

Kubernetes objects on Microsoft Azure

An introduction to deployment options based on a simple ASP.NET Core web application

By Mahesh Kshirsagar
Azure Customer Advisory Team (AzureCAT)

January 2018

Contents

Overview	4
In this guide	5
Sample code	5
PART 1 GET STARTED WITH KUBERNETES	6
Set up a local Kubernetes environment.....	6
Enable a hypervisor	7
Install kubectl	8
Install Minikube	9
Validate the prerequisite installation	9
Open the Kubernetes dashboard from Minikube	10
Start a local Kubernetes cluster.....	11
Troubleshoot deployment.....	12
Set up Kubernetes on Azure.....	12
Create a Kubernetes cluster on Container Service	12
Manage a Kubernetes cluster on Azure.....	14
Open the Kubernetes dashboard for an Azure-based cluster	15
PART 2 WORK WITH THE KUBERNETES DASHBOARD	18
Switch context between a local cluster and an Azure cluster.....	18
Tour the Kubernetes dashboard	20
Admin	20
Namespaces	20
Nodes	20
Persistent volume	21
Storage classes	21
Roles	21
Workloads.....	21
Deployments	22
Replica sets	22
Replication controllers.....	22
Daemon sets.....	22
Stateful sets	23
Jobs	23
Pods	23
Services and discovery.....	23
Ingresses	24

Services	24
Storage and config	24
Persistent volume claims	24
ConfigMap	24
Secrets	24
Summary of key Kubernetes resources for workloads	25
PART 3 DEPLOY THE TIERS	26
Considerations for SQL Server back-end deployment	26
Create a secret	27
Create a storage class	27
Create a persistent volume claim	27
Create a service	28
Create a stateful set	29
Create the database	30
Verify data persistence	31
Considerations for the ASP.NET Core web front-end deployment	33
Install Heapster with InfluxDB and Grafana	33
Change the grafana.yaml file	33
Create services	35
Use the Horizontal Pod Autoscaler	36
Create deployment	36
Create service	37
Create horizontalPodAutoScaler	38
Troubleshoot issues	38
Verify how operations scale	38
Verify the application	41
Verify the database	42
Conclusion	42

Authored by Mahesh Kshirsagar. Edited by Ed Price. Reviewed by Brendon Burns, Wasim Bloch, and Kimmo Forss.

© 2018 Microsoft Corporation. This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Overview

More people are turning to Kubernetes as a container orchestrator—a rapidly growing trend according to search statistics.¹ As more enterprises adopt architectures based on containers or microservices, they also need to adopt a container orchestrator, and they need options for managing their containerized workloads. Many developers start with Docker, an excellent ecosystem for learning to work with containers. But Kubernetes pushes the boundaries with its exhaustive set of options for managing containerized workloads. The sheer number of options can be overwhelming, especially when you’re getting started.

This paper attempts to demystify Kubernetes by focusing on a real-life scenario in which a basic tiered application is deployed using pods and controllers. Step by step, using a sample application, you’ll see how to choose the right [Kubernetes objects](#) for running workloads in Azure. Kubernetes provides multiple deployment objects, each with its unique advantages that are explained in the context of a sample two-tier workload on Kubernetes. The concepts you learn by deploying this workload can easily be extended to multitier architecture implementations.

The workload discussed in this paper consists of an application with a web front end running ASP.NET Core 1.0 (with some ASP.NET SignalR) shown in Figure 1. The back end consists of a SQL Server container running on Linux.

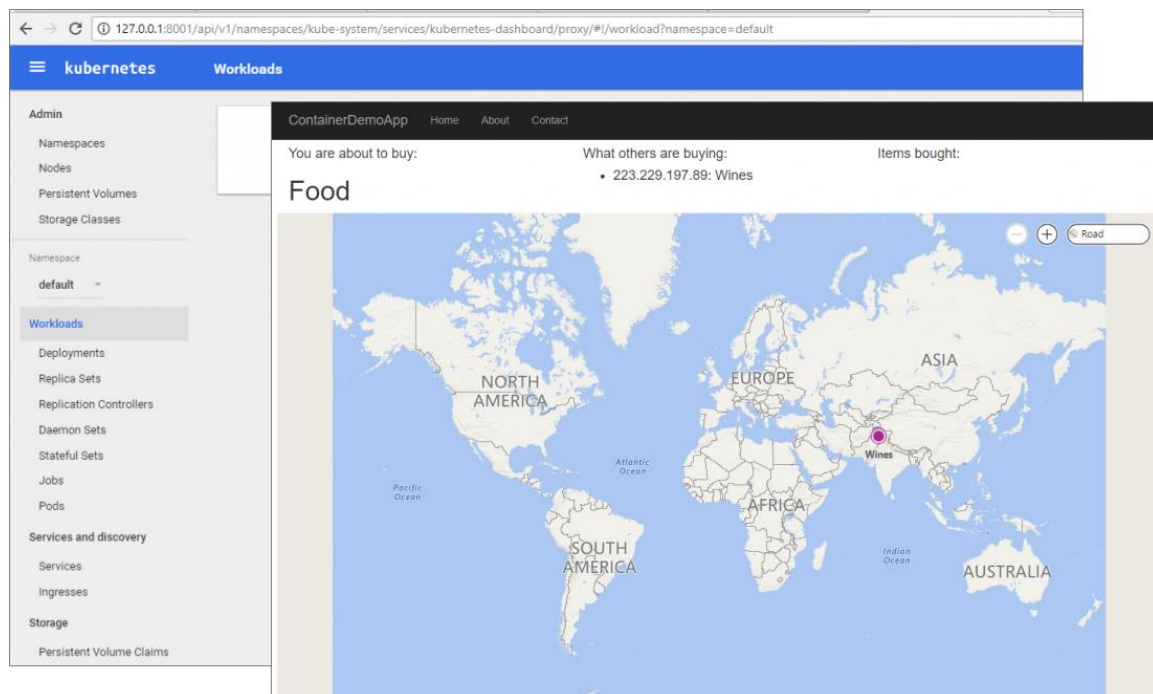


Figure 1. Use the sample app to see how to use Azure Container Service with Kubernetes.

¹ Check out Google Trends for “Kubernetes” vs. “Docker Swarm” at <https://trends.google.co.uk/trends/explore?q=kubernetes,docker%20swarm>.

In this guide

This paper has three parts that follow the typical order you might use to start up and learn Kubernetes:

- In Part 1, get started with Kubernetes, first by setting up a local development environment, then a Kubernetes cluster on Azure using [Azure Container Service](#). Learn about deployment options and process.
- In Part 2, get a high-level overview of key Kubernetes concepts through a tour of the dashboard, the Kubernetes web interface. Learn how to switch between clusters and work with the dashboard options.
- In Part 3, walk through a sample deployment and learn how to choose the right deployment objects. Set up a SQL Server back end and ASP.NET Core web front end.

The content for this paper was originally written as a series of [blogs](#).

Sample code

This guide demonstrates the deployment of a sample application, ContainerDemoApp. You can copy the source code from [GitHub](#):

Resource	URL
ASP.NET Core application	https://github.com/Mahesh-MSFT/KubernetesDemoApp
SQL Server back-end creation script	https://github.com/Mahesh-MSFT/KubernetesDemoApp/tree/master/sqlscript
Kubernetes SQL Server StatefulSet creation manifest	https://github.com/Mahesh-MSFT/KubernetesDemoApp/blob/master/k8smanifests/sqlserver-backend.yaml
Kubernetes ASP.NET Core HorizontalPodAutoScaler manifest	https://github.com/Mahesh-MSFT/KubernetesDemoApp/blob/master/k8smanifests/aspnetcore-frontend.yaml



PART 1

GET STARTED WITH KUBERNETES

In a typical development scenario, developers start on their local computers before deploying to a remote cluster, most likely in the cloud. Part 1 shows the basics you need to set up a Kubernetes cluster.

First, you'll set up the development environment for Kubernetes on your local computer, then start a local Kubernetes cluster. Next, you'll get started on Azure and set up a Kubernetes cluster there.

Set up a local Kubernetes environment

A local development environment for Kubernetes has three prerequisites:

- **Hypervisor** enables virtualization, which is the foundation on which all container orchestrators operate, including Kubernetes.
- **Kubectl** is the Kubernetes command-line tool used to manage the Kubernetes cluster.
- **Minikube** is a tool that runs a single-node Kubernetes cluster inside a virtual machine on your development computer.

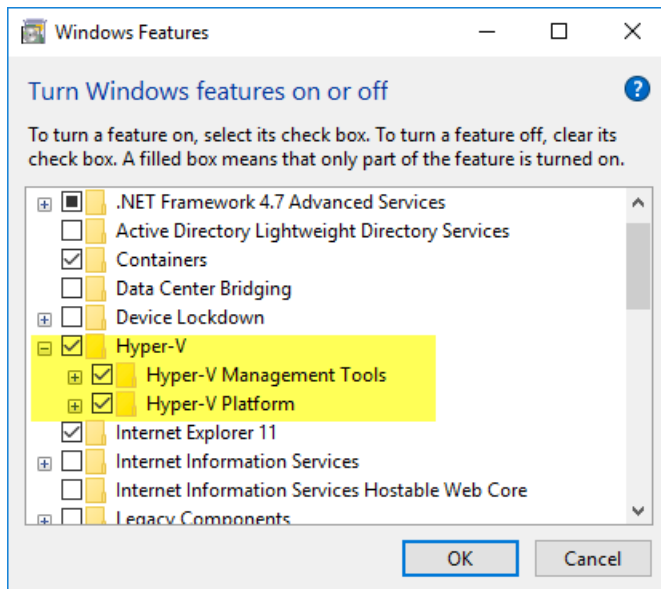
Enable a hypervisor

This guide uses Hyper-V as the hypervisor. On many Windows 10 versions, Hyper-V is already installed—for example, on 64-bit versions of Windows Professional, Enterprise, and Education in Windows 8 and later. It is not available on Windows Home edition.

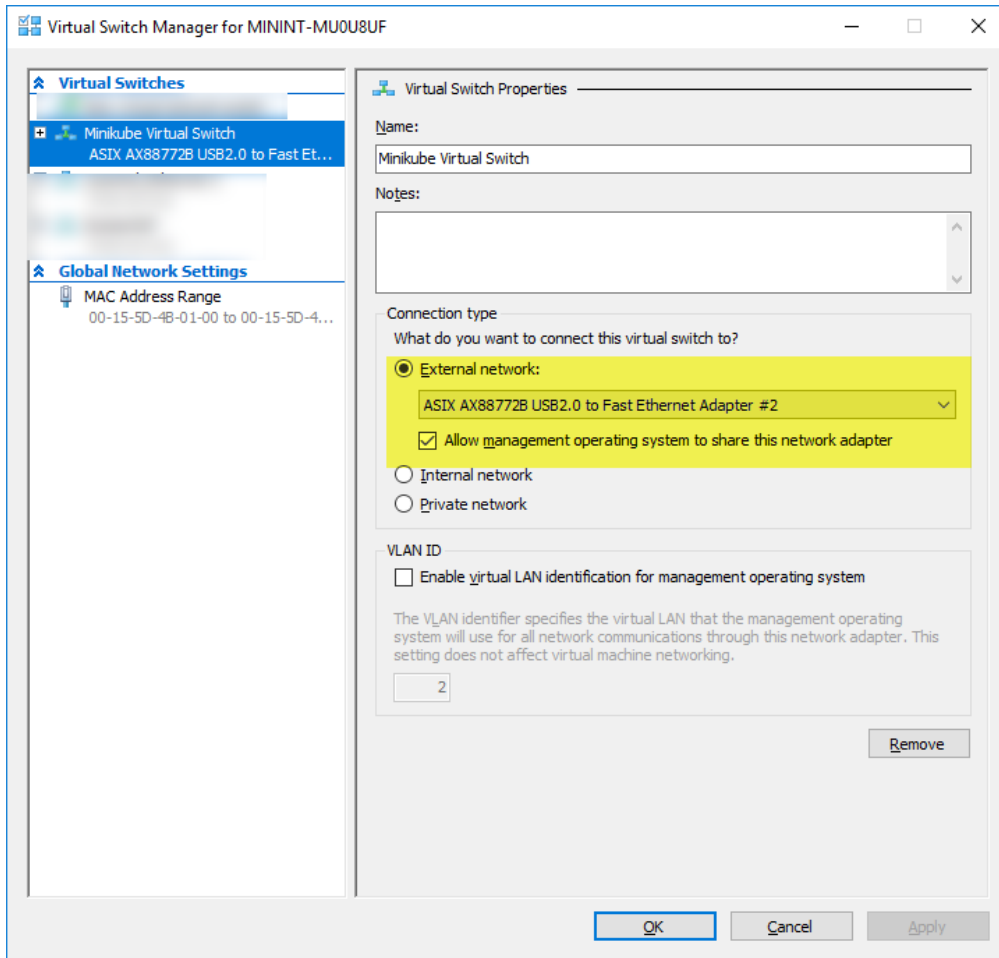
NOTE: If you're running something other than Windows 10 on your development platforms, another hypervisor option is to use [VirtualBox](#), a cross-platform virtualization application. For a list of hypervisors, see "Install a Hypervisor" on the [Minikube](#) page of the Kubernetes documentation.

