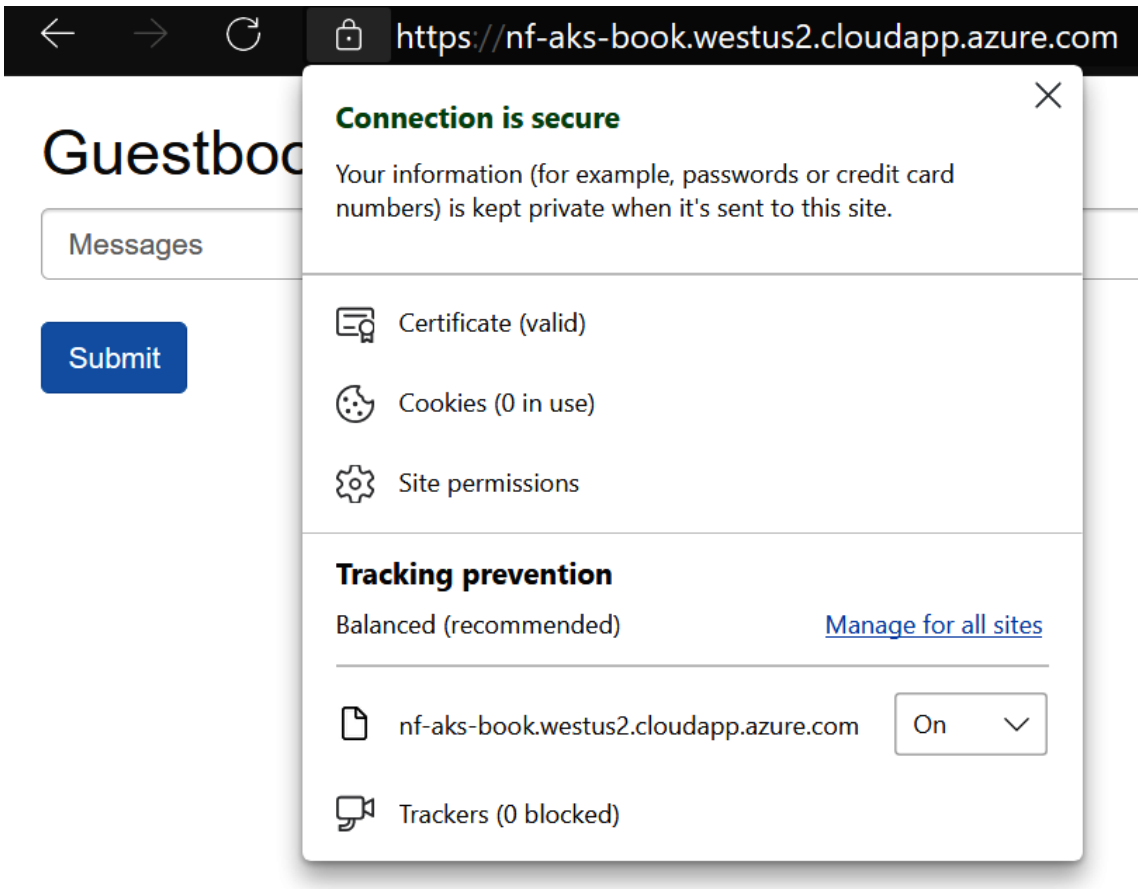


Figure 6.14: The web page displaying a valid certificate

In this section, you have learned how to add TLS support to an ingress. You did this by installing the `cert-manager` Kubernetes add-on. `cert-manager` got a free certificate from Let's Encrypt and added this to the existing ingress deployed on the application gateway. The process that was described here is not specific to Azure and Azure Application Gateway. This process of adding TLS to an ingress works with other ingress controllers as well.

Let's delete the resources you created during this chapter:

```
kubectl delete -f https://github.com/jetstack/cert-manager/releases/
download/v1.1.0/cert-manager.yaml
az aks disable-addons -n handsonaks \
  -g rg-handsonaks -a ingress-appgw
```

Summary

In this chapter, you added HTTPS security to the guestbook application without actually changing the source code. You started by setting up a new application gateway and configured AGIC on AKS. This gives you the ability to create Kubernetes ingresses that can be configured on the application gateway.

Then, you installed a certificate manager that interfaces with the Let's Encrypt API to request a certificate for the domain name we subsequently specified. You leveraged a certificate issuer to get the certificate from Let's Encrypt. You then reconfigured the ingress to request a certificate from this issuer in the cluster. Using these capabilities of both the certificate manager as well as the ingress, you are now able to secure your websites using TLS.

In the next chapter, you will learn how to monitor your deployments and set up alerts. You will also learn how to quickly identify root causes when errors do occur, and how to debug applications running on AKS. At the same time, you'll learn how to perform the correct fixes once you have identified the root causes.

7

Monitoring the AKS cluster and the application

Now that you know how to deploy applications on an AKS cluster, let's focus on how you can ensure that your cluster and applications remain available. In this chapter, you will learn how to monitor your cluster and the applications running on it. You'll explore how Kubernetes makes sure that your applications are running reliably using readiness and liveness probes.

You will also learn how **AKS Diagnostics** and **Azure Monitor** are used, and how they are integrated within the Azure portal. You will see how you can use AKS Diagnostics to monitor the status of the cluster itself, and how Azure Monitor helps monitor the pods on the cluster and allows you to get access to the logs of the pods at scale.

In brief, the following topics will be covered in this chapter:

- Monitoring and debugging applications using `kubectl`
- Reviewing metrics reported by Kubernetes
- Reviewing metrics from Azure Monitor

Let's start the chapter by reviewing some of the commands in `kubectl` that you can use to monitor your applications.

Commands for monitoring applications

Monitoring the health of applications deployed on Kubernetes as well as the Kubernetes infrastructure itself is essential for providing a reliable service to your customers. There are two primary use cases for monitoring:

- Ongoing monitoring to get alerts if something is not behaving as expected
- Troubleshooting and debugging application errors

When observing an application running on top of a Kubernetes cluster, you'll need to examine multiple things in parallel, including containers, pods, services, and the nodes in the cluster. For ongoing monitoring, you'll need a monitoring system such as Azure Monitor or Prometheus. Azure Monitor will be introduced later in this chapter. Prometheus (https://prometheus.io/) is a popular open-source solution within the Kubernetes ecosystem to monitor Kubernetes environments. For troubleshooting, you'll need to interact with the live cluster. The most common commands used for troubleshooting are as follows:

```
kubectl get <resource type> <resource name>
kubectl describe <resource type> <resource name>
kubectl logs <pod name>
```

Each of these commands will be described in detail later in this chapter.

To begin with the practical examples, recreate the guestbook example again using the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

While the `create` command is running, you will watch its progress in the following sections. Let's start by exploring the `get` command.

The kubectl get command

To see the overall picture of deployed applications, `kubectl` provides the `get` command. The `get` command lists the resources that you specify. Resources can be pods, ReplicaSets, ingresses, nodes, deployments, secrets, and so on. You have already run this command in the previous chapters to verify that an application was ready for use.

Perform the following steps:

1. Run the following `get` command, which will get us the resources and their statuses:

   ```
   kubectl get all
   ```

   This will show you all the deployments, ReplicaSets, pods, and services in your namespace:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get all
NAME                                 READY   STATUS        RESTARTS   AGE
pod/frontend-766d4f77cb-859v8        1/1     Running       0          17s
pod/frontend-766d4f77cb-grfcx        1/1     Running       0          17s
pod/frontend-766d4f77cb-vq5dd        1/1     Running       0          17s
pod/redis-master-f46ff57fd-fb5k2     1/1     Running       0          17s
pod/redis-replica-57c8c66cc4-8jx7t   1/1     Running       0          17s
pod/redis-replica-57c8c66cc4-crltb   1/1     Running       0          17s
pod/redis-replica-57c8c66cc4-fddvg   0/1     Terminating   0          76m
pod/redis-replica-57c8c66cc4-mwtp9   0/1     Terminating   0          76m

NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)        AGE
service/frontend        LoadBalancer   10.0.184.216   51.143.114.234   80:30977/TCP   17s
service/kubernetes      ClusterIP      10.0.0.1       <none>           443/TCP        34h
service/redis-master    ClusterIP      10.0.102.11    <none>           6379/TCP       17s
service/redis-replica   ClusterIP      10.0.48.214    <none>           6379/TCP       17s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/frontend           3/3     3            3           17s
deployment.apps/redis-master       1/1     1            1           17s
deployment.apps/redis-replica      2/2     2            2           17s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/frontend-766d4f77cb           3         3         3       17s
replicaset.apps/redis-master-f46ff57fd        1         1         1       17s
replicaset.apps/redis-replica-57c8c66cc4      2         2         2       17s
```

Figure 7.1: All the resources running in the default namespace

2. Focus your attention on the pods in your deployment. You can get the status of the pods with the following command:

   ```
   kubectl get pods
   ```

   You will see that only the pods are shown, as seen in *Figure 7.2*. Let's investigate this in detail:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-ds6gb         1/1     Running   0          103s
frontend-766d4f77cb-gvbjj         1/1     Running   0          103s
frontend-766d4f77cb-qw8kv         1/1     Running   0          103s
redis-master-f46ff57fd-qs6gk      1/1     Running   0          103s
redis-replica-57c8c66cc4-fddvg    1/1     Running   0          103s
redis-replica-57c8c66cc4-mwtp9    1/1     Running   0          103s
```

Figure 7.2: All the pods in your namespace

   The first column indicates the pod name, for example, `frontend-766d4f77cb-ds6gb`. The second column indicates how many containers in the pod are ready against the total number of containers in the pod. Readiness is defined via a readiness probe in Kubernetes. There is a dedicated section called *Readiness and liveness probes* later in this chapter.

   The third column indicates the status, for example, `Pending`, `ContainerCreating`, `Running`, and so on. The fourth column indicates the number of restarts, while the fifth column indicates the age when the pod was asked to be created.

3. If you need more information about your pod, you can add extra columns to the output of a `get` command by adding `-o wide` to the command like this:

   ```
   kubectl get pods -o wide
   ```

   This will show you additional information, as shown in *Figure 7.3*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE     IP            NODE                                NOMINATED NODE   READINESS GATES
frontend-766d4f77cb-ds6gb         1/1     Running   0          3m56s   10.244.0.44   aks-agentpool-39838025-vmss000000   <none>           <none>
frontend-766d4f77cb-gvbjj         1/1     Running   0          3m56s   10.244.2.29   aks-agentpool-39838025-vmss000002   <none>           <none>
frontend-766d4f77cb-qw8kv         1/1     Running   0          3m56s   10.244.0.45   aks-agentpool-39838025-vmss000000   <none>           <none>
redis-master-f46ff57fd-qs6gk      1/1     Running   0          3m56s   10.244.0.42   aks-agentpool-39838025-vmss000000   <none>           <none>
redis-replica-57c8c66cc4-fddvg    1/1     Running   0          3m56s   10.244.2.28   aks-agentpool-39838025-vmss000002   <none>           <none>
redis-replica-57c8c66cc4-mwtp9    1/1     Running   0          3m56s   10.244.0.43   aks-agentpool-39838025-vmss000000   <none>           <none>
```

Figure 7.3: Adding -o wide shows more details on the pods

The extra columns include the IP address of the pod, the node it is running on, the nominated node, and readiness gates. A nominated node is only set when a higher-priority pod preempts a lower-priority pod. The nominated node field would then be set on the higher-priority pod. It signifies the node that the higher-priority pod will be scheduled once the lower-priority pod has terminated gracefully. A readiness gate is a way to introduce external system components as the readiness for a pod.

Executing a `get pods` command only shows the state of the current pod. As we will see next, things can fail at any of the states, and we need to use the `kubectl describe` command to dig deeper.

The kubectl describe command

The `kubectl describe` command gives you a detailed view of the object you are describing. It contains the details of the object itself, as well as any recent events related to that object. While the `kubectl get events` command lists all the events for the entire namespace, with the `kubectl describe` command, you would get only the events for that specific object. If you are interested in just pods, you can use the following command:

```
kubectl describe pods
```

The preceding command lists all the information pertaining to all pods. This is typically too much information to contain in a typical shell.

If you want information on a particular pod, you can type the following:

```
kubectl describe pod/<pod-name>
```

> **Note**
>
> You can either use a slash or a space in between pod and `<pod-name>`.
> The following two commands will have the same output:
> `kubectl describe pod/<pod-name>`
> `kubectl describe pod <pod-name>`

You will get an output similar to *Figure* 7.4, which will be explained in detail later:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl describe pod frontend-766d4f77cb-ds6gb
Name:         frontend-766d4f77cb-ds6gb
Namespace:    default
Priority:     0
Node:         aks-agentpool-39838025-vmss000000/10.240.0.4
Start Time:   Tue, 26 Jan 2021 02:10:33 +0000
Labels:       app=guestbook
              pod-template-hash=766d4f77cb
              tier=frontend
Annotations:  <none>
Status:       Running
IP:           10.244.0.44
IPs:
  IP:           10.244.0.44
Controlled By:  ReplicaSet/frontend-766d4f77cb
Containers:
  php-redis:
    Container ID:   containerd://f202c0fc671be873362ff3a097c30193b04182ffdd1f5065aeb9b5daac724762
    Image:          gcr.io/google-samples/gb-frontend:v4
    Image ID:       sha256:c8cb3a8f677bc4b7fb210d98368dae7b6268451897d43ebbc4add5265574b610
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Tue, 26 Jan 2021 02:10:34 +0000
    Ready:          True
    Restart Count:  0
    Requests:
      cpu:     10m
      memory:  10Mi
    Environment:
      GET_HOSTS_FROM:  dns
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-ktl66 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-ktl66:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-ktl66
    Optional:    false
QoS Class:       Burstable
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/memory-pressure:NoSchedule op=Exists
                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                 node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  50m   default-scheduler  Successfully assigned default/frontend-766d4f77cb-ds6gb to
 aks-agentpool-39838025-vmss000000
  Normal  Pulled     50m   kubelet            Container image "gcr.io/google-samples/gb-frontend:v4" alr
eady present on machine
  Normal  Created    49m   kubelet            Created container php-redis
  Normal  Started    49m   kubelet            Started container php-redis
```

Figure 7.4: Describing an object shows the detailed output of that object

From the description, you can get the node on which the pod is running, how long it has been running, its internal IP address, the Docker image name, the ports exposed, the env variables, and the events (from within the past hour).

In the preceding example, the pod name is `frontend-766d4f77cb-ds6gb`. As mentioned in *Chapter 1*, *Introduction to containers and Kubernetes*, it has the `<ReplicaSet name>-<random 5 chars>` format. The `replicaset` name itself is randomly generated from the deployment name front end: `<deployment name>-<random-string>`.

Figure 7.5 shows the relationship between a deployment, a ReplicaSet, and pods:

Figure 7.5: Relationship between a deployment, a ReplicaSet, and pods

The namespace under which this pod runs is `default`. So far, you have just been using the `default` namespace, appropriately named `default`.

Another section that is important from the preceding output is the node section:

```
Node:           aks-agentpool-39838025-vmss000000/10.240.0.4
```

The node section lets you know which physical node/VM the pod is running on. If the pod is repeatedly restarting or having issues running and everything else seems OK, there might be an issue with the node itself. Having this information is essential to perform advanced debugging.

The following is the time the pod was initially scheduled:

```
Start Time:   Tue, 26 Jan 2021 02:10:33 +0000
```

This doesn't mean that the pod has been running since that time, so the time can be misleading in that sense. If a health event occurs (for example, a container crashes), the pod will reset automatically.

You can add more information about a workload in Kubernetes using `Labels`, as shown here:

```
Labels:app=guestbook
pod-template-hash=57d8c9fb45
tier=frontend
```

Labels are a commonly used functionality in Kubernetes. For example, this is how links between objects, such as `service` to pod and `deployment` to `ReplicaSet` to pod (*Figure 7.5*), are made. If you see that traffic is not being routed to a pod from a service, this is the first thing you should check. Also, you'll notice that the `pod-template-hash` label also occurs in the pod name. This is how the link between the ReplicaSet and the pod is made. If the labels don't match, the resources won't attach.

The following shows the internal IP of the pod and its status:

```
Status:       Running
IP:           10.244.0.44
IPs:
  IP:           10.244.0.44
```

As mentioned in previous chapters, when building out your application, the pods can be moved to different nodes and get a different IP, so you should avoid using these IP addresses. However, when debugging application issues, having a direct IP for a pod can help with troubleshooting. Instead of connecting to your application through a service object, you can connect directly from one pod to another using the other pod's IP address to test connectivity.

The containers running in the pod and the ports that are exposed are listed in the following block:

```
Containers:
  php-redis:
    ...
    Image:          gcr.io/google-samples/gb-frontend:v4
    ...
    Port:           80/TCP
    ...
    Requests:
      cpu:     10m
```

```
      memory:  10Mi
    Environment:
      GET_HOSTS_FROM:  dns
    ...
```

In this case, you are getting the `gb-frontend` container with the `v4` tag from the `gcr.io` container registry, and the repository name is `google-samples`.

Port `80` is exposed to outside traffic. Since each pod has its own IP, the same port can be exposed for multiple instances of the same pod even when running on the same host. For instance, if you had two pods running a web server on the same node, both could use port `80`, since each pod has its own IP address. This is a huge management advantage as you don't have to worry about port collisions on the same node.

Any events that occurred in the previous hour show up here:

```
Events:
```

Using `kubectl describe` is very useful to get more context about the resources you are running. The final section contains events related to the object you were describing. You can get all events in your cluster using the `kubectl get events` command.

To see the events for all resources in the system, run the following command:

```
kubectl get events
```

> **Note**
>
> Kubernetes maintains events for only 1 hour by default.

If everything goes well, you should have an output similar to *Figure 7.6*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get events
LAST SEEN   TYPE     REASON             OBJECT                          MESSAGE
11m         Normal   Scheduled          pod/frontend-766d4f77cb-7w5gz   Successfully assigned default/frontend-766d4f77cb-7
w5gz to aks-agentpool-39838025-vmss000000
11m         Normal   Pulled             pod/frontend-766d4f77cb-7w5gz   Container image "gcr.io/google-samples/gb-frontend:
v4" already present on machine
11m         Normal   Created            pod/frontend-766d4f77cb-7w5gz   Created container php-redis
11m         Normal   Started            pod/frontend-766d4f77cb-7w5gz   Started container php-redis
```

Figure 7.6: Getting the events shows all events from the past hour

Figure 7.6 only shows the event for one pod, but as you can see in your output, the output for this command contains the events for all resources that were recently created, updated, or deleted.

In this section, you have learned about the commands you can use to inspect a Kubernetes application. In the next section, you'll focus on debugging application failures.

Debugging applications

Now that you have a basic understanding of how to inspect applications, you can start seeing how you can debug issues with deployments.

In this section, common errors will be introduced, and you'll determine how to debug and fix them.

If you haven't implemented the Guestbook application already, run the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

After a couple of seconds, the application should be up and running.

Image pull errors

In this section, you are going to introduce image pull errors by setting the image tag value to a non-existent one. An image pull error occurs when Kubernetes cannot download the image for the container it needs to run.

1. Run the following command on Azure Cloud Shell:

   ```
   kubectl edit deployment/frontend
   ```

   Next, change the image tag from `v4` to `v_non_existent` by executing the following steps.

2. Type `/gb-frontend` and hit the *Enter* key to have your cursor brought to the image definition.

Hit the *I* key to go into insert mode. Delete v4 and type `v_non_existent` as shown in *Figure 7.7*:

```
spec:
  containers:
  - env:
    - name: GET_HOSTS_FROM
      value: dns
    image: gcr.io/google-samples/gb-frontend:v_non_existent
    imagePullPolicy: IfNotPresent
    name: php-redis
    ports:
```

Figure 7.7: Changing the image tag from v4 to v_non_existent

3. Now, close the editor by first hitting the *Esc* key, then type `:wq!` and hit *Enter*.
4. Run the following command to list all the pods in the current namespace:

```
kubectl get pods
```

The preceding command should indicate errors, as shown in *Figure 7.8*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods
NAME                              READY   STATUS             RESTARTS   AGE
frontend-69f4b6d547-2xq6n         0/1     ErrImagePull       0          32s
frontend-766d4f77cb-ds6gb         1/1     Running            0          64m
frontend-766d4f77cb-gvbjj         1/1     Running            0          64m
frontend-766d4f77cb-qw8kv         1/1     Running            0          64m
redis-master-f46ff57fd-qs6gk      1/1     Running            0          64m
redis-replica-57c8c66cc4-fddvg    1/1     Running            0          64m
redis-replica-57c8c66cc4-mwtp9    1/1     Running            0          64m
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods
NAME                              READY   STATUS             RESTARTS   AGE
frontend-69f4b6d547-2xq6n         0/1     ImagePullBackOff   0          66s
frontend-766d4f77cb-ds6gb         1/1     Running            0          64m
frontend-766d4f77cb-gvbjj         1/1     Running            0          64m
frontend-766d4f77cb-qw8kv         1/1     Running            0          64m
redis-master-f46ff57fd-qs6gk      1/1     Running            0          64m
redis-replica-57c8c66cc4-fddvg    1/1     Running            0          64m
redis-replica-57c8c66cc4-mwtp9    1/1     Running            0          64m
```

Figure 7.8: One of the pods has the status of either ErrImagePull or ImagePullBackOff

You might see either a status called `ErrImagePull` or `ImagePullBackOff`. Both errors refer to the fact that Kubernetes cannot pull the image from the registry. The `ErrImagePull` error describes just this; `ImagePullBackOff` describes that Kubernetes will back off (wait) before retrying to download the image. This back-off has an exponential delay, going from 10 to 20 to 40 seconds and beyond, up to 5 minutes.

5. Run the following command to get the full error details:

```
kubectl describe pods/<failed pod name>
```

A sample error output is shown in *Figure 7.9*. The key error message is highlighted in red:

```
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  5m3s                 default-scheduler  Successfully assigned default/frontend-69
f4b6d547-2xq6n to aks-agentpool-39838025-vmss000000
  Normal   Pulling    3m33s (x4 over 5m3s)  kubelet           Pulling image "gcr.io/google-samples/gb-f
rontend:v_non_existent"
  Warning  Failed     3m33s (x4 over 5m2s)  kubelet           Failed to pull image "gcr.io/google-sampl
es/gb-frontend:v_non_existent": rpc error: code = NotFound desc = failed to pull and unpack image "gcr.i
o/google-samples/gb-frontend:v_non_existent": failed to resolve reference "gcr.io/google-samples/gb-fron
tend:v_non_existent": gcr.io/google-samples/gb-frontend:v_non_existent: not found
  Warning  Failed     3m33s (x4 over 5m2s)  kubelet           Error: ErrImagePull
  Warning  Failed     3m4s (x7 over 5m2s)   kubelet           Error: ImagePullBackOff
  Normal   BackOff    1s (x20 over 5m2s)    kubelet           Back-off pulling image "gcr.io/google-sam
ples/gb-frontend:v_non_existent"
```

Figure 7.9: Using describe shows more details on the error

The events clearly show that the image does not exist. Errors such as passing invalid credentials to private Docker repositories will also show up here.

6. Let's fix the error by setting the image tag back to `v4`. First, type the following command in Cloud Shell to edit the deployment:

```
kubectl edit deployment/frontend
```

7. Type `/gb-frontend` and hit *Enter* to have your cursor brought to the image definition.
8. Hit the *I* key to go into insert mode. Delete `v_non_existent`, and type `v4`.
9. Now, close the editor by first hitting the *Esc* key, then type `:wq!` and hit *Enter*.
10. This should automatically fix the deployment. You can verify it by getting the events for the pods again.

> **Note**
>
> Because Kubernetes did a rolling update, the front end was continuously available with zero downtime. Kubernetes recognized a problem with the new specification and stopped rolling out additional changes automatically.

Image pull errors can occur when images aren't available or when you don't have access to the container registry. In the next section, you'll explore an error within the application itself.

Application errors

You will now see how to debug an application error. The errors in this section will be self- induced, similar to the last section. The method for debugging the issue is the same as the one we used to debug errors on running applications.

1. To start, get the public IP of the front-end service:

```
kubectl get service
```

2. Connect to the service by pasting its public IP in a browser. Create a couple of entries:

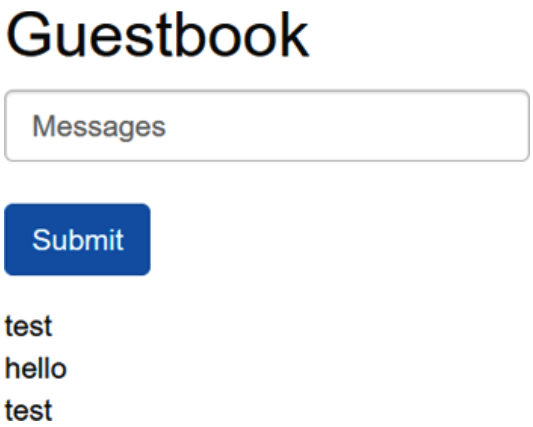


Figure 7.10: Make a couple of entries in the guestbook application

You now have an instance of the guestbook application running. To improve the experience with the example, it's best to scale down the front end so there is only a single replica running.

Scaling down the front end

In *Chapter 3*, *Application deployment on AKS*, you learned how the deployment of the front end has a configuration of `replicas=3`. This means that the requests the application receives can be handled by any of the pods. To introduce the application error and note the errors, you'll need to make changes in all three of them.

But to make this example easier, set `replicas` to `1`, so that you have to make changes to only one pod:

```
kubectl scale --replicas=1 deployment/frontend
```

Having only one replica running will make introducing the error easier. Let's now introduce this error.

Introducing an app error

In this case, you are going to make the **Submit** button fail to work. You will need to modify the application code for this:

> **Note:**
>
> It is not advised to make production changes to your application by using `kubectl exec` to execute commands in your pods. If you need to make changes to your application, the preferred way is to create a new container image and update your deployment.

