**Note**: Check the /Windows/System32/etc/drivers/hosts file for conflict between docker-k8s Hadoop sandbox config.

**Table of Contents**

[Overview 10](#_Toc201777024)

[Kubernetes Architecture Overview 10](#_Toc201777025)

[Kubernetes Cluster Archiercture 10](#_Toc201777026)

[Desired State Management 12](#_Toc201777027)

[Kubernetes Cluster 13](#_Toc201777028)

[Pods 14](#_Toc201777029)

[Pod Life-Cycle Phases 16](#_Toc201777030)

[Kubernetes Node Status 17](#_Toc201777031)

[Container Orchestration 17](#_Toc201777032)

[Volumes 18](#_Toc201777033)

[Stateful vs Stateless Applications on Kubernetes 19](#_Toc201777034)

[Stateless services aren’t actually ‘stateless’ 19](#_Toc201777035)

[Stateful services and the CAP theorem 20](#_Toc201777036)

[Kubernetes Core Concepts 21](#_Toc201777037)

[Kubernetes - Master Machine Components 21](#_Toc201777038)

[Kubernetes - Node Components 22](#_Toc201777039)

[Kubernetes - Master and Node Structure 23](#_Toc201777040)

[What is the Difference Between Kubernetes and Docker Swarm 24](#_Toc201777041)

[Introduction 24](#_Toc201777042)

[Kubernetes 24](#_Toc201777043)

[Pros of Using Kubernetes 24](#_Toc201777044)

[Cons of Using Kubernetes 25](#_Toc201777045)

[Docker Swarm 25](#_Toc201777046)

[Pros of Using Docker Swarm 25](#_Toc201777047)

[Cons of Using Docker Swarm 26](#_Toc201777048)

[Kubernetes or Docker: Which Is the Perfect Choice? 27](#_Toc201777049)

[Use Kubernetes if: 27](#_Toc201777050)

[Use Docker if, 27](#_Toc201777051)

[Final Thoughts: Kubernetes and Docker as Friends 27](#_Toc201777052)

[Installation 28](#_Toc201777053)

[Check Ubuntu Version 28](#_Toc201777054)

[Terminal 28](#_Toc201777055)

[GUI 28](#_Toc201777056)

[Installation Steps 28](#_Toc201777057)

[Install Docker on Ubuntu 20.04 30](#_Toc201777058)

[Initial Settings 30](#_Toc201777059)

[Steps 30](#_Toc201777060)

[Install Minikube on ubuntu 20.04 31](#_Toc201777061)

[Enable Kubernetes on Docker and Enable Dashboard 32](#_Toc201777062)

[Setup the k8s Dashboard 32](#_Toc201777063)

[Kubernetes Cluster Setup 34](#_Toc201777064)

[Prerequisites 34](#_Toc201777065)

[Install Docker Engine 34](#_Toc201777066)

[Install etcd 2.0 35](#_Toc201777067)

[Configure kube-apiserver 36](#_Toc201777068)

[Configure the kube Controller Manager 36](#_Toc201777069)

[Kubernetes Node Configuration 37](#_Toc201777070)

[Tutorial #0: Quick Start 39](#_Toc201777071)

[Pods 39](#_Toc201777072)

[Deployments 39](#_Toc201777073)

[Tutorial #1: Quick Start 41](#_Toc201777074)

[Tutorial #2: Deploy, Expose as a Service, Multiple Instances, Rolling Updates 42](#_Toc201777075)

[Create a Cluster 42](#_Toc201777076)

[Using kubectl to Create a Deployment 42](#_Toc201777077)

[Deploy the app 42](#_Toc201777078)

[View the app 43](#_Toc201777079)

[Using a Service to Expose Your App 43](#_Toc201777080)

[Services and Labels 44](#_Toc201777081)

[Step 1: Creating a new Service 45](#_Toc201777082)

[Step 2: Using labels 46](#_Toc201777083)

[Step 3: Deleting a service 47](#_Toc201777084)

[Running Multiple Instances of Your App 47](#_Toc201777085)

[Scaling a Deployment 48](#_Toc201777086)

[Load Balancing 49](#_Toc201777087)

[Scale Down 50](#_Toc201777088)

[Performing a Rolling Update 50](#_Toc201777089)

[Update the version of the app 50](#_Toc201777090)

[Verify an update 51](#_Toc201777091)

[Roll back an update 52](#_Toc201777092)

[Tutorial #3 53](#_Toc201777093)

[Create a cluster 53](#_Toc201777094)

[Deploy an app 53](#_Toc201777095)

[View our app 54](#_Toc201777096)

[Viewing Pods and Nodes 55](#_Toc201777097)

[Show app in the terminal 55](#_Toc201777098)

[View Container logs 55](#_Toc201777099)

[Executing command on the container 56](#_Toc201777100)

[Using a Service to expose your app 56](#_Toc201777101)

[Create a new service 56](#_Toc201777102)

[Using Labels 57](#_Toc201777103)

[Deleting a service 58](#_Toc201777104)

[Running multiple instances of your app 58](#_Toc201777105)

[Load balancing 59](#_Toc201777106)

[Scale down 60](#_Toc201777107)

[Performing a rolling update 60](#_Toc201777108)

[Update the version of the app 60](#_Toc201777109)

[Verify an update 60](#_Toc201777110)

[Rollback an update 61](#_Toc201777111)

[Cleanup 62](#_Toc201777112)

[Delete the service 62](#_Toc201777113)

[Delete the deployment 62](#_Toc201777114)

[Delete the pods 62](#_Toc201777115)

[Use a Service to Access an Application in a Cluster 62](#_Toc201777116)

[Creating a service for an application running in two pods: 62](#_Toc201777117)

[Kubernetes Volumes 65](#_Toc201777118)

[What are Persistent Volumes? 65](#_Toc201777119)

[What is the difference between Volume and Persistent Volume in Kubernetes? 65](#_Toc201777120)

[Types of Persistent Volume 66](#_Toc201777121)

[Persistent Volume access modes 66](#_Toc201777122)

[The lifecycles of PVs and PVCs 66](#_Toc201777123)

[Example: How to create and use a Persistent Volume 67](#_Toc201777124)

[Kubernetes Secrets 71](#_Toc201777125)

[What are secrets in Kubernetes? 71](#_Toc201777126)

[Encoding and decoding data for Kubernetes Secrets 71](#_Toc201777127)

[1. Create Kubernetes Secrets using kubectl 72](#_Toc201777128)

[2. Create Kubernetes Secrets from a YAML manifest file 73](#_Toc201777129)

[List existing Kubernetes Secrets 74](#_Toc201777130)

[View a Kubernetes Secret value with kubectl describe 74](#_Toc201777131)

[Decode a Kubernetes Secret 74](#_Toc201777132)

[Delete a Kubernetes Secret 75](#_Toc201777133)

[Using Secret data as container environment variables 75](#_Toc201777134)

[Using Secret data as files in a volume mounted on a Pod’s container(s) 76](#_Toc201777135)

[More 77](#_Toc201777136)

[Built-in Secrets 77](#_Toc201777137)

[Creating your own Secrets 77](#_Toc201777138)

[Creating a Secret Using kubectl create secret 77](#_Toc201777139)

[Creating a Secret Manually 78](#_Toc201777140)

[Using Secrets 80](#_Toc201777141)

[Using Secrets with Volumes 81](#_Toc201777142)

[Consuming Secret Values from Volumes 81](#_Toc201777143)

[Using Secrets as Environment Variables 81](#_Toc201777144)

[Consuming Secret Values from Volumes 82](#_Toc201777145)

[Cleanup 82](#_Toc201777146)

[Kubernetes ConfigMaps 83](#_Toc201777147)

[Key Features 83](#_Toc201777148)

[Creating a ConfigMap 83](#_Toc201777149)

[Using a ConfigMap in Pods 83](#_Toc201777150)

[Updating and Managing ConfigMaps 84](#_Toc201777151)

[Typical Use Cases 84](#_Toc201777152)

[Summary Table 84](#_Toc201777153)

[Kubernetes on Azure (AKS) 85](#_Toc201777154)

[Access, security, and monitoring 85](#_Toc201777155)

[Identity and security management 85](#_Toc201777156)

[Integrated logging and monitoring 86](#_Toc201777157)

[Clusters and nodes 86](#_Toc201777158)

[Cluster node and pod scaling 86](#_Toc201777159)