To enable Hyper-V manually on Windows 10 and set up a virtual switch:

1. Right-click the Windows button and select **Apps and Features** to open the **Windows Features** dialog box.



2. Select the **Hyper-V** check boxes, then click **OK**.
3. To set up a virtual switch, type *hyper* in the Windows Start menu, then select **Hyper-V Manager**.
4. In Hyper-V Manager, select **Virtual Switch Manager**.
5. Select **External** as the type of virtual switch.
6. Select the **Create Virtual Switch** button.
7. Ensure that the **Allow management operating system to share this network adapter** checkbox is selected.



Install kubectl

In this step, you install kubectl from the Kubernetes site and set it as an environment variable.

1. Go to [Install and Set Up kubectl](#) in the Kubernetes documentation and follow the instructions for Windows to copy the executable (.exe) file to your C: directory.
2. In the Windows Search box, type *View advanced system settings* and click on the result from the Control Panel.
3. In the **Advanced** tab of the **System Properties** dialog box, click the **Environment Variables** button.
4. In the **System variables** frame of the **Environment Variables** dialog box, select the row with Path as a variable name, then select the **Edit** button.
5. In the **Edit System Variable** dialog box, click **New**, then add that path to where you copy and pasted the kubectl.exe. Click **OK**.
6. Click **OK** in the **Environment Variables** dialog box.
7. Click **OK** in the **System Properties** dialog box.

Install Minikube

Like kubectl, you set up Minikube by copying the latest Minikube executable (.exe) file and making it an environment variable.

1. Go to the [kubernetes/minikube GitHub page](#) and download Minikube. For the examples shown in this guide, the file was downloaded to the same location (C:\) as the kubectl file.
2. Rename the downloaded .exe file to *minikube.exe*.
3. Add the path to minikube.exe in the **System variables** frame of the **Environment Variables** dialog box as you did for kubectl.

Validate the prerequisite installation

It is easy to verify an installation by running the version command. For Minikube, use the following command:

```
minikube version
```

For kubectl, use this command:

```
kubectl version
```

The first time you run this command, no cluster has been set up yet, so errors are generated for "Server Version." Ignore these. Figure 2 compares the output from minikube and kubectl.

```
C:\WINDOWS\system32>minikube version
minikube version: v0.20.0
```

```
C:\WINDOWS\system32>kubectl version
Client Version: version.Info{Major:"1", Minor:"7", GitVersion:"v1.7.0", GitCommit:"d3ada0119e776222f11ec7945e6d860061339aad", GitTreeState:"clean", BuildDate:"2017-06-29T23:15:59Z", GoVersion:"go1.8.3", Compiler:"gc", Platform:"windows/amd64"}
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.6", GitCommit:"7fa1c1756d8bc963f1a389f4a6937dc71f08ada2", GitTreeState:"clean", BuildDate:"2017-06-16T18:21:54Z", GoVersion:"go1.7.6", Compiler:"gc", Platform:"linux/amd64"}
```

Figure 2. Using the *version* command in minikube and kubectl.

Next, run the following command to verify the latest version of Kubernetes that is available for installation:

```
Minikube get-k8s-versions
```

Figure 3 shows that the latest Kubernetes version available for installation is V1.7.0.

```
C:\WINDOWS\system32>minikube get-k8s-versions
The following Kubernetes versions are available:
- v1.7.0
- v1.7.0-rc.1
- v1.7.0-alpha.2
- v1.6.4
- v1.6.3
- v1.6.0
- v1.6.0-rc.1
- v1.6.0-beta.4
- v1.6.0-beta.3
- v1.6.0-beta.2
- v1.6.0-alpha.1
- v1.6.0-alpha.0
- v1.5.3
- v1.5.2
- v1.5.1
- v1.4.5
- v1.4.3
- v1.4.2
- v1.4.1
- v1.4.0
```

Figure 3. List of the latest Kubernetes versions.

Open the Kubernetes dashboard from Minikube

When running Minikube, access the dashboard using the `minikube dashboard` command (Figure 4). A browser window opens with the Kubernetes dashboard (Figure 5). For details about using the dashboard, see [Part 2: Work with the Kubernetes dashboard](#).

```
C:\WINDOWS\system32>minikube dashboard
Opening kubernetes dashboard in default browser...
```

Figure 4. The minikube dashboard command.

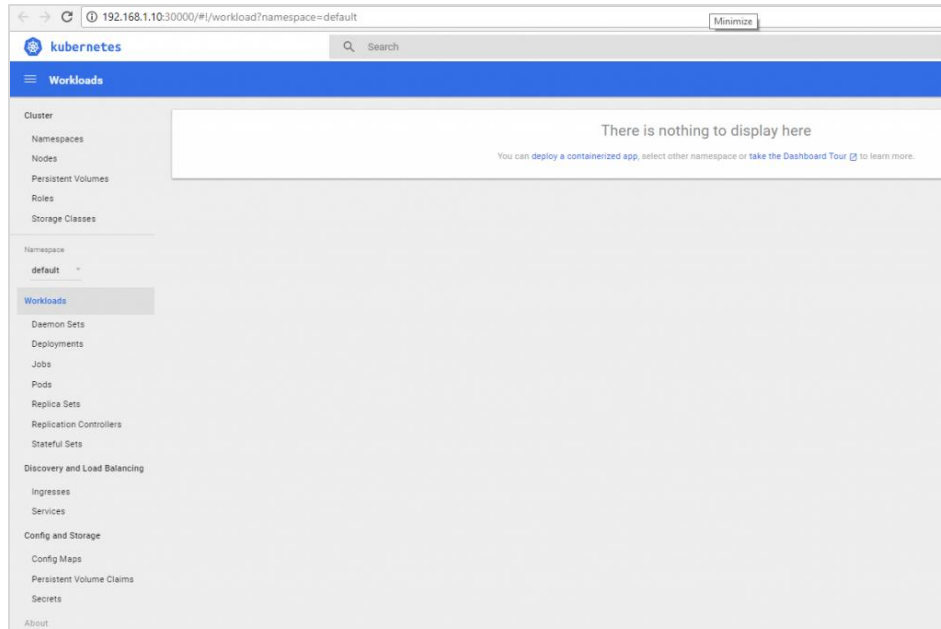


Figure 5. Kubernetes dashboard for an empty local cluster.

Start a local Kubernetes cluster

Now that you know the latest Kubernetes version, use Minikube to start a one-node Kubernetes cluster. Run the following command:

```
Minikube start --kubernetes-version="v1.7.0" --vm-driver="hyperv" --hyperv-virtual-switch="Minikube Virtual Switch" --logtostderr
```

NOTE: The screenshots in this document show a Windows 10 setup that includes a Hyper-V virtual switch.

After a while, the following verification message appears:

```
Connecting to cluster...
Setting up kubeconfig...
I0717 17:55:36.791316    9800 config.go:75] Using kubeconfig: C:\Users\maksh\.kube/config
Kubectl is now configured to use the cluster.
```

Figure 6. Verification message after starting a cluster.

To verify the Minikube status, run the `minikube status` command as Figure 7 shows.

```
C:\WINDOWS\system32>minikube status
minikube: Running
localkubernetes: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.1.6
```

Figure 7. The `minikube status` command.

Now use kubectl to verify status. Run the cluster-info and get nodes commands as Figure 8 shows.

```
C:\WINDOWS\system32>kubectl cluster-info
Kubernetes master is running at https://192.168.1.6:8443

C:\WINDOWS\system32>kubectl get nodes
NAME          STATUS    AGE      VERSION
minikube      Ready     9m       v1.7.0
```

Figure 8. Getting the kubectl status.

Troubleshoot deployment

If you encounter deployment issues, follow these steps:

1. Ensure that you are running the command prompt in Administrator mode.
2. Stop Minikube by running minikube.stop command followed by the minikube.delete command.
3. Stop and delete the Minikube virtual machine in Hyper-V Manager.
4. Delete the .kube and .minikube folders in the HOME path (generally found at C:\Windows\Users\<your-user-id>).

Set up Kubernetes on Azure

For the sample deployment described in this document, [Container Service](#) is used to deploy and manage a Kubernetes cluster in the cloud. Container Service enables you to run containerized workloads on Azure. You can choose among multiple container orchestrators, including Mesos DC/OS, Docker Swarm, and Kubernetes.

NOTE: In October 2017, Microsoft [announced](#) a managed Kubernetes service called AKS that offers new deployment options and other benefits. This guide uses Container Service, which has been available since 2015 and features an Azure-backed SLA. However, in the setup steps below, you can substitute the equivalent [AKS](#) command for the Container Service command.

Other Azure services support containers, including [Azure Service Fabric](#), [Azure App Service](#), and [Azure Container Instances](#). The focus here is on Container Service because its of support for Kubernetes.

Create a Kubernetes cluster on Container Service

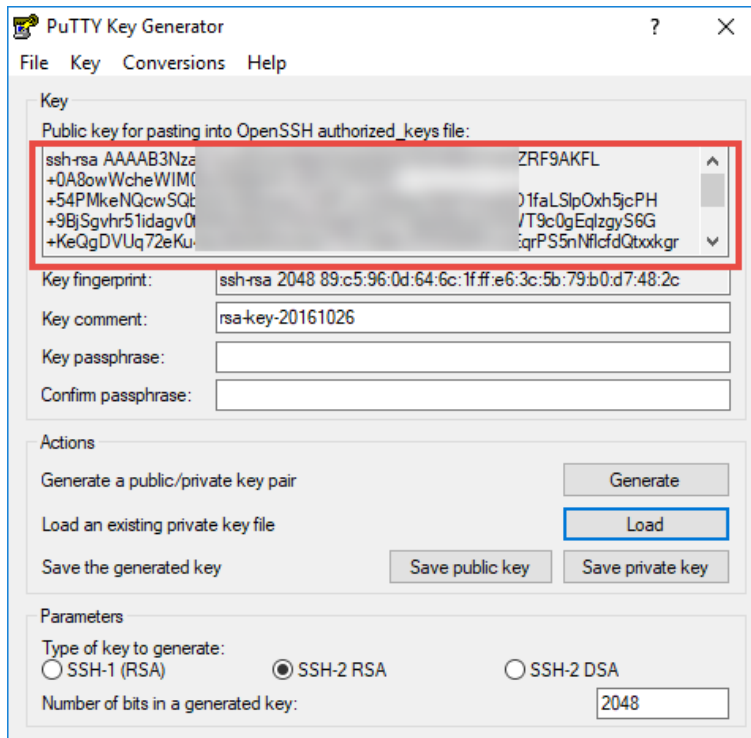
You can create a Kubernetes cluster on Container Service through the Azure portal or with [Azure CLI \(command-line interface\)](#). For detailed instructions based on Azure CLI, see [Deploy Kubernetes cluster for Linux containers](#).