1. You will use the `kubectl exec` command. This command lets you run commands on the command line of that pod. With the `-it` option, it attaches an interactive terminal to the pod and gives you a shell that you can run commands on. The following command launches a Bash terminal on the pod:

   ```
   kubectl exec -it <frontend-pod-name> -- bash
   ```

This will enter a Bash shell environment as shown in *Figure 7.11*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
frontend-7c8cb4c59f-pcphb         1/1     Running   0          4m27s
redis-master-f46ff57fd-wkhtb      1/1     Running   0          4m27s
redis-replica-57c8c66cc4-qqlqs    1/1     Running   0          4m27s
redis-replica-57c8c66cc4-xrt5v    1/1     Running   0          4m27s
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl exec -it frontend-7c8cb4c59f-pcphb -- sh
# 
```

Figure 7.11: Getting a pod's name and getting access to a shell inside the pod

2. Once you are in the container shell, run the following command:

```
apt update
apt install -y vim
```

   The preceding code installs the `vim` editor so that we can edit the file to introduce an error.

3. Now, use `vim` to open the `guestbook.php` file:

```
vim guestbook.php
```

4. Add the following code at line 17, below the line `if ($_GET['cmd'] == 'set') {`. Remember, to edit a line in `vim`, you hit the *I* key. After you are done editing, you can exit by hitting *Esc*, and then type `:wq!` and press *Enter*:

```
$host = 'localhost';
if(!defined('STDOUT')) define('STDOUT', fopen('php://stdout', 'w'));
fwrite(STDOUT, "hostname at the beginning of 'set' command ");
fwrite(STDOUT, $host);
fwrite(STDOUT, "\n");
```

The file will look like *Figure 7.12*:

```
<?php

error_reporting(E_ALL);
ini_set('display_errors', 1);

require 'Predis/Autoloader.php';

Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
  $host = 'redis-master';
  if (getenv('GET_HOSTS_FROM') == 'env') {
    $host = getenv('REDIS_MASTER_SERVICE_HOST');
  }
  header('Content-Type: application/json');
  if ($_GET['cmd'] == 'set') {
    $host = 'localhost';
    if(!defined('STDOUT')) define('STDOUT', fopen('php://stdout', 'w'));
    fwrite(STDOUT, "hostname at the beginning of 'set' command ");
    fwrite(STDOUT, $host);
    fwrite(STDOUT, "\n");

    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);
```

Figure 7.12: The updated code that introduced an error and additional logging

5. You have now introduced an error where reading messages will work, but not writing them. You have done this by asking the front end to connect to the Redis master at the non-existent localhost server. The writes should fail. At the same time, to make this demo more visual, we added some additional logging to this section of the code.

Open your guestbook application by browsing to its public IP, and you should see the entries from earlier:

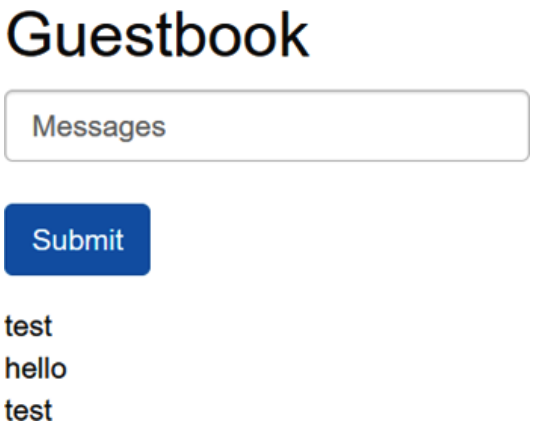


Figure 7.13: The entries from earlier are still present

6. Now, create a new message by typing a message and hitting the **Submit** button:

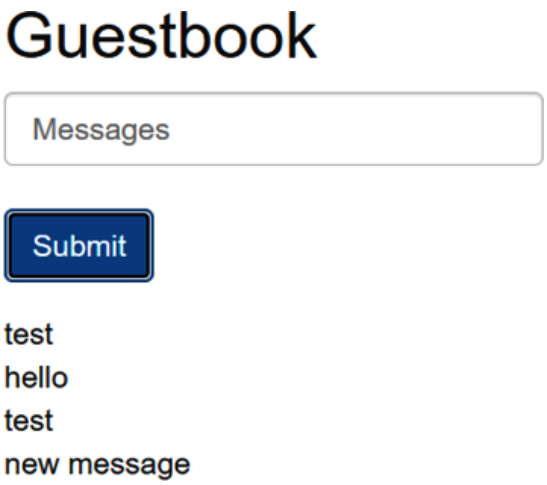


Figure 7.14: A new message was created

Submitting a new message makes it appear in the application. If you did not know any better, you would have thought the entry was written successfully to the database. However, if you refresh your browser, you will see that the message is no longer there.

7. To verify that the message has not been written to the database, hit the **Refresh** button in your browser; you will see just the initial entries, and the new entry has disappeared:

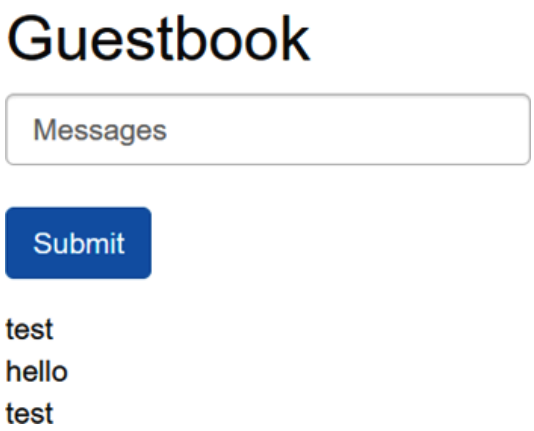


Figure 7.15: The new message has disappeared

As an app developer or operator, you'll probably get a ticket like this: `After the new deployment, new entries are not persisted. Fix it.`

Using logs to identify the root cause

The first step toward resolution is to get the logs.

1. Exit out of the front-end pod for now and get the logs for this pod:

```
exit
kubectl logs <frontend-pod-name>
```

> **Note:**
>
> You can add the `-f` flag after `kubectl logs` to get a live log stream, as follows: `kubectl logs <pod-name> -f`. This is useful during live debugging sessions.

2. You will see entries such as those seen in *Figure 7.16*:

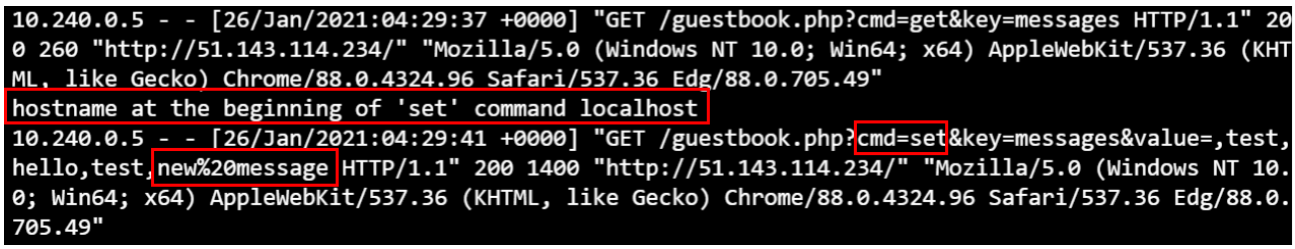


Figure 7.16: The new message shows up as part of the application logs

3. Hence, you know that the error is somewhere when writing to the database in the `set` section of the code. When you see the entry `hostname at the beginning of 'set' command localhost`, you know that the error is between this line and the start of the client, so the setting of `$host = 'localhost'` must be the offending error. This error is not as uncommon as you would think and, as you just saw, could have easily gone through QA unless there had been a specific instruction to refresh the browser. It could have worked perfectly well for the developer, as they could have a running Redis server on the local machine.

Now that you have used logs in Kubernetes to root cause the issue, let's get to resolving the error and getting our application back to a healthy state.

Solving the issue

There are two options to fix this bug you introduced: you can either navigate into the pod and make the code changes, or you can ask Kubernetes to give us a healthy new pod. It is not recommended to make manual changes to pods, so in the next step, you will use the second approach. Let's fix this bug by deleting the faulty pod:

```
kubectl delete pod <podname>
```

As there is a ReplicaSet that controls the pods, you should immediately get a new pod that has started from the correct image. Try to connect to the guestbook again and verify that messages persist across browser refreshes.

The following points summarize what was covered in this section on how to identify an error and how to fix it:

- Errors can come in many shapes and forms.
- Most of the errors encountered by the deployment team are configuration issues.
- Use logs to identify the root cause.
- Using `kubectl exec` on a container is a useful debugging strategy.
- Note that broadly allowing `kubectl exec` is a serious security risk, as it lets the Kubernetes operator execute commands directly in the pods they have access to. Make sure that only a subset of operators has the ability to use the `kubectl exec` command. You can use role-based access control to manage this access restriction, as you'll learn in *Chapter 8*, *Role-based access control in AKS*.
- Anything printed to `stdout` and `stderr` shows up in the logs (independent of the application/language/logging framework).

In this section, you introduced an application error to the guestbook application and leveraged Kubernetes logs to pinpoint the issue in the code. In the next section, you will learn about a powerful mechanism in Kubernetes called **readiness** and **liveness probes**.

Readiness and liveness probes

Readiness and liveness probes were briefly touched upon in the previous section. In this section, you'll explore them in more depth.

Kubernetes uses liveness and readiness probes to monitor the availability of your applications. Each probe serves a different purpose:

- A **liveness probe** monitors the availability of an application while it is running. If a liveness probe fails, Kubernetes will restart your pod. This could be useful to catch deadlocks, infinite loops, or just a "stuck" application.
- A **readiness probe** monitors when your application becomes available. If a readiness probe fails, Kubernetes will not send any traffic to the unready pods. This is useful if your application has to go through some configuration before it becomes available, or if your application has become overloaded but is recovering from the additional load. By having a readiness probe fail, your application will temporarily not get any more traffic, giving it the ability to recover from the increased load.

Liveness and readiness probes don't need to be served from the same endpoint in your application. If you have a smart application, that application could take itself out of rotation (meaning no more traffic is sent to the application) while still being healthy. To achieve this, it would have the readiness probe fail but have the liveness probe remain active.

Let's build this out in an example. You will create two nginx deployments, each with an index page and a health page. The index page will serve as the liveness probe.

Building two web containers

For this example, you'll use a couple of web pages that will be used to connect to a readiness and a liveness probe. The files are present in the code files for this chapter. Let's first create `index1.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Server 1</title>
  </head>
  <body>
    Server 1
  </body>
</html>
```

After that, create `index2.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Server 2</title>
  </head>
  <body>
    Server 2
  </body>
</html>
```

Let's also create a health page, `healthy.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>All is fine here</title>
  </head>
  <body>
    OK
  </body>
</html>
```

In the next step, you'll mount these files to your Kubernetes deployments. To do this, you'll turn each of these into a `configmap` that you will connect to your pods. You've already learned about configmaps in *Chapter 3, Application deployment on AKS*. Use the following commands to create the `configmap`:

```
kubectl create configmap server1 --from-file=index1.html
kubectl create configmap server2 --from-file=index2.html
kubectl create configmap healthy --from-file=healthy.html
```

With that out of the way, you can go ahead and create your two web deployments. Both will be very similar, with just the `configmap` changing. The first deployment file (`webdeploy1.yaml`) looks like this:

```
1   apiVersion: apps/v1
2   kind: Deployment
...
17     spec:
18       containers:
19         - name: nginx-1
20           image: nginx:1.19.6-alpine
21           ports:
22             - containerPort: 80
23           livenessProbe:
24             httpGet:
25               path: /healthy.html
26               port: 80
27             initialDelaySeconds: 3
28             periodSeconds: 3
29           readinessProbe:
30             httpGet:
31               path: /index.html
32               port: 80
33             initialDelaySeconds: 3
34             periodSeconds: 3
35           volumeMounts:
36             - name: html
37               mountPath: /usr/share/nginx/html
38             - name: index
39               mountPath: /tmp/index1.html
40               subPath: index1.html
41             - name: healthy
42               mountPath: /tmp/healthy.html
43               subPath: healthy.html
44           command: ["/bin/sh", "-c"]
45           args: ["cp /tmp/index1.html /usr/share/nginx/html/index.
html; cp /tmp/healthy.html /usr/share/nginx/html/healthy.html; nginx;
sleep inf"]
46       volumes:
47         - name: index
48           configMap:
49             name: server1
50         - name: healthy
51           configMap:
52             name: healthy
53         - name: html
54           emptyDir: {}
```

There are a few things to highlight in this deployment:

- **Lines 23-28**: This is the liveness probe. The liveness probe points to the health page. Remember, if the health page fails, the container will restart.
- **Lines 29-32**: This is the readiness probe. The readiness probe in our case points to the index page. If this page fails, the pod will temporarily not be sent any traffic but will remain running.
- **Lines 44-45**: These two lines contain a couple of commands that get executed when the container starts. Instead of simply running the nginx server, this copies the index and ready files in the right location, then starts nginx, and then uses a `sleep` command (so the container keeps running).

You can create this deployment using the following command. You can also deploy the second version for `server 2`, which is similar to `server 1`:

```
kubectl create -f webdeploy1.yaml
kubectl create -f webdeploy2.yaml
```

Finally, you can also create a service (`webservice.yaml`) that routes traffic to both deployments:

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: web
5   spec:
6     selector:
7       app: web-server
8     ports:
9     - protocol: TCP
10     port: 80
11     targetPort: 80
12   type: LoadBalancer
```

You can create that service using the following:

```
kubectl create -f webservice.yaml
```

You now have the application up and running. In the next section, you'll introduce some failures to verify the behavior of the liveness and readiness probes.

Experimenting with liveness and readiness probes

In the previous section, the functionality of the liveness and readiness probes was explained, and you created a sample application. In this section, you will introduce errors in this application and verify the behavior of the liveness and readiness probes. You will see how a failure of the readiness probe will cause the pod to remain running but no longer accept traffic. After that, you will see how a failure of the liveness probe will cause the pod to be restarted.

Let's start by failing the readiness probe.

Failing the readiness probe causes traffic to temporarily stop

Now that you have a simple application up and running, you can experiment with the behavior of the liveness and readiness probes. To start, let's get the service's external IP to connect to our web server using the browser:

```
kubectl get service
```

If you hit the external IP in the browser, you should see a single line that either says **Server 1** or **Server 2**:

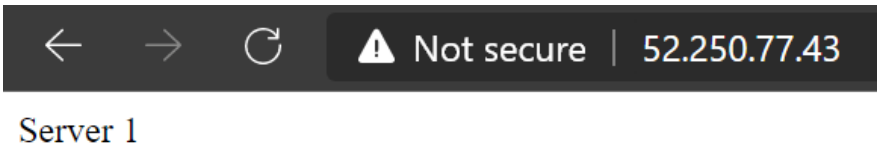


Figure 7.17: Our application is returning traffic from server 1

During the upcoming tests, you'll use a small script called `testWeb.sh` that has been provided in the code samples for this chapter to connect to your web page 50 times, so you can monitor a good distribution of results between servers 1 and 2. You'll first need to make that script executable, and then you can run that script while your deployment is fully healthy:

```
chmod +x testWeb.sh
./testWeb.sh <external-ip>
```

During healthy operations, we can see that server 1 and server 2 are hit almost equally, with 24 hits for server 1 and 26 for server 2:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ ./testWeb.sh 52.250.77.43
server 1:
     24      48     216
server 2:
     26      52     234
```

Figure 7.18: While the application is healthy, traffic is load-balanced between server 1 and server 2

Let's now move ahead and fail the readiness probe in server 1. To do this, you will use the `kubectl exec` command to move the index file to a different location:

```
kubectl get pods #note server1 pod name
kubectl exec <server1 pod name> -- \
  mv /usr/share/nginx/html/index.html \
  /usr/share/nginx/html/index1.html
```

Once this is executed, we can view the change in the pod status with the following command:

```
kubectl get pods -w
```

You should see the readiness state of the server 1 pod change to `0/1`, as shown in *Figure 7.19*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl exec server1-698686949-tffwp -- \
>   mv /usr/share/nginx/html/index.html \
>   /usr/share/nginx/html/index1.html
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods -w
NAME                       READY   STATUS    RESTARTS   AGE
server1-698686949-tffwp    1/1     Running   0          2m30s
server2-6c9f779df7-k7gm7   1/1     Running   0          2m30s
server1-698686949-tffwp    0/1     Running   0          2m33s
```

Figure 7.19: The failing readiness probes causes server 1 to not have any READY containers

This should direct no more traffic to the server 1 pod. Let's verify that:

```
./testWeb.sh <external-ip>
```

Traffic should be redirected to server 2:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ ./testWeb.sh 52.250.77.43
server 1:
      0       0       0
server 2:
     50     100     450
```

Figure 7.20: All traffic is now served by server 2

You can now restore the state of server 1 by moving the file back to its rightful place:

```
kubectl exec <server1 pod name> -- mv \
  /usr/share/nginx/html/index1.html \
  /usr/share/nginx/html/index.html
```

This will return the pod to a **Ready** state and should again split traffic equally:

```
./testWeb.sh <external-ip>
```

This will show an output similar to *Figure 7.21*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ ./testWeb.sh 52.250.77.43
server 1:
     29      58     261
server 2:
     21      42     189
```

Figure 7.21: Restoring the readiness probe causes traffic to be load-balanced again

A failing readiness probe will cause Kubernetes to no longer send traffic to the failing pod. You have verified this by causing a readiness probe in your example application to fail. In the next section, you'll explore the impact of a failing liveness probe.

A failing liveness probe restarts the pod

You can repeat the previous process with the liveness probe as well. When the liveness probe fails, Kubernetes is expected to restart that pod. Let's try this by deleting the health file:

```
kubectl exec <server 2 pod name> -- \
  rm /usr/share/nginx/html/healthy.html
```

Let's see what this does to the pod:

```
kubectl get pods -w
```

You should see that the pod restarts within a couple of seconds:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl exec server2-6c9f779df7-k7gm7 -- \
>   rm /usr/share/nginx/html/healthy.html
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods -w
NAME                       READY   STATUS    RESTARTS   AGE
server1-698686949-tffwp    1/1     Running   0          3m56s
server2-6c9f779df7-k7gm7   1/1     Running   0          3m56s
server2-6c9f779df7-k7gm7   0/1     Running   1          4m30s
server2-6c9f779df7-k7gm7   1/1     Running   1          4m35s
```

Figure 7.22: A failing liveness probe will cause the pod to be restarted

As you can see in *Figure 7.22*, the pod was successfully restarted, with limited impact. You can inspect what was going on in the pod by running a `describe` command:

```
kubectl describe pod <server2 pod name>
```

The preceding command will give you an output similar to *Figure 7.23*:

```
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  5m9s               default-scheduler  Successfully assigned default/server2-6c9f779df7-k7gm7 to aks-agentpool-39838025-vmss000000
  Warning  Unhealthy  69s (x3 over 75s)  kubelet            Liveness probe failed: HTTP probe failed with statuscode: 404
  Normal   Killing    69s                kubelet            Container nginx-2 failed liveness probe, will be restarted
  Normal   Pulled     39s (x2 over 5m8s) kubelet            Container image "nginx:1.19.6-alpine" already present on machine
  Normal   Created    39s (x2 over 5m8s) kubelet            Created container nginx-2
  Normal   Started    39s (x2 over 5m8s) kubelet            Started container nginx-2
```

Figure 7.23: More details on the pod showing how the liveness probe failed

In the `describe` command, you can clearly see that the pod failed the liveness probe. After three failures, the container was killed and restarted.

This concludes the experiment with liveness and readiness probes. Remember that both are useful for your application: a readiness probe can be used to temporarily stop traffic to your pod, so it has to deal with less load. A liveness probe is used to restart your pod if there is an actual failure in the pod.

Let's also make sure to clean up the deployments you just created:

```
kubectl delete deployment server1 server2
kubectl delete service web
```

Liveness and readiness probes are useful to ensure that only healthy pods will receive traffic in your cluster. In the next section, you will explore different metrics reported by Kubernetes that you can use to verify the state of your application.

Metrics reported by Kubernetes

Kubernetes reports multiple metrics. In this section, you'll first use a number of `kubectl` commands to get these metrics. Afterward, you'll look into Azure Monitor for containers to see how Azure helps with container monitoring.

Node status and consumption

The nodes in your Kubernetes are the servers running your application. Kubernetes will schedule pods to different nodes in the cluster. You need to monitor the status of your nodes to ensure that the nodes themselves are healthy and that the nodes have enough resources to run new applications.

Run the following command to get information about the nodes on the cluster:

```
kubectl get nodes
```

The preceding command lists their name, status, and age:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-agentpool-39838025-vmss000000   Ready    agent   3d      v1.19.6
aks-agentpool-39838025-vmss000002   Ready    agent   2d23h   v1.19.6
```

Figure 7.24: There are two nodes in this cluster

You can get more information by passing the `-o wide` option:

```
kubectl get -o wide nodes
```

The output lists the underlying `OS-IMAGE` and `INTERNAL-IP`, and other useful information, which can be viewed in *Figure 7.25*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get -o wide nodes
NAME                                STATUS   ROLES   AGE     VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION     CONTAINER-RUNTIME
aks-agentpool-39838025-vmss000000   Ready    agent   3d      v1.19.6   10.240.0.4    <none>        Ubuntu 18.04.5 LTS   5.4.0-1036-azure   containerd://1.4.3+azure
aks-agentpool-39838025-vmss000002   Ready    agent   2d23h   v1.19.6   10.240.0.5    <none>        Ubuntu 18.04.5 LTS   5.4.0-1036-azure   containerd://1.4.3+azure
```

Figure 7.25: Using -o wide adds more details about the nodes

You can find out which nodes are consuming the most resources by using the following command:

```
kubectl top nodes
```

It shows the CPU and memory usage of the nodes:

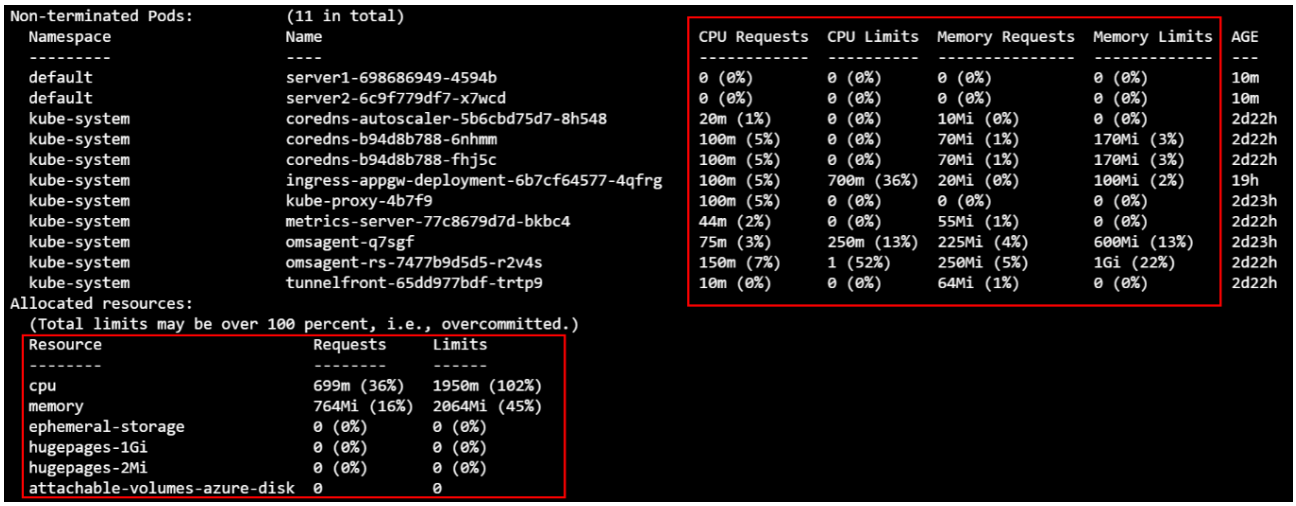
```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl top nodes
NAME                                CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
aks-agentpool-39838025-vmss000000   41m          2%     648Mi           14%
aks-agentpool-39838025-vmss000002   154m         8%     966Mi           21%
```

Figure 7.26: CPU and memory utilization of the nodes

Note that this is the actual consumption at that point in time, not the number of requests a certain node has. To get the requests, you can execute the following:

```
kubectl describe node <node name>
```

This will show you the requests and limits per pod, as well as the cumulative amount for the whole node:



```
Non-terminated Pods:          (11 in total)
  Namespace                   Name                                        CPU Requests  CPU Limits  Memory Requests  Memory Limits  AGE
  ---------                   ----                                        ------------  ----------  ---------------  -------------  ---
  default                     server1-698686949-4594b                     0 (0%)        0 (0%)      0 (0%)           0 (0%)         10m
  default                     server2-6c9f779df7-x7wcd                    0 (0%)        0 (0%)      0 (0%)           0 (0%)         10m
  kube-system                 coredns-autoscaler-5b6cbd75d7-8h548         20m (1%)      0 (0%)      10Mi (0%)        0 (0%)         2d22h
  kube-system                 coredns-b94d8b788-6nhmm                     100m (5%)     0 (0%)      70Mi (1%)        170Mi (3%)     2d22h
  kube-system                 coredns-b94d8b788-fhj5c                     100m (5%)     0 (0%)      70Mi (1%)        170Mi (3%)     2d22h
  kube-system                 ingress-appgw-deployment-6b7cf64577-4qfrg   100m (5%)     700m (36%)  20Mi (0%)        100Mi (2%)     19h
  kube-system                 kube-proxy-4b7f9                            100m (5%)     0 (0%)      0 (0%)           0 (0%)         2d23h
  kube-system                 metrics-server-77c8679d7d-bkbc4             44m (2%)      0 (0%)      55Mi (1%)        0 (0%)         2d22h
  kube-system                 omsagent-q7sgf                              75m (3%)      250m (13%)  225Mi (4%)       600Mi (13%)    2d23h
  kube-system                 omsagent-rs-7477b9d5d5-r2v4s                150m (7%)     1 (52%)     250Mi (5%)       1Gi (22%)      2d22h
  kube-system                 tunnelfront-65dd977bdf-trtp9                10m (0%)      0 (0%)      64Mi (1%)        0 (0%)         2d22h
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource                    Requests     Limits
  --------                    --------     ------
  cpu                         699m (36%)   1950m (102%)
  memory                      764Mi (16%)  2064Mi (45%)
  ephemeral-storage           0 (0%)       0 (0%)
  hugepages-1Gi               0 (0%)       0 (0%)
  hugepages-2Mi               0 (0%)       0 (0%)
  attachable-volumes-azure-disk  0            0
```

Figure 7.27: Describing the nodes shows details on requests and limits

As you can see in *Figure* 7.27, the `describe node` command outputs the requests and limits per pod, across namespaces. This is a good way for cluster operators to verify how much load is being put on the cluster, across all namespaces.

You now know where you can find information about the utilization of your nodes. In the next section, you will look into how you can get the same metrics for individual pods.

Pod consumption

Pods consume CPU and memory resources from an AKS cluster. Requests and limits are used to configure how much CPU and memory a pod can consume. Requests are used to reserve a minimum amount of CPU and memory, while limits are used to set a maximum amount of CPU and memory per pod.

In this section, you will learn how you can use `kubectl` to get information about the CPU and memory utilization of pods.

Let's start by exploring how you can see the requests and limits for a pod that you currently have running:

1. For this example, you will use the pods running in the `kube-system` namespace. Get all the pods in this namespace:

   ```
   kubectl get pods -n kube-system
   ```

   This should show something similar to *Figure 7.28*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get pods -n kube-system
NAME                                    READY   STATUS    RESTARTS   AGE
coredns-autoscaler-5b6cbd75d7-8h548     1/1     Running   1          2d22h
coredns-b94d8b788-6nhmm                 1/1     Running   1          2d22h
coredns-b94d8b788-fhj5c                 1/1     Running   1          2d22h
ingress-appgw-deployment-6b7cf64577-4qfrg   1/1     Running   0          19h
kube-proxy-4b7f9                        1/1     Running   1          2d23h
kube-proxy-x66h9                        1/1     Running   2          3d
metrics-server-77c8679d7d-bkbc4         1/1     Running   1          2d22h
omsagent-n796q                          1/1     Running   2          3d
omsagent-q7sgf                          1/1     Running   1          2d23h
omsagent-rs-7477b9d5d5-r2v4s            1/1     Running   1          2d22h
tunnelfront-65dd977bdf-trtp9            1/1     Running   1          2d22h
```

Figure 7.28: The pods running in the kube-system namespace

2. Let's get the requests and limits for one of the `coredns` pods. This can be done using the `describe` command:

   ```
   kubectl describe pod coredns-<pod id> -n kube-system
   ```

   In the `describe` command, there should be a section similar to *Figure 7.29*:

```
Limits:
  memory:  170Mi
Requests:
  cpu:      100m
  memory:   70Mi
```

Figure 7.29: Limits and requests for the CoreDNS pod

This shows you that this pod has a memory limit of `170Mi`, no CPU limit, and has a request for 100 m CPU (which means 0.1 CPU) and `70Mi` of memory. This means that if this pod were to consume more than 170 MiB of memory, Kubernetes would restart that pod. Kubernetes has also reserved 0.1 CPU core and 70 MiB of memory for this pod.

Requests and limits are used to perform capacity management in a cluster. You can also get the actual CPU and memory consumption of a pod. Run the following command and you'll get the actual pod consumption in all namespaces:

```
kubectl top pods --all-namespaces
```

This should show you anoutput similar to *Figure* 7.30:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl top pods --all-namespaces
NAMESPACE     NAME                                    CPU(cores)   MEMORY(bytes)
default       server1-698686949-4594b                 1m           3Mi
default       server2-6c9f779df7-x7wcd                1m           3Mi
kube-system   coredns-autoscaler-5b6cbd75d7-8h548     1m           7Mi
kube-system   coredns-b94d8b788-6nhmm                 3m           10Mi
kube-system   coredns-b94d8b788-fhj5c                 3m           10Mi
kube-system   ingress-appgw-deployment-6b7cf64577-4qfrg   4m           12Mi
kube-system   kube-proxy-4b7f9                        1m           19Mi
kube-system   kube-proxy-x66h9                        1m           22Mi
kube-system   metrics-server-77c8679d7d-bkbc4         2m           13Mi
kube-system   omsagent-n796q                          8m           142Mi
kube-system   omsagent-q7sgf                          8m           149Mi
kube-system   omsagent-rs-7477b9d5d5-r2v4s            6m           156Mi
kube-system   tunnelfront-65dd977bdf-trtp9            83m          63Mi
```

Figure 7.30: Seeing the CPU and memory consumption of pods

Using the `kubectl top` command shows the CPU and memory consumption at the point in time when the command was run. In this case, you can see that the `coredns` pods are using `3m` CPU and `10Mi` of memory.

In this section, you have used the `kubectl` command to get an insight into the resource utilization of the nodes and pods in your cluster. This is useful information, but it is limited to that specific point in time. In the next section, you'll use the Azure portal to get more detailed information on the cluster and the applications on top of the cluster. You'll start by exploring the **AKS Diagnostics** pane.

Using AKS Diagnostics

When you are experiencing issues in AKS, a good place to start your exploration is the **AKS Diagnostics** pane. It provides you with tools that help investigate any issues related to underlying infrastructure or system cluster components.

> **Note:**
>
> AKS Diagnostics is in preview at the time of writing this book. This means functionality might be added or removed.