[Cluster node upgrades 86](#_Toc201777160)

[Storage volume support 86](#_Toc201777161)

[Virtual networks and ingress 87](#_Toc201777162)

[Ingress with HTTP application routing 87](#_Toc201777163)

[Development tooling integration 87](#_Toc201777164)

[Docker image support and private container registry 87](#_Toc201777165)

[Quotas, virtual machine size restrictions, and region availability in Azure Kubernetes Service (AKS) 87](#_Toc201777166)

[Autoscaling on AKS 88](#_Toc201777167)

[Tutorial 01: Prepare an application for Azure Kubernetes Service (AKS) 89](#_Toc201777168)

[Before you begin 89](#_Toc201777169)

[Get application code 89](#_Toc201777170)

[Create container images 90](#_Toc201777171)

[Test application locally 91](#_Toc201777172)

[Clean up resources 91](#_Toc201777173)

[Tutorial 02: Deploy and use Azure Container Registry (ACR) 92](#_Toc201777174)

[Before you begin 92](#_Toc201777175)

[Create an Azure Container Registry 92](#_Toc201777176)

[Log in to the container registry 92](#_Toc201777177)

[Tag a container image 93](#_Toc201777178)

[Push images to registry 93](#_Toc201777179)

[List images in registry 94](#_Toc201777180)

[Tutorial 03: Deploy an Azure Kubernetes Service (AKS) cluster 95](#_Toc201777181)

[Before you begin 95](#_Toc201777182)

[Create a Kubernetes cluster 95](#_Toc201777183)

[Install the Kubernetes CLI 96](#_Toc201777184)

[Connect to cluster using kubectl 96](#_Toc201777185)

[Tutorial 04: Run applications in Azure Kubernetes Service (AKS) 97](#_Toc201777186)

[Before you begin 97](#_Toc201777187)

[Update the manifest file 97](#_Toc201777188)

[Deploy the application 98](#_Toc201777189)

[Test the application 98](#_Toc201777190)

[Tutorial 05: Scale applications in Azure Kubernetes Service (AKS) 100](#_Toc201777191)

[Before you begin 100](#_Toc201777192)

[Manually scale pods 100](#_Toc201777193)

[Autoscale pods 101](#_Toc201777194)

[Manually scale AKS nodes 102](#_Toc201777195)

[Tutorial 06: Update an application in Azure Kubernetes Service (AKS) 104](#_Toc201777196)

[Before you begin 104](#_Toc201777197)

[Update an application 104](#_Toc201777198)

[Update the container image 105](#_Toc201777199)

[Test the application locally 105](#_Toc201777200)

[Tag and push the image 105](#_Toc201777201)

[Deploy the updated application 106](#_Toc201777202)

[Test the updated application 106](#_Toc201777203)

[Tutorial 07: Upgrade Kubernetes in Azure Kubernetes Service (AKS) 107](#_Toc201777204)

[Using Azure CLI 107](#_Toc201777205)

[Get available cluster versions 107](#_Toc201777206)

[Upgrade a cluster 107](#_Toc201777207)

[View the upgrade events 108](#_Toc201777208)

[Validate an upgrade 109](#_Toc201777209)

[Delete the cluster 109](#_Toc201777210)

[Using Azure Portal 109](#_Toc201777211)

[Get available cluster versions 109](#_Toc201777212)

[Upgrade a cluster 110](#_Toc201777213)

[View the upgrade events 110](#_Toc201777214)

[Validate an upgrade 111](#_Toc201777215)

[Delete the cluster 111](#_Toc201777216)

[Quickstart: Deploy an Azure Kubernetes Service cluster using the Azure CLI 112](#_Toc201777217)

[Prerequisites 112](#_Toc201777218)

[Create a resource group 113](#_Toc201777219)

[Create AKS cluster 114](#_Toc201777220)

[Connect to the cluster 114](#_Toc201777221)

[Deploy the application 114](#_Toc201777222)

[Test the application 117](#_Toc201777223)

[Delete the cluster 117](#_Toc201777224)

[Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using the Azure portal 118](#_Toc201777225)

[Prerequisites 118](#_Toc201777226)

[Create an AKS cluster 118](#_Toc201777227)

[Connect to the cluster 121](#_Toc201777228)

[Deploy the application 122](#_Toc201777229)

[Test the application 124](#_Toc201777230)

[Delete cluster 125](#_Toc201777231)

[Kubernetes core concepts for Azure Kubernetes Service (AKS) 126](#_Toc201777232)

[What is Kubernetes? 126](#_Toc201777233)

[Kubernetes cluster architecture 126](#_Toc201777234)

[Control plane 127](#_Toc201777235)

[Nodes and node pools 128](#_Toc201777236)

[Resource reservations 129](#_Toc201777237)

[Node pools 130](#_Toc201777238)

[Node selectors 130](#_Toc201777239)

[Pods 131](#_Toc201777240)

[Deployments and YAML manifests 131](#_Toc201777241)

[Package management with Helm 134](#_Toc201777242)

[StatefulSets and DaemonSets 134](#_Toc201777243)

[StatefulSets 134](#_Toc201777244)

[DaemonSets 135](#_Toc201777245)

[Namespaces 135](#_Toc201777246)

[Kubernetes Ingress Controller 137](#_Toc201777247)

[Security: AKS-managed Azure Active Directory integration 138](#_Toc201777248)

[Azure AD authentication overview 138](#_Toc201777249)

[Limitations 138](#_Toc201777250)

[Prerequisites 138](#_Toc201777251)

[Before you begin 138](#_Toc201777252)

[Create an AKS cluster with Azure AD enabled 139](#_Toc201777253)

[Access an Azure AD enabled cluster 139](#_Toc201777254)

[Troubleshooting access issues with Azure AD 139](#_Toc201777255)

[Enable AKS-managed Azure AD integration on your existing cluster 140](#_Toc201777256)

[Security: Use Azure role-based access control for Kubernetes Authorization 141](#_Toc201777257)

[Before you begin 141](#_Toc201777258)

[Create a new AKS cluster with managed Azure AD integration and Azure RBAC for Kubernetes Authorization 141](#_Toc201777259)

[Enable Azure RBAC on an existing AKS cluster 142](#_Toc201777260)

[Disable Azure RBAC for Kubernetes Authorization from an AKS cluster 142](#_Toc201777261)

[Create role assignments for users to access the cluster 142](#_Toc201777262)

[Create custom roles definitions 143](#_Toc201777263)

[Use Azure RBAC for Kubernetes Authorization with kubectl 143](#_Toc201777264)

[Use Azure RBAC for Kubernetes Authorization with kubelogin 144](#_Toc201777265)

[Clean up resources 144](#_Toc201777266)

[Delete role assignment 144](#_Toc201777267)

[Delete role definition 144](#_Toc201777268)

[Delete resource group and AKS cluster 144](#_Toc201777269)

[Security: Use Kubernetes role-based access control with Azure Active Directory in Azure Kubernetes Service 145](#_Toc201777270)

[Before you begin 145](#_Toc201777271)

[Create demo groups in Azure AD 146](#_Toc201777272)

[Create demo users in Azure AD 147](#_Toc201777273)

[Set the user principal name and password for application developers 147](#_Toc201777274)

[Create the user accounts 147](#_Toc201777275)

[Create AKS cluster resources for app devs 148](#_Toc201777276)

[Create the AKS cluster resources for SREs 150](#_Toc201777277)

[Interact with cluster resources using Azure AD identities 151](#_Toc201777278)

[Create and view cluster resources outside of the assigned namespace 151](#_Toc201777279)

[Test the SRE access to the AKS cluster resources 152](#_Toc201777280)

[Clean up resources 153](#_Toc201777281)

[Storage: Kubernetes Storage by Example 154](#_Toc201777282)

[Writable Container Layer 154](#_Toc201777283)

[Volumes (Temporary) 154](#_Toc201777284)

[Volumes (Persistent) 155](#_Toc201777285)

[PersistentVolume and PersistentVolumeClaim 156](#_Toc201777286)

[StorageClass 158](#_Toc201777287)

[Storage: Container Storage Interface (CSI) drivers on Azure Kubernetes Service (AKS) 161](#_Toc201777288)

[Prerequisites 162](#_Toc201777289)

[Enable CSI storage drivers on an existing cluster 162](#_Toc201777290)

[Disable CSI storage drivers on a new or existing cluster 162](#_Toc201777291)