A common practice is to use the switch --generate-ssh-keys to autogenerate SSH keys, but many enterprises prefer to have their operations or security teams generate and provide SSH keys to the development teams. This paper uses the latter approach instead of autogenerating SSH keys.

To support this approach, use the `--ssh-key-file` switch. For details about the various switches available when creating a Kubernetes cluster, see the [az acs](#) command reference.

NOTE: Due to a current [known issue](#), please compact the SSH public key by completing the following procedure on Windows 10 using PuTTY.

1. Open PuTTYGen and load a private SSH key.



2. Copy the public key, create a new text file, paste the public key there and save the file. Note that there are no comments at the beginning or the end of this public key, which you would generally see if you open it as-is.
3. Use the `az acs create` command with the full path of this file to create a new Kubernetes cluster as follows:

```
az acs create --orchestrator-type=kubernetes --resource-group acs-kuber-rg
--name=acs-kuber-cluster --ssh-key-value="<full-path-name>"
```

```
C:\WINDOWS\system32>az acs create --orchestrator-type=kubernetes --resource-group acs-kuber-rg
--name=acs-kuber-cluster --ssh-key-value="C:\Users\maksh\AppData\Local\VirtualStore\Program
Files (x86)\PuTTY\acs-swarm-pub-key-txt.txt"
```

4. After a while, the following message appears, indicating the successful deployment of a Kubernetes cluster on Azure:

```
C:\WINDOWS\system32>az acs create --orchestrator-type=kubernetes --resource-group acs-kuber-r
g --name=acs-kuber-cluster --ssh-key-value="C:\Users\maksh\AppData\Local\VirtualStore\Program
Files (x86)\PuTTY\acs-swarm-pub-key-txt.txt"
waiting for AAD role to propagate.done
[| Finished ..
  "id": "/subscriptions/5dd3998d-b447-44b5-884a-2da7751e365a/resourceGroups/acs-kuber-rg/prov
iders/Microsoft.Resources/deployments/azurecli1500461262.346693529231",
  "name": "azurecli1500461262.346693529231",
  "properties": {
    "correlationId": "a972b798-db2f-4d60-92c9-7e5b7e455f08",
    "debugSetting": null,
    "dependencies": [],
    "mode": "Incremental",
    "outputs": null,
    "parameters": {
      "clientSecret": {
        "type": "SecureString"
      }
    }
  }
}
```

Manage a Kubernetes cluster on Azure

Kubectl (the Kubernetes command-line interface) is used to manage the Kubernetes cluster in Azure. To interact with an Azure-based Kubernetes cluster, run the [get-credentials](#) command:

```
az acs kubernetes get-credentials --resource-group=acs-kuber-rg
--name=acs-kuber-cluster
```

When the command completes, run the `config current-context` command to see if the current context of kubectl is pointing to the Azure cluster:

```
kubectl config current-context acs-kuber-cluster-ac-kuber-rg-5dd399
```

After you confirm that the current context is pointing to the Azure cluster, run the `get nodes` command to get more details about the nodes (Figure 9).

```
C:\WINDOWS\system32>kubectl get nodes
```

NAME	STATUS	AGE	VERSION
k8s-agent-10e054b7-0	Ready	26m	v1.6.6
k8s-agent-10e054b7-1	Ready	26m	v1.6.6
k8s-agent-10e054b7-2	Ready	26m	v1.6.6
k8s-master-10e054b7-0	Ready,SchedulingDisabled	26m	v1.6.6

Figure 9. Node status message.

These nodes correspond to virtual machines on Azure. As Figure 10 shows, the names match.

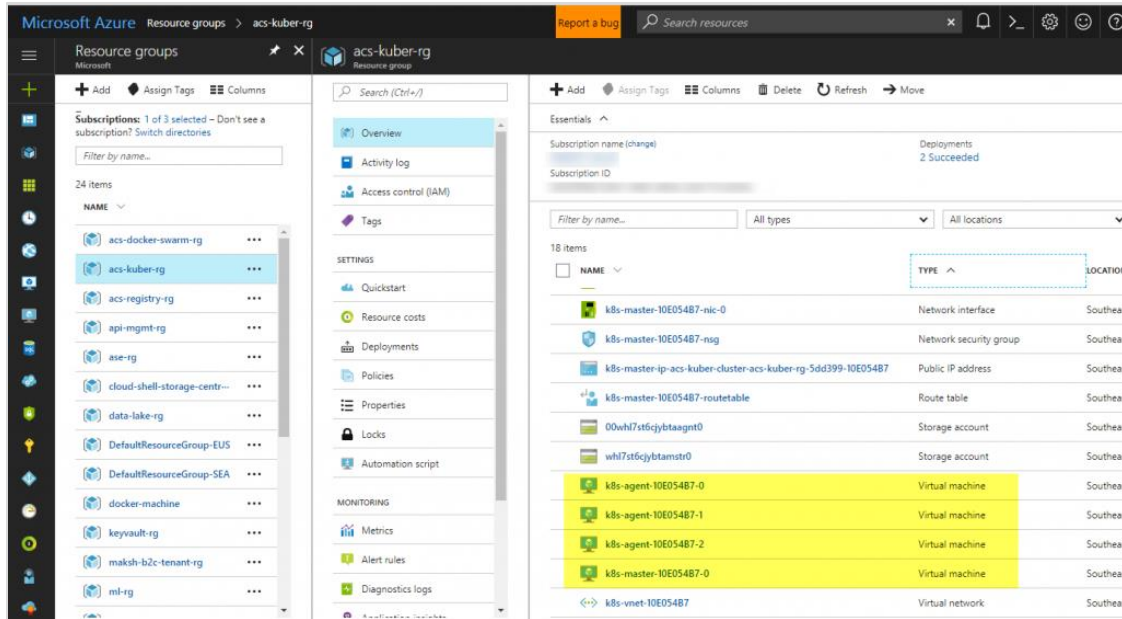


Figure 10. Nodes and corresponding virtual machines.

To identify the various endpoints of the Kubernetes cluster on Azure, run the `cluster-info` command (Figure 11).

```
C:\WINDOWS\system32>kubectl cluster-info
Kubernetes master is running at https://...a.cloudapp.azure.com
Heapster is running at https://...a.cloudapp.azure.com/api/v1/namespaces/kube-system/services/heapster/proxy
KubeDNS is running at https://...a.cloudapp.azure.com/api/v1/namespaces/kube-system/services/kube-dns/proxy
kubernetes-dashboard is running at https://...a.cloudapp.azure.com/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy
```

Figure 11. Identifying the endpoints of the Kubernetes cluster on Azure.

Open the Kubernetes dashboard for an Azure-based cluster

To access the Kubernetes dashboard of the Azure-based cluster, run the `kubectl proxy` command. It returns an IP address and port on which various services are available on proxy.

```
C:\WINDOWS\system32>kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Figure 12. Getting the IP address for the dashboard with the `kubectl proxy` command.

When you go to that address, you'll see a list of all the endpoints of the Azure-based Kubernetes cluster (Figure 13).


 A screenshot of a web browser window. The address bar shows the URL "127.0.0.1:8001". The main content area displays a JSON object representing a list of API paths. The JSON is as follows:


```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/apps",
    "/apis/apps/v1beta1",
    "/apis/authentication.k8s.io",
    "/apis/authentication.k8s.io/v1",
    "/apis/authentication.k8s.io/v1beta1",
    "/apis/authorization.k8s.io",
    "/apis/authorization.k8s.io/v1",
    "/apis/authorization.k8s.io/v1beta1",
    "/apis/autoscaling",
    "/apis/autoscaling/v1",
    "/apis/autoscaling/v2alpha1",
    "/apis/batch",
    "/apis/batch/v1",
    "/apis/batch/v2alpha1",
    "/apis/certificates.k8s.io",
    "/apis/certificates.k8s.io/v1beta1",
    "/apis/extensions",
    "/apis/extensions/v1beta1",
    "/apis/policy",
    "/apis/policy/v1beta1",
    "/apis/rbac.authorization.k8s.io",
    "/apis/rbac.authorization.k8s.io/v1alpha1",
    "/apis/rbac.authorization.k8s.io/v1beta1",
    "/apis/settings.k8s.io",
    "/apis/settings.k8s.io/v1alpha1",
    "/apis/storage.k8s.io",
    "/apis/storage.k8s.io/v1",
    "/apis/storage.k8s.io/v1beta1",
    "/healthz",
    "/healthz/ping",
    "/healthz/poststarthook/bootstrap-controller",
    "/healthz/poststarthook/ca-registration",
    "/healthz/poststarthook/extensions/third-party-resources",
    "/logs",
    "/metrics",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}
```

Figure 13. List of endpoints.

To see the dashboard for the Azure-based Kubernetes cluster, add the **/ui** path to the address—for example, <http://127.0.0.1:8001/ui>. The Kubernetes dashboard opens (Figure 14), and you can now access the cluster through this browser interface.

For details about using the dashboard, see [Part 2: Work with the Kubernetes dashboard](#).

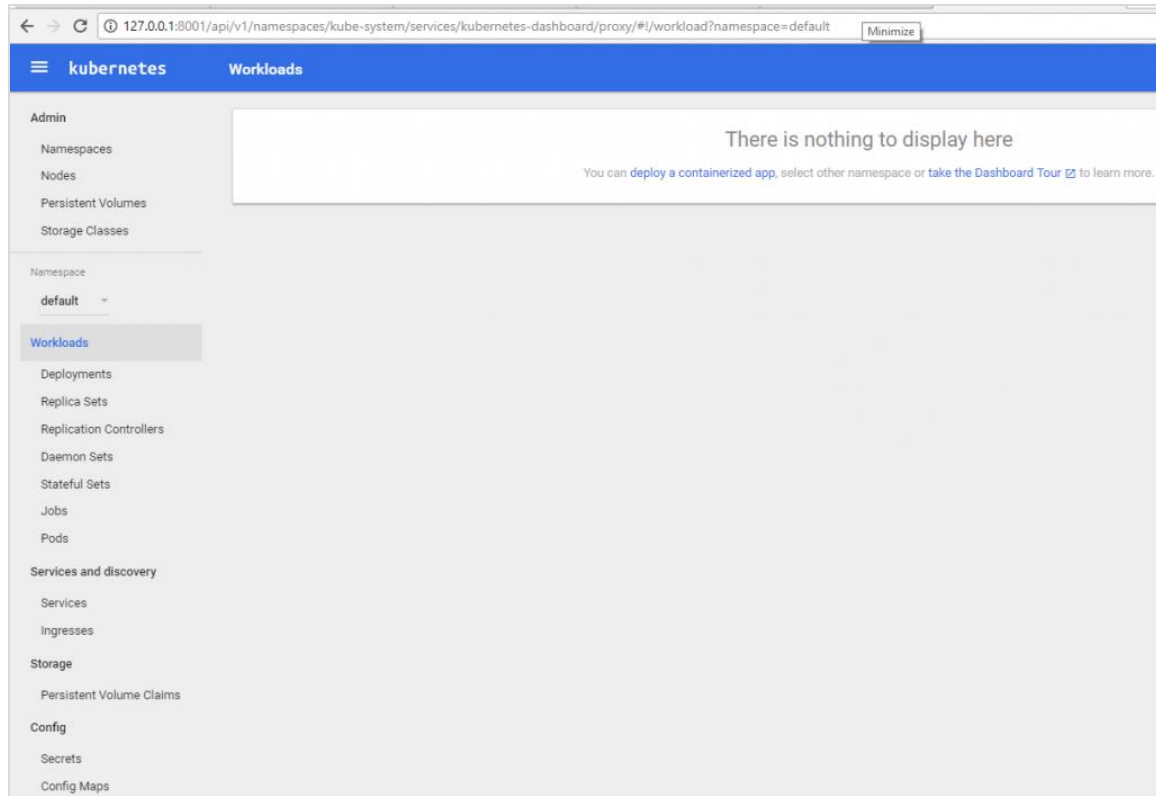


Figure 14. Kubernetes dashboard for Azure-based cluster showing the welcome page of an empty cluster.



PART 2

WORK WITH THE KUBERNETES DASHBOARD

Kubernetes gives you a multitude of options, and the dashboard helps you discover them. First, you need to know how to switch between your local clusters and your clusters on Azure. Kubernetes calls this switching *context*.

This section offers a high-level overview of the dashboard and introduces you to common options for getting information about your clusters.

Switch context between a local cluster and an Azure cluster

In earlier sections, `kubectl` is used to interact with a local and cloud-based cluster through the `cluster-info` and `proxy` commands. To identify a cluster, `kubectl` uses context. Think of context as a variant of the database *connectionString* property, but for a Kubernetes cluster.

`Kubectl` issues commands against a specific cluster using the context. To see which contexts are available to receive commands, use the `config view` command as shown in Figure 15.