To access AKS Diagnostics, hit the **Diagnose and solve problems** option in the AKS menu. This will open up Diagnostics, as shown in *Figure 7.31*:

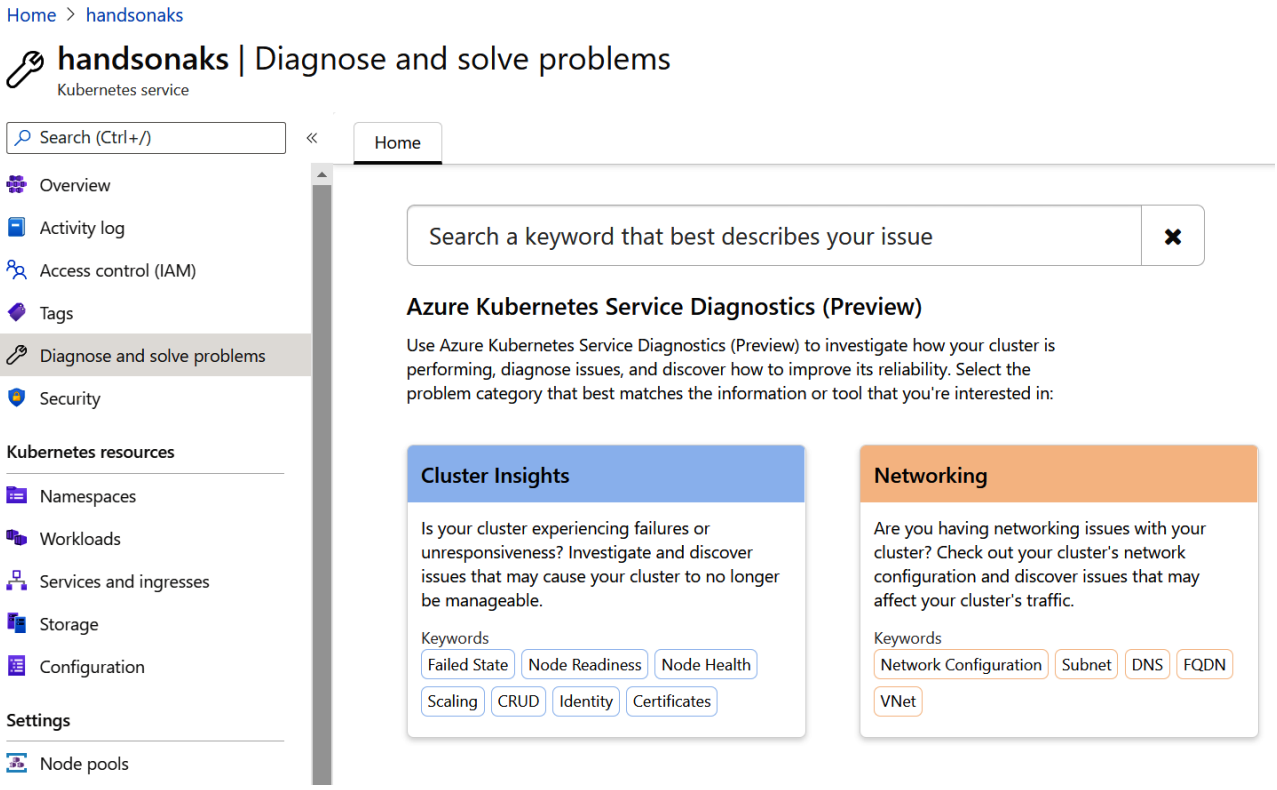


Figure 7.31: Accessing AKS Diagnostics

AKS Diagnostics gives you two tools to diagnose and explore issues. One is **Cluster Insights**, and the other is **Networking**. Cluster Insights uses cluster logs and configuration on your cluster to perform a health check and compare your cluster against best practices. It contains useful information and relevant health indicators in case anything is misconfigured in your cluster. An example output of Cluster Insights is shown in *Figure* 7.32:

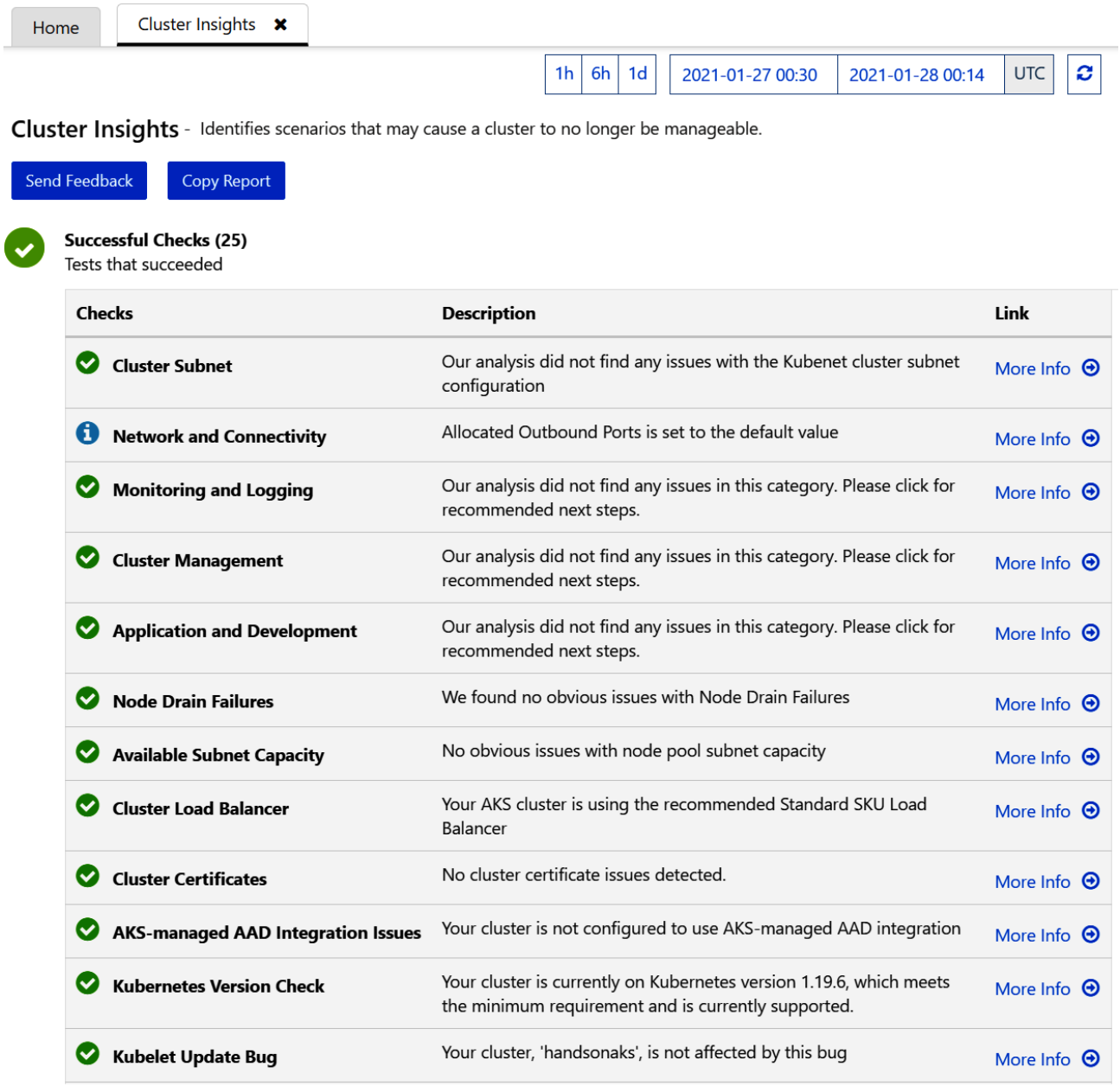


Figure 7.32: Example output from Cluster Insights

The **Networking** section of AKS Diagnostics allows you to interactively troubleshoot networking issues in your cluster. As you open the **Networking** view, you are presented with several questions that will then trigger network health checks and configuration reviews. Once you select one of those options, the interactive tool will give you the output from those checks, as shown in *Figure* 7.33:

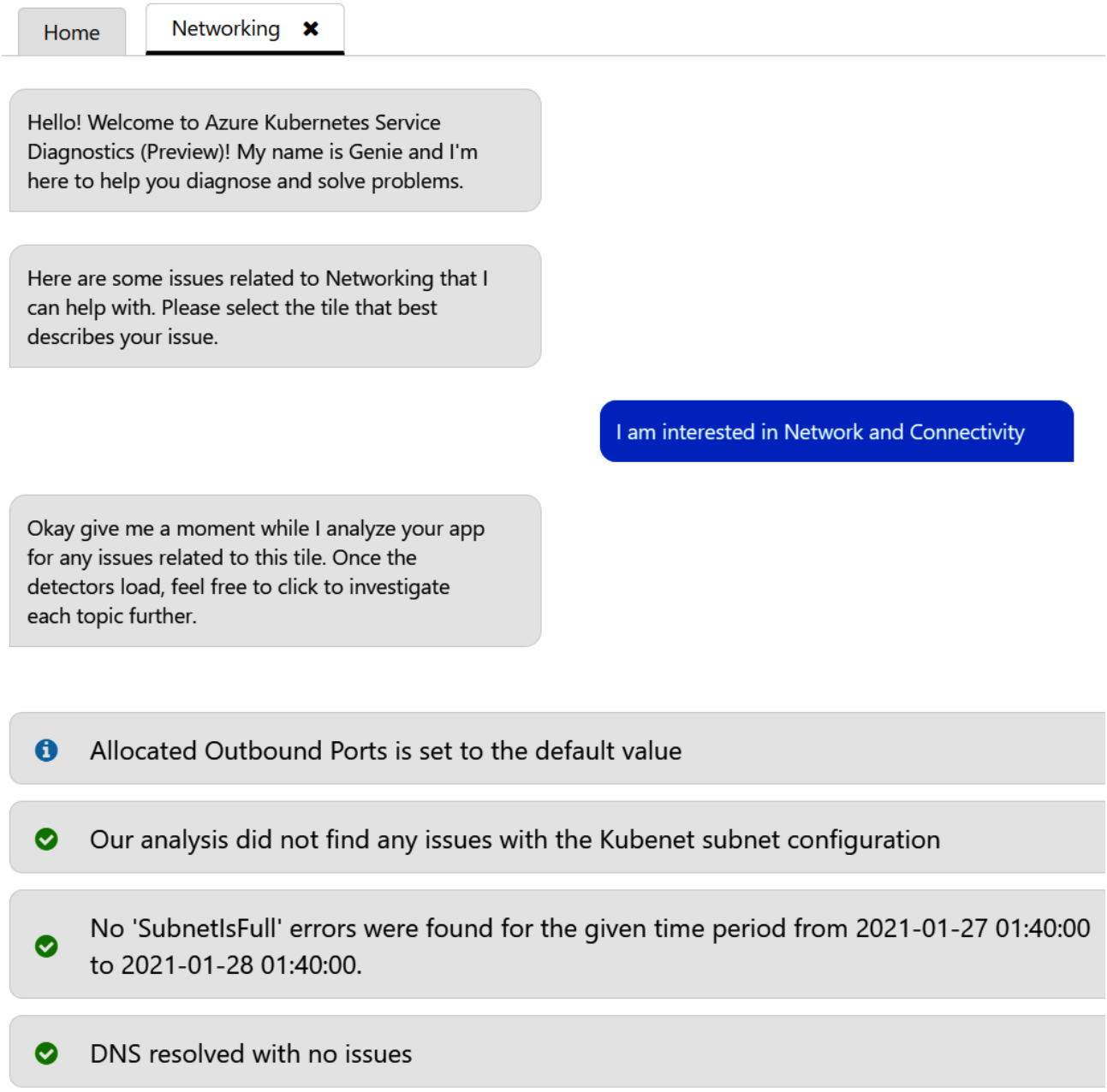


Figure 7.33: Diagnosing networking issues using AKS Diagnostics

Using AKS Diagnostics is very useful when you are facing infrastructure issues on your cluster. The tool does a scan of your environment and verifies whether everything is running and configured well. However, it does not scan your applications. That is where Azure Monitor comes in; it allows you to monitor your application and access your application logs.

Azure Monitor metrics and logs

Previously in this chapter, you explored the status and metrics of nodes and pods in your cluster using the `kubectl` command-line tool. In Azure, you can get more metrics from nodes and pods and explore the logs from pods in your cluster. Let's start by exploring AKS Insights in the Azure portal.

AKS Insights

The **Insights** section of the AKS pane provides most of the metrics you need to know about your cluster. It also has the ability to drill down to the container level. You can also see the logs of the container.

> **Note:**
>
> The Insights section of the AKS pane relies on Azure Monitor for containers. If you created the cluster using the portal defaults, this is enabled by default.

Kubernetes makes metrics available but doesn't store them. Azure Monitor can be used to store these metrics and make them available to query over time. To collect the relevant metrics and logs into Insights, Azure connects to the Kubernetes API to collect the metrics and logs to then store them in Azure Monitor.

> **Note:**
>
> Logs of a container could contain sensitive information. Therefore, the rights to review logs should be controlled and audited.

Let's explore the **Insights** tab of the AKS pane, starting with the cluster metrics.

Cluster metrics

Insights shows the cluster metrics. *Figure* 7.34 shows the CPU utilization and the memory utilization of all the nodes in the cluster. You can optionally add additional filters to filter to a particular namespace, node, or node pool. There also is a live option, which gives you more real-time information on your cluster status:

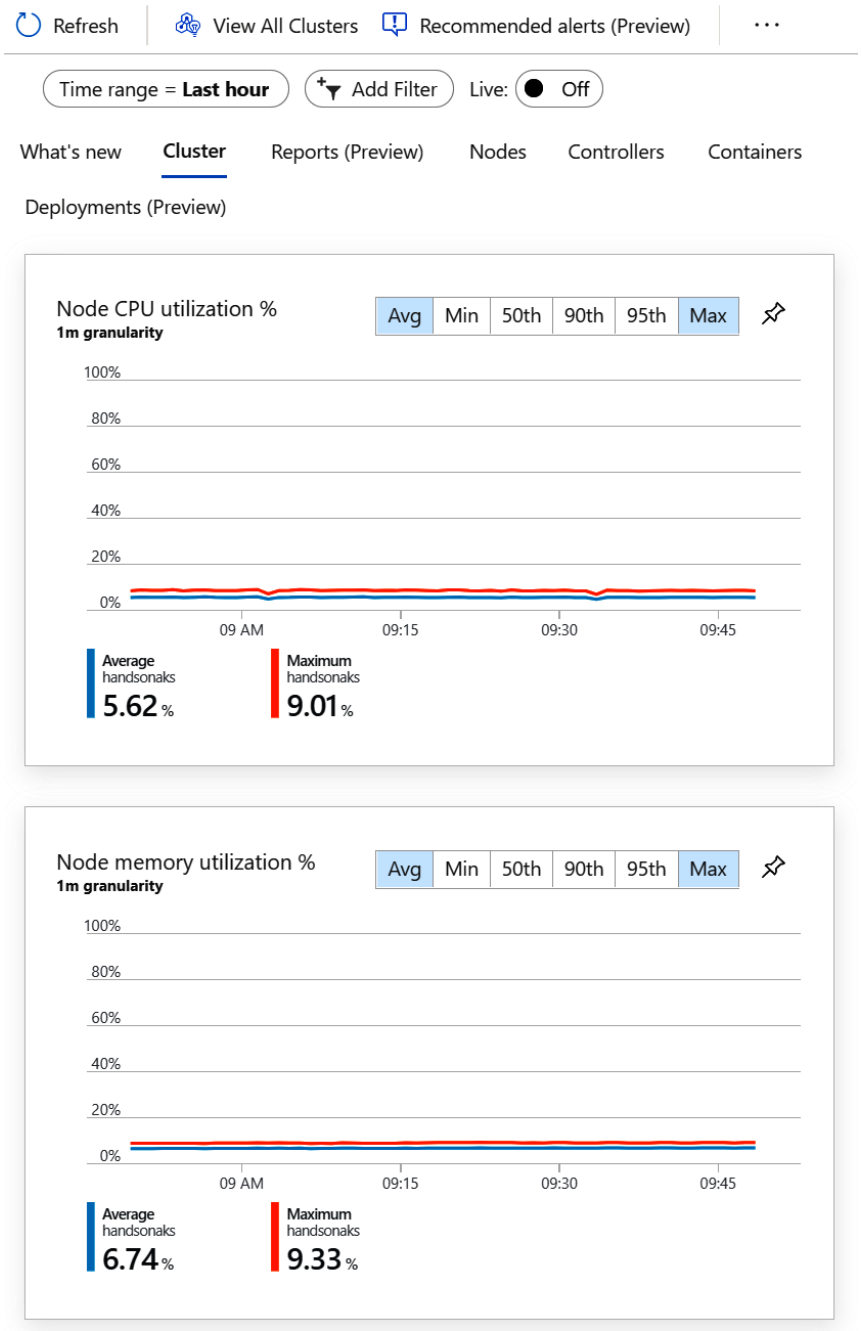


Figure 7.34: The Cluster tab shows CPU and memory utilization for the cluster

The cluster metrics also show the node count and the number of active pods. The node count is important, as you can track whether you have any nodes that are in a **Not Ready** state:

Figure 7.35: The Cluster tab shows the node count and the number of active pods

The **Cluster** tab can be used to monitor the status of the nodes in the cluster. Next, you'll explore the **Reports** tab.

Reports

The **Reports** tab in AKS Insights gives you access to a number of preconfigured monitoring workbooks. These workbooks combine text, log queries, metrics, and parameters together and give you rich interactive reports. You can drill down into each individual report to get more information and prebuilt log queries. The available reports are shown in *Figure* 7.36:

> **Note**
>
> The Reports functionality is in preview at the time of writing this book.

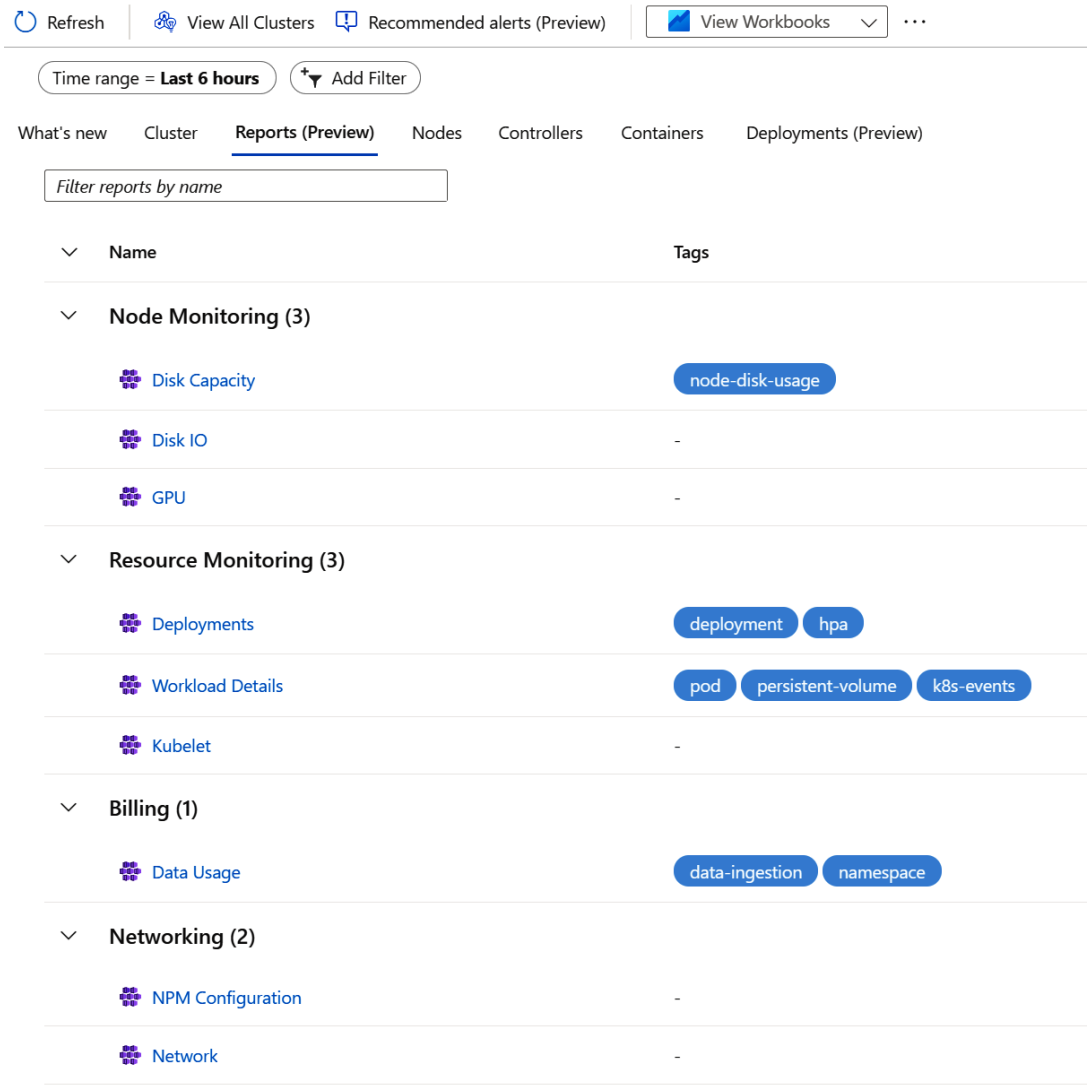


Figure 7.36: The Reports tab gives you access to preconfigured monitoring workbooks

As an example, you can explore the **Deployments** workbook. This is shown in *Figure 7.37*:

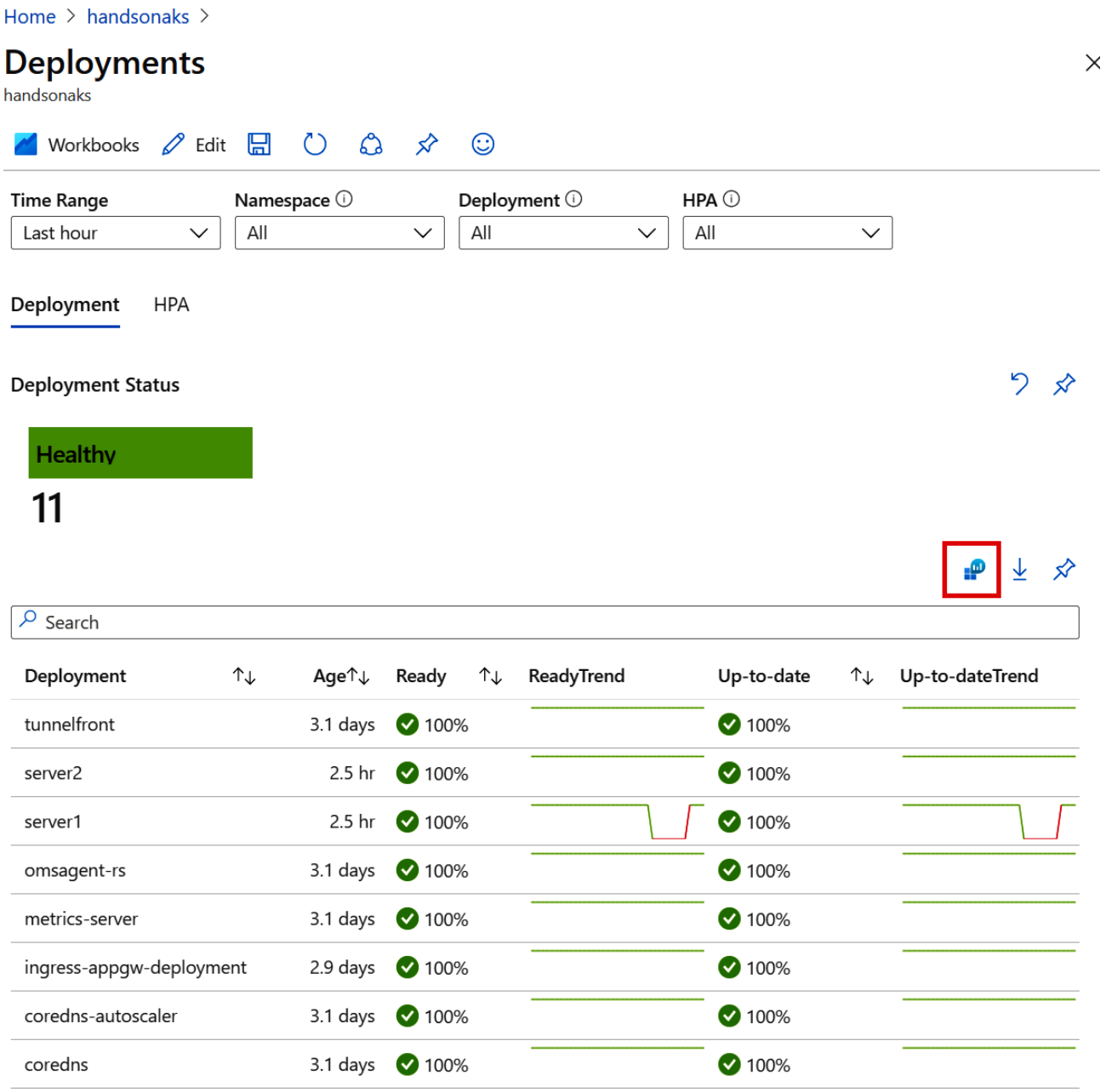


Figure 7.37: The Deployments workbook shows you the status of your deployments

This shows you all the deployments by default, their health, and up-to-date status. As you can see, it shows you that **server1** was temporarily unavailable when you were doing the exploration with liveness and readiness probes earlier in this chapter.

You can drill down further into the status of the individual deployments. If you click on the **Log** button highlighted in *Figure* 7.37, you get redirected to Log Analytics with a prebuilt query. You can then modify this query and get deeper insights into your workload, as shown in *Figure* 7.38.

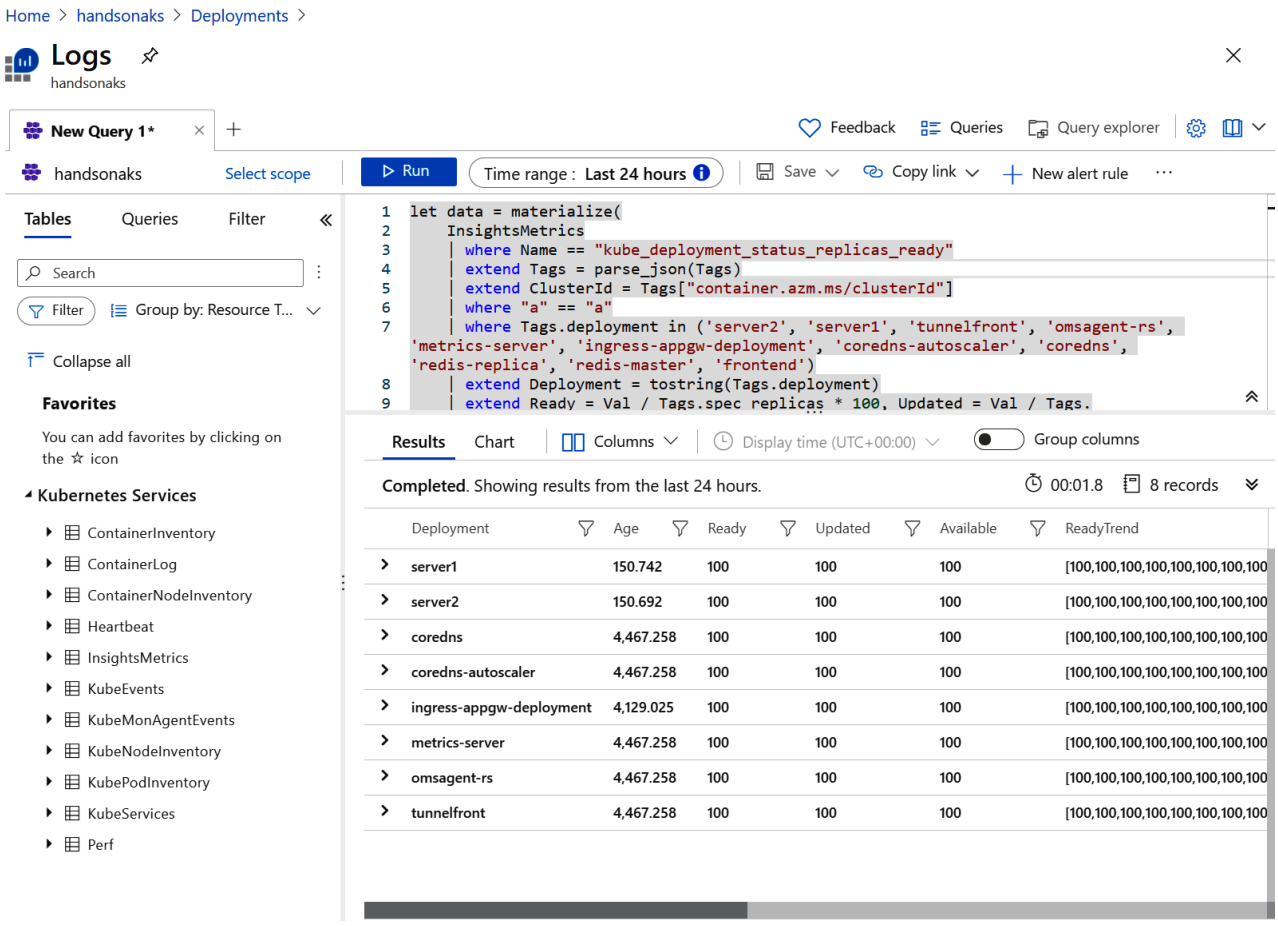


Figure 7.38: Drilling down in Log Analytics to get more details on your deployments

> ## Note:
>
> The queries used in Log Analytics make use of the **Kusto Query Language** (**KQL**). To learn more about KQL, please refer to the documentation: https://docs.microsoft.com/azure/data-explorer/kusto/concepts/

The **Reports** tab in AKS Insights gives you a number of prebuilt monitoring workbooks. The next tab is the **Nodes** tab.