[Storage: Use the Azure Disks Container Storage Interface (CSI) driver in Azure Kubernetes Service (AKS) 164](#_Toc201777292)

[Azure Disks CSI driver features 164](#_Toc201777293)

[Use CSI persistent volumes with Azure Disks 164](#_Toc201777294)

[Dynamically create Azure Disks PVs by using the built-in storage classes 165](#_Toc201777295)

[Create a Kubernetes dev space with Azure Dev Spaces (.NET Core and VS Code) 167](#_Toc201777296)

[Debug a container in Kubernetes 169](#_Toc201777297)

[Join a new node to the master 172](#_Toc201777298)

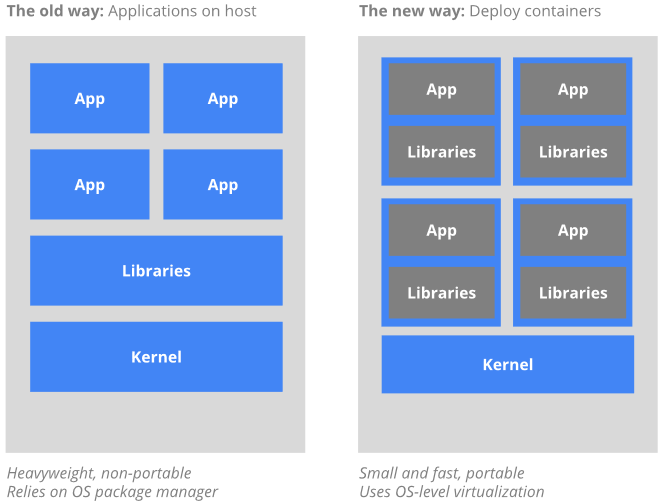
[Expose Deployment as a Service 173](#_Toc201777299)

[Example 1: Direct Run 173](#_Toc201777300)

[Example 2: With a YAML file 173](#_Toc201777301)

# Overview

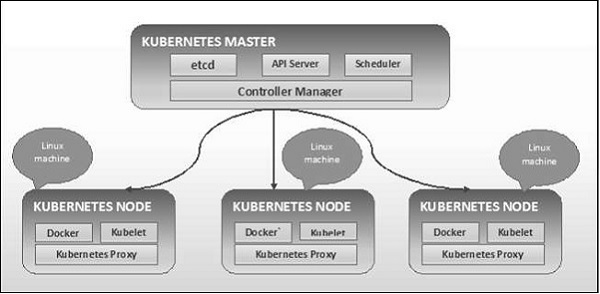
* Kubernetes is a
  + Portable,
  + Extensible,
  + Open-sourced platform
* for managing containerized workloads and services
* Google open-sourced K8s in 2014
* K8s features
  + Is a container platform
  + A microservices platform
  + A portable cloud platform
* Provides a container-centric management environment
* Orchestrates computing, networking and storage
* It is **not** a traditional, all-inclusive PaaS system
  + Since it operates at a container level rather than at the h/w level
* Why Containers?

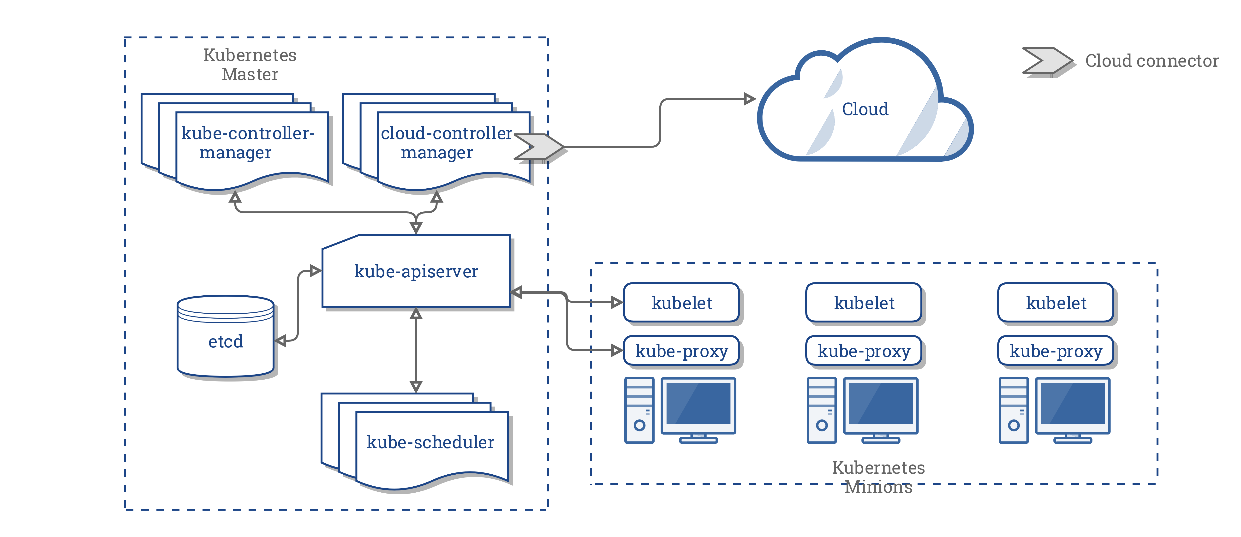


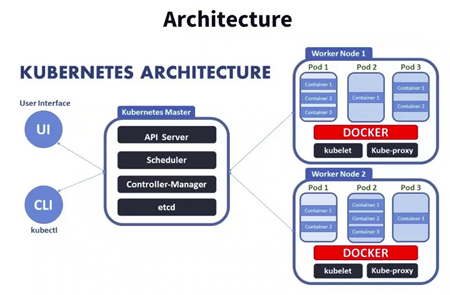
## Kubernetes Architecture Overview

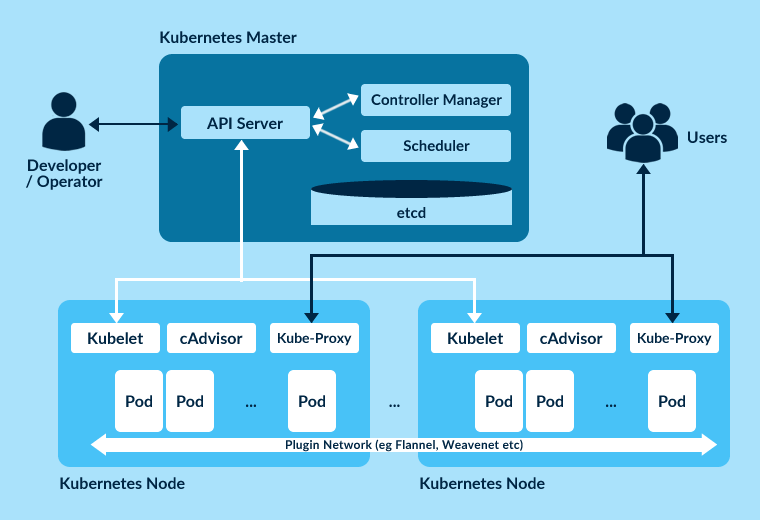
## Kubernetes Cluster Archiercture

Kubernetes follows client-server architecture. Wherein, we have master installed on one machine and the node on separate Linux machines.