```

C:\WINDOWS\system32>kubect1 config view
apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: REDACTED
  server: https://<redacted>.cloudapp.azure.com
  name: acs-kuber-cluster-ac-s-kuber-rg-5dd399
- cluster:
  certificate-authority: C:\Users\maksh\.minikube\ca.crt
  server: https://192.168.1.10:8443
  name: minikube
contexts:
- context:
  cluster: acs-kuber-cluster-ac-s-kuber-rg-5dd399
  user: acs-kuber-cluster-ac-s-kuber-rg-5dd399-admin
  name: acs-kuber-cluster-ac-s-kuber-rg-5dd399
- context:
  cluster: minikube
  user: minikube
  name: minikube
current-context: acs-kuber-cluster-ac-s-kuber-rg-5dd399

```

Figure 15. See the available contexts.

The `kubect1 config view` command lists *clusters*, *configs*, and shows the *current config*.

Use the command `kubect1 config use-context <name-of-context>` to switch between contexts. For example:

```
kubect1 config use-context minikube
```

When you are connected to a specific context, you can issue further commands that work on that cluster. For example, Figure 16 shows the `cluster-info` command.

```

C:\WINDOWS\system32>kubect1 cluster-info
Kubernetes master is running at https://192.168.1.10:8443

```

Figure 16. Getting information about a context using the `cluster-info` command.

Tour the Kubernetes dashboard

After switching to the appropriate context, use either the `kubectl proxy` or `minikube dashboard` command to navigate to the Kubernetes dashboard (Figure 17).

This section walks through the dashboard to acquaint you with the options needed to manage your clusters on Azure.

You'll learn the key concepts needed to work with containerized workloads. All the Kubernetes objects you create are accessed from the dashboard.

The organization of the next sections follows the order of the items on the dashboard menu.

Admin

This view is for cluster and namespace administrators and includes Namespaces, Nodes, Persistent Volume, Storage Classes, and Roles (V1.7.0 only).

Namespaces

A *namespace* is a virtual cluster within a physical cluster and offers a great solution for running multitenant applications. Physical cluster resources can be distributed across multiple namespaces using policies and resource quotas. Each namespace is isolated from other namespaces.

You can also use a namespace to isolate multiple areas within an application or microservice. Each application or microservice can be deployed within namespaces, and access to it can be controlled.

Each Kubernetes cluster has at least two built-in namespaces. These are *default* and *kube-system*. You can also create custom namespaces, but each time any operation or command is issued on any component within a custom namespace, you must qualify it with a namespace. This extra step is time-consuming when operating a large number of custom namespaces. This situation can be remedied by creating users and credentials that are unique to the namespaces.

Nodes

A *node* is a host within a Kubernetes cluster. Either physical or virtual servers, nodes keep the Kubernetes cluster up and running. Each node runs multiple components, such as *kubelet* and *kube proxy*.

Nodes play a key role in cluster capacity planning, high-availability, security, problem detection

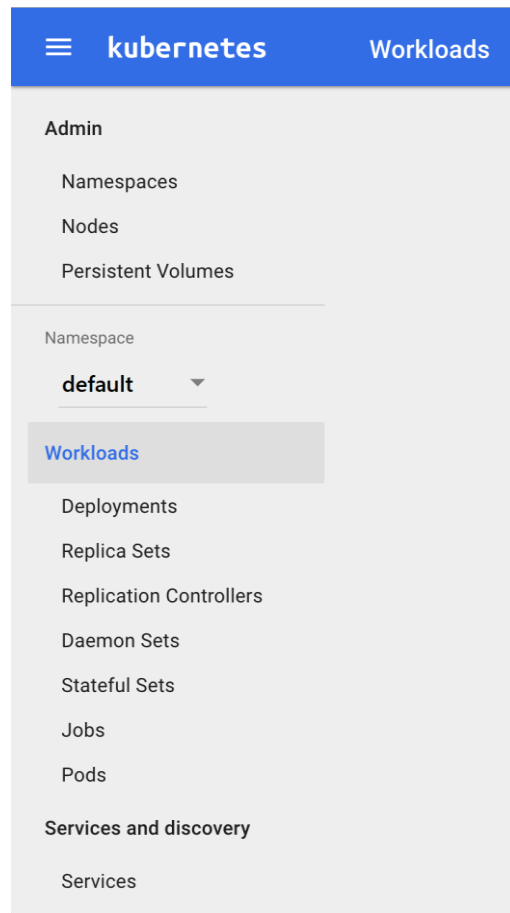


Figure 17. Menu of the Kubernetes dashboard.

and resolution, and monitoring. Nodes are managed by *master*, a collection of components, such as API server, scheduler, and controller manager. Master itself can run on one of the nodes and should be distributed across nodes for redundancy. During capacity planning, use an odd number of nodes) so you can form a quorum and have higher redundancy and reliability.

Depending upon the workload you plan to run on the Kubernetes cluster, it's a good idea to have multiple node types. These node types can include—but are not restricted to—CPU, memory, or IOPS-intensive. Node types can be labelled, so when it is time to run a containerized workload, you can easily identify and select a suitable node type to run a workload on.

Persistent volume

While nodes represent the compute capacity of a Kubernetes cluster, a *persistent volume* represents its storage capacity. To understand why persistent volume is important, consider how container orchestration works. Containers and cluster nodes are transient by nature. When they become unresponsive or fail, the orchestrator handles it—Kubernetes creates a new container or a *pod*, a basic building block that encapsulates a single container or multiple containers.

In such a volatile environment, storage might be seen as a problem. How do you keep data in storage when the container or pod is transient and nodes can fail? Kubernetes offers options for both persistent and nonpersistent volumes:

- Persistent volumes are resources that need to be provisioned separately from the Kubernetes cluster. Kubernetes can use these resources but does not manage them. They need to be managed by the cluster administrator. In Azure, you can use [Azure Disk](#) or [Azure File Storage](#) as a persistent volume with a Kubernetes cluster.
- Nonpersistent volumes include the *emptyDir* and *HostPath* options to enable communication between pods and between nodes respectively.

Storage classes

Storage classes give you the flexibility to choose multiple volume options for a Kubernetes cluster. You can use multiple storage solutions from various vendors, including Azure, Amazon Web Service, and Google Cloud Platform.

Containers *request* storage from a cluster via a *volume claim*, which is a declarative claim for a specific type of volume access mode, capacity, and so on. The Kubernetes cluster evaluates this claim request, and then assigns a volume from its storage class.

Roles

The roles feature enables role-based access (RBAC) to cluster resources. RBAC replaces the attribute-based access control (ABAC) used prior to Kubernetes 1.6.0.

You can use roles to grant access to resources within the scope of a namespace or a cluster, including multiple namespaces. Roles address authorization concerns about accessing API Server, which provides core services to manage Kubernetes services that use RESTful API endpoints. Roles and users are associated using *RoleBinding*.

Workloads

The dashboard shows all the applications running in a selected workspace in the workloads view.

To see the status and relationship between objects, choose one of the details views for the type of

workload: deployments, replica sets, replication controllers, daemon sets, jobs, pods, and stateful sets.

Deployments

This view shows workload deployments. In Kubernetes, a *deployment* is a resource that ensures the reliable rollout of an application. A deployment creates a *replica set* that is used to manage a group of pods of the same type. If a pod crashes, another one is automatically created through the replica set.

It is possible to create an individual pod without using a deployment controller, but the pod will lack the mechanism needed to recreate itself without any manual intervention. That makes the process unsustainable.

Deployments uses a declarative syntax to define the replicas that ensure the many instances of a pod are always running. They are also a useful feature to trigger a version upgrade.

Replica sets

This dashboard view shows the replica sets used to manage a group of pods of the same type. Replica sets ensure that a specific number of pods are always running. They support *set-based label selectors*, a type of selection that enables a set of pods to be grouped so that the entire set is available for an operation defined in the ReplicaSet controller.

If a pod crashes, a replica set restores it. This functionality is certainly useful, but there are not many use cases—pods are often version-updated. In these cases, the expectation is to update the pod and ensure that a specific number of pods is always kept running.

The recommendation is to use replica sets along with higher-level controllers such as deployment. Replica sets and deployments can also be used with the HorizontalPodAutoscaler controller, a related resource that provides an auto-scale functionality for pods.

Replication controllers

Replication controllers are the predecessors to replica sets. Unlike replica sets, they operate on *name equality*. The official recommendation going forward is to use the ReplicaSet controller.

Daemon sets

This dashboard view shows daemon sets, which are a bit like the demons in movies that never die no matter how many times they are killed, and they play a versatile role in a Kubernetes cluster. A DaemonSet is a pod that runs on every node. If that pod crashes, Kubernetes tries to bring it back up automatically. Similarly, any new node that gets added to cluster automatically gets a DaemonSet pod.

This functionality is very useful in scenarios where every node needs to be monitored for a potential problem. For those scenarios, you can use the [node-problem-detector](#) component, which is deployed as a DaemonSet.

You can also use daemon sets to scale a container using the **nodeSelector** node in their template. This node instructs Kubernetes to run a container on a qualifying node. So, when any new node gets added in the cluster and qualifies for the **nodeSelector** criteria (typically via labelling), it automatically runs a container specified in the DaemonSet template.

Another use case is to label a set of nodes with a particular hardware configuration that is

dedicated to persistent storage, and run stateful workloads as a daemon set on those labelled nodes. It's also possible to use the `.spec.updateStrategy.type` node in the DaemonSet template for rolling updates.

Stateful sets

This dashboard view shows the workload API object called StatefulSets that were designed to solve the problems associated with data loss when pods crash. Stateful sets use persistent storage volume. By comparison, ReplicaSet or ReplicationControllers help ensure that a specific number of pods are always running, and they simply spin up a new pod instance when a pod crashes. With a crash, any data that is written to the pod volume disappears as well. A new pod gets its own new volume—prohibitive behavior in some workloads, such as distributed databases and stateful applications.

Stateful sets can be configured to mount a persistent storage volume. So even if a pod crashes, data is preserved on persistent data storage. The StatefulSets object also has a unique and stable hostname that can be queried through DNS. They ensure that pods are named (such as pod-1, pod-2, and so on). A new pod instance (such as pod-2) is not created until pod-1 is created and is healthy.

Jobs

This dashboard view shows the jobs that are running. Jobs are a Kubernetes resources that create one or more pods and ensure that they run until they succeed. Jobs are ideal for tasks that run and achieve a goal, and then stop.

You can customize jobs with restart policies, completions, and parallelism. Internally, jobs make sure the optimum number of pods are created to execute parallel operations.

By design, when a job completes its execution, it and its pods are kept for telemetry purposes. If any of the logging information is not needed, a job and its related pod should be *manually* deleted. Cron jobs are a special type of jobs that run on schedule either as a single instance or repeatedly on a schedule.

Pods

This dashboard lists the status of all pods, the Kubernetes building block that simplifies deployment, scaling, and replication. A pod encapsulates one or more containers and can also include storage, an IP address, and a configuration option for managing each container.

Storage within a pod is shared among containers and is mounted as a volume on all the containers inside it. All containers in a pod can communicate by using either `localhost` or inter-process communication.

Pods can be created as a Kubernetes resource. However, they lack self-healing capabilities. So they are usually created using *controllers* (DaemonSets, Deployment, StatefulSets).

Services and discovery

This dashboard view shows the Kubernetes resources that can expose services and the pods targeted by them.

Ingresses

An ingress is a layer 7 HTTP load balancer. Ingresses are Kubernetes resources that expose one or more services to the outside world. An ingress provides externally visible URLs to services and load-balance traffic with SSL termination. This resource is useful in scenarios where rolling out such services is not possible or is expensive.

Ingresses can also be used to define network routes between namespaces and pods in conjunction with network policies. They are managed using ingress controllers, which can limit requests, redirect URLs, and control access. For details, see the [Ingress](#) topic in the Kubernetes documentation.

Services

A service is a layer 4 TCP load balancer. They are a way to expose an application functionality to users or other services. A service encompasses one or more pods and can be *internal* or *external*. Internal services are accessed only by other services or jobs in a cluster. An *external* service is marked by the presence of either NodePort or load balancer. For details, see the [Services](#) topic in the Kubernetes documentation.

Storage and config

This dashboard view shows resources used for persistent storage and configuration of applications running in clusters.

Persistent volume claims

Persistent volume claims are storage specifications. You can use external storage providers, including Azure Disk and Azure Files, that match required storage specification. Containers claim their storage needs expressed by persistent storage claims. They typically specify the following:

- **Capacity:** Expressed in Gibibytes (GiB).
- **Access mode:** Determines a container's rights on volume when mounted. Supports one of following options: ReadOnlyMany, ReadWriteOnce, and ReadWriteMany.
- **Reclaim policy:** Determines what happens with storage when a claim is deleted.

ConfigMap

ConfigMap helps keep image configuration options separate from containers and pods. It lists configuration options as a key-value pair. This configuration information is exposed as an environment variable. When creating pods, a pod template can read the values for ConfigMap and can provision a pod. In similar fashion, it can also be used to provision volume.

Secrets

ConfigMap provides an easy configuration separation, but it stores a key-value pair as plaintext—not the ideal solution for sensitive information such as passwords and keys. This is where secrets come in handy.

Secrets are scoped at namespaces, so their access is restricted to users for that namespace. Secrets can be mounted as a volume in pod so they can be used to consume information. Note that secrets are stored in plaintext in an etcd cluster (a data store used by Kubernetes to store its state). Access to etcd is restricted.

Summary of key Kubernetes resources for workloads

The following table provides a quick summary of the resources described in this section and when you might use them.

Resource	Notes
Pod	<ul style="list-style-type: none"> • Good starting point. • Simple way to deploy containers on Kubernetes. • Not to be used for deploying an application. Simplicity is an advantage, but pods lack autohealing and other features.
Deployment	<ul style="list-style-type: none"> • Solves problems associated with pods. • Uses the replicas value to ensure that a specified number of pods are always running, even if there are pod failures. • Uses ReplicaSet in the background for pod creation, updates, and deletions.
HorizontalPodAutoscaler	<ul style="list-style-type: none"> • Provides an autoscale functionality for pods. • Typically used in a front-end web application where the number of instances (or pods) needs to be based on resource utilization (such as CPU or memory). • Helpful when the deployment controller is not suitable (based on keeping a specified number of pods running), and scale needs to be a workload-dependent range.
DaemonSet	<ul style="list-style-type: none"> • Good choice if functionality is tied to a physical node of a Kubernetes cluster. For example, node resource utilization and node monitoring. • Helps offset infrastructure concerns.
StatefulSets	<ul style="list-style-type: none"> • Used for a workload when data loss is unacceptable. • Useful when you need replicas for availability or scalability, and the replicas need stable volume mounts across restarts. • Often used when deploying database workloads that store their state on external storage providers. With StatefulSets, a crash of the pod or cluster does not result in data loss.



PART 3

DEPLOY THE TIERS

It can be a challenge to select the right deployment resource for a given workload. To show which choices to make when deploying a tiered workload on Kubernetes, this section uses the [sample](#) Kubernetes demo application:

- A back-end SQL Server database used to store the app's data.
- A front-end web UI for a simple ASP.NET Core app called Food.

The goal here is to show how to use the resources from Part 2 and demonstrate an approach you can extend to multitier deployments.

Considerations for SQL Server back-end deployment

This section covers running a SQL Server database on Linux in Kubernetes. Running SQL Server as a container gives you the chance to work with Kubernetes resources, including pods, Deployment controllers, HorizontalPodAutoscaler, DaemonSet, and StatefulSet.

In this sample scenario, if a cluster crashes, data loss is unacceptable for the database files (.mdf, .ndf, and .ldf). The Kubernetes StatefulSet deployment object allows you to use external persistent

storage. This scenario uses Azure Disk Storage as the storage provider, but [other storage solutions](#) can be used.

To deploy a SQL Server database on Azure Storage on Kubernetes, complete the following steps.

Create a secret

Running SQL Server as a container involves passing at least two environment variables: `ACCEPT_EULA` and `SA_PASSWORD`. The latter is obviously a secret.

To create a Kubernetes secrets resource, use the Kubernetes manifest. As Figure 18 shows, `sapassword` is declared as a secret with its value specified as a base64-encoded value of a real password.