Nodes

The **Nodes** view shows you detailed metrics for your nodes. It also shows you which pods are running on each node, as you can see in *Figure* 7.39:

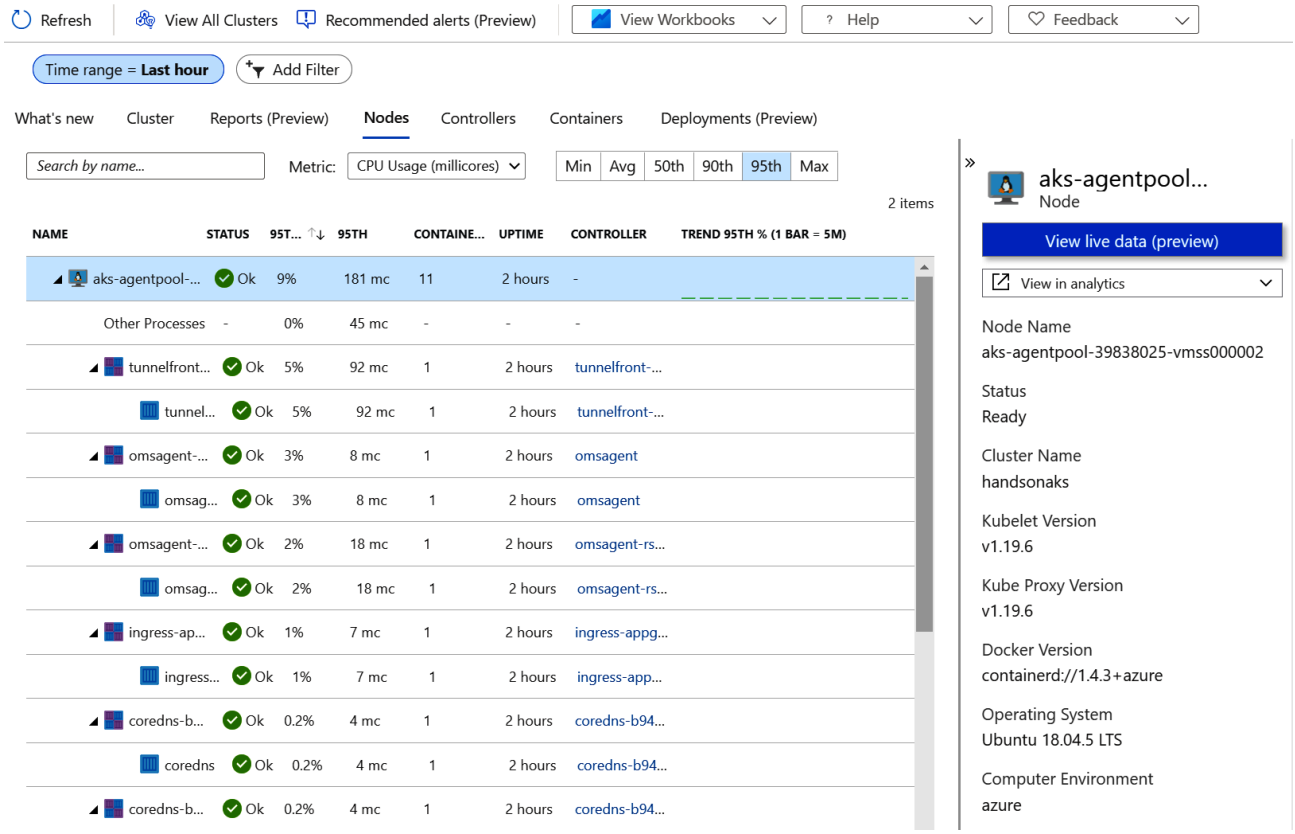


Figure 7.39: Detailed metrics of the nodes in the Nodes pane

Note that different metrics can be viewed from the dropdown menu right next to the search bar. If you need even more details, you can click through and get Kubernetes event logs from your nodes as well:

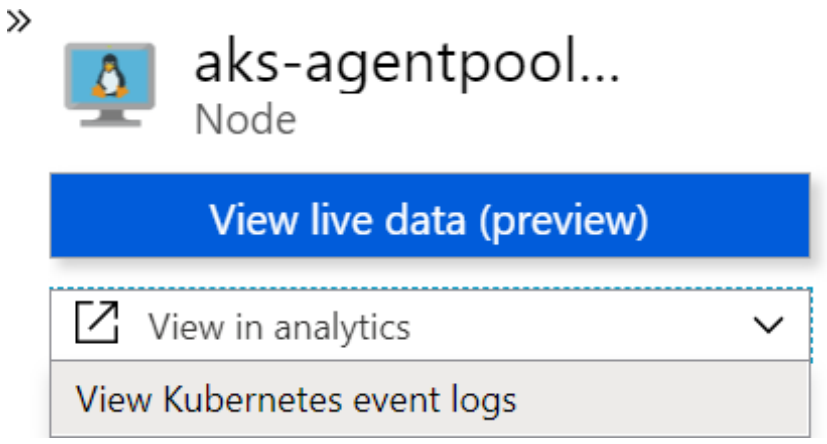


Figure 7.40: Click on View Kubernetes event logs to get the logs from a cluster

This will open Azure Log Analytics and will have pre-created a query for you that shows the logs for your node. In the example in *Figure 7.41*, you can see that the node was rebooted a couple of times and hit an `InvalidDiskCapacity` warning as well:

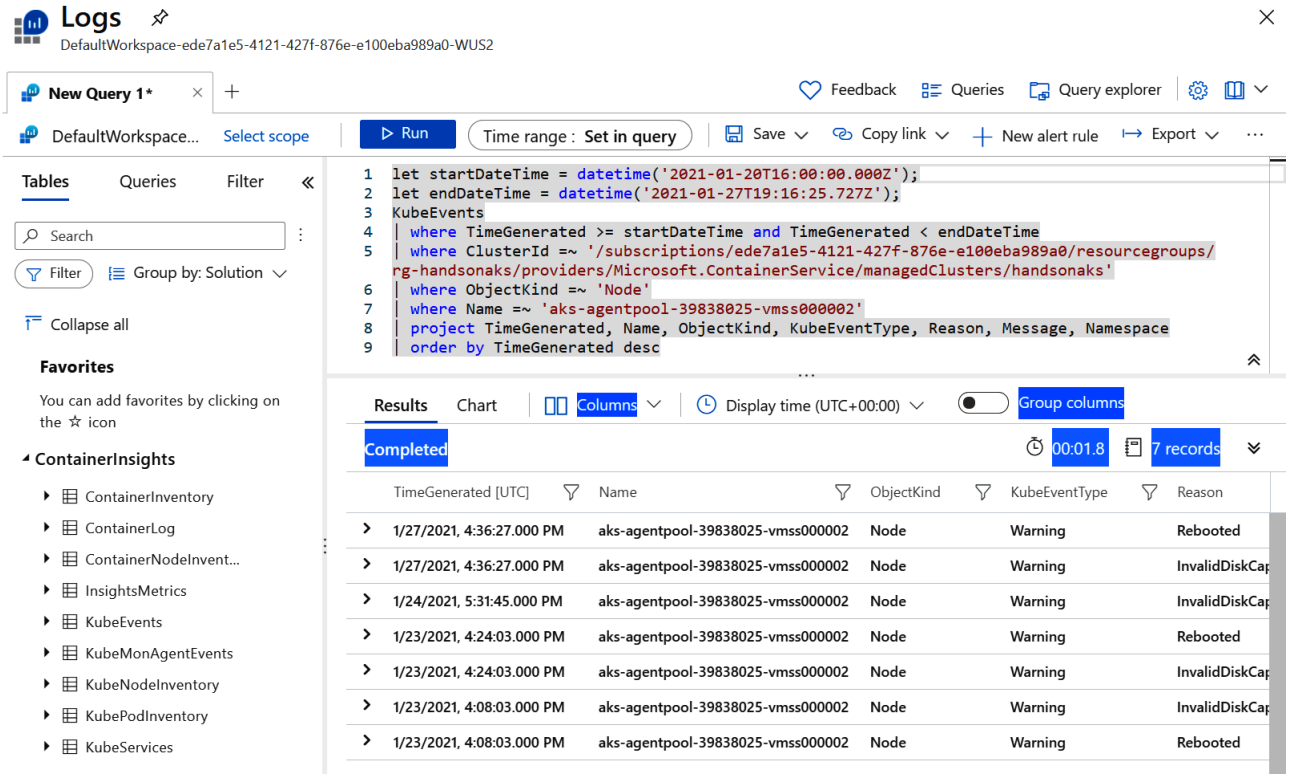


Figure 7.41: Log Analytics showing the logs for the nodes

This gives you information about the status of your nodes. Next, you'll explore the **Controllers** tab.

Controllers

The **Controllers** tab shows you details on all the controllers (that is, ReplicaSets, DaemonSets, and so on) on your cluster and the pods running in them. This shows you a controller-centric view of running pods. For instance, you can find the **server1** ReplicaSet and see all the pods and containers running in it, as shown in *Figure 7.42*:

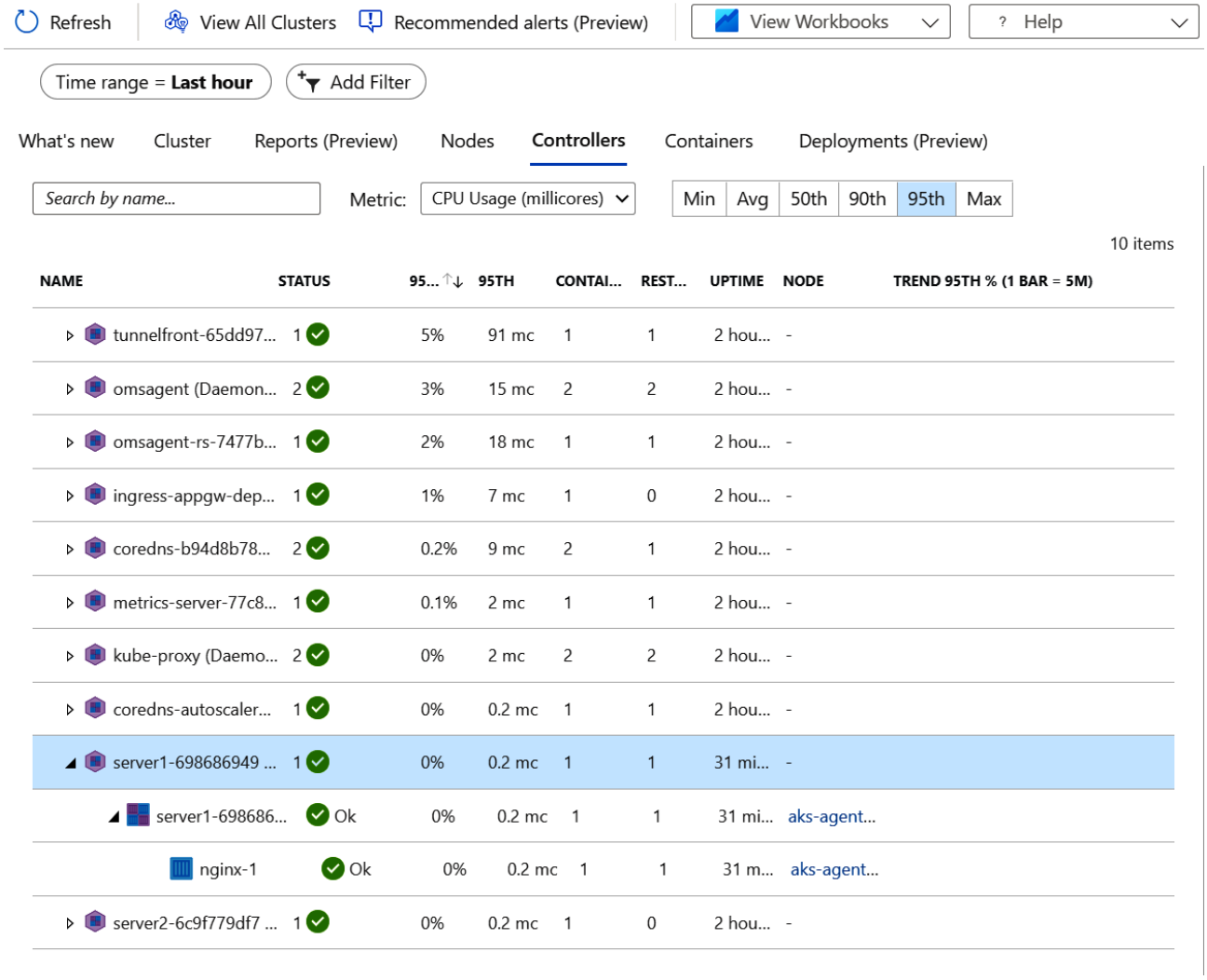


Figure 7.42: The Controllers tab shows you all the pods running in a ReplicaSet

The next tab is the **Containers** tab, which will show you the metrics, logs, and environment variables for a container.

Container metrics, logs, and environment variables

Clicking on the **Containers** tab lists the container metrics, environment variables, and access to its logs, as shown in *Figure 7.43*:

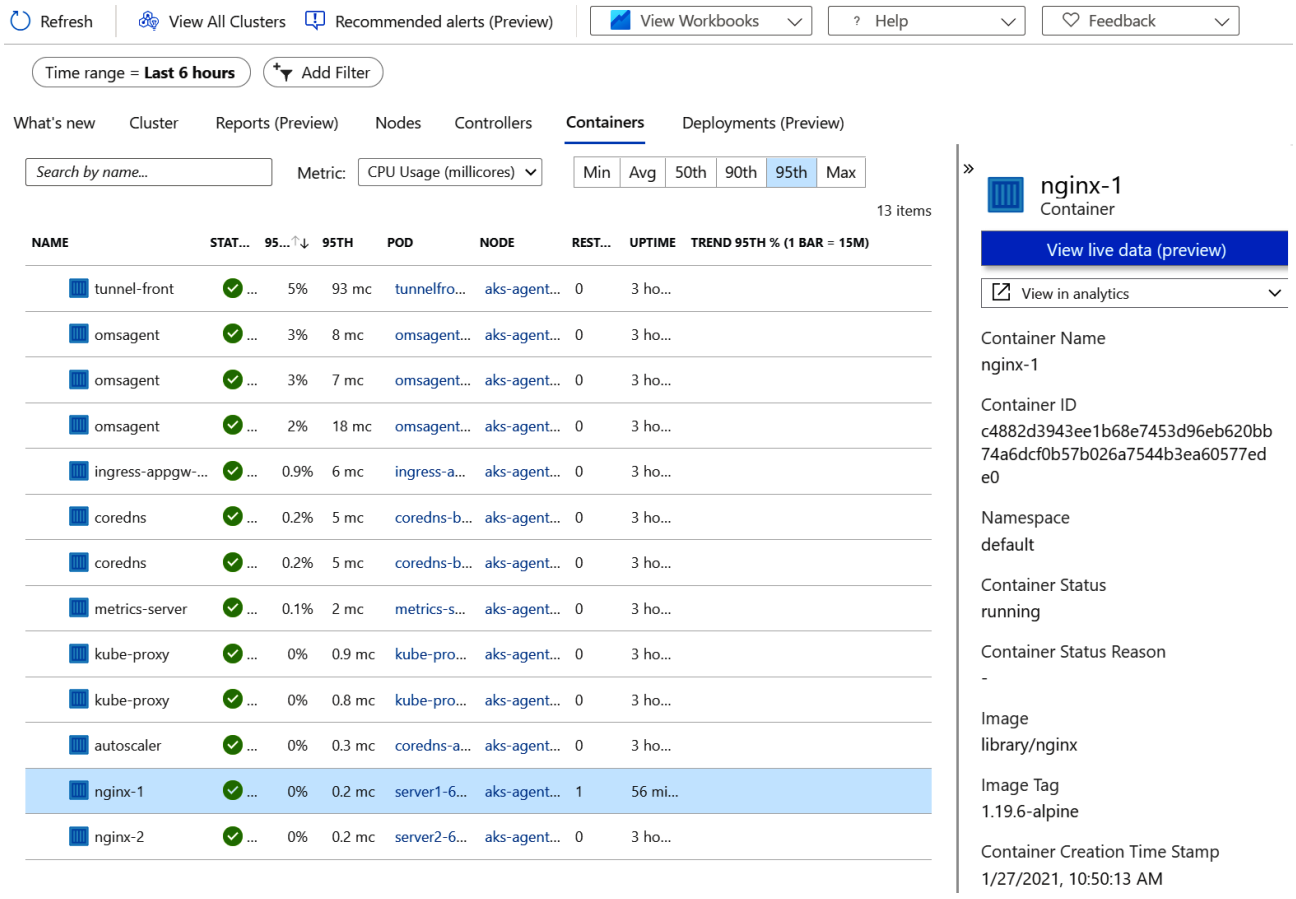


Figure 7.43: The Containers tab shows us all the individual containers

> **Note:**
>
> You might notice a couple of containers with an `Unknown` state. If a container in the **Insights** pane has an unknown status, that is because Azure Monitor has logs and information about that container, but the container is no longer running on the cluster.

You can get access to the container's logs from this view as well:

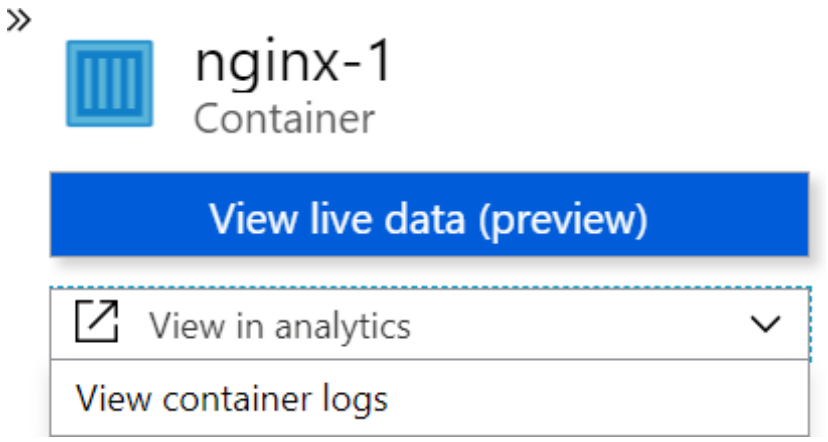


Figure 7.44: Access the container's logs

This will show you all the logs that Kubernetes logged from your application. Earlier in the chapter, you used `kubectl` to get access to container logs. Using this approach can be a lot more productive, as you can edit the log queries and correlate logs from different pods and applications in a single view:

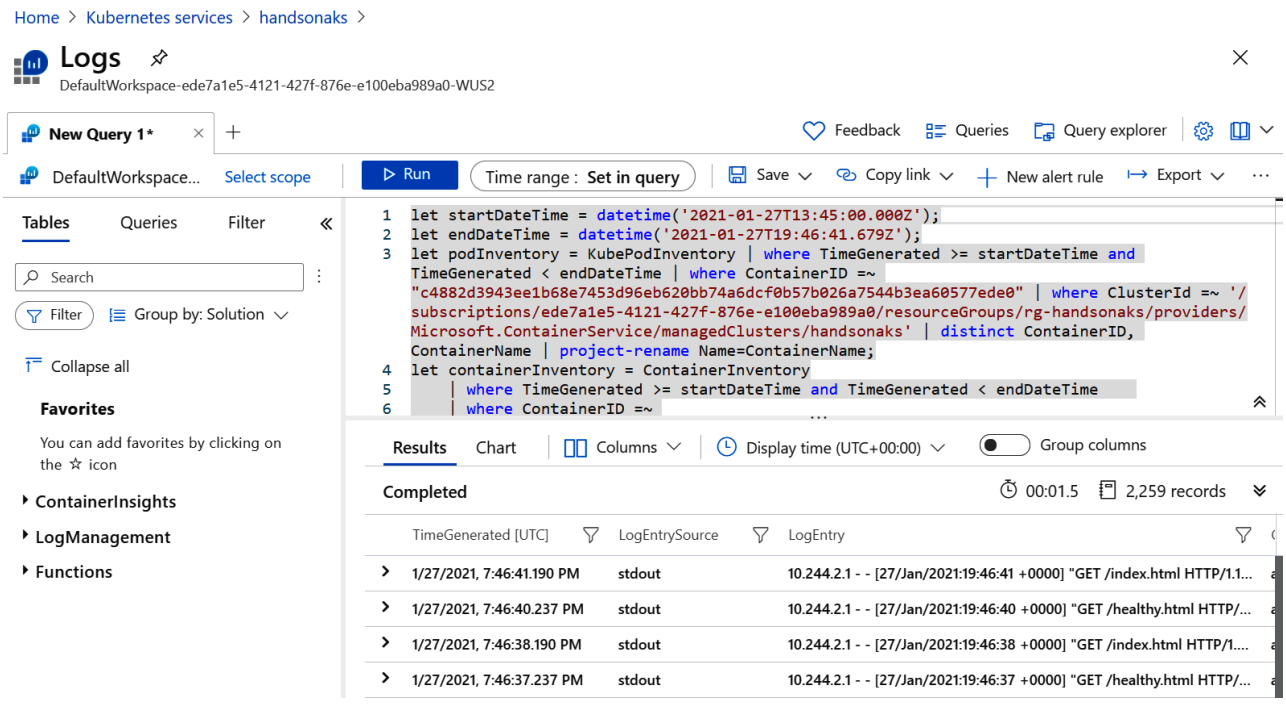


Figure 7.45: Logs are collected and can be queried

Apart from the logs, this view also shows the environment variables that are set for the container. To see the environment variables, scroll down in the right cell of the **Containers** view:

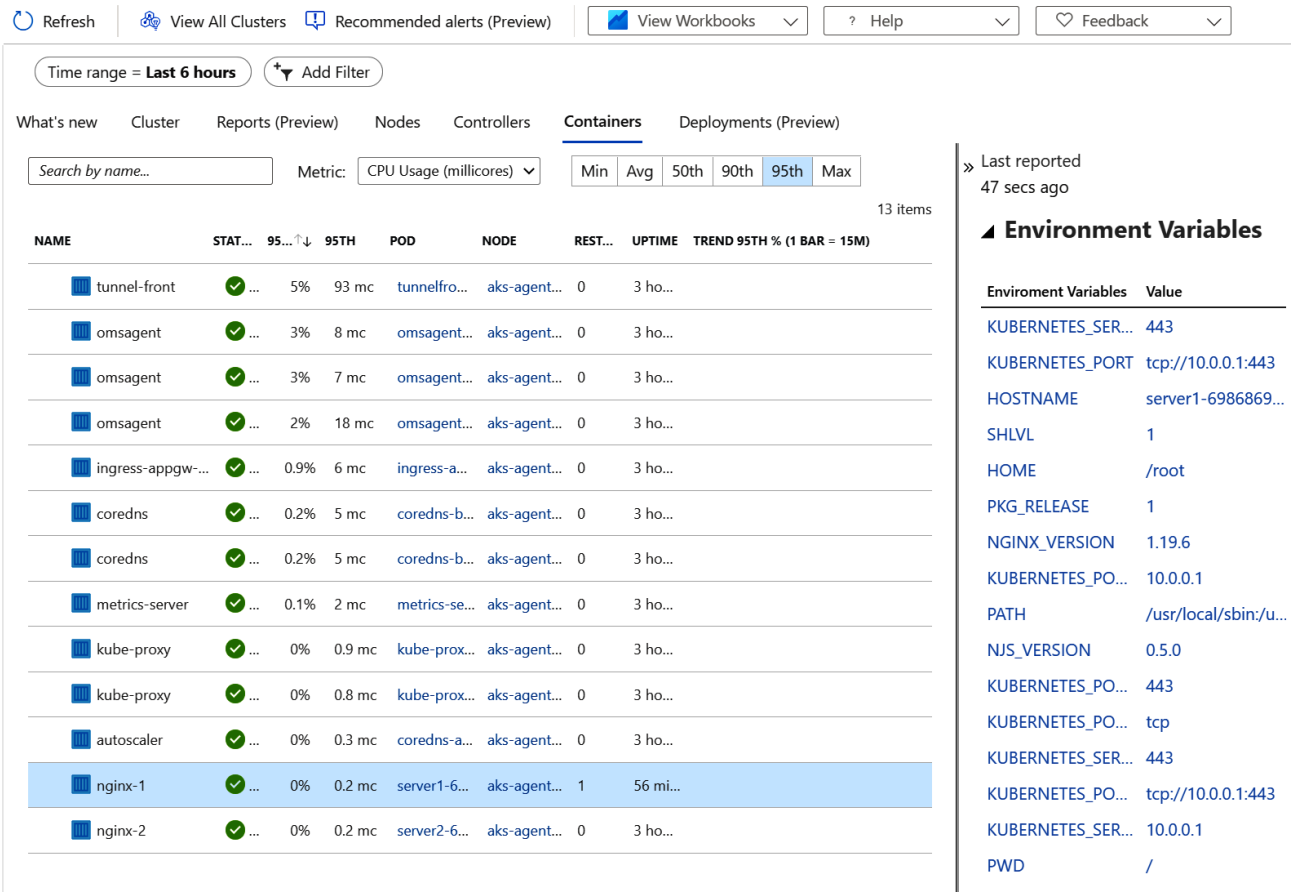


Figure 7.46: The environment variables set for the container

The final tab in AKS Insights is the **Deployments** tab, which you'll explore next.

Deployments

The final tab is the **Deployments** tab. This tab gives you an overview of all deployments in the cluster and allows you to get the definition of the deployment by selecting it. As you can see in *Figure 7.47*, you can get this view either in **Describe** (in text format) or in **RAW** (YAML format):

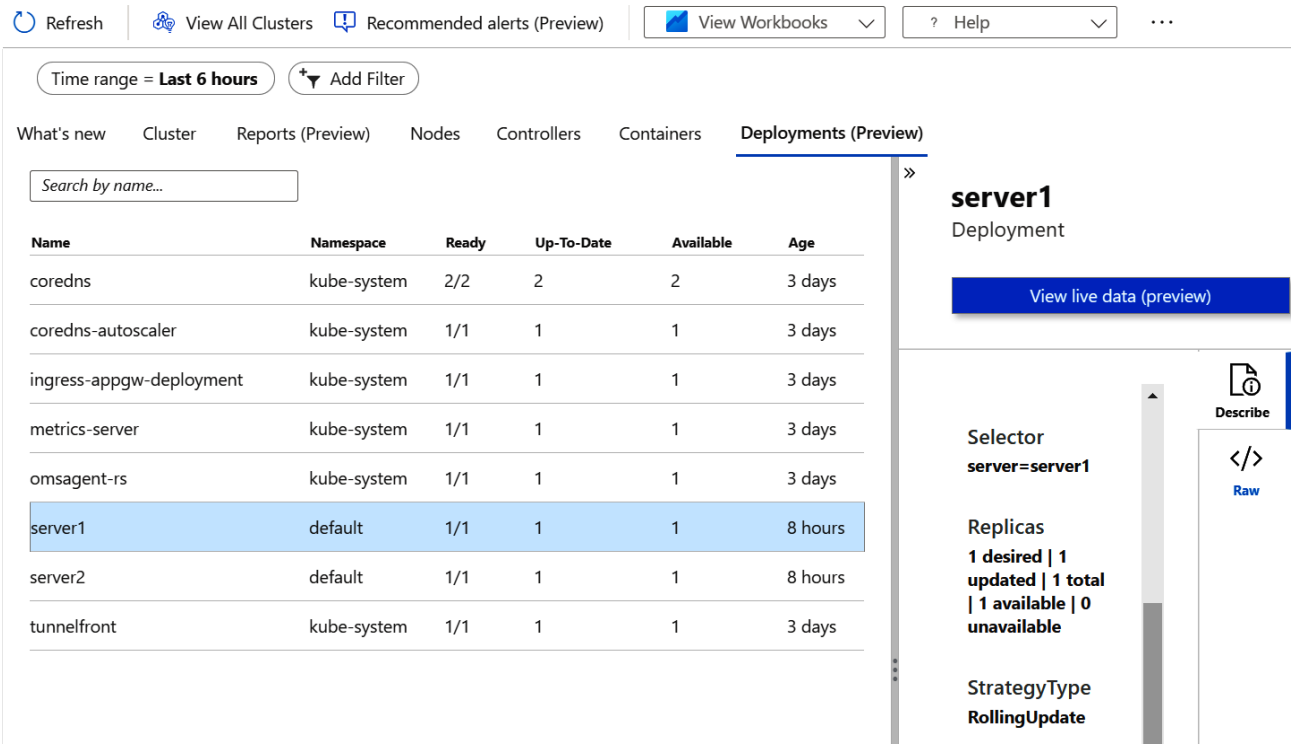


Figure 7.47: The Deployments tab in AKS Insights

By using the **Insights** pane in AKS, you can get detailed information about your cluster. You explored the different tabs in this section and learned how you can drill down and get access to customizable log queries to get even more information.

And that concludes this section. Let's make sure to clean up all the resources created in this chapter by using the following command:

```
kubectl delete -f
```

In this section, you explored monitoring applications running on top of Kubernetes. You used the AKS **Insights** tab in the Azure portal to get a detailed view of your cluster and the containers running on the cluster.

Summary

You started this chapter by learning how to use different `kubectl` commands to monitor an application. Then, you explored how logs created in Kubernetes can be used to debug that application. The logs contain all the information that is written to `stdout` and `stderr`.

After that, you switched to the Azure portal and started using AKS Diagnostics to explore infrastructure issues. Lastly, you explored the use of Azure Monitor and AKS Insights to show the AKS metrics and environment variables, as well as logs with log filtering.

In the next chapter, you will learn how to connect an AKS cluster to Azure PaaS services. You will specifically focus on how you can connect an AKS cluster to a MySQL database managed by Azure.

Section 3: Securing your AKS cluster and workloads

Loose lips sink ships is a phrase that describes how easy it can be to jeopardize the security of a Kubernetes-managed cluster (*Kubernetes*, by the way, is Greek for *helmsman*, as in the helmsman of a *ship*). If your cluster is left open with the wrong ports or services exposed, or plain text is used for secrets in application definitions, bad actors can take advantage of this negligent security and do pretty much whatever they want in your cluster.

There are multiple items to consider when securing an **Azure Kubernetes Service** (**AKS**) cluster and workloads running on top of it. In this section, you will learn about four ways to secure your cluster and applications. You will learn about role-based access control in Kubernetes and how this can be integrated with **Azure Active Directory** (**Azure AD**). After that, you'll learn how to allow your pods to get access to Azure resources such as Blob Storage or Key Vault using an Azure AD pod identity. Subsequently, you'll learn about Kubernetes secrets and how to safely integrate them with Key Vault. Finally, you'll learn about network security and how to isolate your Kubernetes cluster.

In this chapter, you will be routinely deleting clusters and creating new clusters with new functionalities enabled. The reason you will delete existing clusters is to save costs and optimize the free trial, if you are using it.

This section contains the following chapters:

- *Chapter 8*, *Role-based access control in AKS*
- *Chapter 9*, *Azure Active Directory pod-managed identities in AKS*
- *Chapter 10*, *Storing secrets in AKS*
- *Chapter 11*, *Network security in AKS*

You will start this section with *Chapter 8*, *Role-based access control in AKS*, in which you will configure role-based access control in Kubernetes and integrate this with Azure AD.

8

Role-based access control in AKS

Up to this point, you've been using a form of access to **Azure Kubernetes Service** (**AKS**) that gave you permissions to create, read, update, and delete all objects in your cluster. This has worked great for testing and development but is not recommended on production clusters. On production clusters, the recommendation is to leverage **role-based access control** (**RBAC**) in Kubernetes to only grant a limited set of permissions to users.

In this chapter, you will explore Kubernetes RBAC in more depth. You will be introduced to the concept of RBAC in Kubernetes. You will then configure RBAC in Kubernetes and integrate it with **Azure Active Directory** (**Azure AD**).

The following topics will be covered in this chapter:

- RBAC in Kubernetes
- Enabling Azure AD integration in your AKS cluster
- Creating a user and a group in Azure AD
- Configuring RBAC in AKS
- Verifying RBAC for a user

> **Note**
>
> To complete the example on RBAC, you need access to an Azure AD instance, with global administrator permissions.

Let's start this chapter by explaining RBAC.

RBAC in Kubernetes explained

In production systems, you need to allow different users different levels of access to certain resources; this is known as **RBAC**. The benefit of establishing RBAC is that it not only acts as a guardrail against the accidental deletion of critical resources but also is an important security feature that limits full access to the cluster to roles that really need it. On an RBAC-enabled cluster, users can only access and modify those resources for which they have permission.

Up until now, using Cloud Shell, you have been acting as *root*, which allowed you to do anything and everything in the cluster. For production use cases, root access is dangerous and should be restricted as much as possible. It is a generally accepted best practice to use the **principle of least privilege** (**PoLP**) to sign in to any computer system. This prevents both access to secure data and unintentional downtime through the deletion of key resources. Anywhere between 22% and 29% of data loss is attributed to human error. You don't want to be a part of that statistic.

Kubernetes developers realized this was a problem and added RBAC to Kubernetes along with the concept of service roles to control access to clusters. Kubernetes RBAC has three important concepts:

- **Role**: A role contains a set of permissions. A role defaults to `no permissions`, and every permission needs to be specifically called out. Examples of permissions include `get`, `watch`, and `list`. The role also contains which resources these permissions are given to. Resources can be either all pods, deployments, and so on, or can be a specific object (such as `pod/mypod`).

- **Subject**: The subject is either a person or a service account that is assigned a role. In AKS clusters integrated with Azure AD, these subjects can be Azure AD users or groups.
- **RoleBinding**: A RoleBinding links a subject to a role in a certain namespace or, in the case of a ClusterRoleBinding, the whole cluster.