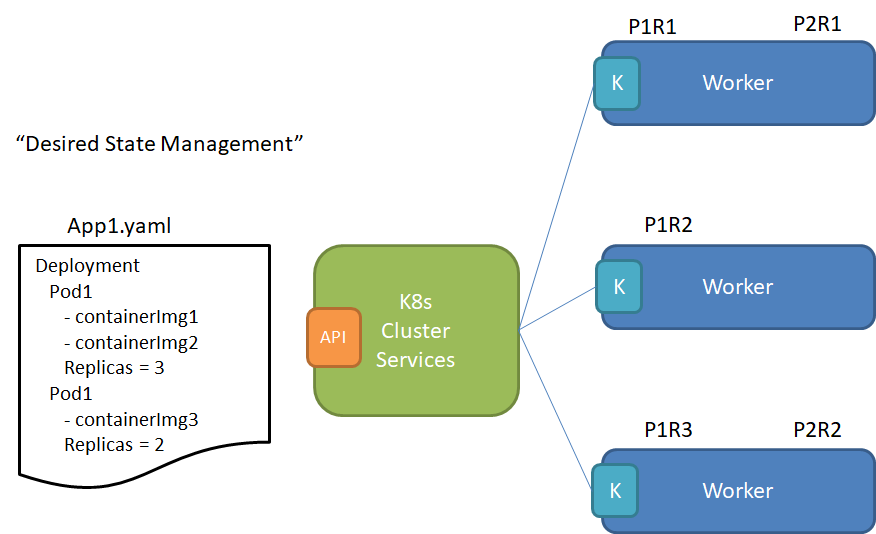


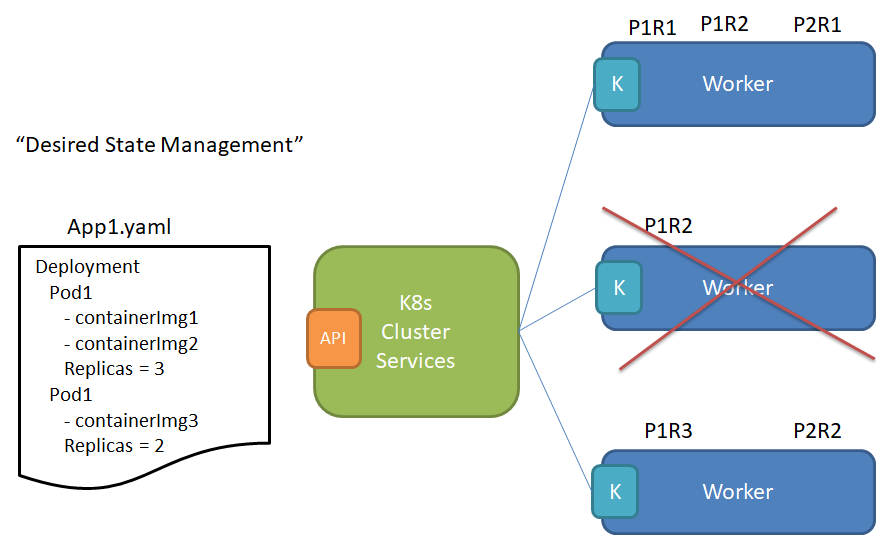






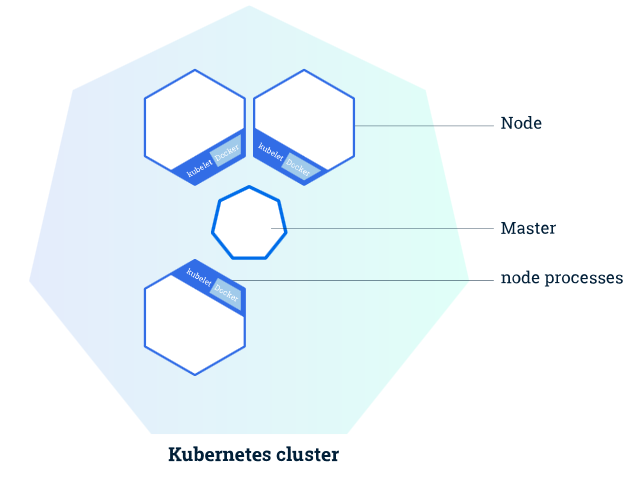
## Desired State Management





## Kubernetes Cluster

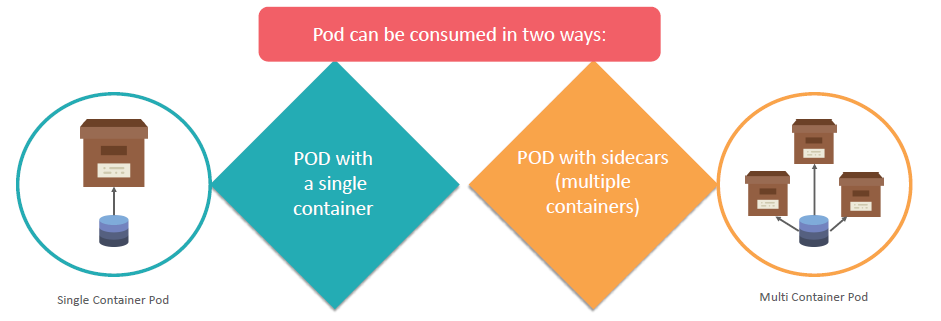
* Consists of two types of resources
  + Master: coordinates the cluster
  + Nodes: workers that run applications



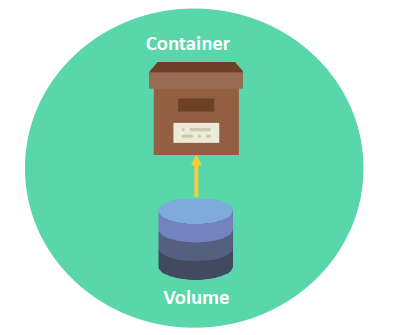
* Master
  + is responsible for managing the cluster
  + coordinates all activities in the cluster
    - scheduling apps
    - maintaining apps’ desired state
    - scaling apps
    - rolling out new updates
  + Node
    - Is a VM or physical server
    - aka as ***minions***
    - Serves as a worker machine in a k8s cluster
    - Each node has a Kubelet, which is an agent for managing the node and communicating with the k8s master
    - Should also have tools for handling container operations
      * Such as Docker
    - A k8s cluster that handles production traffic should have a minimum three nodes
* When you deploy an app on k8s,
  + You tell the master to start the app containers
  + The master schedules the containers to run on the cluster’s nodes
  + The nodes communicate with the master using k8s API, which the master exposes

## Pods

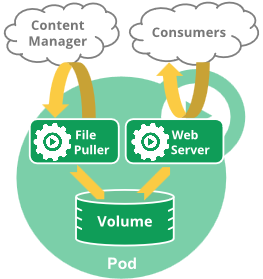
* A *Pod* is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy
* A *Pod* represents a unit of deployment in k8s cluster
* It is very easy to horizontally scale a Pod
* A pod (as in a pod of whales or pea pod) is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers
* It contains one or more application containers that are relatively tightly-coupled
* A Pod represents a running process on your cluster
* A Pod encapsulates an
  + application container (or, in some cases, multiple containers),
  + storage resources,
  + a unique network IP, and
  + options that govern how the container(s) should run
* A Pod represents a unit of deployment:
  + a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources
* ***Docker*** is the most common container runtime used in a Kubernetes Pod,
  + but Pods support other container runtimes as well
* Pods in a Kubernetes cluster can be used in two main ways:



* + **Pods that run a single container**.
    - The “one-container-per-Pod” model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly

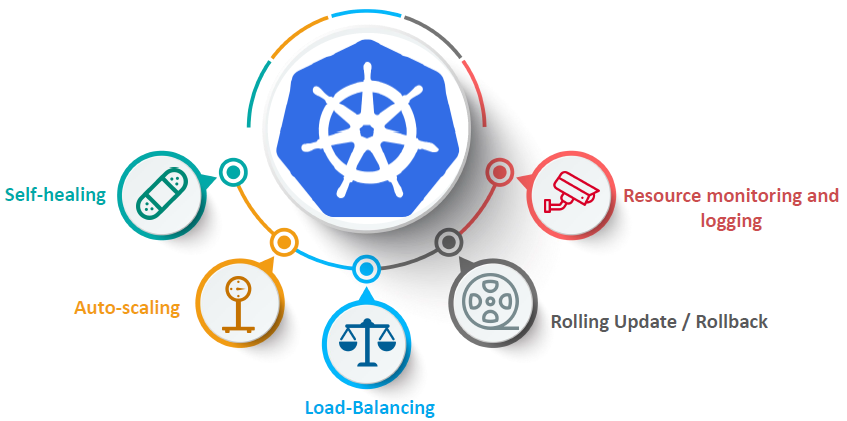


* + **Pods that run multiple containers that need to work together**
    - Pod with side-cars (*multiple containers*)
    - A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources
    - These co-located containers might form a single cohesive unit of service–one container serving files from a shared volume to the public, while a separate “sidecar” container refreshes or updates those files
      * Second containers are generally called “***side-car***”
    - The Pod wraps these containers and storage resources together as a single manageable entity