```
apiVersion: v1
kind: Secret
metadata:
  name: sqlsecret
type: Opaque
data:
  sapassword: [REDACTED]
```

Figure 18. Creating a secret.

Create a storage class

The **StorageClass** type provides the specifications of the external persistent storage for a SQL Server database. Figure 19 uses Azure Disk Storage as an external persistent storage.

Ensure that `storageAccount` is in the same Azure resource group as the Kubernetes cluster.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurestorageclass
provisioner: kubernetes.io/azure-disk
parameters:
  skuName: Standard_LRS
  location: southeastasia
  storageAccount: [REDACTED]
```

Figure 19. Creating a StorageClass.

Create a persistent volume claim

The **PersistentVolumeClaim** type ensures that data is stored on an external storage device outside the Kubernetes cluster. It refers to **StorageClass** created earlier. It also specified `accessMode` and the `storage resource` requirement—10 GiB as shown in Figure 20.

```

apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "sqlldb-pv"
  annotations:
    volume.beta.kubernetes.io/storage-class: "azurestorageclass"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "10Gi"

```

Figure 20. Creating a PersistentVolumeClaim.

Create a service

When a database runs as a pod, it needs to include fixed endpoints (protocol, port, and IP address) that can be called. Kubernetes Service objects provide this abstraction. Unlike a pod, a service is a set of pods identified by a selector. Running as a Service object decouples the service from the way it is accessed by other pods or applications.

Figure 21 shows how a service is created in the manifest.

Two notable specifications in Figure 21 are **type: LoadBalancer** and **selector**. The LoadBalancer service type provides a load-balanced endpoint to connect with SQL Server. In most cases, this is desirable when multiple applications, including the ones that run outside a cluster, are expected to connect with SQL Server. These applications can connect to SQL Server using the load balancer IP and port number as server name (Figure 22).

```

apiVersion: v1
kind: Service
metadata:
  name: sqlservice
  labels:
    app: sqlservice
spec:
  type: LoadBalancer
  ports:
    - port: 1433
      targetPort: 1433
  selector:
    app: sqllinux

```

Figure 21. Creating a service.

In some scenarios, SQL Server is expected to be accessed only by services or pods that are running inside a cluster. In such cases, a service can be run as a [headless service](#). The **selector** option identifies a set of pods that run together as a service. The **sqlservice** service looks for pods with the label **app: sqllinux** and groups them together to form a service.

Figure 22. Connecting the applications to SQL Server.

Create a stateful set

StatefulSet uses all the components created earlier. Figure 23 shows a template to create it.

- 1) **Service:** Under **spec: serviceName**.
The same service created earlier.
- 2) **Replicas:** The number of pods to be created.
- 3) **Labels:** Identify the pods that will make up the service. The value of **app: sqlinux** should be same in both **Service** and **StatefulSet**.
- 4) **Secret:** Sets environment variable **SA_PASSWORD**.
- 5) **volumeMounts:** Specifies directory on pod that will be mounted on external storage.
- 6) **persistentVolumeClaim:** Sets the persistent volume claim defined earlier.

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: sqlserverstatefulset
spec:
  serviceName: sqlservice 1
  replicas: 1 2
  template:
    metadata:
      labels:
        app: sqlinux 3
    spec:
      containers:
        - name: sqlinux
          image: microsoft/mssql-server-linux
          env:
            - name: SA_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: sqlsecret 4
                  key: sapassword
            - name: ACCEPT_EULA
              value: "Y"
          ports:
            - containerPort: 1433
          volumeMounts:
            - mountPath: /var/opt 5
              name: sqldb-home
      volumes:
        - name: sqldb-home
          persistentVolumeClaim:
            claimName: sqldb-pv 6
```

Figure 23. Creating the StatefulSet resource.

Combining these pieces and creating a YAML file to run yields the result shown in Figure 24.