An important concept to understand is that when interfacing with AKS, there are two layers of RBAC: Azure RBAC and Kubernetes RBAC, as shown in *Figure 8.1*. Azure RBAC deals with the roles given to people to make changes in Azure, such as creating, modifying, and deleting clusters. Kubernetes RBAC deals with the access rights to resources in a cluster. Both are independent control planes but can use the same users and groups originating in Azure AD.

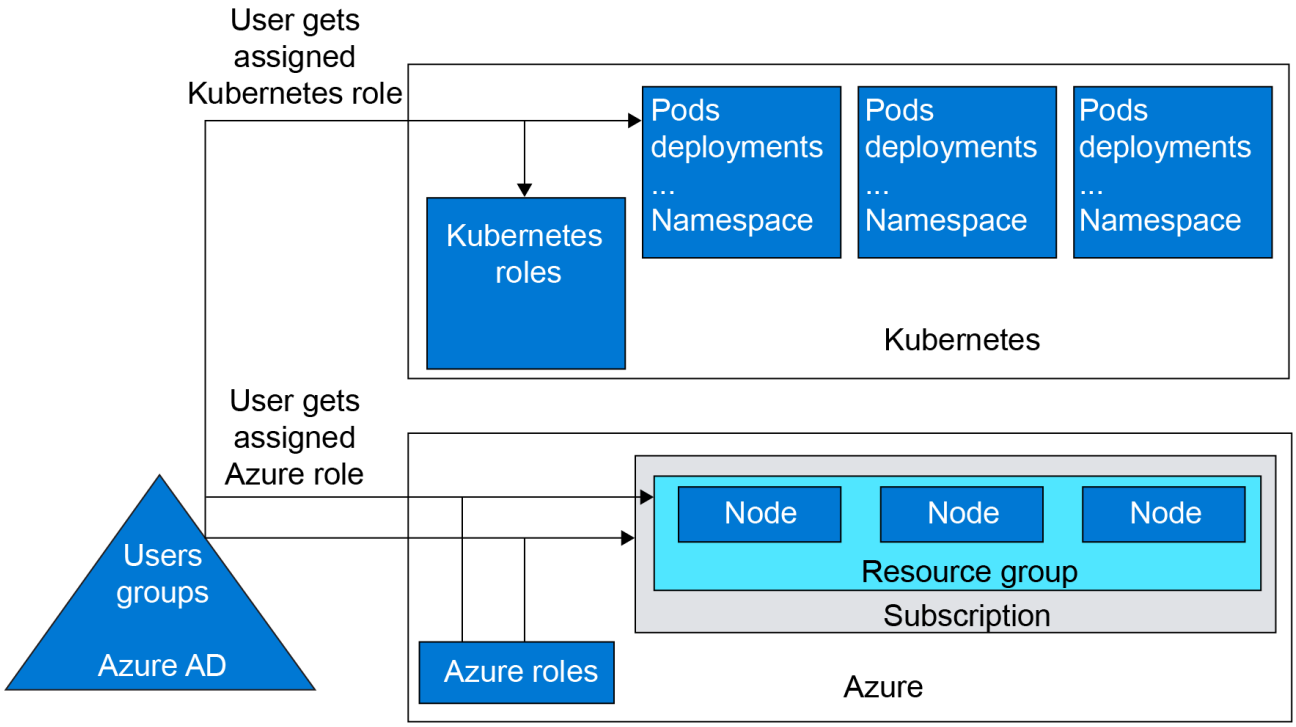


Figure 8.1: Two different RBAC planes, Azure and Kubernetes

RBAC in Kubernetes is an optional feature. The default in AKS is to create clusters that have RBAC enabled. However, by default, the cluster is not integrated with Azure AD. This means that by default you cannot grant Kubernetes permissions to Azure AD users. In the coming section, you will enable Azure AD integration in your cluster.

Enabling Azure AD integration in your AKS cluster

In this section, you will update your existing cluster to include Azure AD integration. You will do this using the Azure portal:

> **Note**
>
> Once a cluster has been integrated with Azure AD, this functionality cannot be disabled.

1. To start, you will need an Azure AD group. You will later give admin privileges for your AKS cluster to this group. To create this group, search for `azure active directory` in the Azure search bar:

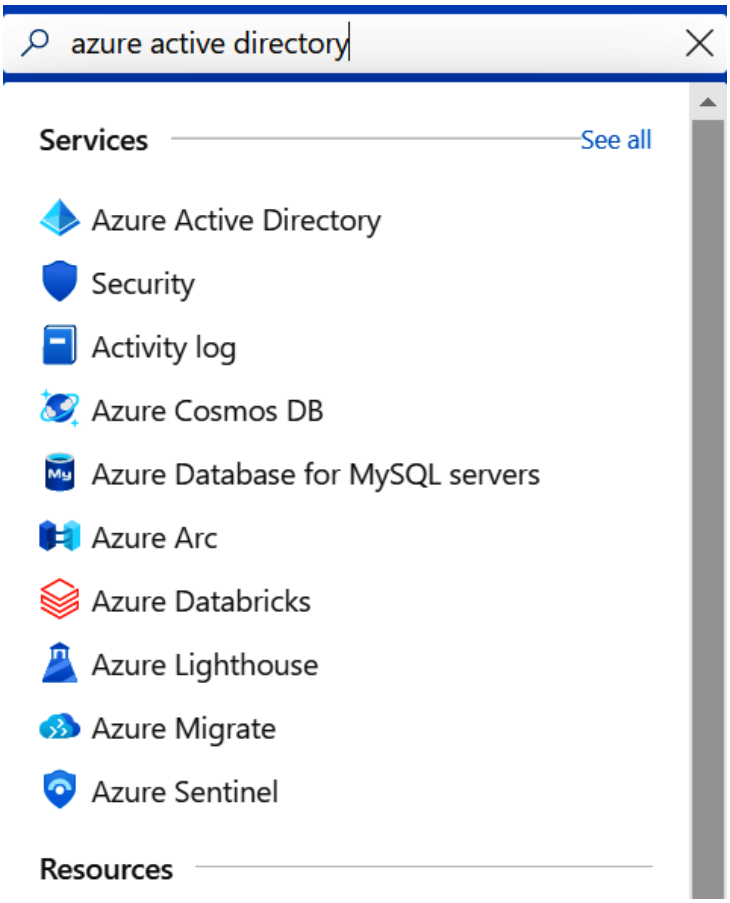


Figure 8.2: Searching for azure active directory in the Azure search bar

2. In the left pane, select **Groups**, which will bring you to the **All groups** screen. Click **+ New Group**, as shown in *Figure 8.3*:

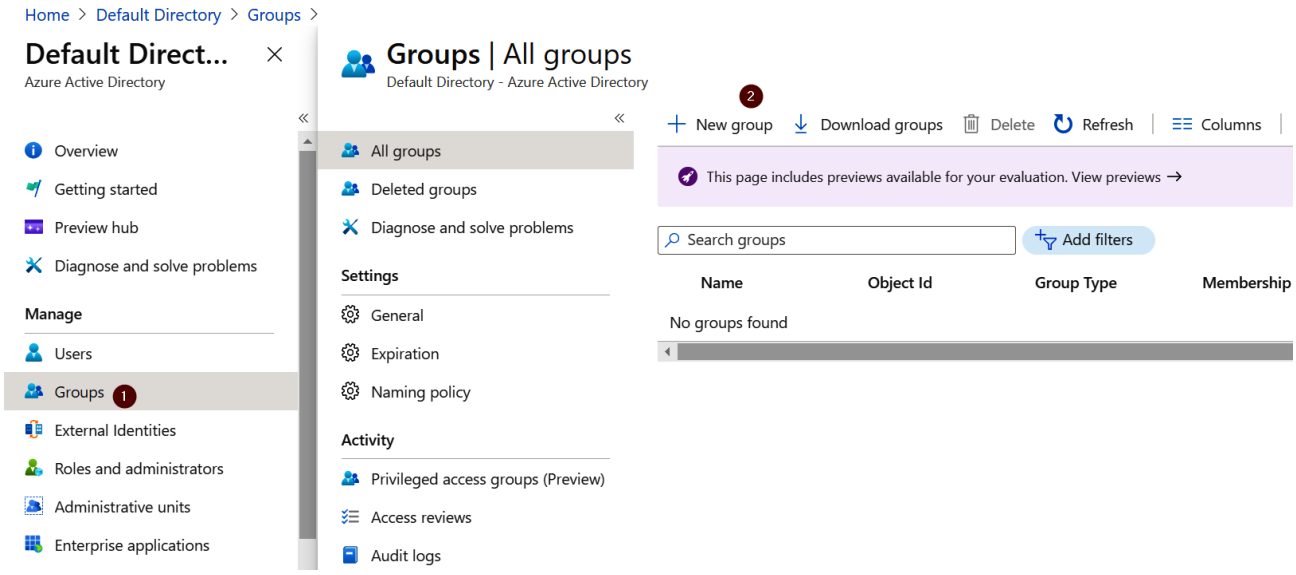


Figure 8.3: Creating a new Azure AD group

3. On the resulting page, create a security group and give it a name and description. Select your user as the owner and a member of this group. Click the **Create** button on the screen:

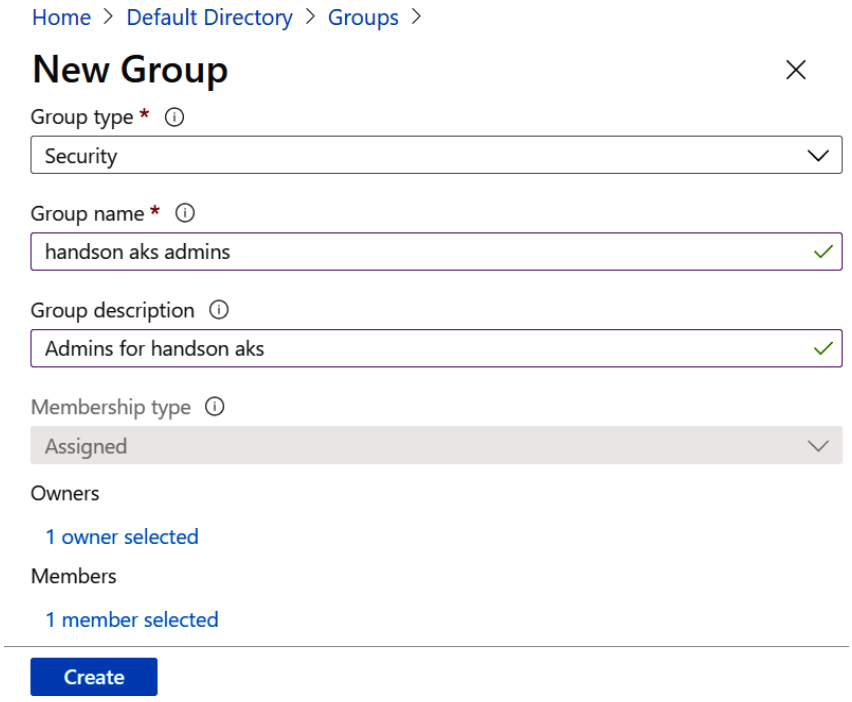


Figure 8.4: Providing details for creating the Azure AD group

4. Now that this group is created, search for your Azure cluster in the Azure search bar to open the AKS pane:

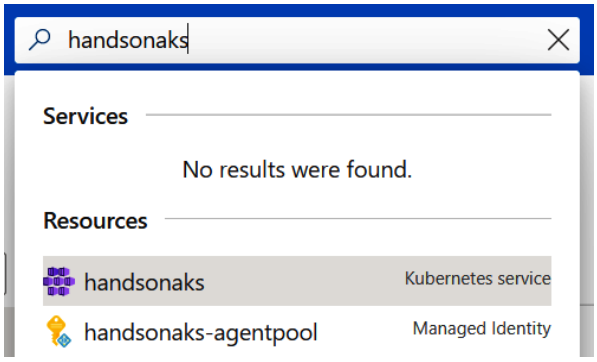


Figure 8.5: Searching for your cluster in the Azure search bar

5. In the AKS pane, select **Cluster configuration** under **Settings**. In this pane, you will be able to turn on **AKS-managed Azure Active Directory**. Enable the functionality and select the Azure AD group you created earlier to set as the admin Azure AD group. Finally, hit the **Save** button in the command bar, as shown in *Figure 8.6*:

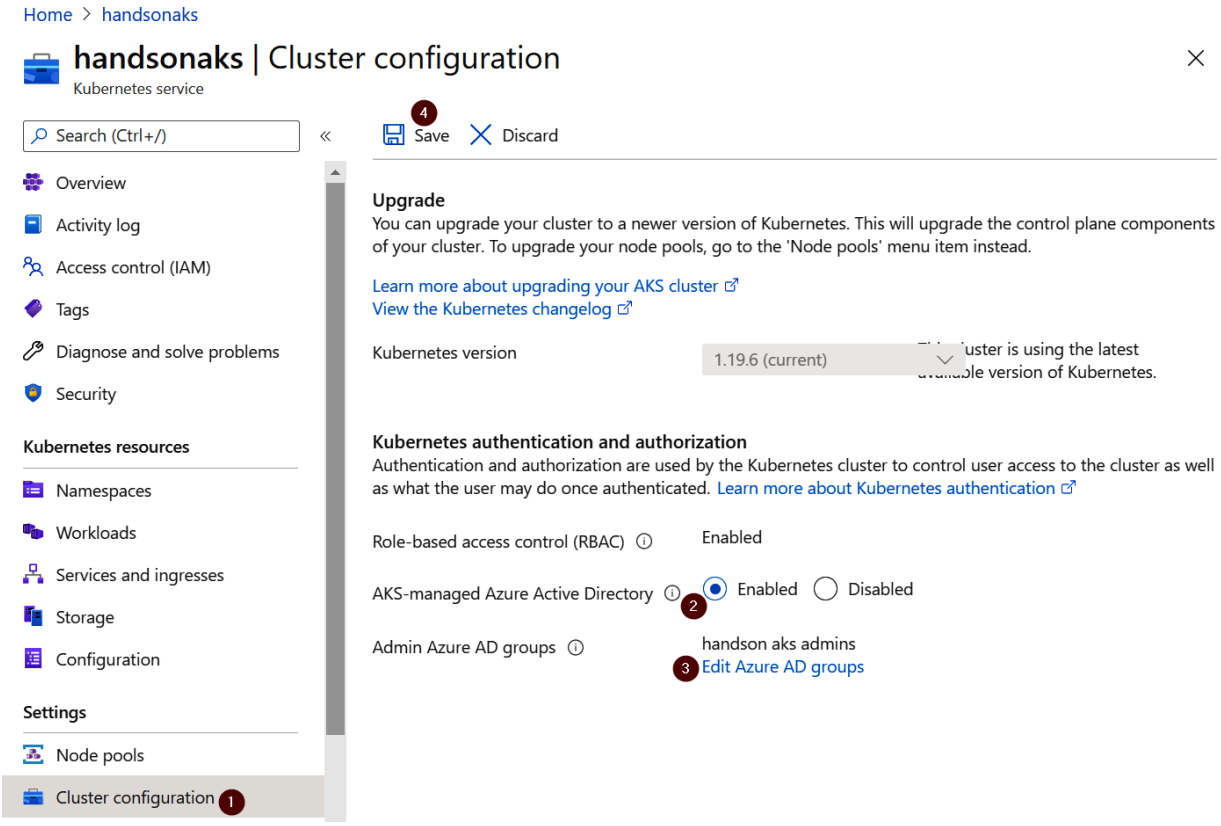


Figure 8.6: Enabling AKS-managed Azure Active Directory and clicking the Save button

This enables Azure AD–integrated RBAC on your AKS cluster. In the next section, you will create a new user and a new group that will be used in the section afterward to set up and test RBAC in Kubernetes.

Creating a user and group in Azure AD

In this section, you will create a new user and a new group in Azure AD. You will use them later on in the chapter to assign them permissions to your AKS cluster:

> **Note**
>
> You need the *User Administrator role* in Azure AD to be able to create users and groups.

1. To start with, search for `azure active directory` in the Azure search bar:

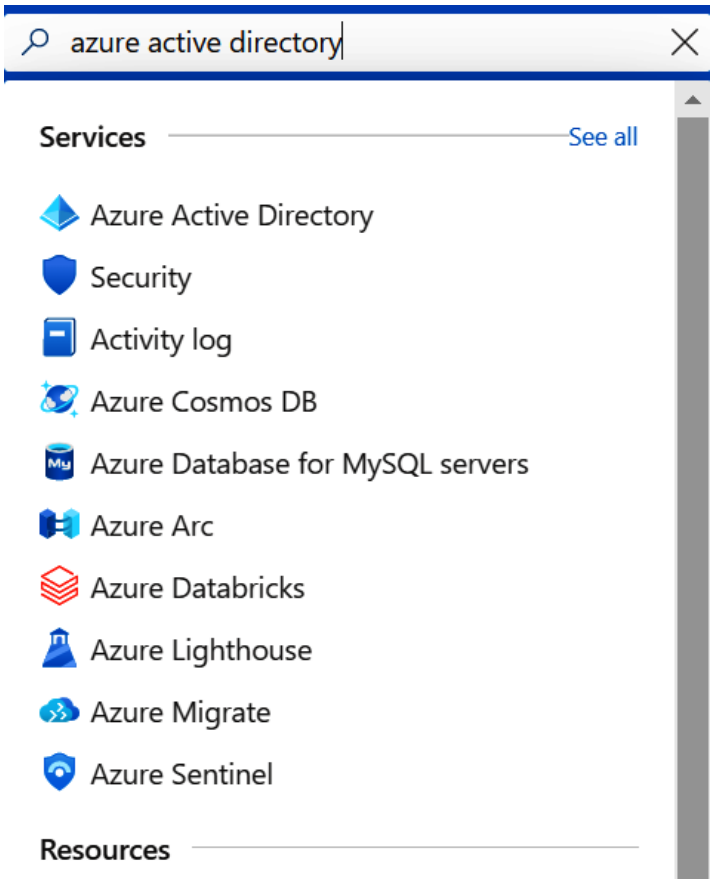


Figure 8.7: Searching for azure active directory in the search bar

2. Click on **All users** in the left pane. Then select **+ New user** to create a new user:

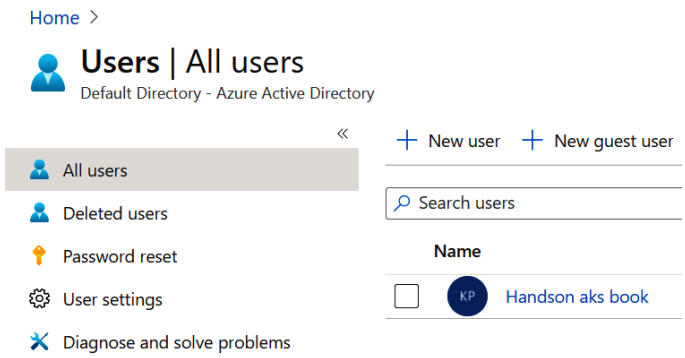


Figure 8.8: Clicking on + New user to create a new user

3. Provide the information about the user, including the username. Make sure to note down the password, as this will be required to sign in:

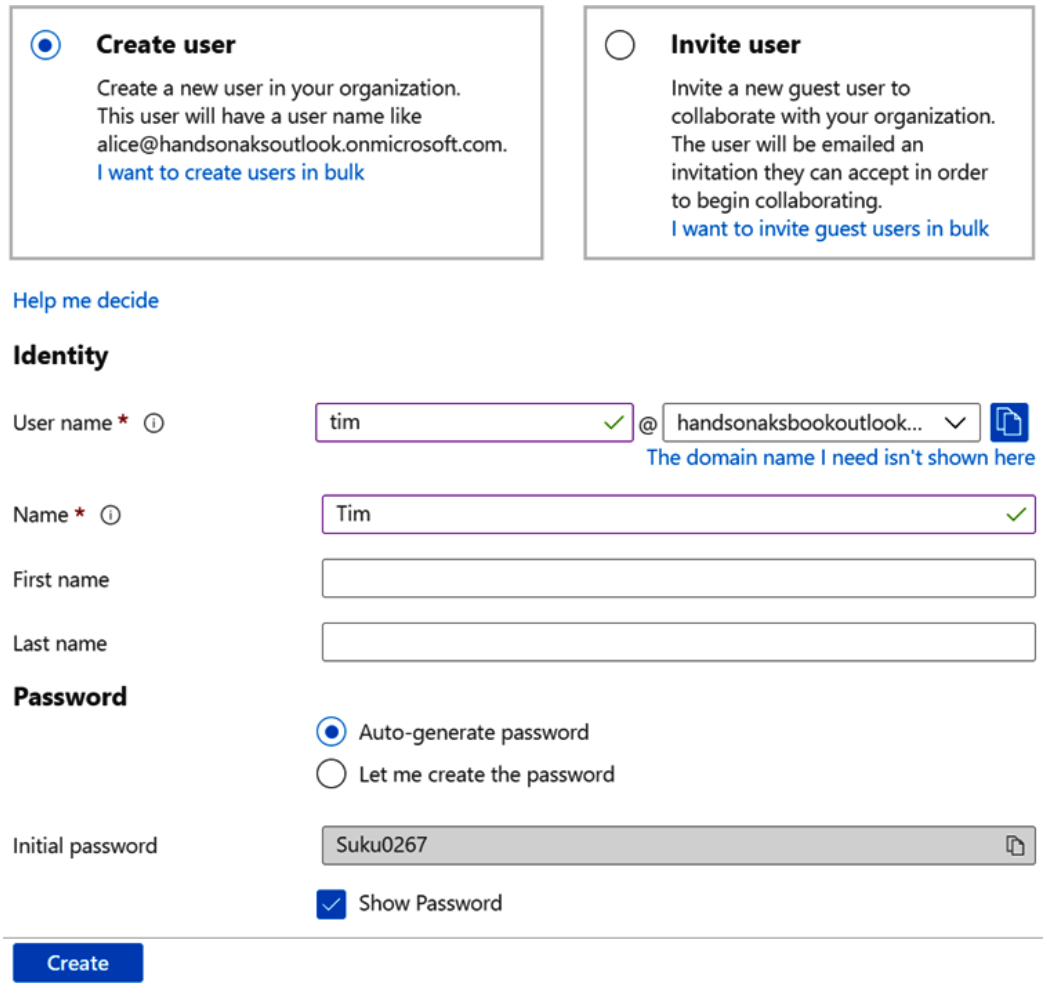


Figure 8.9: Providing the user details

4. Once the user is created, go back to the Azure AD pane and select **Groups**. Then click the **+ New group** button to create a new group:

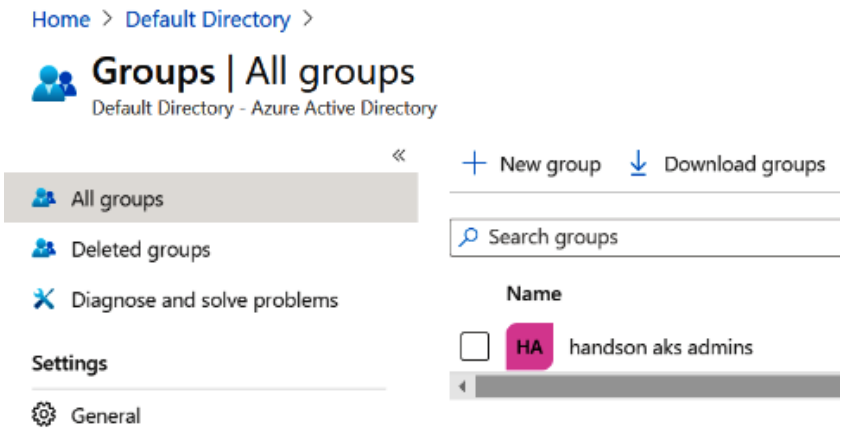


Figure 8.10: Clicking on + New group to create a new group

5. Create a new security group. Call the group `handson aks users` and add `Tim` as a member of the group. Then hit the **Create** button at the bottom:

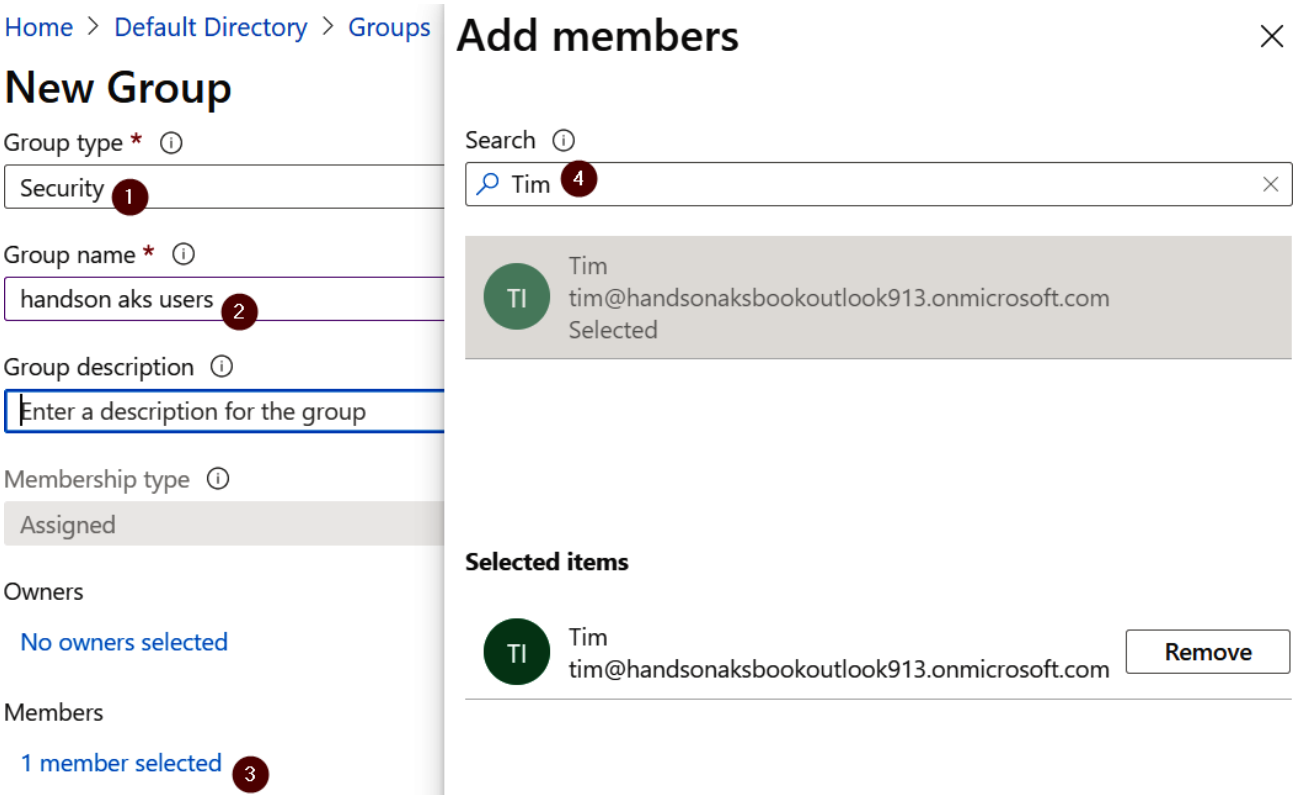


Figure 8.11: Providing the group type, group name, and group description

6. You have now created a new user and a new group. Next, you'll make that user a cluster user in AKS RBAC. This enables them to use the Azure CLI to get access to the cluster. To do that, search for your cluster in the Azure search bar:

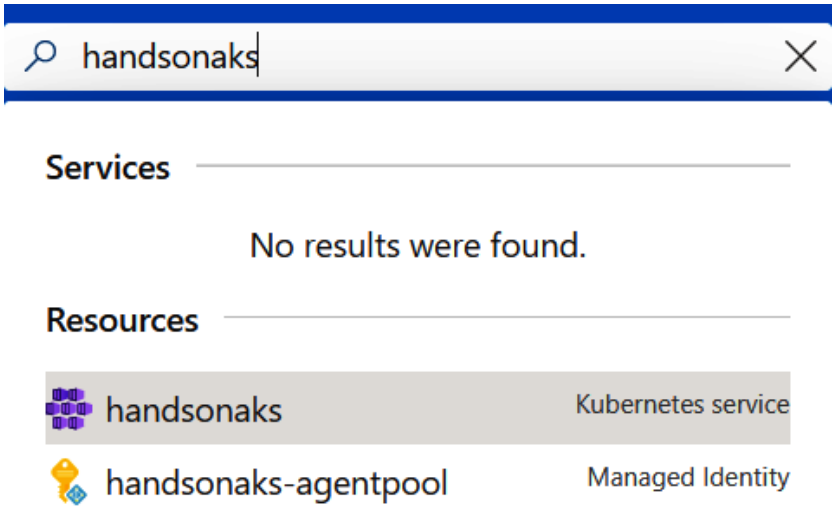


Figure 8.12: Searching for your cluster in the Azure search bar

7. In the cluster pane, click on **Access control (IAM)** and then click on the **+ Add** button to add a new role assignment. Select **Azure Kubernetes Service Cluster User Role** and assign that to the new user you just created:

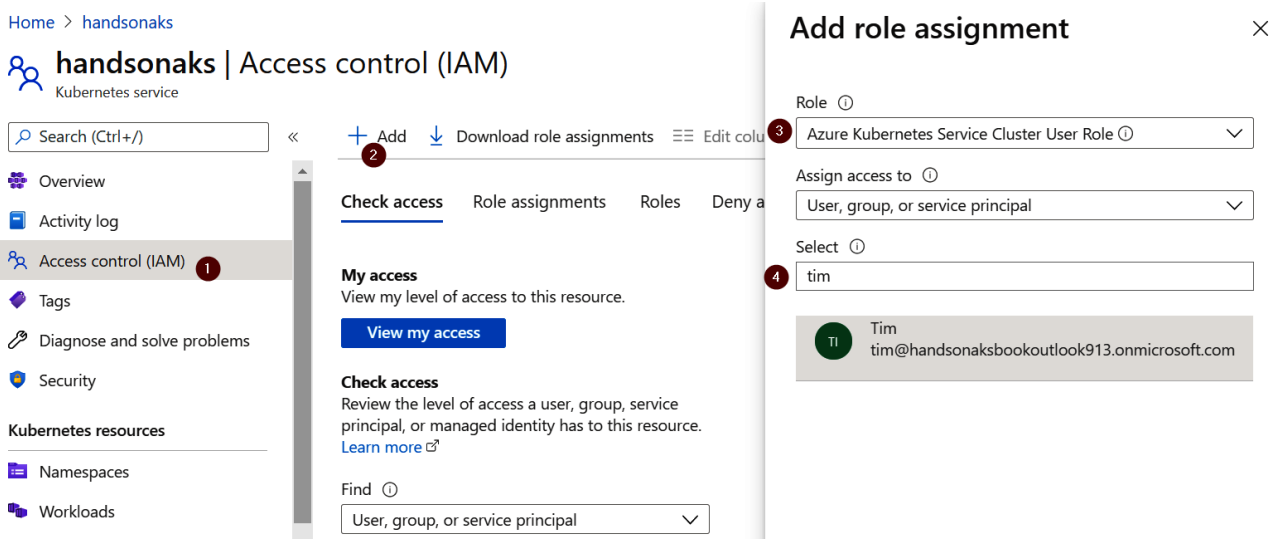


Figure 8.13: Assigning the cluster user role to the new user you created

8. As you will also be using Cloud Shell with the new user, you will need to give them contributor access to the Cloud Shell storage account. First, search for `storage` in the Azure search bar:

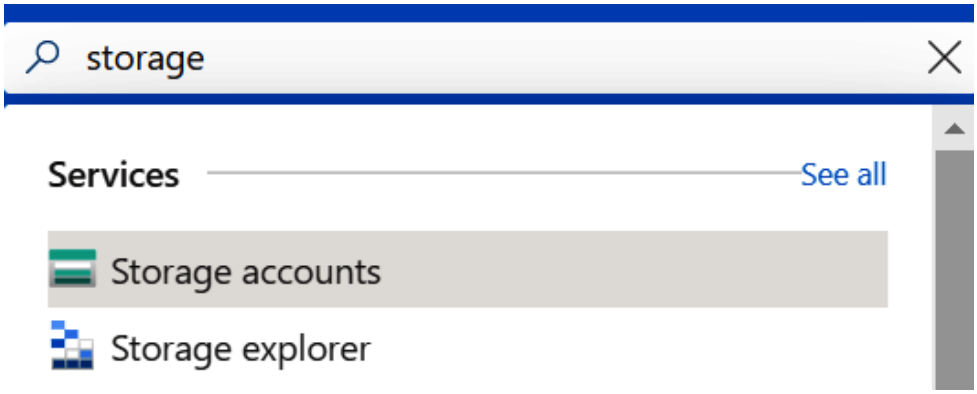


Figure 8.14: Searching for storage in the Azure search bar

9. There should be a storage account under **Resource group** with a name that starts with **cloud-shell-storage**. Click on the resource group:

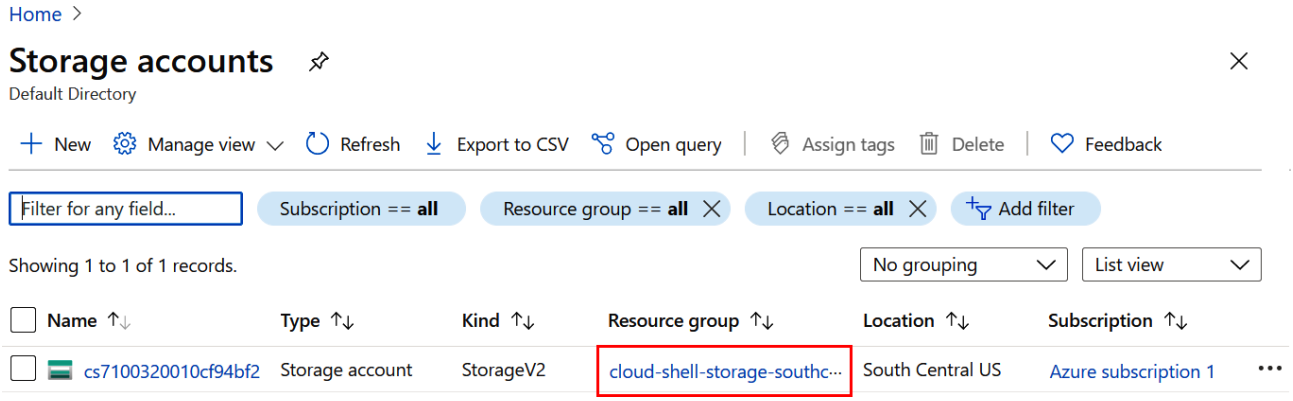


Figure 8.15: Selecting the resource group

10. Go to **Access control (IAM)** and click on the **+ Add** button. Give the **Storage Account Contributor** role to your newly created user:

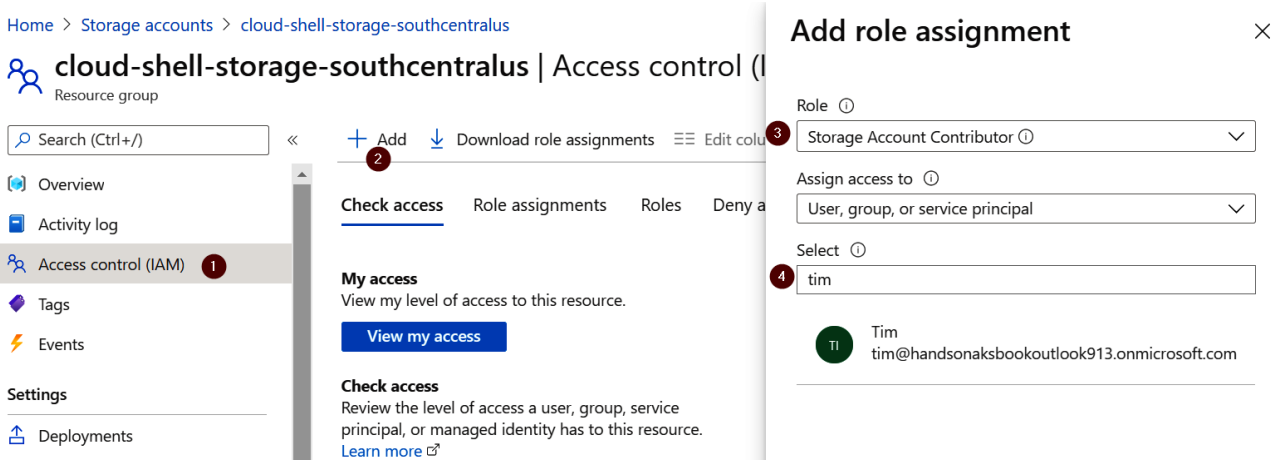


Figure 8.16: Assigning Storage Account Contributor role to the new user

This has concluded the creation of a new user and a group and giving that user access to AKS. In the next section, you will configure RBAC for that user and group in your AKS cluster.

Configuring RBAC in AKS

To demonstrate RBAC in AKS, you will create two namespaces and deploy the Azure voting application in each namespace. You will give the group cluster-wide read-only access to pods, and you will give the user the ability to delete pods in only one namespace. Practically, you will need to create the following objects in Kubernetes:

- `ClusterRole` to give read-only access
- `ClusterRoleBinding` to grant the group access to this role
- `Role` to give delete permissions in the `delete-access` namespace
- `RoleBinding` to grant the user access to this role

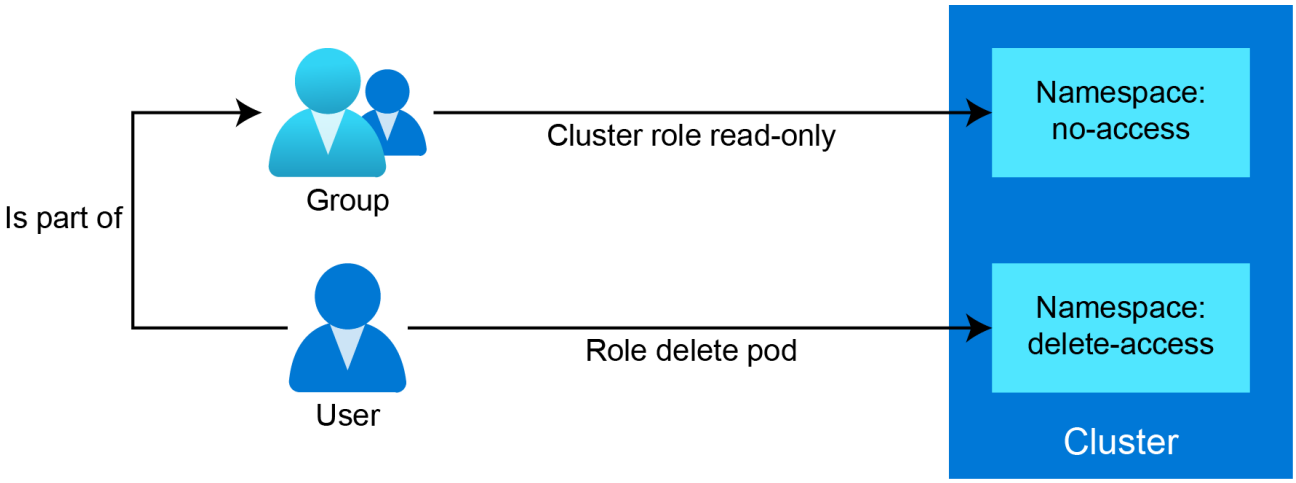


Figure 8.17: The group getting read-only access to the whole cluster, and the user getting delete permissions to the delete-access namespace

Let's set up the different roles on your cluster:

1. To start our example, you will need to retrieve the ID of the group. The following commands will retrieve the group ID:

   ```
   az ad group show -g 'handson aks users' \
     --query objectId -o tsv
   ```

   This will show your group ID. Note this down because you'll need it in the next steps:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter08$ az ad group show -g 'handson aks users' \
>   --query objectId -o tsv
2f5de9a5-dc1b-4ee6-b4a2-4898a6ee3fb6
```

Figure 8.18: Getting the group ID

2. In Kubernetes, you will create two namespaces for this example:

   ```
   kubectl create ns no-access
   kubectl create ns delete-access
   ```

3. You will also deploy the `azure-vote` application in both namespaces:

   ```
   kubectl create -f azure-vote.yaml -n no-access
   kubectl create -f azure-vote.yaml -n delete-access
   ```

4. Next, you will create the `ClusterRole` object. This is provided in the `clusterRole.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: ClusterRole
3   metadata:
4     name: readOnly
5   rules:
6   - apiGroups: [""]
7     resources: ["pods"]
8     verbs: ["get", "watch", "list"]
```

   Let's have a closer look at this file:

   - **Line 2**: Defines the creation of a `ClusterRole` instance
   - **Line 4**: Gives a name to our `ClusterRole` instance
   - **Line 6**: Gives access to all API groups
   - **Line 7**: Gives access to all pods
   - **Line 8**: Gives access to the actions `get`, `watch`, and `list`

   We will create `ClusterRole` using the following command:

```
kubectl create -f clusterRole.yaml
```

5. The next step is to create a cluster role binding. The binding links the role to a user or a group. This is provided in the `clusterRoleBinding.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: ClusterRoleBinding
3   metadata:
4     name: readOnlyBinding
5   roleRef:
6     kind: ClusterRole
7     name: readOnly
8     apiGroup: rbac.authorization.k8s.io
9   subjects:
10  - kind: Group
11    apiGroup: rbac.authorization.k8s.io
12    name: "<group-id>"
```

Let's have a closer look at this file:

- **Line 2**: Defines that we are creating a `ClusterRoleBinding` instance.
- **Line 4**: Gives a name to `ClusterRoleBinding`.
- **Lines 5–8**: Refer to the `ClusterRole` object we created in the previous step
- **Lines 9–12**: Refer to your group in Azure AD. Make sure to replace `<group-id>` on *line 12* with the group ID you got earlier.

We can create `ClusterRoleBinding` using the following command:

```
kubectl create -f clusterRoleBinding.yaml
```

6. Next, you'll create a role that is limited to the `delete-access` namespace. This is provided in the `role.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: Role
3   metadata:
4     name: deleteRole
5     namespace: delete-access
6   rules:
7   - apiGroups: [""]
8     resources: ["pods"]
9     verbs: ["delete"]
```

This file is similar to the `ClusterRole` object from earlier. There are two meaningful differences:

- **Line 2**: Defines that you are creating a `Role` instance and not a `ClusterRole` instance
- **Line 5**: Defines the namespace this role is created in

You can create `Role` using the following command:

```
kubectl create -f role.yaml
```

7. Finally, you will create a `RoleBinding` instance that links our user to the namespace role. This is provided in the `roleBinding.yaml` file:

```
1   apiVersion: rbac.authorization.k8s.io/v1
2   kind: RoleBinding
3   metadata:
4     name: deleteBinding
5     namespace: delete-access
6   roleRef:
7     kind: Role
8     name: deleteRole
9     apiGroup: rbac.authorization.k8s.io
10  subjects:
11  - kind: User
12    apiGroup: rbac.authorization.k8s.io
13    name: "<user e-mail address>"
```

   This file is similar to the `ClusterRoleBinding` object from earlier. There are a couple of meaningful differences:

   - **Line 2**: Defines the creation of a `RoleBinding` instance and not a `ClusterRoleBinding` instance
   - **Line 5**: Defines the namespace this `RoleBinding` instance is created in
   - **Line 7**: Refers to a regular role and not a `ClusterRole` instance
   - **Lines 11–13**: Defines a user instead of a group

   You can create `RoleBinding` using the following command:

```
kubectl create -f roleBinding.yaml
```

This has concluded the requirements for RBAC. You have created two roles—`ClusterRole` and one namespace-bound role, and set up two `RoleBindings` objects—`ClusterRoleBinding` and the namespace-bound `RoleBinding`. In the next section, you will explore the impact of RBAC by signing in to the cluster as the new user.

Verifying RBAC for a user

To verify that RBAC works as expected, you will sign in to the Azure portal using the newly created user. Go to https://portal.azure.com in a new browser, or an InPrivate window, and sign in with the newly created user. You will be prompted immediately to change your password. This is a security feature in Azure AD to ensure that only that user knows their password:

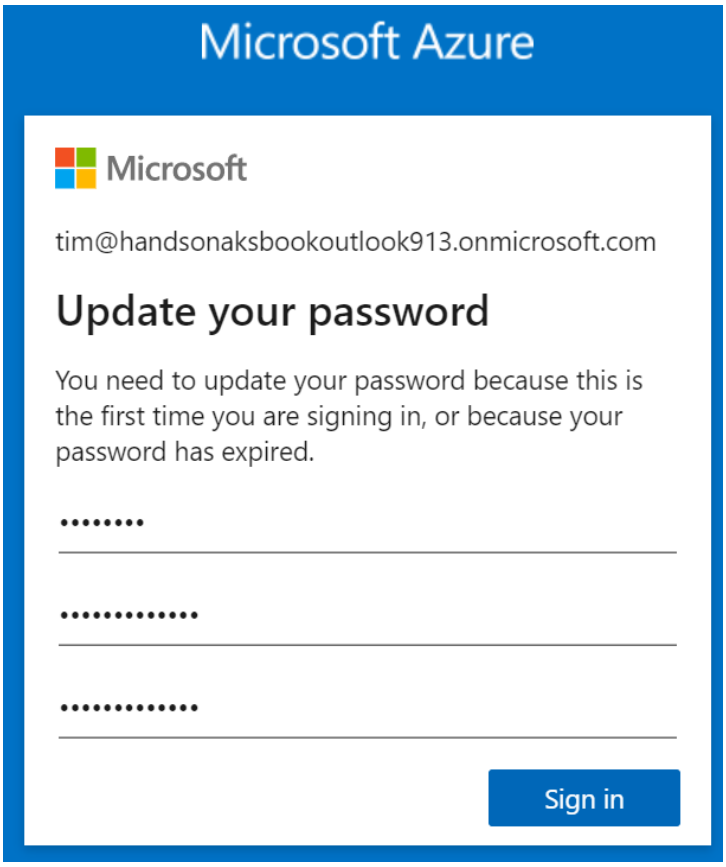


Figure 8.19: You will be asked to change your password

Once you have changed your password, you can start testing the different RBAC roles:

1. You will start this experiment by setting up Cloud Shell for the new user. Launch Cloud Shell and select **Bash**:

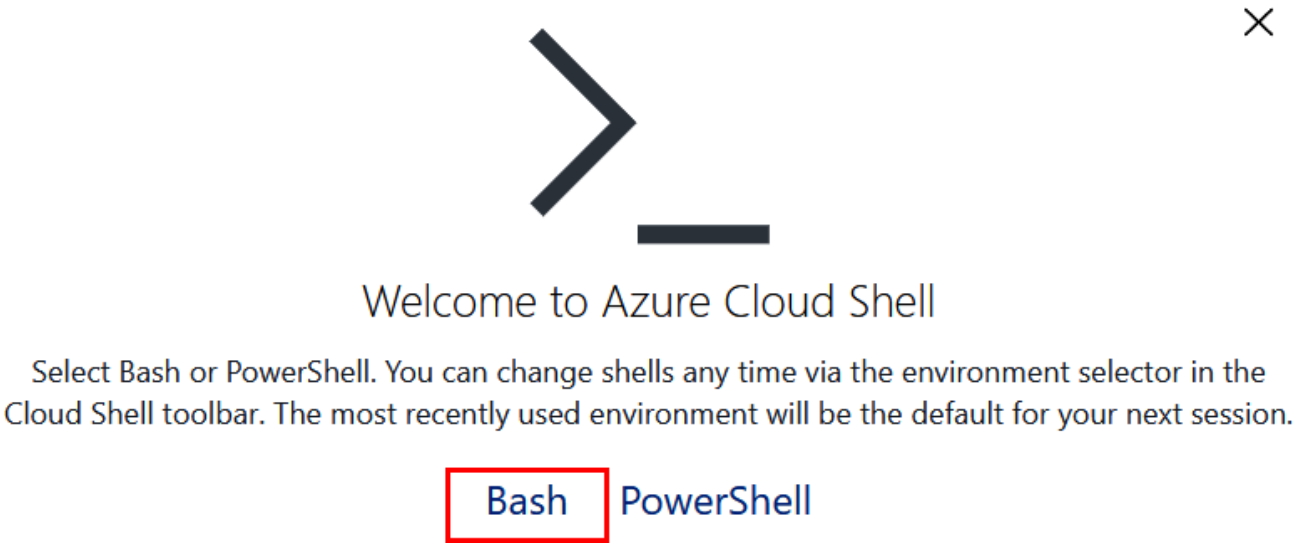


Figure 8.20: Selecting Bash in Cloud Shell

2. In the next dialog box, select **Show advanced settings**:

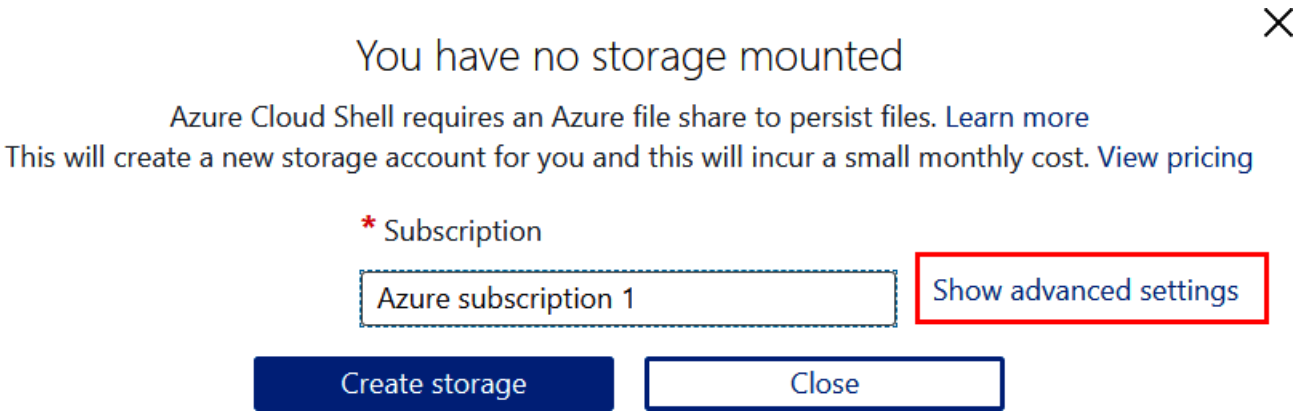


Figure 8.21: Selecting Show advanced settings

3. Then, point Cloud Shell to the existing storage account and create a new file share:

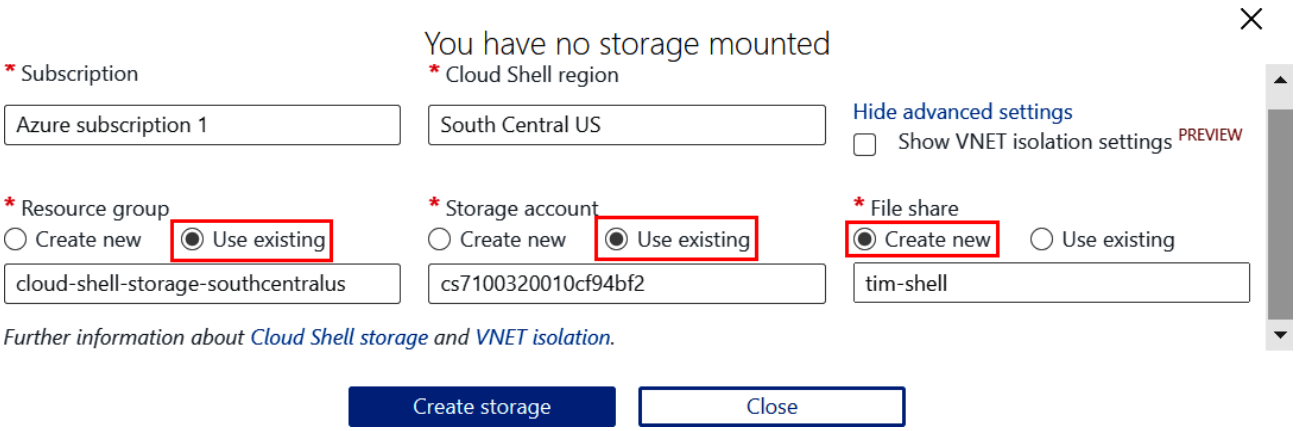


Figure 8.22: Pointing to the existing storage account and creating a new file share

4. Once Cloud Shell is available, get the credentials to connect to the AKS cluster:

```
az aks get-credentials -n handsonaks -g rg-handsonaks
```

Then, try a command in `kubectl`. Let's try to get the nodes in the cluster:

```
kubectl get nodes
```

Since this is the first command executed against an RBAC-enabled cluster, you are asked to sign in again. Browse to https://microsoft.com/devicelogin and provide the code Cloud Shell showed you (this code is highlighted in *Figure 8.24*). Make sure you sign in here with your new user credentials:

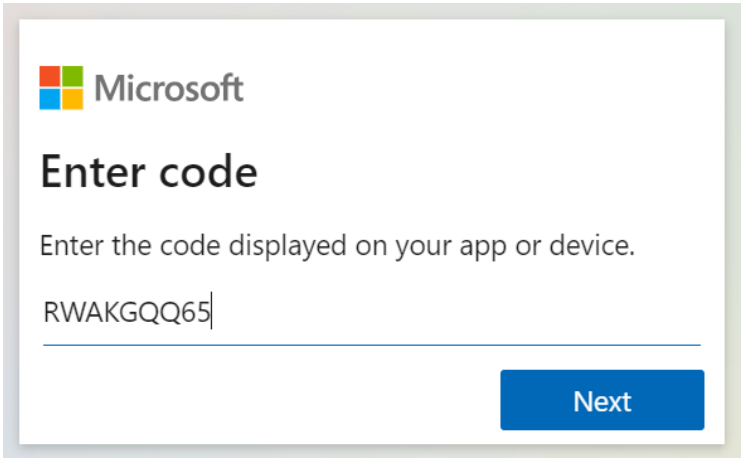


Figure 8.23: Copying and pasting the code Cloud Shell showed you in the prompt

After you have signed in, you should get a `Forbidden` error message from `kubectl`, informing you that you don't have permission to view the nodes in the cluster. This was expected since the user is configured only to have access to pods:

```
tim@Azure:~$ kubectl get nodes
To sign in, use a web browser to open the page https://microsoft.com/devicelogi
n and enter the code RWAKGQQ65 to authenticate.
Error from server (Forbidden): nodes is forbidden: User "tim@handsonaksbookoutl
ook913.onmicrosoft.com" cannot list resource "nodes" in API group "" at the clu
ster scope
```

Figure 8.24: The prompt asking you to sign in and the Forbidden message

5. Now you can verify that your user has access to view pods in all namespaces and that the user has permission to delete pods in the `delete-access` namespace:

```
kubectl get pods -n no-access
kubectl get pods -n delete-access
```

This should succeed for both namespaces. This is due to the `ClusterRole` object configured for the user's group:

```
tim@Azure:~$ kubectl get pods -n no-access
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-6bbcb89698-tjpv7    1/1     Running   0          27m
azure-vote-front-64b6fd89d6-vm9c8   1/1     Running   0          27m
tim@Azure:~$ kubectl get pods -n delete-access
NAME                                READY   STATUS    RESTARTS   AGE
azure-vote-back-6bbcb89698-qrbl4    1/1     Running   0          27m
azure-vote-front-64b6fd89d6-wkp9l   1/1     Running   0          27m
```

Figure 8.25: The user has access to view pods in both namespaces

6. Let's also verify the `delete` permissions:

```
kubectl delete pod --all -n no-access
kubectl delete pod --all -n delete-access
```

   As expected, this is denied in the `no-access` namespace and allowed in the `delete-access` namespace, as seen in *Figure 8.26*:

```
tim@Azure:~$ kubectl delete pod --all -n no-access
Error from server (Forbidden): pods "azure-vote-back-6bbcb89698-tjpv7" is forbi
dden: User "tim@handsonaksbookoutlook913.onmicrosoft.com" cannot delete resourc
e "pods" in API group "" in the namespace "no-access"
Error from server (Forbidden): pods "azure-vote-front-64b6fd89d6-vm9c8" is forb
idden: User "tim@handsonaksbookoutlook913.onmicrosoft.com" cannot delete resour
ce "pods" in API group "" in the namespace "no-access"
tim@Azure:~$ kubectl delete pod --all -n delete-access
pod "azure-vote-back-6bbcb89698-qrbl4" deleted
pod "azure-vote-front-64b6fd89d6-wkp9l" deleted
```

Figure 8.26: Deletes are denied in the no-access namespace and allowed in the delete-access namespace

In this section, you have verified the functionality of RBAC on your Kubernetes cluster. Since this is the last section of this chapter, let's make sure to clean up the deployments and namespaces in the cluster. Make sure to execute these steps from Cloud Shell with your main user, not the new user:

```
kubectl delete -f azure-vote.yaml -n no-access
kubectl delete -f azure-vote.yaml -n delete-access
kubectl delete -f .
kubectl delete ns no-access
kubectl delete ns delete-access
```

This concludes the overview of RBAC on AKS.

Summary

In this chapter, you learned about RBAC on AKS. You enabled Azure AD–integrated RBAC in your cluster. After that, you created a new user and group and set up different RBAC roles on your cluster. Finally, you signed in using that user and were able to verify that the RBAC roles that were configured gave you limited access to the cluster you were expecting.

This deals with how users can get access to your Kubernetes cluster. The pods running on your cluster might also need an identity in Azure AD that they can use to access resources in Azure services such as Blob Storage or Key Vault. You will learn more about this use case and how to set this up using an Azure AD pod identity in AKS in the next chapter.

9

Azure Active Directory pod-managed identities in AKS

In the previous chapter, *Chapter 8*, *Role-based access control in* AKS, you integrated your AKS cluster with **Azure Active Directory** (**Azure AD**). You then assigned Kubernetes roles to users and groups in Azure AD. In this chapter, you will explore how you can integrate your applications running on AKS with Azure AD, and you will learn how you can give your pods an identity in Azure so they can interact with other Azure resources.

In Azure, application identities use a functionality called service principals. A service principal is the equivalent of a service account in the cloud. An application can use a service principal to authenticate to Azure AD and get access to resources. Those resources could be either Azure resources such as Azure Blob Storage or Azure Key Vault, or they could be applications that you developed that are integrated with Azure AD.

There are two ways to authenticate a service principal: you can either use a password or a combination of a certificate and a private key. Although these are secure ways to authenticate your applications, managing passwords or certificates and the rotation associated with them can be cumbersome.

Managed identities in Azure are a functionality that makes authenticating to a service principal easier. It works by assigning an identity to a compute resource in Azure, such as a virtual machine or an Azure function. Those compute resources can authenticate using that managed identity by calling an endpoint that only that machine can reach. This is a secure type of authentication that does not require you to manage passwords or certificates.

Azure AD pod-managed identities allow you to assign managed identities to pods in Kubernetes. Since pods in Kubernetes run on virtual machines, by default, each pod would be able to access the managed identity endpoint and authenticate using that identity. Using Azure AD pod-managed identities, pods can no longer reach the internal endpoint for the virtual machine, and rather only get access to identities assigned to that specific pod.

In this chapter, you'll configure an Azure AD pod-managed identity on an AKS cluster and use it to get access to Azure Blob Storage. In the next chapter, you will then use these Azure AD pod-managed identities to get access to Azure Key Vault and manage Kubernetes secrets.

The following topics will be covered briefly in this chapter:

- An overview of Azure AD pod-managed identities
- Setting up a new cluster with Azure AD pod-managed identities
- Linking an identity to your cluster
- Using a pod with managed identity

Let's start with an overview of Azure AD pod-managed identities.

An overview of Azure AD pod-managed identities

The goal of this section is to describe Azure managed identities and Azure AD pod-managed identities.

As explained in the introduction, managed identities in Azure are a way to securely authenticate applications running inside Azure. There are two types of managed identities in Azure. The difference between them is how they are linked to resources:

- **System assigned**: This type of managed identity is linked 1:1 to the resource (such as a virtual machine) itself. This managed identity also shares the lifecycle of the resource, meaning that once the resource is deleted, the managed identity is also deleted.
- **User assigned**: User-assigned managed identities are standalone Azure resources. A user-assigned managed identity can be linked to multiple resources. When a resource is deleted, the managed identity is not deleted.