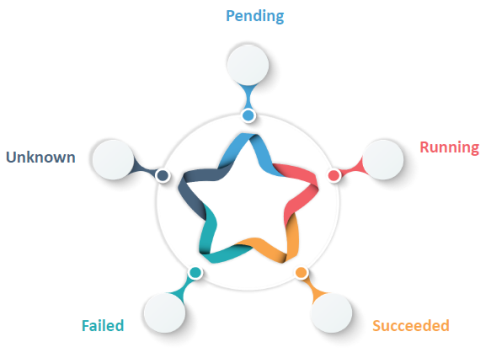


A pod diagram

* + - A multi-container pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers There are 2 containers that share the persistent volume
      * One of the containers is writing to the storage volume
      * The other container processes and publishes the same data to its consumers through a web server
* Pod features:



## Pod Life-Cycle Phases



|  |  |
| --- | --- |
| Pending | * It reflects the time spent in downloading the container images and creating them * It also means that system has accepted the Pod |
| Running | * Pod is tied-up with the node, and at least one container is running |
| Succeeded | * A container’s termination in k8s was successful * It will not be restarted |
| Failed | * When one or more container’s termination is unsuccessful * Termination due to failure is because of non-zero exit status of container |
| Unknown | * Where there is a communication problem of a container with the host machine, status of the container cannot be obtained, i.e.; *unknown*. * Since there is no status update, k8s system marks it as *unknown* * For such errors, communication channels should be checked first |

## Kubernetes Node Status

|  |  |  |
| --- | --- | --- |
| Address | Hostname | Container engine provides the hostname, however you can override it and put a more meaningful hostname using –hostname-override parameter |
| Internal IP Address | Internally routable IP address within the cluster (for internal communication only) |
| External IP Address | Externally routable IP address to connect to/from outside the cluster |
| Condition | OutOfDisk | True, if there is insufficient disk space, otherwise false |
| DiskPressure | True, if Disk capacity is low, else false |
| MemoryPressure | True, if the node memory is slow, else false |
| Networkunavailable | True, if network node is misconfigured, else false |
| ConfigOK | True, if kubelet configuration is correct, else false |
| Ready | True, if node is healthy  False, if something is wrong  Unknown, if nothing is heard from the node |
| Capacity | Describes the resources available on the node.  Using this information, you can decide on the no. of pods that can be scheduled.  These resources could be:   * CPU * Memory * Storage | |
| Info | * Kernel details * O/S details * K8s details (like version no. etc.) | |

## Container Orchestration

Container Orchestration refers to the automated arrangement, coordination, and management of software containers.

|  |  |
| --- | --- |
| **Why do we need this? Let’s start with the following diagram:** | **https://miro.medium.com/max/875/1*ffvv2BRLtnwRfZaoEPYi_g.png** |

**If your current software infrastructure looks something like this — maybe Nginx/Apache + PHP/Python/Ruby/Node.js app running on a few containers that talk to a replicated DB — then you might not require container orchestration, you can probably manage everything yourself.**

**What if your application keeps growing? Let’s say you keep adding more and more functionality until it becomes a massive monolith that is almost impossible to maintain and eats way too much CPU and RAM. You finally decide to split your application into smaller chunks, each responsible for one specific task, maintained by a team, aka. microservices.**

|  |  |
| --- | --- |
| **Your infrastructure now kind of looks like this:** | **https://miro.medium.com/max/875/1*gGm9l6xEiZQlJgIGOpFqSw.png** |

**You now need a caching layer — maybe a queuing system as well — to increase performance, be able to process tasks asynchronously and quickly share data between the services. You also might want to run multiple instances of each microservice spanning multiple servers to make it highly available in a production environment…you see where I’m going with this.**

**You now have to think about challenges like:**

* **Service Discovery**
* **Load Balancing**
* **Secrets/configuration/storage management**
* **Health checks**
* **Auto-[scaling/restart/healing] of containers and nodes**
* **Zero-downtime deploys**

**This is where container orchestration platforms become extremely useful and powerful, because they offer a solution for most of those challenges.**

So what choices do we have? Today, the main players are Kubernetes, AWS ECS and Docker Swarm, in order of popularity. Kubernetes has the largest community and is the most popular by a big margin (usage doubled in 2016, expected to 3–4x in 2017). There is also Kontena, which is much easier to setup compared to Kubernetes, but not as configurable and not very mature.

## Volumes

A Kubernetes volume has an explicit lifetime - the same as the Pod that encloses it. Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.

## Stateful vs Stateless Applications on Kubernetes

<https://linuxhint.com/stateful-vs-stateless-kubernetes/>

An important criterion to consider before running a new application, in production, is the app’s underlying architecture. A term often used in this context is that the application is ‘stateless’ or that the application is ‘stateful’. Both types have their own pros and cons. We will be having a Kubernetes cluster in the back of our mind when we talk about an application or a service running in production. You can install a Kubernetes cluster of your own on the cloud or you can have it running as a single node on your PC to get some practice with it.

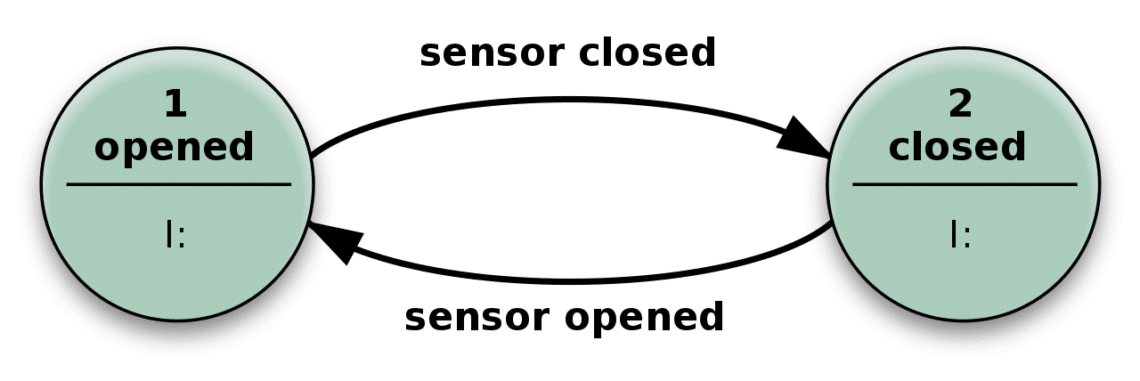
Let’s start with a naïve definition of ‘statelessness’ and then slowly progress to a more rigorous and real-world view.

A stateless application is one which depends on no persistent storage. The only thing your cluster is responsible for is the code, and other static content, being hosted on it. That’s it, no changing databases, no writes and no left over files when the pod is deleted.

A stateful application, on the other hand, has several other parameters it is supposed to look after in the cluster. There are dynamic databases which, even when the app is offline or deleted, persist on the disk. On a distributed system, like Kubernetes, this raises several issues. We will look at them in detail, but first let’s clarify some misconceptions.

### ****Stateless services aren’t actually ‘stateless’****

What does it mean when we say the state of a system? Well, let’s consider the following simple example of an automatic door.



The door opens when the sensor detects someone approaching, and it closes once the sensor gets no relevant input.

In practice, your stateless app is similar to this mechanism above. It can have many more states than just closed or open, and many different types of input as well making it more complex but essentially the same.

It can solve complicated problems by just receiving an input and performing actions which depend on both the input, and ‘state’ it is in. The number of possible states are predefined.

So, statelessness is a misnomer.

Stateless applications, in practice, can also cheat a little by saving details about, say, the client sessions on the client itself (HTTP cookies are a great example) and still have a nice statelessness which would make them run flawlessly on the cluster.

For example, a client’s session details like what products were saved in the cart and not checked out can all be stored on the client and the next time a session begins these relevant details are also recollected.

On a Kubernetes cluster, a stateless application has no persistent storage or volume associated with it. From an operations perspective, this is great news. Different pods all across the cluster can work independently with multiple requests coming to them simultaneously. If something goes wrong, you can just restart the application and it will go back to the initial state with little downtime.

### ****Stateful services and the CAP theorem****

The stateful services, on the other hand, will have to worry about lots and lots of edge-cases and weird issues. A pod is accompanied with at least one volume and if the data in that volume is corrupted then that persists even if the entire cluster gets rebooted.

For example, if you are running a database on a Kubernetes cluster, all the pods must have a local volume for storing the database. All of the data must be in perfect sync.

So, if someone modifies an entry to the database, and that was done on pod A, and a read request comes on pod B to see that modified data, then pod B must show that latest data or give you an error message. This is known as consistency.

**Consistency**, in the context of a Kubernetes cluster, means every read receives the most recent write or an error message.