```
C:\>kubectl create -f "c:\Users\maksh\Documents\Visual Studio 2017\Projects\ContainerDemoApp\k8smanifests\sqlserver-backend.yaml"
secret "sqlsecret" created
storageclass "azurestorageclass" created
persistentvolumeclaim "sqldb-pv" created
service "sqlservice" created
statefulset "sqlserverstatefulset" created
```

Figure 24. YAML file.

Create the database

Record the service's external endpoint IP, PortNo, from the Kubernetes dashboard by clicking **Services** (item 1 in Figure 25), locating **sqlservice** (2), and noting **External endpoints** (3).

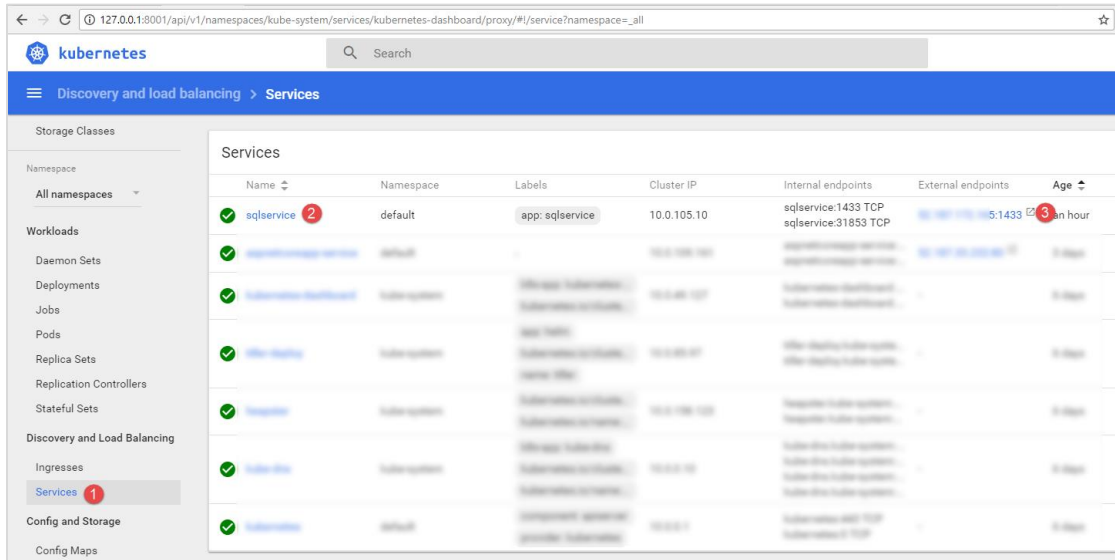


Figure 25. Finding Kubernetes Services (1), the `sqlservice` service (2), and endpoint (3).

Use that IP address and the SA password to connect from SQL Server Management Studio as shown in Figure 26.

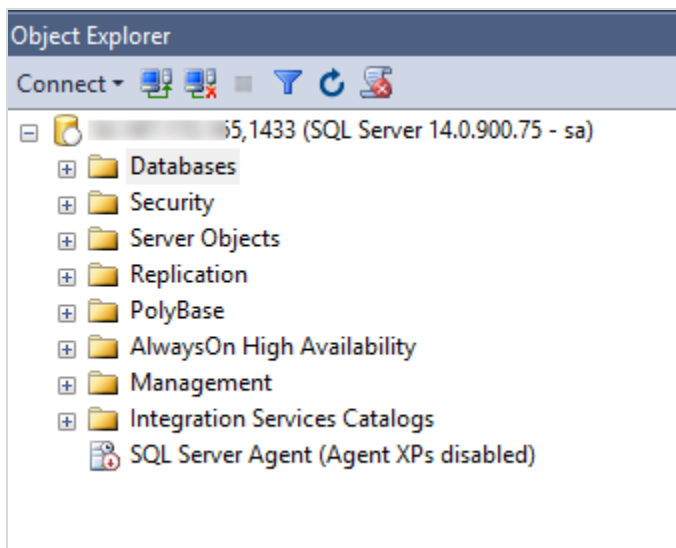


Figure 26. Connecting from SQL Server Management Studio.

To create a new database, run the [custom database creation script](https://github.com/Mahesh-MSFT/KubernetesDemoApp/blob/master/sqlscript/dbcreationscript.sql) (https://github.com/Mahesh-MSFT/KubernetesDemoApp/blob/master/sqlscript/dbcreationscript.sql).

StatefulSet creates data disks in the Azure Storage account. You can validate that as well. First check the volume name in the Kubernetes dashboard as Figure 27 shows.

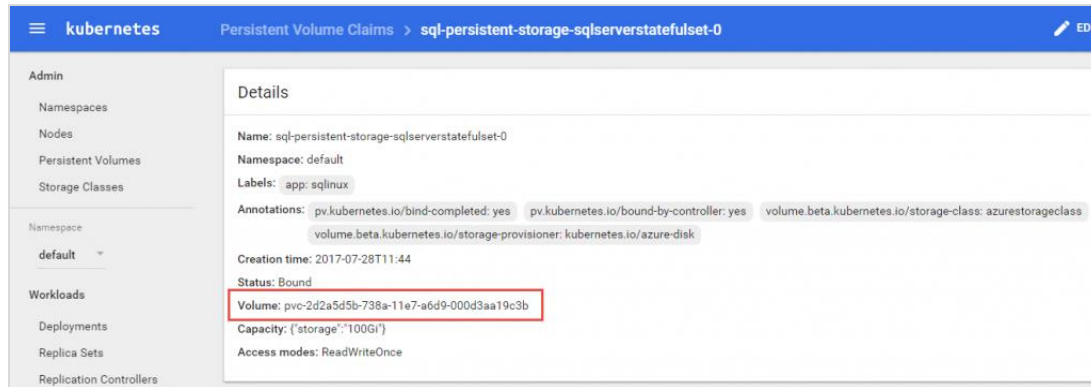


Figure 27. Volume name in Kubernetes dashboard.

Make sure this volume is available in the Azure Storage account as well by opening the Azure portal. Figure 28 shows that the external storage is available.

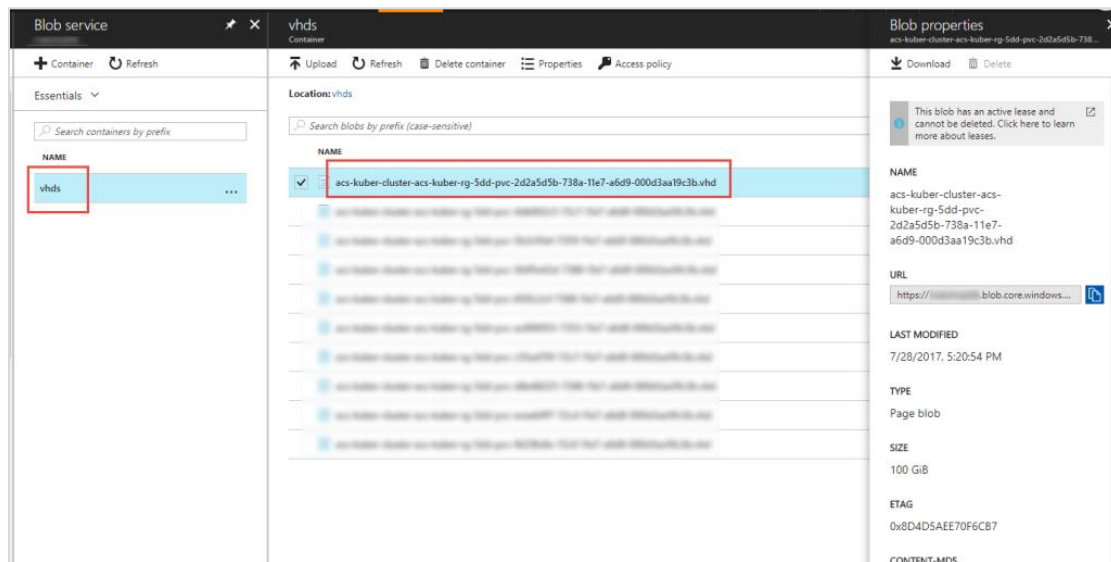


Figure 28. Volume in an Azure Storage account.

Verify data persistence

Now you can verify that the data is actually being stored on external storage. [SSH onto the node](#) that is running the pod. Run the `fdisk -l` command to display the partition tables for the devices as shown in Figure 29. Azure Disk is attached as a device on the node.

```

@] ~$ sudo fdisk -l
Disk /dev/sda: 30 GiB, 32212254720 bytes, 62914560 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x1a7d4c6a

Device      Boot Start          End  Sectors  Size Id Type
/dev/sda1   *      2048 62914526 62912479   30G 83 Linux

Disk /dev/sdb: 100 GiB, 107374182400 bytes, 209715200 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disklabel type: dos
Disk identifier: 0xaed15283

Device      Boot Start          End  Sectors  Size Id Type
/dev/sdb1           2048 209713151 209711104   100G  7 HPFS/NTFS/exFAT

Disk /dev/sdc: 10 GiB, 10737418240 bytes, 20971520 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

```

Figure 29. Using the `fdisk -l` command to verify Azure Disk is attached as a device on the node.

As Figure 29 shows, `/dev/sdc` matches the **10 GiB** storage from Azure Disk. This information can be used to mount on a directory and verify its content as Figure 30 shows.

```

root@k8s-agent-00350A71:~# sudo su
root@k8s-agent-00350A71:~# mkdir /test ①
root@k8s-agent-00350A71:~# mount /dev/sdc /test ②
root@k8s-agent-00350A71:~# cd /test
root@k8s-agent-00350A71:~# /test# ls ③
lost+found  mssql
root@k8s-agent-00350A71:~# /test# cd mssql/data
root@k8s-agent-00350A71:~# /test/mssql/data# ls ④
master.mdf  modellog.ldf  msdbdata.mdf  snoopyshoppingcart_log.ldf  tempdb.mdf
mastlog.ldf  model.mdf    msdblog.ldf   snoopyshoppingcart.mdf      templog.ldf
root@k8s-agent-00350A71:~# /test/mssql/data# █

```

Figure 30. 1) Create a temporary directory. 2) Mount it from the external volume. 3) List contents. 4) Go to `/mssql/data` (the SQL Server data and log files). 5) List contents; verify that the `.mdf` and `.ldf` files are created for the new database.

This verification step proves that SQL Server data and log files are stored on Azure Disk. Even if the Kubernetes cluster crashes, the data can be safely recovered from those files.

Considerations for the ASP.NET Core web front-end deployment

The resource demands of front-end services usually vary over time, so for this type of workload, you need a flexible way to scale up or down. For these scenarios, the Kubernetes Horizontal Pod Autoscaler is the option you want.

This section shows you how to deploy the sample ASP.NET Core front end using the `horizontalpodautoscaler` resource. It demonstrates scaling up and down based on calculating a pod's CPU utilization percentage. For details about this tool, see [Horizontal Pod Autoscaler Walkthrough](#) in the Kubernetes documentation.

You might wonder how we know the CPU utilization of a pod. Monitoring provides the answer. It's important to monitor a Kubernetes cluster and collect metrics from each node. For our demo app, [Heapster](#) is installed with InfluxDB, a time-series database, and Grafana, a monitoring board. Metrics are collected, and then used to scale operations up and down.

Install Heapster with InfluxDB and Grafana

The basic process is to first clone the [heapster github repo](#). Get the latest code base using any tool you like, such as [GitHub Desktop](#).

When the cloning is complete, go to `heapster\deploy\kube-config\influxdb`, where the YAML files are stored for the three components. Change the `grafana.yaml` file and create the services as described in the following sections.

For more information about the levels of customization that are available, see [How to Utilize the "Heapster + InfluxDB + Grafana" Stack in Kubernetes for Monitoring Pods](#) and [Deploying Heapster to Kubernetes](#).

Change the grafana.yaml file

By changing the `grafana.yaml` file, you can customize the dashboard setup. For example, our sample scenario uses Azure Load Balancer. In addition, environment variables that are not needed have been commented out for the production setup as Figure 31 shows.

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: monitoring-grafana
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        task: monitoring
        k8s-app: grafana
    spec:
      containers:
      - name: grafana
        image: gcr.io/google_containers/heapster-grafana-amd64:v4.4.1
        ports:
        - containerPort: 3000
          protocol: TCP
        volumeMounts:
        - mountPath: /var
          name: grafana-storage
        env:
        - name: INFLUXDB_HOST
          value: monitoring-influxdb
        - name: GF_SERVER_HTTP_PORT
          value: "3000"
        # The following env variables are required to make Grafana accessible via
        # the kubernetes api-server proxy. On production clusters, we recommend
        # removing these env variables, setup auth for grafana, and expose the grafana
        # service using a LoadBalancer or a public IP.
        #- name: GF_AUTH_BASIC_ENABLED
        #  value: "false"
        #- name: GF_AUTH_ANONYMOUS_ENABLED
        #  value: "true"
        #- name: GF_AUTH_ANONYMOUS_ORG_ROLE
        #  value: Admin
        #- name: GF_SERVER_ROOT_URL
        #  # If you're only using the API Server proxy, set this value instead:
        #  # value: /api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/
        #  value: /
      volumes:
      - name: grafana-storage
        emptyDir: {}

```

Figure 31. Grafana customization.