Both types of managed identities work the same way once they are created and linked to a resource. This is how managed identities work from an application perspective:

1. Your application running in Azure requests a token to the **Instance Metadata Service** (**IMDS**). The IMDS is only available to that resource itself, at a non-routable IP address (`169.254.169.254`).
2. The IMDS will request a token from Azure AD. It uses a certificate that is configured for your managed identity and is only known by the IMDS.
3. Azure AD will return a token to the IMDS, which will, in turn, return that token to your application.
4. Your application can use this token to authenticate to other resources, for instance, Azure Blob Storage.

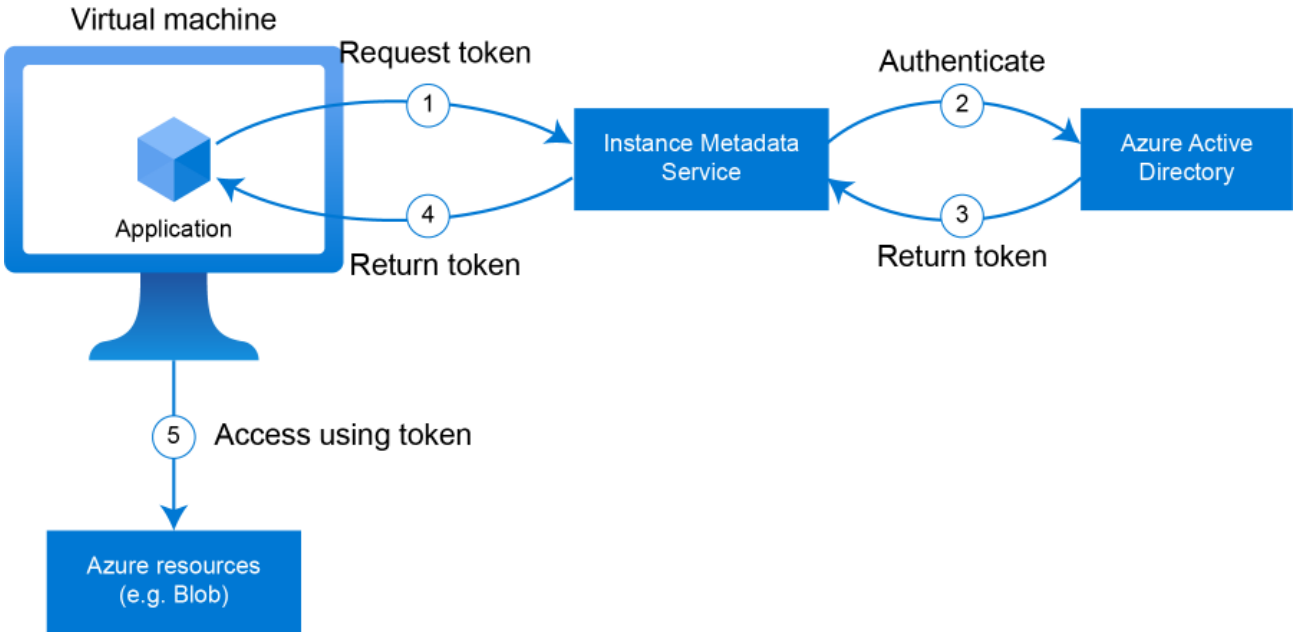


Figure 9.1: Managed identity in an Azure virtual machine

When running multiple pods on a single virtual machine in a Kubernetes cluster, by default each pod can reach the IMDS endpoint. This means that each pod could get access to the identities configured for that virtual machine.

The Azure AD pod-managed identities add-on for AKS configures your cluster in such a way that pods can no longer access the IMDS endpoint directly to request an access token. It configures your cluster in such a way that pods trying to access to IMDS endpoint (1) will connect to a DaemonSet running on the cluster. This DaemonSet is called the **node managed identity** (**NMI**). The NMI will verify which identities that pod should have access to. If the pod is configured to have access to the requested identity, then the DaemonSet will connect to the IMDS (2 to 5) to get the token, and then deliver the token to the pod (6). The pods can then use this token to access Azure resources (7).

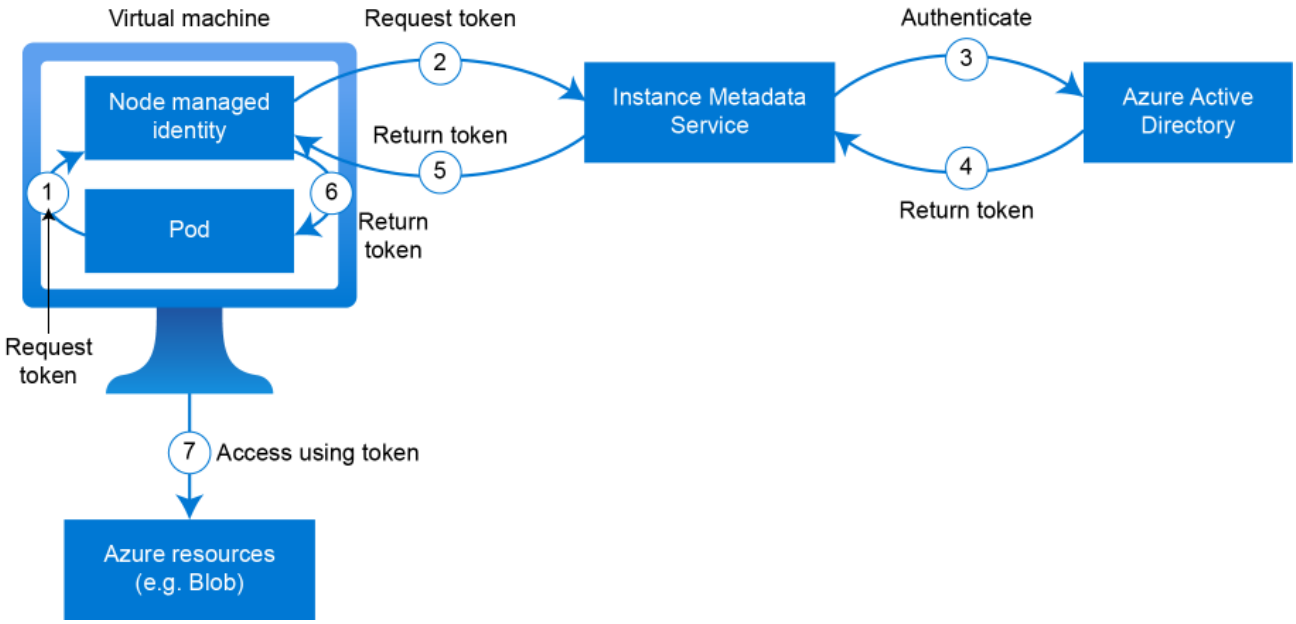


Figure 9.2: Azure AD pod-managed identity

This way, you can control which pods on your cluster have access to certain identities.

Azure AD pod-managed identities were initially developed as an open-source project by Microsoft on GitHub. More recently, Microsoft has released Azure AD pod-managed identities as an AKS add-on. The benefit of using Azure AD pod-managed identities as an AKS add-on is that the functionality is supported by Microsoft and the software will be updated automatically as part of regular cluster operations.

> **Note**
>
> At the time of writing, the Azure AD pod-managed identities add-on is in preview. Currently, it is also not supported for Windows containers. Using preview functionality for product use cases is not recommended.

Now that you know how Azure AD pod-managed identities work, let's set it up on an AKS cluster in the next section.

Setting up a new cluster with Azure AD pod-managed identities

As mentioned in the previous section, there are two ways to set up Azure AD pod-managed identities in AKS. It can either be done using the open-source project on GitHub, or by setting it up as an AKS add-on. By using the add-on, you'll get a supported configuration, which is why you'll set up a cluster using the add-on in this section.

At the time of writing, it is not yet possible to enable the Azure AD pod-managed identities add-on on an existing cluster, which is why in the following instructions you'll delete your existing cluster and create a new one with the add-on installed. By the time you are reading this, it might be possible to enable this add-on on an existing cluster without recreating your cluster.

Also, because the functionality is in preview at the time of this writing, you'll have to register for the preview. That'll be the first step in this section:

1. Start by opening Cloud Shell and registering for the preview of Azure AD pod-managed identities:

```
az feature register --name EnablePodIdentityPreview \
  --namespace Microsoft.ContainerService
```

2. You'll also need a preview extension of the Azure CLI, which you can install using the following command:

```
az extension add --name aks-preview
```

3. Now you can go ahead and delete your existing cluster. This is required to ensure you have enough core quota available in Azure. You can do this using the following command:

```
az aks delete -n handsonaks -g rg-handsonaks --yes
```

4. Once your previous cluster is deleted, you'll have to wait until the pod identity preview is registered on your subscription. You can use the following command to verify this status:

```
az feature show --name EnablePodIdentityPreview \
  --namespace Microsoft.ContainerService -o table
```

   Wait until the status shows as registered, as shown in *Figure 9.3*:

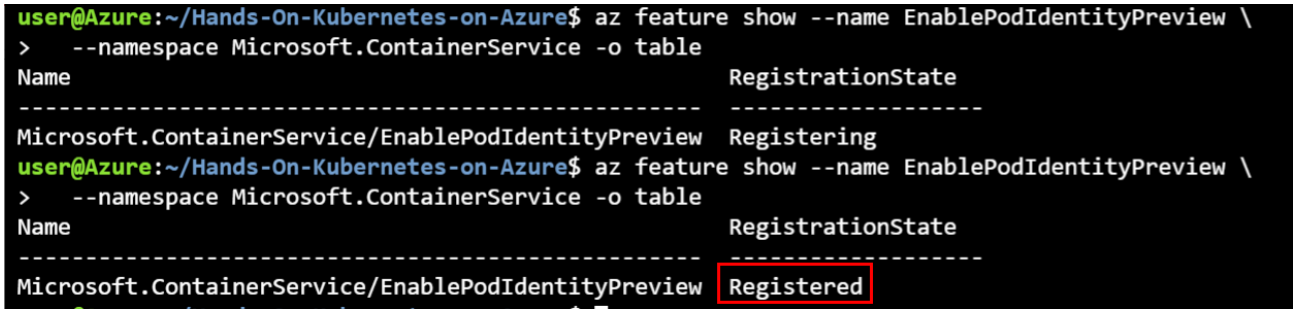


Figure 9.3: Waiting for the feature to be registered

5. If the feature is registered and your old cluster is deleted, you need to refresh the registration of the namespace before creating a new cluster. Let's first refresh the registration of the namespace:

```
az provider register --namespace Microsoft.ContainerService
```

6. And now you can create a new cluster using the Azure AD pod-managed identities add-on. You can use the following command to create a new cluster with the add-on enabled:

```
az aks create -g rg-handsonaks -n handsonaks \
  --enable-managed-identity --enable-pod-identity \
  --network-plugin azure --node-vm-size Standard_DS2_v2 \
  --node-count 2 --generate-ssh-keys
```

7. This will take a couple of minutes to finish. Once the command finishes, obtain the credentials to access your cluster and verify you can access your cluster using the following commands:

```
az aks get-credentials -g rg-handsonaks \
  -n handsonaks --overwrite-existing
kubectl get nodes
```

This should return an output similar to *Figure 9.4*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ az aks get-credentials -g rg-handsonaks \
>    -n handsonaks --overwrite-existing
The behavior of this command has been altered by the following extension: aks-preview
Merged "handsonaks" as current context in /home/kelly/.kube/config
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ kubectl get nodes
NAME                                STATUS   ROLES   AGE    VERSION
aks-nodepool1-36910094-vmss000000   Ready    agent   107s   v1.18.14
aks-nodepool1-36910094-vmss000001   Ready    agent   105s   v1.18.14
```

Figure 9.4: Getting cluster credentials and verifying access

Now you have a new AKS cluster with Azure AD pod-managed identities enabled. In the next section, you will create a managed identity and link it to your cluster.

Linking an identity to your cluster

In the previous section, you created a new cluster with Azure AD pod-managed identities enabled. Now you are ready to create a managed identity and link it to your cluster. Let's get started:

1. To start, you will create a new managed identity using the Azure portal. In the Azure portal, look for `managed identity` in the search bar, as shown in *Figure 9.5*:

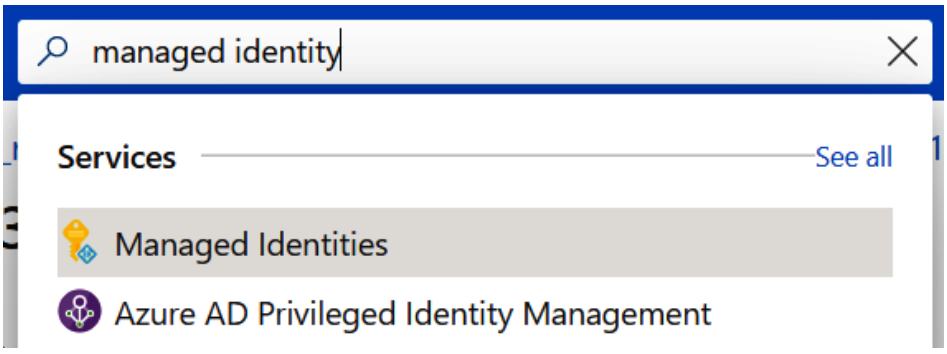


Figure 9.5: Navigating to Managed Identities in the Azure portal

2. In the resulting pane, click the **+ New** button at the top. To organize the resources for this chapter together, it's recommended to create a new resource group. In the resulting pane, click the **Create new** button to create a new resource group. Call it `aad-pod-id`, as shown in *Figure 9.6*:

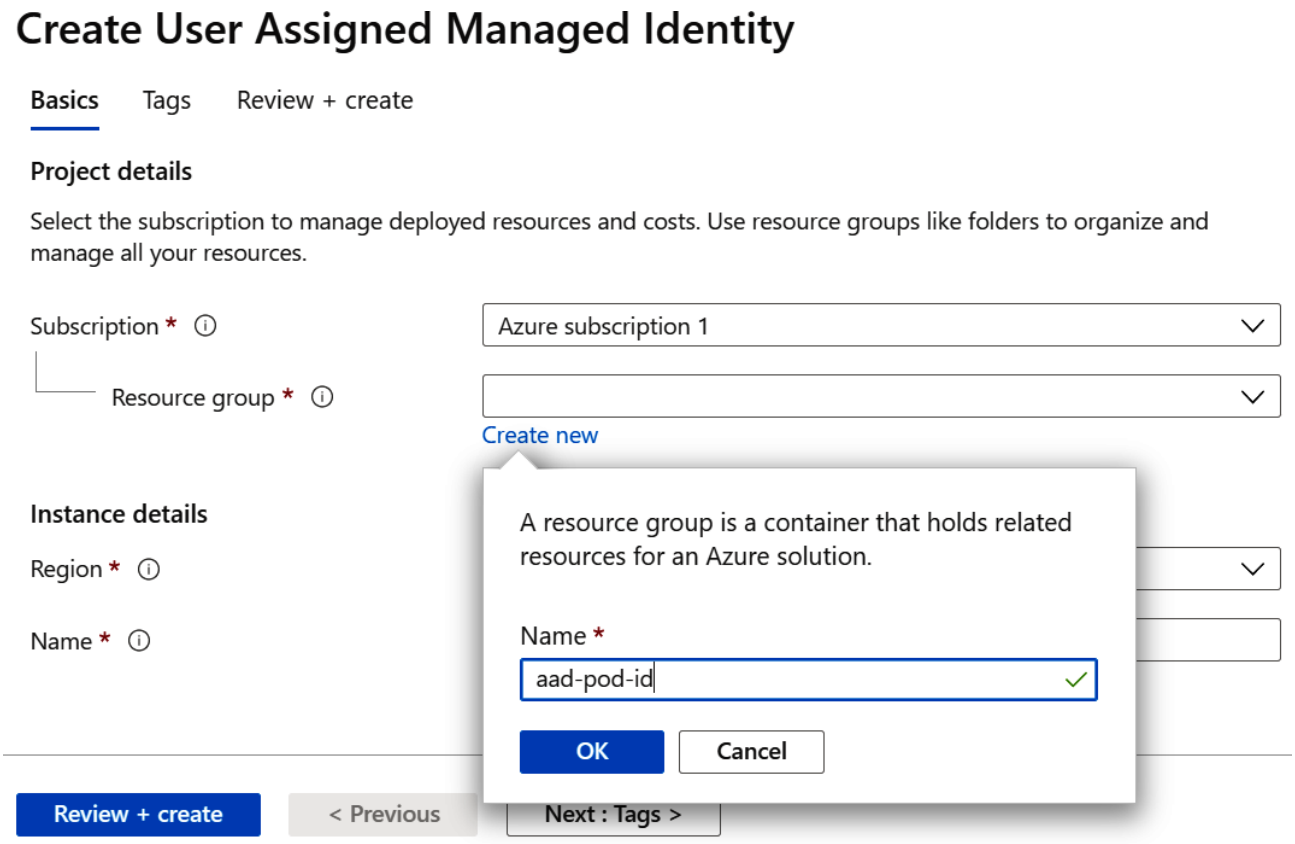


Figure 9.6: Creating a new resource group

3. Now, select the region you created your cluster in as the region for your managed identity and give it a name (`aad-pod-id` in this example), as shown in *Figure 9.7*. To finish, click the **Review + create** button and in the final window click the **Create** button to create your managed identity:

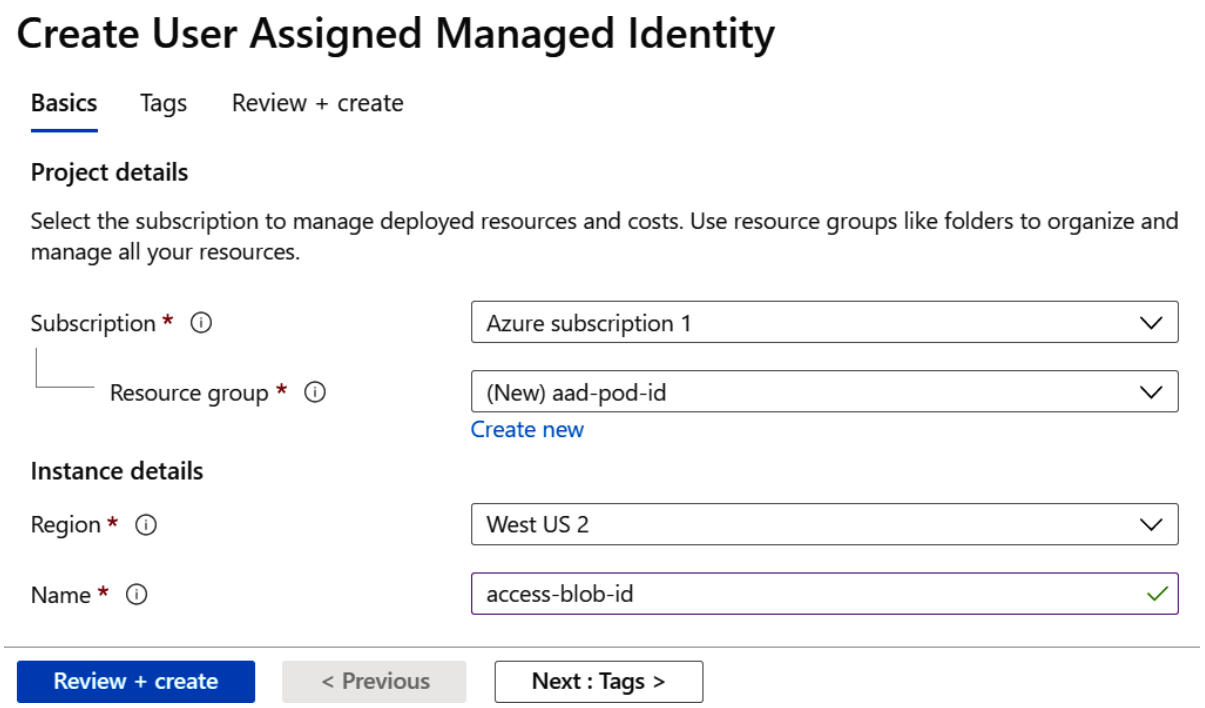


Figure 9.7: Providing Instance details for the managed identity

4. Once the managed identity has been created, hit the **Go to resource** button to go to the resource. Here, you will need to copy the client ID and the resource ID. They will be used later in this chapter. Copy and paste the values somewhere that you can access later. First, you will need the client ID of the managed identity. You can find that in the **Overview** pane of the managed identity, as shown in *Figure 9.8*:

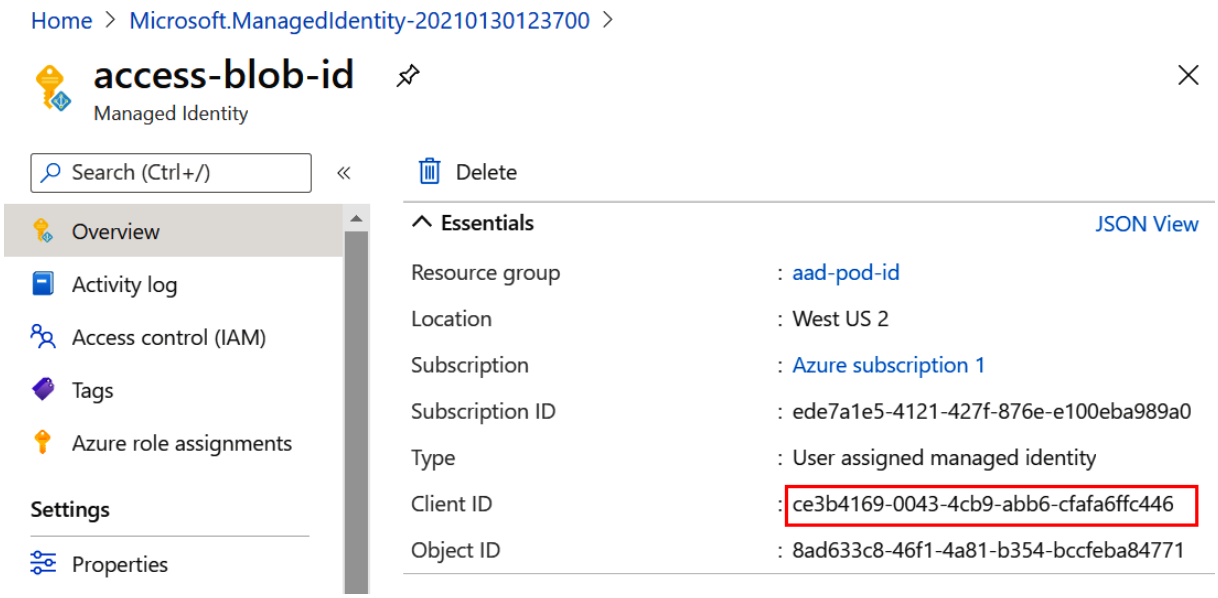


Figure 9.8: Getting the client ID of the managed identity

5. Finally, you will also need the resource ID of the managed identity. You can find that in the Properties pane of the managed identity, as shown in *Figure* 9.9:

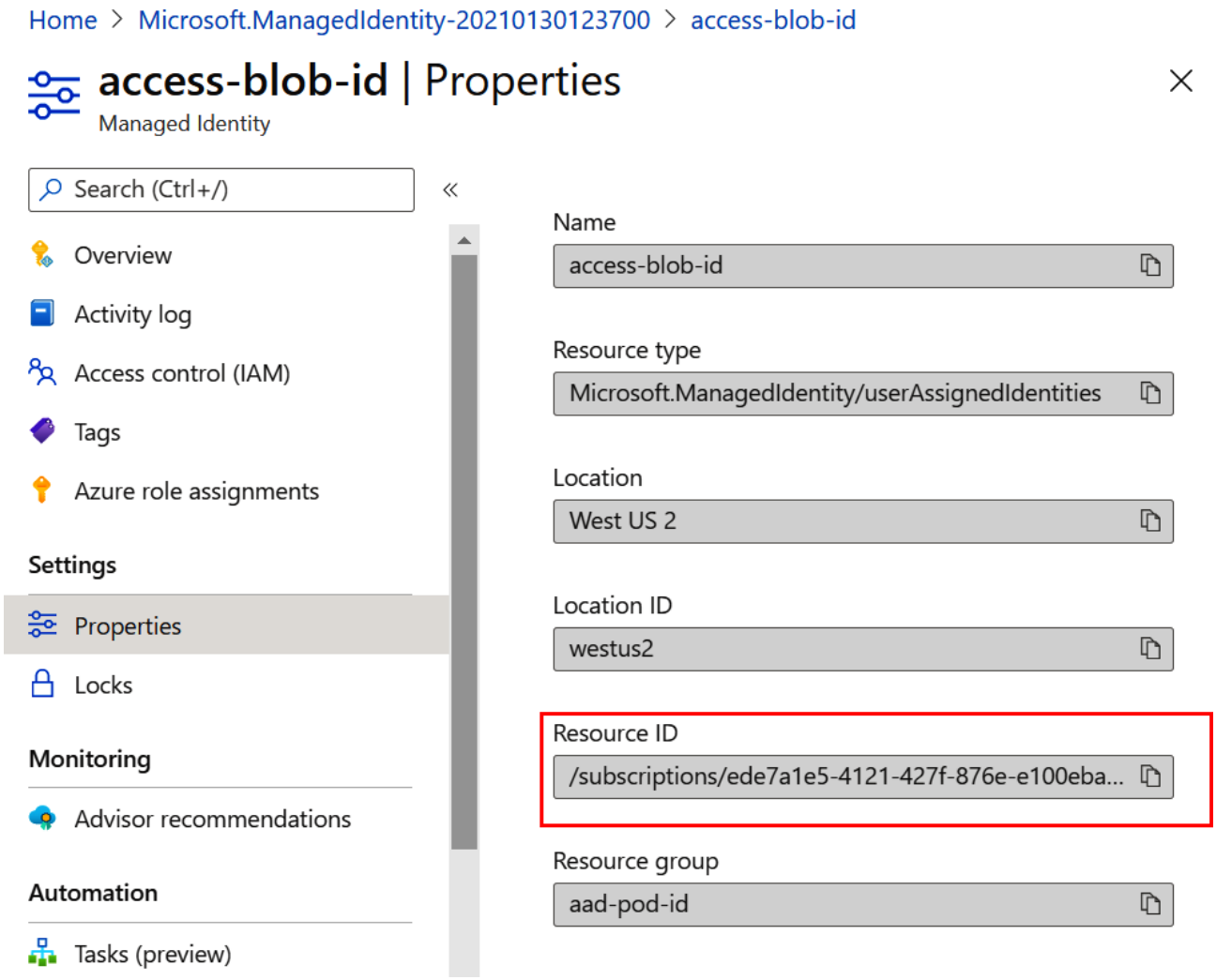


Figure 9.9: Getting the resource ID of the managed identity

6. Now you are ready to link the managed identity to your AKS cluster. To do this, you will run a command in Cloud Shell, and afterward you will be able to verify that the identity is available in your cluster. Let's start with linking the identity. Make sure to replace `<Managed identity resource ID>` with the resource you copied earlier:

```
az aks pod-identity add --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace default \
  --name access-blob-id \
  --identity-resource-id <Managed identity resource ID>
```

7. You can verify that your identity was successfully linked to your cluster by running the following command:

```
kubectl get azureidentity
```

   This should give you an output similar to *Figure 9.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ kubectl get azureidentity
NAME              AGE
access-blob-id    59s
```

Figure 9.10: Verifying the availability of the identity in the cluster

This means that the identity is now available for you to use in your cluster. How you do this will be explained in the next section.