But this cuts against **availability**, one of the most important reasons for having a distributed system. Availability implies that your application functions as close to perfection as possible, around the clock, with as little error as possible.

One may argue that you can avoid all of this if you have just one centralized database which is responsible for handling all of the persistent storage needs. Now we are back to having a single point of failure, which is yet another problem that a Kubernetes clusters is supposed to solve in the first place.

You need to have a decentralized way of storing persistent data in a cluster. Commonly referred to as network partitioning. Moreover, your cluster must be able to survive the failure of nodes running the stateful application. This is known as **partition tolerance**.

Any stateful service (or application), being run on a Kubernetes cluster, needs to have a balance between these three parameters. In the industry, it is known as the CAP theorem where the tradeoffs between Consistency and Availability are considered in presence of network Partitioning.

## Kubernetes Core Concepts

**Master node**: Runs multiple controllers that are responsible for the health of the cluster, replication, scheduling, endpoints (linking Services and Pods), Kubernetes API, interacting with the underlying cloud providers etc. Generally it makes sure everything is running as it should be and looks after worker nodes.

**Worker node (minion)**: Runs the Kubernetes agent that is responsible for running Pod containers via Docker or rkt, requests secrets or configurations, mounts required Pod volumes, does health checks and reports the status of Pods and the node to the rest of the system.

**Pod**: The smallest and simplest unit in the Kubernetes object model that you can create or deploy. It represents a running process in the cluster. Can contain one or multiple containers.

**Deployment**: Provides declarative updates for Pods (like the template for the Pods), for example the Docker image(s) to use, environment variables, how many Pod replicas to run, labels, node selectors, volumes etc.

**DaemonSet**: It’s like a Deployment but instead runs a copy of a Pod (or multiple) on all (or some) nodes. Useful for things like log collection daemons (sumologic, fluentd), node monitoring daemons (datadog) and cluster storage daemons (glusterd).

**ReplicaSet**: Controller that ensures a specified number of Pod replicas (defined in the Deployment) is running at any given time.

**Service**: An abstraction which defines a logical set of Pods and a policyby which to access them (determined by a label selector). Generally it’s used to expose Pods to other services within the cluster (using targetPort) or externally (using NodePort or LoadBalancer objects).

A screenshot of a computer

AI-generated content may be incorrect.

## Kubernetes - Master Machine Components

**etcd:** It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key value store that can be distributed among multiple nodes. It is accessible only by Kubernetes API server as it may have some sensitive information. It is a distributed key value Store which is accessible to all.

**API Server:** Kubernetes is an API server which provides all the operation on cluster using the API. API server implements an interface, which means different tools and libraries can readily communicate with it. Kubeconfig is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.

**Controller Manager:** This component is responsible for most of the collectors that regulates the state of cluster and performs a task. In general, it can be considered as a daemon which runs in nonterminating loop and is responsible for collecting and sending information to API server. It works toward getting the shared state of cluster and then make changes to bring the current status of the server to the desired state. The key controllers are replication controller, endpoint controller, namespace controller, and service account controller. The controller manager runs different kind of controllers to handle nodes, endpoints, etc.

**Scheduler:** This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload. It is responsible for tracking utilization of working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to available nodes. The scheduler is responsible for workload utilization and allocating pod to new node.

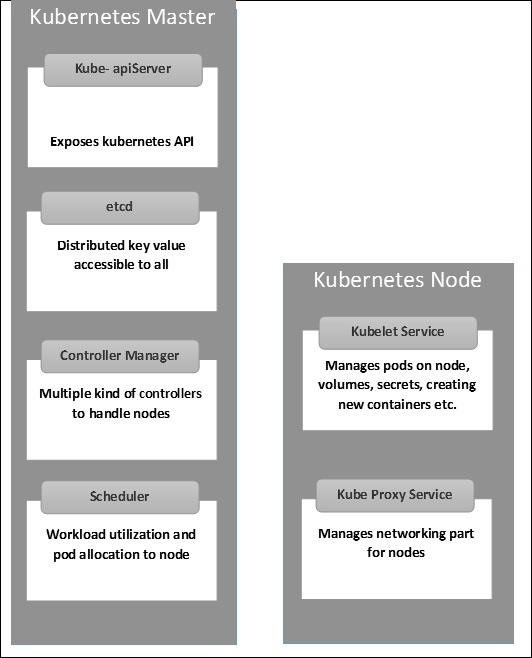
## Kubernetes - Node Components

**Docker:** The first requirement of each node is Docker which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment.

**Kubelet Service:** This is a small service in each node responsible for relaying information to and from control plane service. It interacts with etcd store to read configuration details and wright values. This communicates with the master component to receive commands and work. The kubelet process then assumes responsibility for maintaining the state of work and the node server. It manages network rules, port forwarding, etc.

**Kubernetes Proxy Service:** This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containers’ health checkup, etc.

## Kubernetes - Master and Node Structure

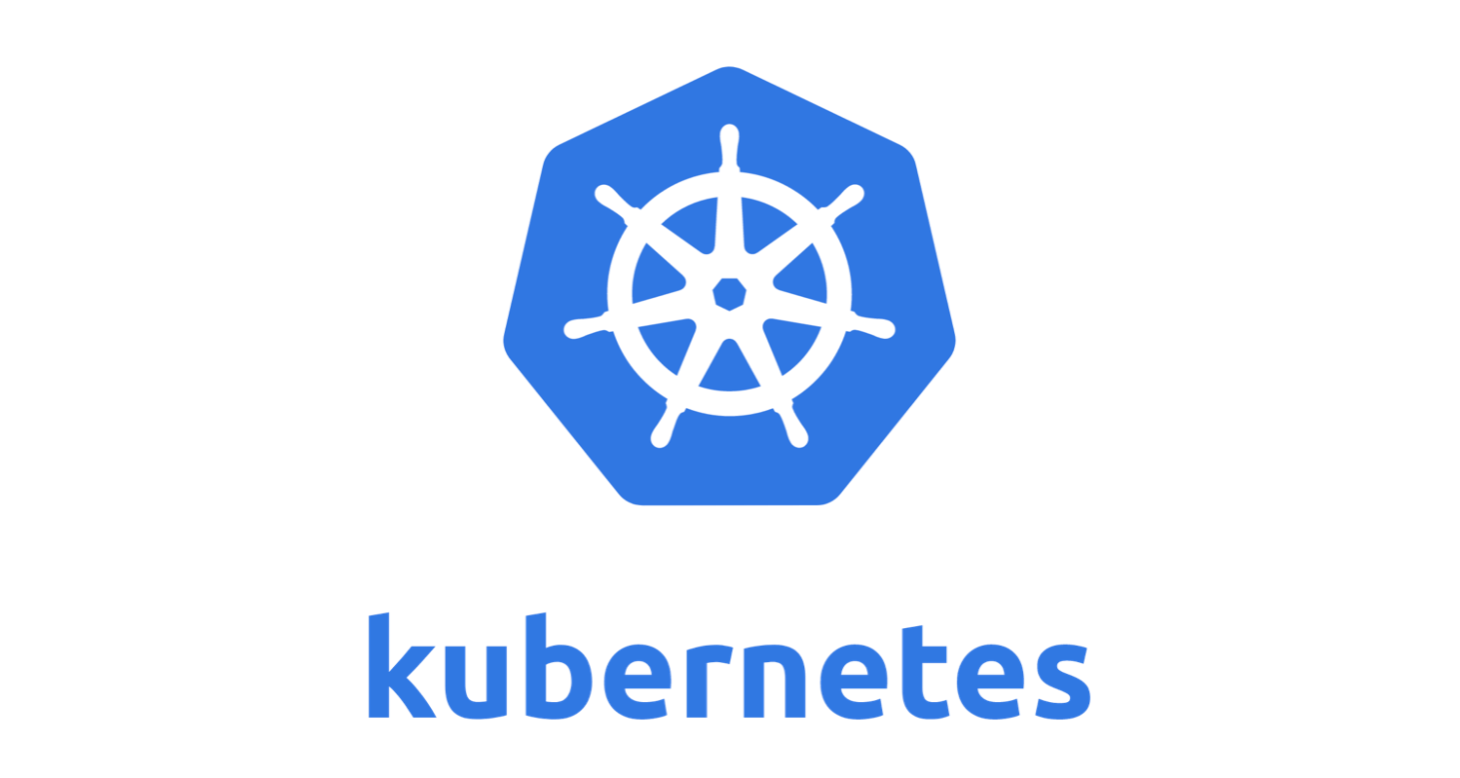


# What is the Difference Between Kubernetes and Docker Swarm

## Introduction

We have been coming across many container management engines, and while Kubernetes is the most popular container orchestration engine, Docker has Docker Swarm to do the same job and it easily integrates with Docker.