Next, specify a public static IP address that is different from the one used for creating the SQL Server service in the previous section as Figure 32 shows.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    # For use as a Cluster add-on (https://github.com/kubernetes/kubernetes/tree/master/cluster/addons)
    # If you are NOT using this as an addon, you should comment out this line.
    kubernetes.io/cluster-service: 'true'
    kubernetes.io/name: monitoring-grafana
  name: monitoring-grafana
  namespace: kube-system
spec:
  # In a production setup, we recommend accessing Grafana through an external Loadbalancer
  # or through a public IP.
  type: LoadBalancer
  loadBalancerIP: 132.233.128.133
  # You could also use NodePort to expose the service at a randomly-generated port.
  # type: NodePort
  ports:
    - port: 80
      targetPort: 3000
  selector:
    k8s-app: grafana
```

Figure 32. Public static IP address.

Create services

This step involves running the `kubectl create -f.` command from a location where the three YAML files exist. First, copy the path as shown in Figure 33.

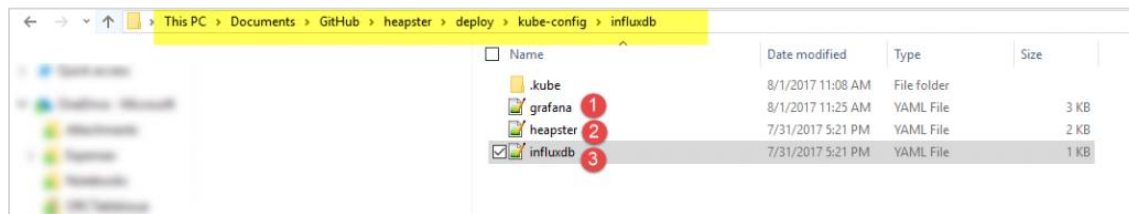


Figure 33. Three YAML files in one folder.

In the command shell, go to that location, then run:

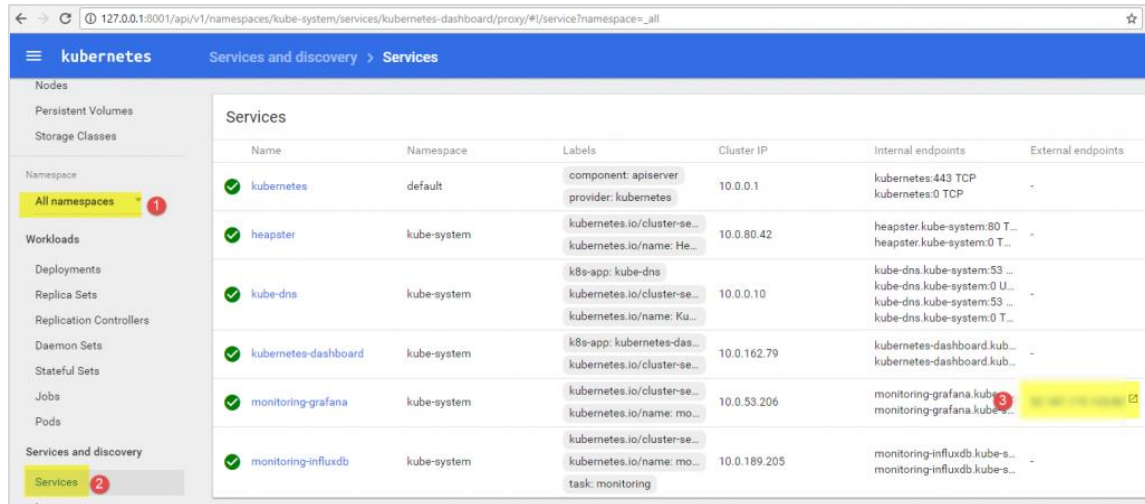
```
kubectl create -f.
```

As Figure 34 shows, all the components in all the YAML files found in the location shown in Figure 33 are created.

```
C:\Users\maksh\Documents\GitHub\heapster\deploy\kube-config\influxdb>kubectl create -f .
deployment "monitoring-grafana" created
service "monitoring-grafana" created
deployment "heapster" created
deployment "monitoring-influxdb" created
service "monitoring-influxdb" created
```

Figure 34. Create the components using the `kubectl create -f.` command.

To verify that all the services have been created and are running, go to the Kubernetes dashboard. In the Namespace box, select All namespaces (item 1 in Figure 35). The **Services** view (2) shows the new services created in the kube-system namespace (3).



Name	Namespace	Labels	Cluster IP	Internal endpoints	External endpoints
kubernetes	default	component: apiserver provider: kubernetes	10.0.0.1	kubernetes:443 TCP kubernetes:0 TCP	-
heapster	kube-system	kubernetes.io/cluster-se... kubernetes.io/name: He...	10.0.80.42	heapster.kube-system:80 T... heapster.kube-system:0 T...	-
kube-dns	kube-system	k8s-app: kube-dns kubernetes.io/cluster-se... kubernetes.io/name: Ku...	10.0.0.10	kube-dns.kube-system:53 ... kube-dns.kube-system:0 U... kube-dns.kube-system:53 ... kube-dns.kube-system:0 T...	-
kubernetes-dashboard	kube-system	k8s-app: kubernetes-das... kubernetes.io/cluster-se...	10.0.162.79	kubernetes-dashboard.kub... kubernetes-dashboard.kub...	-
monitoring-grafana	kube-system	kubernetes.io/cluster-se... kubernetes.io/name: mo...	10.0.53.206	monitoring-grafana.kube... monitoring-grafana.kube...	10.0.53.206
monitoring-influxdb	kube-system	kubernetes.io/cluster-se... kubernetes.io/name: mo... task: monitoring	10.0.189.205	monitoring-influxdb.kube-s... monitoring-influxdb.kube-s...	-

Figure 35. Verifying that new services have been created in the Kubernetes dashboard.

The external endpoint for the Grafana service should be the Azure IP address (item 3 in Figure 35). Click the link to navigate to the Grafana dashboard (Figure 36). The Kubernetes dashboard can now display the resource utilization information for the cluster and for the pods.



Figure 36. Grafana dashboard.

Use the Horizontal Pod Autoscaler

Instead of creating and scaling pods or a container by itself, the Horizontal Pod Autoscaler uses a deployment controller, replication controller, or replica set to create pods. It simply scales using the deployment resources that follow.

Create deployment

A deployment resource uses a custom Docker image of an ASP.NET Core application that connects with SQL Server. For the data source, an ASP.NET Core application uses **connectionstring**, pointing to <External IP Address of SQL Service, 1433> using the five settings shown in Figure 37.

- 1) **Replicas:** mark the number of pods to be created.
- 2) **Labels:** identify the pods that will make up the service. The value of app: aspNetcoreapp should match the value in Service.
- 3) **Image:** is the image from which the container is created.
- 4) **containerPort:** specifies the port on which the container will accept incoming requests.
- 5) **CPU:** is used to set the CPU resource that the deployment is requesting. The *m* suffix is used to request the smallest possible precision of required CPU capacity.

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: aspNetcoreapp-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: aspNetcoreapp
    spec:
      containers:
        - name: aspNetcoreapp
          image: maksh/aspnetcorekuberappimg
          ports:
            - containerPort: 5000
          resources:
            requests:
              cpu: 1m

```

Figure 37. Creating a deployment resource.

For more information about how a deployment resource uses a custom Docker image, see the blog [Running SQL Server + ASP.Net Core in a container on Linux in Azure Container Service on Docker Swarm – Part 2](#).

Create service

A service has a static IP address that can be used to reach an application regardless of any failures of the underlying pods. Service addresses problems associated with dynamic IP addresses that are assigned to pods when they crash and are recreated by the deployment. By comparison, deployment only ensures that a certain number of pods are created and maintained.

In Figure 38 the LoadBalancer type ensures that a load balancer and associated IP address are generated in Azure when this service is created. It also uses the selector value app: aspNetcoreapp, which must match the deployment definition. This service will include all the pods that have the labels app: aspNetcoreapp.

```

apiVersion: v1
kind: Service
metadata:
  name: aspNetcoreapp-service
spec:
  type: LoadBalancer
  ports:
    - port: 80
  selector:
    app: aspNetcoreapp

```

Figure 38. Creating a service resource. The service generates a load balancer (1) and includes pods labeled

aspnetcoreapp (2).

Create horizontalPodAutoScaler

Finally, a horizontalpodautoscaler uses the deployment resource and specifies the scale up rules. As Figure 39 shows, it targets deployment `aspnetcoreapp-deployment` (item 1 in Figure 39). The rules are specified using `minReplicas` and `maxReplicas` and are associated with `targetCPUUtilizationPercentage`.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: aspnetcoreapp-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: aspnetcoreapp-deployment
  minReplicas: 1
  maxReplicas: 2
  targetCPUUtilizationPercentage: 20
```

Figure 39. Creating a horizontalpodautoscaler resource.

For details, see [Horizontal Pod Autoscaler](#) in the Kubernetes documentation.

All three resources—deployment, service, and horizontalpodautoscaler—can be defined in a single YAML file and can be created in a single command execution. For example:

```
kubectl create -f "c:\Users\maksh\Documents\Visual Studio
2017\Projects\ContainerDemoApp\k8smanifests\aspnetcore-frontend.yaml"
```

Deployment replicas can be queried using the `get hpa` command. As Figure 40 shows, HorizontalPodAutoScaler is now set to increase the number of pods to two if the CPU utilization exceeds 20 percent. The value of `REPLICAS` shows that only one pod is currently running.

```
C:\WINDOWS\system32>kubectl get hpa
NAME                REFERENCE                TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
aspnetcoreapp-hpa   Deployment/aspnetcoreapp-deployment  0% / 20%    1         2         1         2m
```

Figure 40. Deployment replicas query using `kubectl get hpa` command.

Troubleshoot issues

Sometimes, Kubernetes doesn't display the current CPU utilization. This known issue is [documented](#), as is a [solution](#).

Verify how operations scale

The Deployment resource is now set to increase the number of pods if the CPU utilization goes above 20 percent for any reason. Note the current number of replicas by running the `get deployment` command. As Figure 41 shows, with the current CPU load (0 percent), only one pod is

needed to run, and it is running.

```
C:\WINDOWS\system32>kubectl get deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
aspnetcoreapp-deployment	1	1	1	1	11m

Figure 41. Current number of replicas using the kubectl get deployment command

Next, launch the application by going to the external URL, and in a new command window, launch another parallel workload, as shown in Figure 42. This command generates a CPU load and generates a pod. After a while, the CPU utilization of this pod, as well as the ASP.NET Core pod, start to increase.

```
C:\WINDOWS\system32>kubectl run -i --tty load-generator --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.
/ # while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done
```

Figure 42. Additional parallel workload.

To verify the results, run the **get hpa** command. As Figure 43 shows, the current CPU utilization is now more than 20 percent.

```
C:\WINDOWS\system32>kubectl get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
aspnetcoreapp-hpa	Deployment/aspnetcoreapp-deployment	7300% / 20%	1	2	1	11m

Figure 43. Showing the scaling results with the kubectl get hpa command.

At this percentage, Horizontalpodautoscaler should add another replica, as specified in its manifest. To make sure, run the **get deployment** command. As Figure 44 shows, the DESIRED replica count has jumped to two, as specified in the manifest.

```
C:\WINDOWS\system32>kubectl get deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
aspnetcoreapp-deployment	2	2	2	2	16m
load-generator	1	1	1	1	8m

Figure 44. Verifying the replica with the kubectl get deployment command.

In addition, the service is now running with two pods, as shown in Figure 45.