Using a pod with managed identity

In the previous section, you created a managed identity and linked it to your cluster. In this section, you will create a new blob storage account and give the managed identity you created permission over this storage account. Then, you will create a new pod in your cluster that can use that managed identity to interact with that storage account. Let's get started by creating a new storage account:

1. To create a new storage account, look for `storage accounts` in the Azure search bar, as shown in *Figure 9.11*:

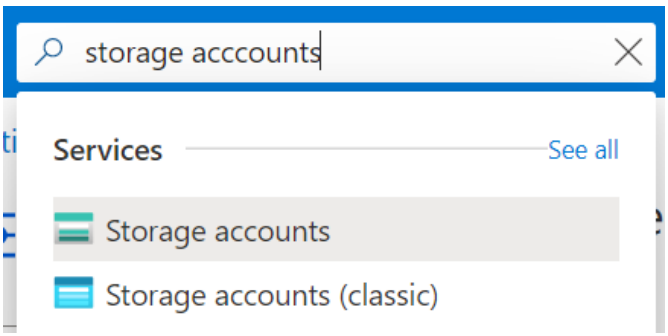


Figure 9.11: Looking for storage accounts in the Azure search bar

   In the resulting pane, click the **+ New** button at the top of the screen as shown in *Figure 9.12*:

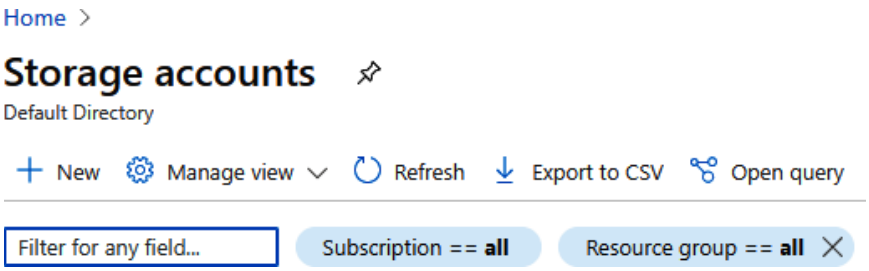


Figure 9.12: Creating a new storage account

Select the `aad-pod-id` resource group you created earlier, give the account a unique name, and select the same region as your cluster. To optimize costs, it is recommended that you select the **Standard** performance, **StorageV2** as the Account kind, and **Locally-redundant storage (LRS)** for Replication, as shown in *Figure 9.13*:

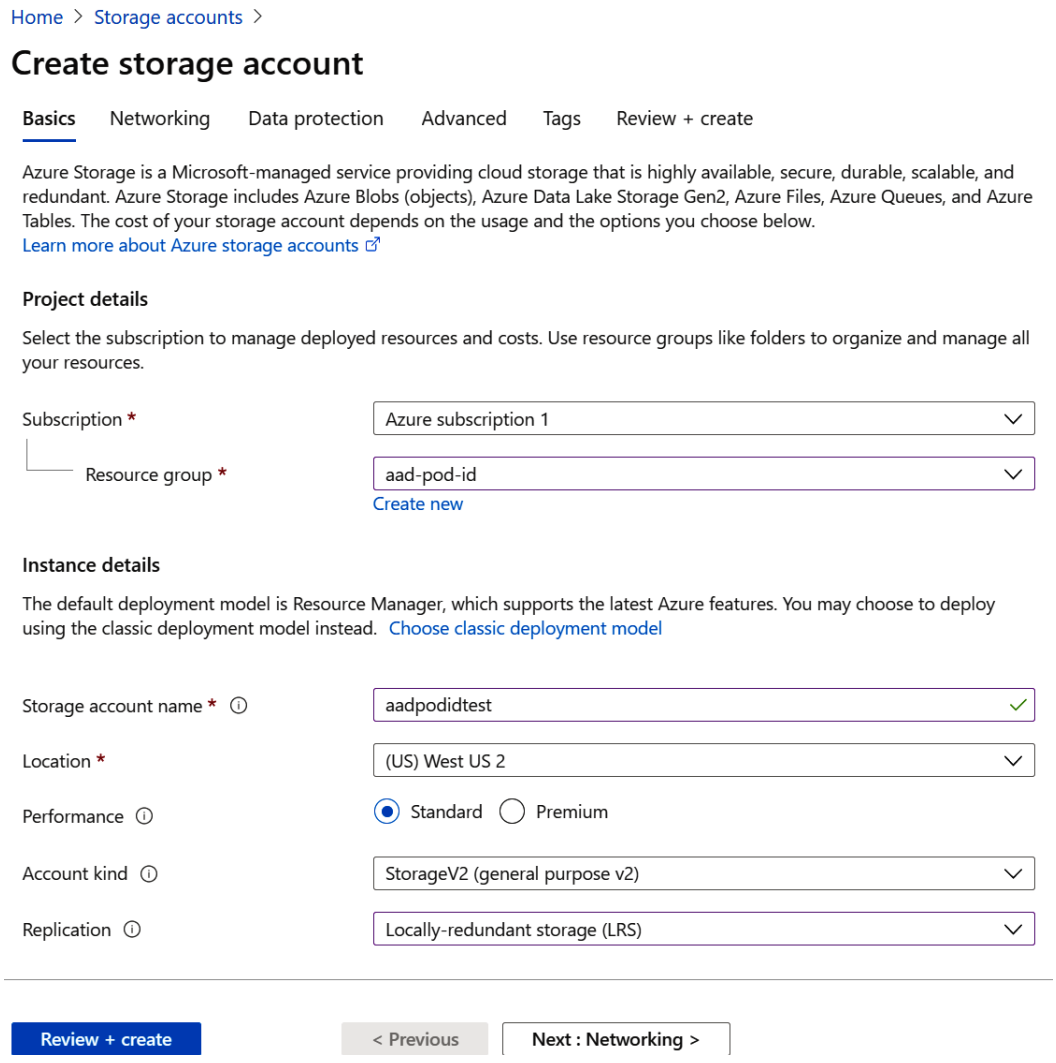


Figure 9.13: Configuring your new storage account

2. After you have provided all the values, click **Review + create** and then the **Create** button on the resulting screen. This will take about a minute to create. Once the storage account is created, click the **Go to resource** button to move on to the next step.
3. First, you will give the managed identity access to the storage account. To do this, click **Access Control (IAM)** in the left-hand navigation bar, click **+ Add** and **Add role assignment**. Then select the **Storage Blob Data Contributor** role, select **User assigned managed identity** in the **Assign access to** dropdown, and select the **access-blob-id** managed identity you created, as shown in *Figure 9.14*. Finally, hit the **Save** button at the bottom of the screen:

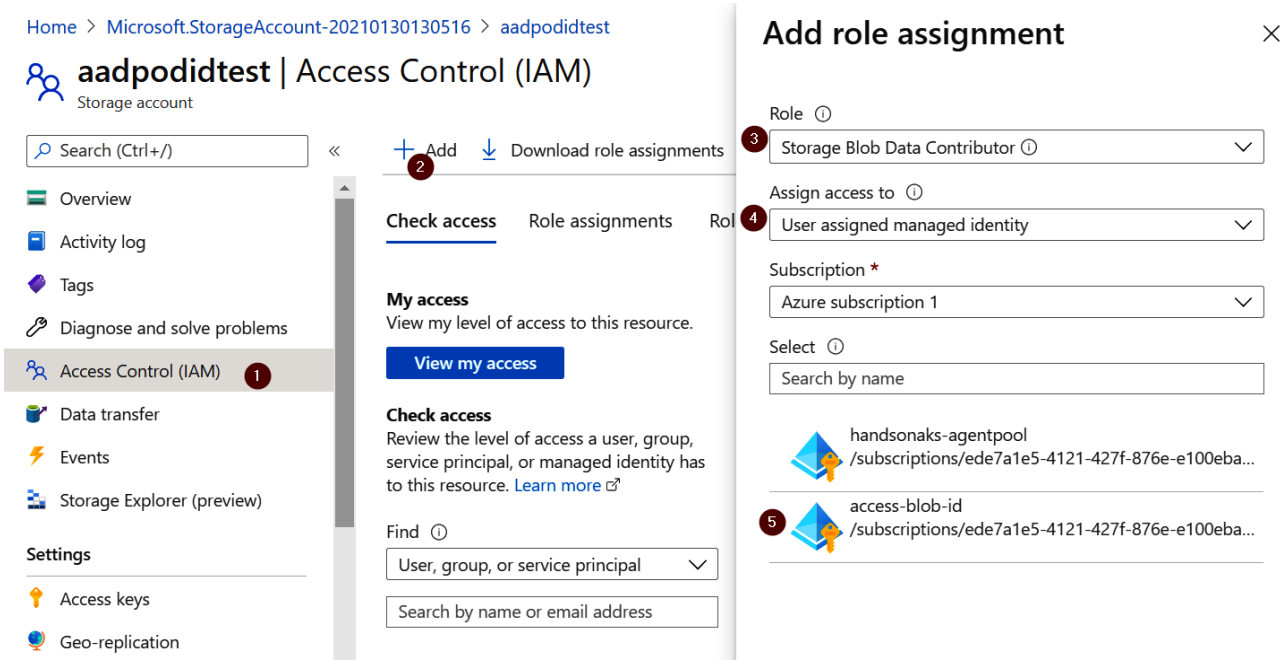


Figure 9.14: Providing access to the storage account for the managed identity

4. Next, you will upload a random file to this storage account. Later, you will try to access this file from within a Kubernetes pod to verify you have access to the storage account. To do this, go back to the **Overview** pane of the storage account. There, click on **Containers**, as shown in *Figure* 9.15:

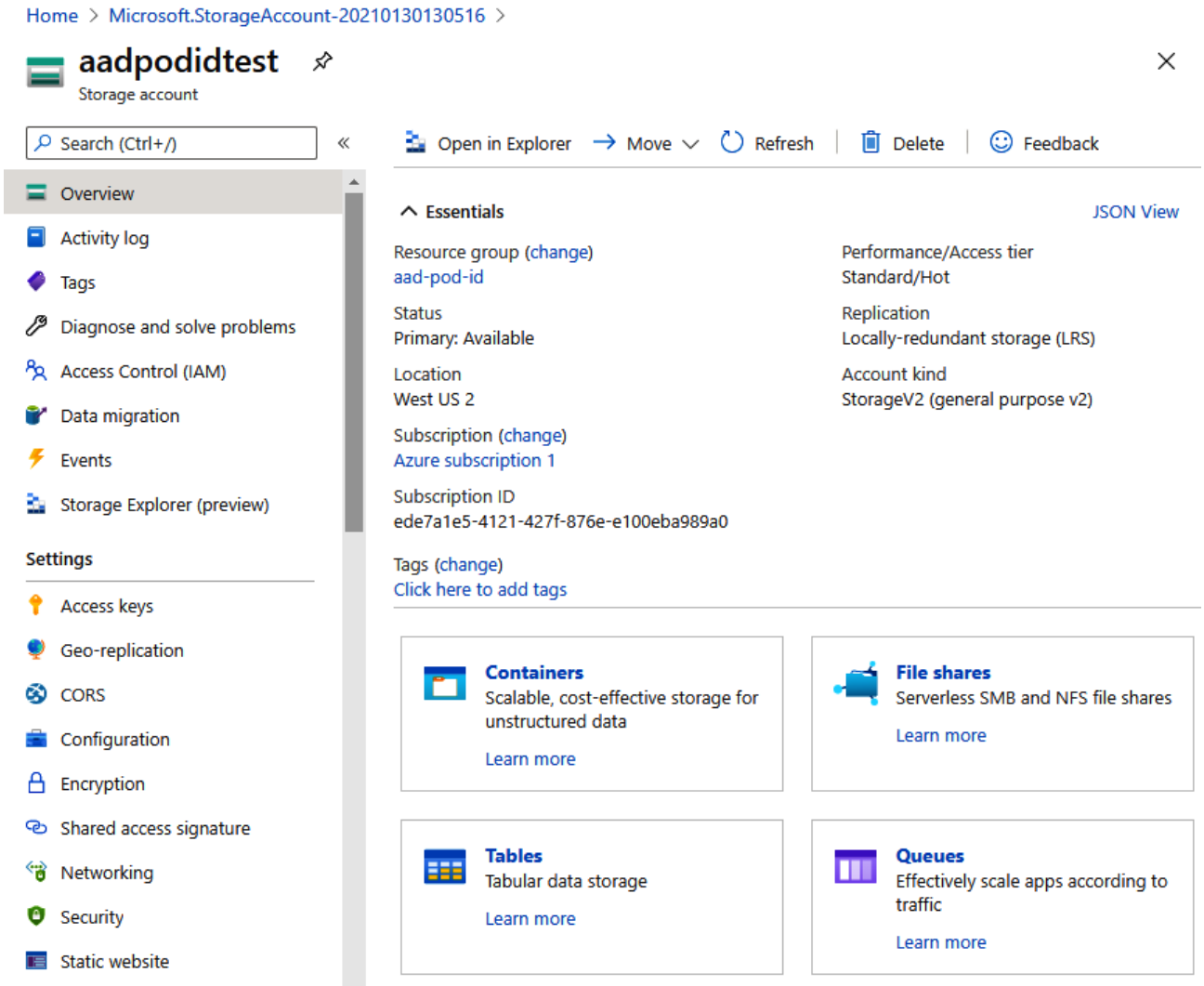


Figure 9.15: Clicking on Containers in the overview pane

5. Then hit the **+ Container** button at the top of the screen. Give the container a name, such as `uploadedfiles`. Make sure to set Public access level to **Private (no anonymous access)**, and then click the **Create** button at the bottom of the screen, as shown in *Figure 9.16*:

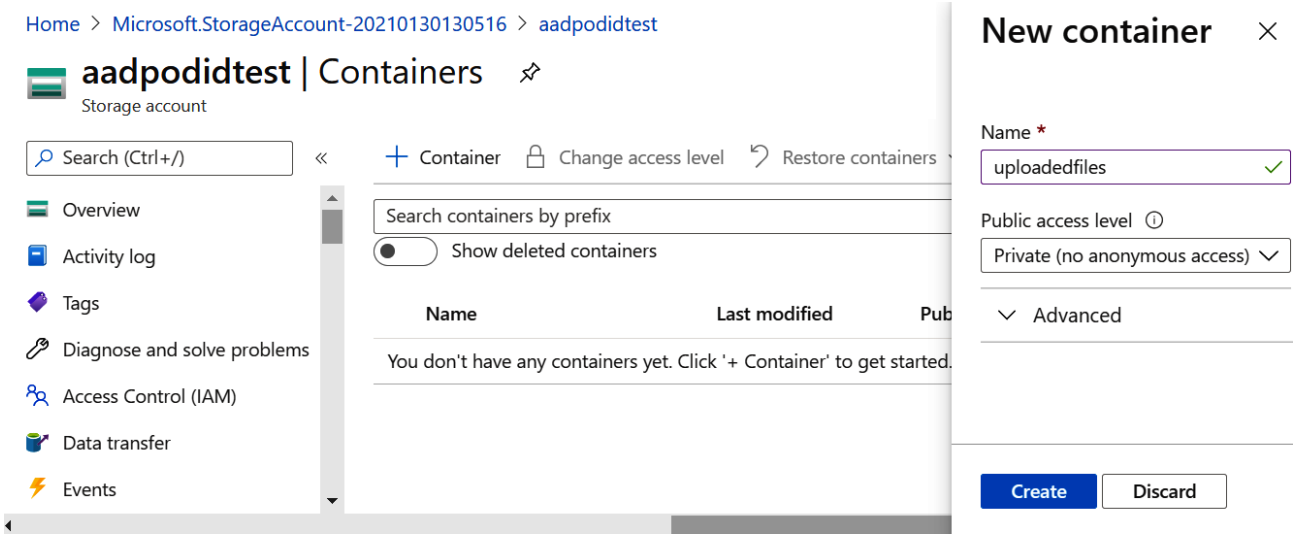


Figure 9.16: Creating a new blob storage container

6. Finally, upload a random file into this storage container. To do this, click on the container name, and then click the **Upload** button at the top of the screen. Select a random file from your computer and click **Upload** as shown in *Figure 9.17*:

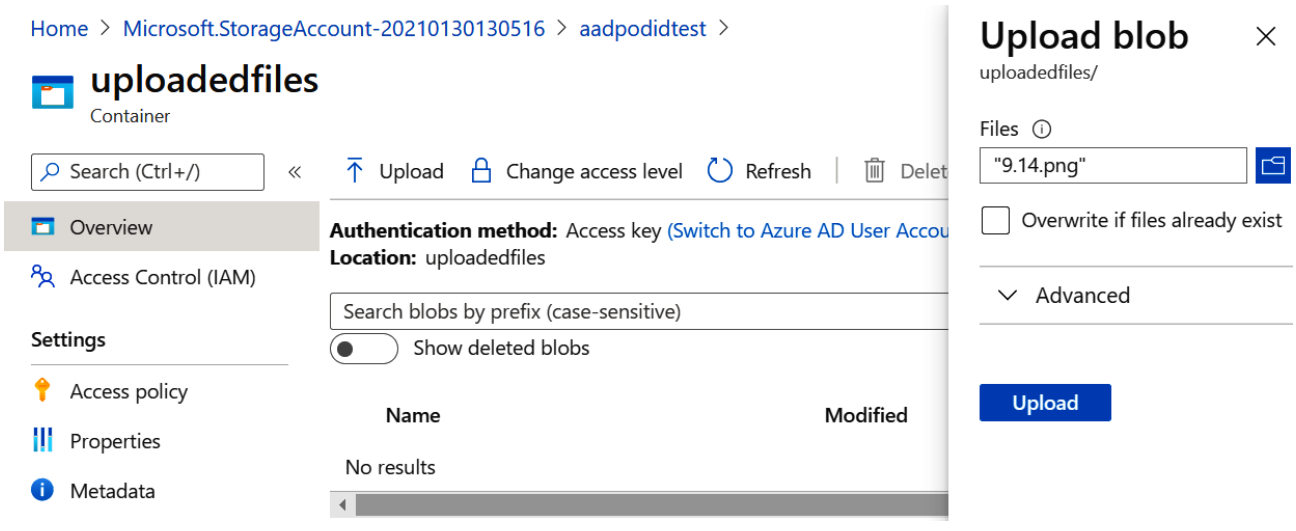


Figure 9.17: Uploading a new file to blob storage

7. Now that you have a file in blob storage, and your managed identity has access to this storage account, you can go ahead and try connecting to it from Kubernetes. To do this, you will create a new deployment using the Azure CLI container image. This deployment will contain a link to the managed identity that was created earlier. The deployment file is provided in the code files for this chapter as `deployment-with-identity.yaml`:

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: access-blob
5   spec:
6     selector:
7       matchLabels:
8         app: access-blob
9     template:
10      metadata:
11        labels:
12          app: access-blob
13          aadpodidbinding: access-blob-id
14      spec:
15        containers:
16        - name: azure-cli
17          image: mcr.microsoft.com/azure-cli
18          command: [ "/bin/bash", "-c", "sleep inf" ]
```

   There are a few things to draw attention to in the definition of this deployment:

   - **Line 13**: This is where you link the pod (created by the deployment) with the managed identity. Any pod with that label will be able to access the managed identity.
   - **Line 16-18**: Here, you define which container will be created in this pod. As you can see, the image (`mcr.microsoft.com/azure-cli`) is referring to the Azure CLI, and you're running a `sleep` command in this container to make sure the container doesn't continuously restart.

8. You can create this deployment using the following command:

```
kubectl create -f deployment-with-identity.yaml
```

9. Watch the pods until the `access-blob` pod is in the **Running** state. Then copy and paste the name of the `access-blob` pod and `exec` into it using the following command:

```
kubectl exec -it <access-blob pod name> -- sh
```

10. Once you are connected to the pod, you can authenticate to the Azure API using the following command. Replace `<client ID of managed identity>` with the client ID you copied earlier:

```
az login --identity -u <client ID of managed identity> \
  --allow-no-subscription -o table
```

This should return you an output similar to *Figure 9.18*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ kubectl exec -it access-blob-5b8dc5bfc6-tknlh -- sh
/ # az login --identity -u ce3b4169-0043-4cb9-abb6-cfafa6ffc446 \
>   --allow-no-subscription -o table
EnvironmentName    HomeTenantId                          IsDefault    Name                 State    TenantId
-----------------  ------------------------------------  -----------  -------------------  -------  ------------------------------------
AzureCloud         1cf4b872-ae04-44c8-8318-2ba43e95f591  True         Azure subscription 1  Enabled  1cf4b872-ae04-44c8-8318-2ba43e95f591
```

Figure 9.18: Logging in to the Azure CLI using the Azure AD pod-managed identity

11. Now, you can try accessing the blob storage account and download the file. You can do this by executing the following command:

```
az storage blob download --account-name <storage account name> \
 --container-name <container name> --auth-mode login \
 --file <filename> --name <filename> -o table
```

This should return you an output similar to *Figure 9.19*:

```
/ # az storage blob download --account-name aadpodidtest \
>  --container-name uploadedfiles --auth-mode login \
>  --file 9.14.png --name 9.14.png -o table
Finished[#############################################################]  100.0000%
Name      Blob Type    Blob Tier    Length    Content Type    Last Modified              Snapshot
--------  -----------  -----------  --------  --------------  -------------------------  ----------
9.14.png  BlockBlob                76133     image/png       2021-01-30T21:20:48+00:00
```

Figure 9.19: Downloading a blob file using the managed identity

12. You can now exit the container using the `exit` command.

13. If you would like to verify that pods that don't have a managed identity configured and cannot download the file, you can use the file called `deployment-without-identity.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: no-access-blob
spec:
  selector:
    matchLabels:
      app: no-access-blob
  template:
    metadata:
      labels:
        app: no-access-blob
    spec:
      containers:
      - name: azure-cli
        image: mcr.microsoft.com/azure-cli
        command: [ "/bin/bash", "-c", "sleep inf" ]
```

    As you can see, this deployment isn't similar to the deployment you created earlier in the chapter. The difference here is that the pod definition doesn't contain the label with the Azure AD pod-managed identity. This means that this pod won't be able to log in to Azure using any managed identity. You can create this deployment using the following:

```
kubectl create -f deployment-without-identity.yaml
```

14. Watch the pods until the `no-access-blob` pod is in the **Running** state. Then copy and paste the name of the `access-blob` pod and `exec` into it using the following command:

```
kubectl exec -it <no-access-blob pod name> -- sh
```

15. Once you are connected to the pod, you can try to authenticate to the Azure API using the following command, which should fail:

```
az login --identity -u <client ID of managed identity> \
  --allow-no-subscription -o table
```

    This should return an output similar to *Figure 9.20*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09$ kubectl exec -it no-access-blob-7db6dcb77c-z4mcv -- sh
/ # az login --identity -u ce3b4169-0043-4cb9-abb6-cfafa6ffc446 \
>   --allow-no-subscription -o table
AzureConnectionError: MSI endpoint is not responding. Please make sure MSI is configured correctly.
Error detail: MSI: Failed to acquire tokens after 12 times
```

Figure 9.20: The new pod cannot authenticate using the managed identity

16. Finally, you can exit the container using the `exit` command.

This has successfully shown you how to use Azure AD pod-managed identities to connect to blob storage from within your Kubernetes cluster. A deployment with an identity label could log in to the Azure CLI and then access blob storage. A deployment without this identity label didn't get permission to log in to the Azure CLI, and hence was also not able to access blob storage.

This has concluded this chapter. Let's make sure to delete the resources you created for this chapter:

```
az aks pod-identity delete --resource-group rg-handsonaks \
  --cluster-name handsonaks --namespace default \
  --name access-blob-id
az group delete -n aad-pod-id --yes
kubectl delete -f
```

You can keep the cluster you created in this chapter since in the next chapter you will use Azure AD pod-managed identities to access Key Vault secrets.

Summary

In this chapter, you've continued your exploration of security in AKS. Whereas *Chapter 8, Role-based access control in AKS*, focused on identities for users, this chapter focused on identities for pods and applications running in pods. You learned about managed identities in Azure and how you can use Azure AD pod-managed identities in Azure to assign those managed identities to pods.

You created a new cluster with the Azure AD pod-managed identities add-on enabled. You then created a new managed identity and linked that to your cluster. In the final section, you gave this identity permissions over a blob storage account and finally verified that pods with the managed identity were able to log in to Azure and download files, but pods without the managed identity couldn't log in to Azure.

In the next chapter, you'll learn more about Kubernetes secrets. You'll learn about the built-in secrets and then also learn how you can securely connect Kubernetes to Azure Key Vault, and even use Azure AD pod-managed identities to do this.

10

Storing secrets in AKS

All production applications require some sensitive information to function, such as passwords or connection strings. Kubernetes has a pluggable back end to manage these secrets. Kubernetes also provides multiple ways of using the secrets in your deployment. The ability to manage secrets and use them properly will make your applications more secure.

You have already used secrets previously in this book. You used them when connecting to the WordPress site to create blog posts in *Chapter 3*, *Application deployment on AKS*, and *Chapter 4*, *Building scalable applications*. You also used secrets in *Chapter 6*, *Securing your application with HTTPS*, when you were configuring the Application Gateway Ingress Controller with TLS.

Kubernetes has a built-in secret system that stores secrets in a semi-encrypted fashion in the default Kubernetes database. This system works well but isn't the most secure way to deal with secrets in Kubernetes. In AKS, you can make use of a project called **Azure Key Vault provider for Secrets Store CSI driver** (**CSI driver**), which is a more secure way of working with Secrets in Kubernetes. This project allows you to store and retrieve secrets in/from Azure Key Vault.

In this chapter, you will learn about the various built-in secret types in Kubernetes and the different ways in which you can create these Secrets. After that, you will install the CSI driver on your cluster, and use it to retrieve Secrets.

Specifically, you will cover the following topics in this chapter:

- Different types of secret in Kubernetes
- Creating and using secrets in Kubernetes
- Installing the Azure Key Vault provider for secrets Store CSI driver
- Using the Azure Key Vault provider for secrets Store CSI driver

Let's start with exploring the different secret types in Kubernetes.

Different secret types in Kubernetes

As mentioned in the introduction to this chapter, Kubernetes comes with a default secrets implementation. This default implementation will store secrets in the etcd database that Kubernetes uses to store all object metadata. When Kubernetes stores secrets in etcd, it will store them in base64-encoded format. Base64 is a way to encode data in an obfuscated manner but is not a secure way of doing encryption. Anybody with access to base64-encoded data can easily decode it. AKS adds a layer of security on top of this by encrypting all data at rest within the Azure platform.

The default secret implementation in Kubernetes allows you to store multiple types of Secrets:

- **Opaque secrets**: These can contain any arbitrary user-defined secret or data.
- **Service account tokens**: These are used by Kubernetes pods for built-in cluster RBAC.
- **Docker config secrets**: These are used to store Docker registry credentials for Docker command-line configuration.
- **Basic authentication secrets**: These are used for storing authentication information in the form of a username and password.
- **SSH authentication secrets**: These are used to store SSH private keys.

- **TLS certificates**: These are used to store TLS/SSL certificates.
- **Bootstrap token Secrets**: These are used to store bearer tokens that are used when creating new clusters or joining new nodes to an existing cluster.

As a user of Kubernetes, you most typically will work with opaque secrets and TLS certificates. You've already worked with TLS secrets in *Chapter 6*, *Securing your application with HTTPS*. In this chapter, you will focus on opaque secrets.

Kubernetes provides three ways of creating secrets, as follows:

- Creating secrets from files
- Creating secrets from YAML or JSON definitions
- Creating secrets from the command line

Using any of the preceding methods, you can create any type of secret.

Kubernetes gives you two ways of consuming secrets:

- Using secrets as an environment variable
- Mounting secrets as a file in a pod

In the next section, you will create secrets using the three ways mentioned here, and you will later consume them using both the methods listed here.

Creating secrets in Kubernetes

In Kubernetes, there are three different ways to create secrets: from files, from YAML or JSON definitions, or directly from the command line. Let's start the exploration of how to create secrets by creating them from files.

Creating Secrets from files

The first way to create secrets in Kubernetes is to create them from a file. In this way, the contents of the file will become the value of the secret, and the filename will be the identifier of each value within the secret.

Let's say that you need to store a URL and a secure token for accessing an API. To achieve this, follow these steps:

1. Store the URL in `secreturl.txt`, as follows:

```
echo https://my-url-location.topsecret.com \
  > secreturl.txt
```

2. Store the token in another file, as follows:

```
echo 'superSecretToken' > secrettoken.txt
```

3. Let Kubernetes create the secret from the files, as follows:

```
kubectl create secret generic myapi-url-token \
  --from-file=./secreturl.txt --from-file=./secrettoken.txt
```

   Please note that you are creating a single secret object in Kubernetes, referring to both text files. In this command, you are creating an opaque secret by using the `generic` keyword.

   The command should return an output similar to *Figure 10.1*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo https://my-url-location.topsecret.com \
>   > secreturl.txt
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'superSecretToken' > secrettoken.txt
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl create secret generic myapi-url-token \
>   --from-file=./secreturl.txt --from-file=./secrettoken.txt
secret/myapi-url-token created
```

Figure 10.1: Creating an opaque secret

4. You can check whether the secrets were created in the same way as any other Kubernetes resource by using the `get` command:

```
kubectl get secrets
```

   This command will return an output similar to *Figure 10.2*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-xphvc   kubernetes.io/service-account-token   3      7h5m
myapi-url-token       Opaque                                2      8s
```

Figure 10.2: List of the created secrets

Here, you will see the secret you just created, and any other secrets that are present in the `default` namespace. The secret is of the `Opaque` type, which means that, from Kubernetes' perspective, the schema of the contents is unknown. It is an arbitrary key-value pair with no constraints, as opposed to, for example, SSH auth or TLS secrets, which have a schema that will be verified as having the required details.

5. For more details about the secret, you can also run the `describe` command:

```
kubectl describe secrets myapi-url-token
```

You will get an output similar to *Figure 10.3*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl describe secrets myapi-url-token
Name:         myapi-url-token
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
secreturl.txt:    38 bytes
secrettoken.txt:  17 bytes
```

Figure 10.3: Description of the created secret

As you can see, neither of the preceding commands displayed the actual secret values.

6. To see the secret's value, you can run the following command:

```
kubectl get -o yaml secrets/myapi-url-token
```

You will get an output similar to *Figure 10.4*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get -o yaml secrets/myapi-url-token
apiVersion: v1
data:
  secrettoken.txt: c3VwZXJTZWNyZXRUb2tlbgo=
  secreturl.txt: aHR0cHM6Ly9teS11cmwtbG9jYXRpb24udG9wc2VjcmV0LmNvbQo=
kind: Secret
metadata:
  creationTimestamp: "2021-01-31T03:32:11Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:secrettoken.txt: {}
        f:secreturl.txt: {}
      f:type: {}
    manager: kubectl-create
    operation: Update
    time: "2021-01-31T03:32:11Z"
  name: myapi-url-token
  namespace: default
  resourceVersion: "59163"
  selfLink: /api/v1/namespaces/default/secrets/myapi-url-token
  uid: 82027703-dda6-449f-a999-7e00f7365662
type: Opaque
```

Figure 10.4: Using the -o yaml switch in kubectl get secret fetches the encoded value of the secret

The data is stored as key-value pairs, with the filename as the key and the base64-encoded contents of the file as the value.

7. The preceding values are base64-encoded. Base64 encoding isn't secure. It obfuscates the secret so it isn't easily readable by an operator, but any bad actor can easily decode a base64-encoded secret. To get the actual values, you can run the following command:

```
echo 'c3VwZXJTZWNyZXRUb2tlbgo=' | base64 -d
echo 'aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHNlY3JldC5jb20K'|
base64 -d
```

You will get the values of the secrets that were originally created:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'c3VwZXJTZWNyZXRUb2tlbgo=' | base64 -d
superSecretToken
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW
9uLnRvcHNlY3JldC5jb20K'| base64 -d
https://my-secret-url-location.topsecret.com
```

Figure 10.5: Base64-encoded secrets can easily be decoded

This shows you that the secrets are not securely encrypted in the default Kubernetes secret store.

In this section, you were able to create a secret containing an example URL with a secure token using files as the source. You were also able to get the actual secret values back by decoding the base64-encoded secrets.

Let's move on and explore the second method of creating Kubernetes secrets, creating secrets from YAML definitions.

Creating secrets manually using YAML files

In the previous section, you created a secret from a text file. In this section, you will create the same secret using YAML files by following these steps:

1. First, you need to encode the secret to `base64`, as follows:

   ```
   echo 'superSecretToken' | base64
   ```

   You will get the following value:

   ```
   c3VwZXJTZWNyZXRUb2tlbgo=
   ```

   You might notice that this is the same value that was present when you got the `yaml` definition of the secret in the previous section.

2. Similarly, for the `url` value, you can get the base64-encoded value, as shown in the following code block:

   ```
   echo 'https://my-secret-url-location.topsecret.com' | base64
   ```

   This will give you the base64-encoded URL:

   ```
   aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHNlY3JldC5jb20K
   ```