## ****Kubernetes****



Kubernetes is an open source system for managing containerized application in a clustered environment. Using Kubernetes in the right way helps the DevOps team to automatically scale an application up or down and update it with zero downtime.

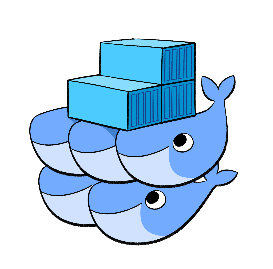
### ****Pros of Using Kubernetes****

* **It's fast:**When it comes to continuously deploying new features without downtime, Kubernetes is a perfect choice. The goal of Kubernetes is to update an application with constant uptime. Its speed is measured through a number of features you can ship per hour while maintaining an available service.
* **Adheres to the principals of immutable infrastructure:**In a traditional way, if anything goes wrong with multiple updates, you don’t have any record of how many updates you deployed and at which point the error occurred. In immutable infrastructure, if you wish to update an application, you need to build a container image with a new tag and deploy it, killing the old container with an old image version. In this way, you will have a record and get an insight of what you did and if there is any error, you can easily roll back to the previous image.
* **Provides declarative configuration:**Users can know in what state the system should be to avoid errors. Source control, unit tests, and other traditional tools can’t be used with imperative configurations, but can be used with declarative configurations.
* **Deploy and update software at scale:**Scaling is easy due to the immutable, declarative nature of Kubernetes. Kubernetes offers several useful features for scaling purposes.
* **Horizontal Infrastructure Scaling:** Operations are done at the individual server level to apply horizontal scaling. The latest servers can be added or detached effortlessly.
* **Auto-scaling:** Based on the usage of CPU resources or other application metrics, you can change the number of containers that are running
* **Manual scaling:** You can manually scale the number of running containers through a command or the interface.
* **Replication controller:** The replication controller makes sure that the cluster has a specified number of equivalent pods in a running condition. If there are too many pods, a replication controller can remove extra pods or vice-versa.
* **Handles the availability of the application:**Kubernetes checks the health of nodes and containers as well as provides self-healing and auto-replacement if in-case pod crashes due to an error. Moreover, it distributes the load across multiple pods to balance the resources quickly during accidental traffic.
* **Storage Volume:**In Kubernetes, data is shared across the containers, but if pods get killed volume is automatically removed. Moreover, data is stored remotely, if the pod is moved to another node, the data will remain until it is deleted by the user.

### ****Cons of Using Kubernetes****

* **Initial process takes time:**When a new process is created, you have to wait for the app to commence before it is available to the users. If you are migrating to Kubernetes, modifications in the codebase need to be done to make a start process more efficient so that users don’t have a bad experience.
* **Migrating to stateless requires many efforts:**If your application is clustered or stateless, extra pods will not get configured and will have to rework on the configurations within your applications.
* **The installation process is tedious:**It is difficult to set up Kubernetes on your cluster if you are not using any cloud provider like Azure, Google or Amazon.

## ****Docker Swarm****



Docker Swarm is Docker’s own native clustering solution for Docker containers which has an advantage of being tightly integrated into the ecosystem of Docker and uses its own API. It monitors the number of containers spread across clusters of servers and is the most convenient way to create clustered docker application without additional hardware. It provides you with a small-scale but useful orchestration system for the Dockerized app.

### ****Pros of Using Docker Swarm****

* **Runs at a faster pace:**When you were using a virtual environment, you may have realized that it takes a long time and includes the tedious procedure of booting up and starting the application that you want to run. With Docker Swarm, this is not a problem. Docker Swarm removes the need to boot up a full virtual machine and enables the app to run in a virtual and software-defined environment quickly and helps in DevOps implementation.
* **Documentation provides every bit of information:**The Docker team stands out when it comes to documentation! Docker is rapidly evolving and has received great applause for the entire platform. When a version gets released in a short interval of time, some platforms don't maintain/take care to maintain documentation. But Docker Swarm never compromises with it. If the information only applies to certain versions of a Docker Swarm, the documentation makes sure that all information is updated.
* **Provides simple and fast configuration:**One of the key benefits of Docker Swarm is that it simplifies matters. Docker Swarm enables the user to take their own configuration, put it into a code and deploy it without any hassle. As Docker Swarm can be used in various environments, requirements are just not bound by the environment of the application.
* **Ensures that application is isolated:**Docker Swarm takes care that each container is isolated from the other containers and has its own resources. Various containers can be deployed for running the separate application in different stacks. Apart from this, Docker Swarm cleans app removal as each application runs on its own container. If the application is no longer required, you can delete its container. It won’t leave any temporary or configuration files on your host OS.
* **Version control and component reuse:**With Docker Swarm, you can track consecutive versions of a container, examine differences or roll-back to the preceding versions. Containers reuse the components from the preceding layers which makes them noticeably lightweight.

### ****Cons of Using Docker Swarm****

* **Docker is platform-dependent:**Docker Swarm is a Linux-agonistic platform. Although Docker supports Windows and Mac OS X, it utilizes virtual machines to run on a non-Linux platform. An application which is designed to run in docker container on Windows can’t run on Linux and vice versa.
* **Doesn’t provide a storage option:**Docker Swarm doesn’t provide a hassle-free way to connect containers to storage and this is one of the major disadvantages. Its data volumes require a lot of improvising on the host and manual configurations. If you’re expecting Docker Swarm to solve the storage issues, it may get done but not in an efficient and user-friendly way.
* **Poor monitoring:**Docker Swarm provides the basic information about the container and if you are looking for a basic monitoring solution, then the stats command is suffice. If you are looking for an advanced monitoring than Docker Swarm is not an option. Although there are third-party tools available like CAdvisor which offers more monitoring, it is not feasible to collect more data about containers in real-time with Docker itself.

**Docker and Kubernetes are Different; But not Rivals**

As discussed earlier, Kubernetes and Docker both work at different levels but both can be used together. Kubernetes can be integrated with the Docker engine to carry out the scheduling and execution of Docker containers. As Docker and Kubernetes are both container orchestrators, they both can help to manage the number containers and also help in DevOps implementation. Both can automate most of the tasks that are involved in running containerized infrastructure and are open source software projects, governed by an Apache Licence 2.0. Apart from this, both use YAML-formatted files to govern how the tools orchestrate container clusters. When both of them are used together, both Docker and Kubernetes are the best tools for deploying modern cloud architecture. In the absence of Docker Swarm, both Kubernetes and Docker complement each other.

Kubernetes uses Docker as the main container engine solution and Docker recently announced that it can support Kubernetes as the orchestration layer of its enterprise edition. Apart from this, Docker approves certified Kubernetes program, which makes sure that all Kubernetes APIs functions as expected. Kubernetes uses the features of Docker Enterprise like Secure Image management, in which Docker EE provides image scanning to check whether there is an issue in the image used in container. Another is Secure Automation in which organizations can remove inefficiencies such as scanning image for vulnerabilities.

## ****Kubernetes or Docker: Which Is the Perfect Choice?****

### ****Use Kubernetes if:****

* You are looking for a mature deployment and monitoring option.
* You are looking for fast and reliable response times.
* You are looking to develop a complex application and requires high resource computing without restrictions.
* You have a pretty big cluster.

### ****Use Docker if,****

* You are looking to initiate with the tool without spending much time on configuration and installation
* You are looking to develop a basic and standard application which is sufficient enough with default docker image
* Testing and running the same application on the different operating system is not an issue for you
* You want docker API experience and compatibility

## ****Final Thoughts: Kubernetes and Docker as Friends****

Whether you choose Kubernetes or Docker, both are considered the best and possess considerable differences. The best way to decide between the two of them is probably to consider which one you already know better or which one fits your existing software stack. If you need to develop the complex app, use Kubernetes, and if you are looking to develop the small-scale app, use Docker Swarm. Moreover, choosing the right one is a very comprehensive task and solely depends on your project requirements and target audience as well.