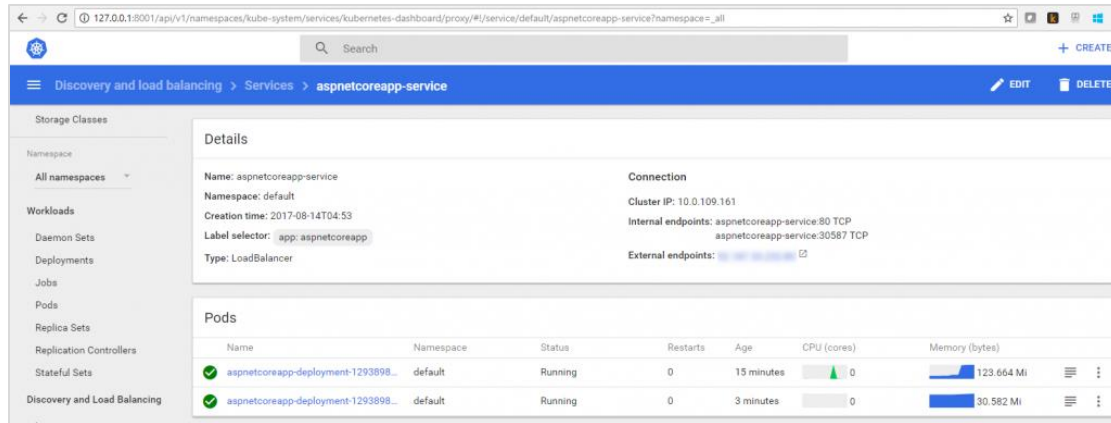


Figure 45. Service running two pods

Next, stop the load-generator deployment and ASP.NET Core application. After a while, use the `kubectl get hpa` command again to note the CPU utilization (Figure 46).

```
C:\WINDOWS\system32>kubectl get hpa
NAME                REFERENCE                      TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
aspnetcoreapp-hpa   Deployment/aspnetcoreapp-deployment  0% / 20%    1         2         1         2m
```

Figure 46. Checking CPU utilization with the `kubectl get hpa` command.

The replica count should also return to one (1) as Figure 47 verifies.

```
C:\WINDOWS\system32>kubectl get deployment
NAME                                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
aspnetcoreapp-deployment            1         1         1             1         11m
```

Figure 47. Checking replica count with the `kubectl get deployment` command.

The Kubernetes dashboard also verifies that the service is back to running with one pod (Figure 48). These steps confirm that Service can scale up pods and replica count when the CPU percentage increases to match the manifest. Service can also scale down based on same metric.

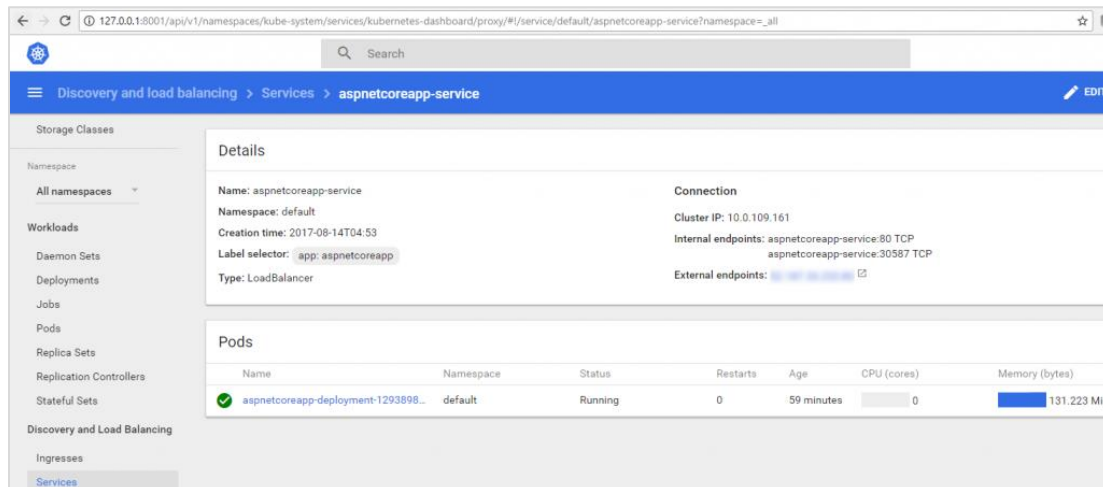
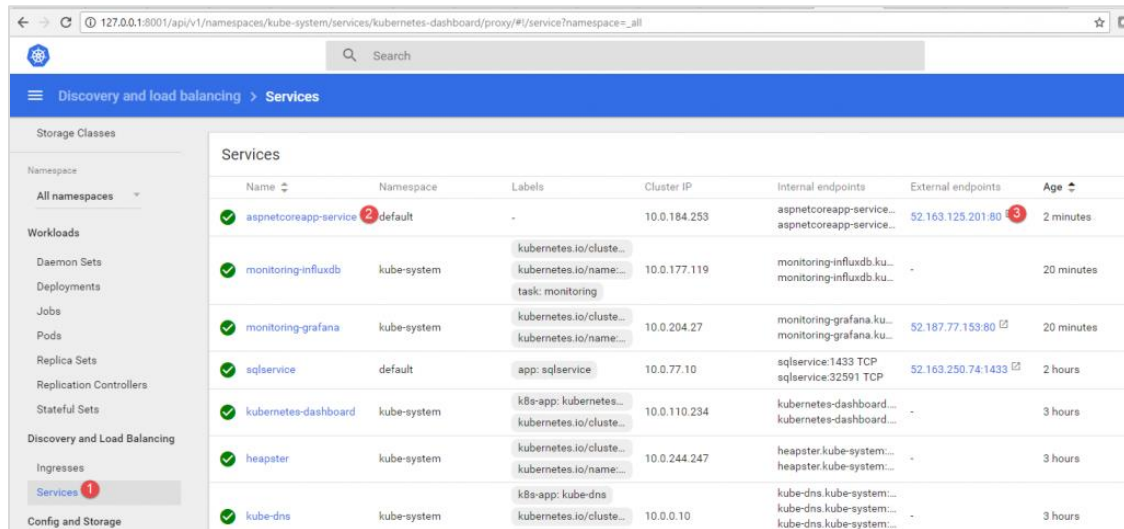


Figure 48. Service running with one pod.

Verify the application

At this stage, a service running the custom ASP.NET Core application image is deployed to the Kubernetes cluster. To launch this application, go to the **Services** menu in the Kubernetes dashboard (item 1 in Figure 49), select the service (2), and then refer to the **External endpoints** column (item 3).



Name	Namespace	Labels	Cluster IP	Internal endpoints	External endpoints	Age
aspnetcoreapp-service	default	-	10.0.184.253	aspnetcoreapp-service...	52.163.125.201:80	2 minutes
monitoring-influxdb	kube-system	kubernetes.io/cluste... kubernetes.io/name:... task: monitoring	10.0.177.119	monitoring-influxdb.ku... monitoring-influxdb.ku...	-	20 minutes
monitoring-grafana	kube-system	kubernetes.io/cluste... kubernetes.io/name:...	10.0.204.27	monitoring-grafana.ku... monitoring-grafana.ku...	52.167.77.153:80	20 minutes
sqlservice	default	app: sqlservice	10.0.77.10	sqlservice:1433 TCP sqlservice:32591 TCP	52.163.250.74:1433	2 hours
kubernetes-dashboard	kube-system	k8s-app: kubernetes... kubernetes.io/cluste...	10.0.110.234	kubernetes-dashboard... kubernetes-dashboard...	-	3 hours
heapster	kube-system	kubernetes.io/cluste... kubernetes.io/name:...	10.0.244.247	heapster.kube-system... heapster.kube-system...	-	3 hours
kube-dns	kube-system	k8s-app: kube-dns... kubernetes.io/cluste...	10.0.0.10	kube-dns.kube-system... kube-dns.kube-system... kube-dns.kube-system...	-	3 hours

Figure 49. Kubernetes dashboard with view of the external endpoint.

Copy the external endpoint, which is a load-balanced URL serving the application. Paste it in a new browser tab to go to the application home page (Figure 50).

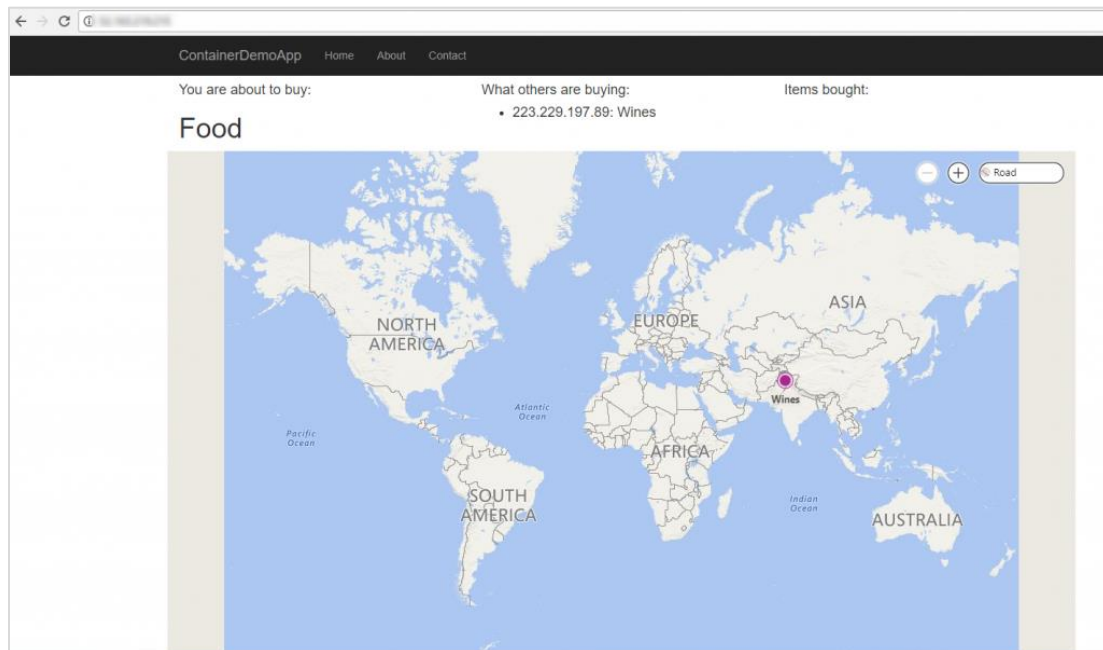


Figure 50. Go to the designated external endpoint to display the application home page.

Verify the database

The sample app's purpose is merely to insert some data in SQL Server, which is also running as a service. To verify that this service is running, connect to SQL Server using SQL Server Management Studio. Query the table application that is inserting the data to see the results (Figure 51).

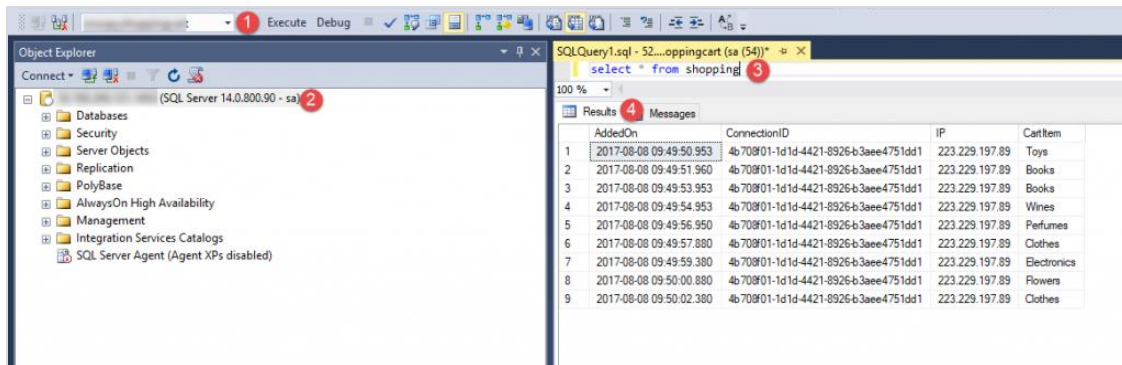


Figure 51. Verifying the database (1) in SQL Server Management Studio. Choose a table (2), select query (3), to see results (4).

Conclusion

I hope this guidance helps you choose the deployment most appropriate for you based on the elements that make up your given workload, whether a stateless web front end or API tier and stateful database back end. Please use this white paper and the [GitHub resources](#) as a jumping off point for other aspects of development using Kubernetes. Consider setting up local development and remote Azure clusters. Try configuring and monitoring workloads on each of these environments. Good luck!