# Installation

## Check Ubuntu Version

## Terminal

lsb\_release -a

## GUI

Settings -> Details -> About

## Installation Steps

* Docker Desktop for Windows: [https://docs.docker.com/docker-for-windows/install/](https://slack-redir.net/link?url=https%3A%2F%2Fdocs.docker.com%2Fdocker-for-windows%2Finstall%2F)
* Docker on Windows 10 Home:
  + Install Oracle Virtual Box and then create a new VM using docker-machine.
  + [https://docs.docker.com/machine/get-started/](https://slack-redir.net/link?url=https%3A%2F%2Fdocs.docker.com%2Fmachine%2Fget-started%2F)
  + <https://docs.docker.com/machine/drivers/virtualbox/>
* Docker Desktop for Mac: [https://docs.docker.com/docker-for-mac/install/](https://slack-redir.net/link?url=https%3A%2F%2Fdocs.docker.com%2Fdocker-for-mac%2Finstall%2F)
* Docker on Linux: [https://runnable.com/docker/install-docker-on-linux](https://slack-redir.net/link?url=https%3A%2F%2Frunnable.com%2Fdocker%2Finstall-docker-on-linux)

1. ~~Open a command line window with administrator privileges.~~
2. ~~Then use the chocolatey package manager to install minikube:~~
   1. ~~Install Chocolatey from https://chocolatey.org/:~~

*~~With cmd.exe as Admin:~~*

~~@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -InputFormat None -ExecutionPolicy Bypass -Command "iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"~~

*~~With Powershell:~~*

~~With PowerShell, there is an additional step. You must ensure~~[**~~Get-ExecutionPolicy~~**](https://go.microsoft.com/fwlink/?LinkID=135170)~~is not Restricted. We suggest using Bypass to bypass the policy to get things installed or AllSigned for quite a bit more security.~~

* ~~Run Get-ExecutionPolicy. If it returns Restricted, then run Set-ExecutionPolicy AllSigned or Set-ExecutionPolicy Bypass -Scope Process.~~
* ~~Now run the following command:  (copy command text)~~

~~Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))~~

#### ~~Additional considerations~~

**~~NOTE:~~**~~Please inspect~~[**~~https://chocolatey.org/install.ps1~~**](https://chocolatey.org/install.ps1)~~prior to running any of these scripts to ensure safety. We already know it's safe, but you should verify the security and contents of~~**~~any~~**~~script from the internet you are not familiar with. All of these scripts download a remote PowerShell script and execute it on your machine.~~

~~Login for minikube VM is docker / tcuser~~

* 1. ~~Then install minikube:~~

~~choco install minikube~~

1. ~~Next install the kubernetes command line program with the chocolatey package manager:~~

~~choco install kubernetes-cli~~

**~~Note: .minikube and .kube folders are in C:\Users\Ajay Singala~~**

**~~To Install minikube on HyperV:~~**

1. ~~Create an external Virtual Switch in Hyper-V Virtual Switch Manager (here, "myswitch")~~
2. ~~Run the following command to start the minikube VM with our applied changes:~~

~~minikube start --vm-driver hyperv --hyperv-virtual-switch myswitch~~

1. ~~Once the VM is running we will have two more steps to do to address a bug (Kube cluster shutsdown after a few minutes MiniKube VM still running) in minikube for Windows. We need to turn off Dynamic Memory for the minikube VM:~~
   1. ~~Run:~~

~~minikube stop --alsologtostderr~~

* 1. ~~In HyperV Manager, disable "Enable Dynamic Memory" for the minikube VM~~
  2. ~~Then, run:~~

~~minikube start~~

**~~To Install minikube on VirtualBox:~~**

1. ~~Make sure Hyper-V is disabled~~
2. ~~Run:~~

~~minikube.exe start –-vm-driver=virtualbox~~

1. ~~Confirm the installation is ready~~

~~kubectl get pods -n kube-system~~

1. ~~Or Run to open the dashboard in a browser:~~

~~minikube dashboard~~

## Install Docker on Ubuntu 20.04

### Initial Settings

When you see the error message:

*"[username] is not in the sudoers file. This incident will be reported."*

it means your user does not have permission to run commands as root using sudo. To fix this in a VirtualBox Ubuntu VM, follow these steps:

**Solution: Add Your User to the Sudo Group**

**1. Log in as Root**

* Open a terminal.
* Type:

su -

* Enter the root password (this is usually the password you set during installation).

**2. Add User to Sudo Group**

* Run:

usermod -aG sudo yourusername

Replace yourusername with your actual username.

**3. Restart or Log Out and In**

* For the changes to take effect, either restart the VM or log out and log back in.

**Alternative: Edit the Sudoers File Directly**

* As root, open the sudoers file:

nano /etc/sudoers

* Add this line below the root entry:

yourusername ALL=(ALL:ALL) ALL

* Save and exit.

### Steps

**Update Your System**

sudo apt-get update

sudo apt upgrade -y

**Install Prerequisite Packages**

sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

gnupg \

lsb\_release

**OR**

sudo apt-get install \

apt-transport-https \

ca-certificates curl \

gnupg-agent \

software-properties-common -y

**Add Docker’s Official GPG Key**

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

**Add the Docker Repository**

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

**Install Docker CE**

sudo apt update

sudo apt install docker-ce -y

**Verify Docker is Running**

sudo systemctl status docker

**Run Docker Without sudo:** Add your user to the docker group:

sudo usermod -aG docker $USER

*# Log out and log back in for this to take effect.*

**Test Docker Installation**

sudo docker run hello-world

docker ps

## Install Minikube on ubuntu 20.04

<https://minikube.sigs.k8s.io/docs/start/>

sudo curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube

minikube version

minikube start

(minikube start --memory=4096)

**Install kubectl:**

**Download:**

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

**Validate:** Downlod checksum file:

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"

**Validate the kubectl binary against the checksum file:**

echo "$(cat kubectl.sha256) kubectl" | sha256sum --check

**Install kubectl**

sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

kubectl version --client

kubectl get nodes

minikube node list

minikube node add –worker

minikube node list

# Enable Kubernetes on Docker and Enable Dashboard

<http://collabnix.com/kubernetes-dashboard-on-docker-desktop-for-windows-2-0-0-3-in-2-minutes/>

In Docker settings, click on Kubernetes and select checkboxes for Enable Kubernetes, Deploy Docker Stacks to K8s by default and Show system containers (advanced). k8s will start and be running in some time.

**Create alias for kubectl (on Linux):**

alias k=kubectl

**Verify:**

kubectl version

## Setup the k8s Dashboard

kubectl version

kubectl apply -f <https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml>

kubectl proxy

Open the following in your browser:

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

You will get:

"status": "Failure",

"message": "no endpoints available for service \"Kubernetes-dashboard\"",

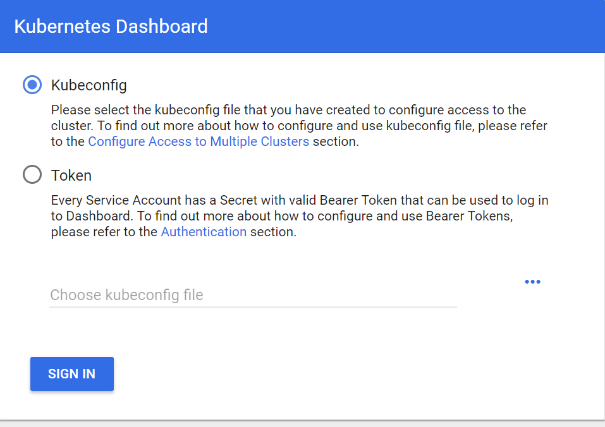
"reason": "ServiceUnavailable",

"code": 503

Open this url in your browser:

<http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/login>

Will show the following:



Enter the following commands on the terminal window:

$TOKEN=((kubectl -n kube-system describe secret default | Select-String "token:") -split " +")[1]

kubectl config set-credentials docker-for-desktop --token="${TOKEN}"

Back in the browser, select Kubeconfig and select the “config” file from the folder C:\USERS\<username>\.kube, then click on Signin.

The dashboard should show up.

For setting up the dashboard on Ubuntu / Linux, follow instructions on this url: <https://www.tutorialspoint.com/kubernetes/kubernetes_dashboard_setup.htm>

# Kubernetes Cluster Setup

**Note** − The setup is shown for Ubuntu machines. The same can be set up on other Linux machines as well.

## Prerequisites

**Installing Docker** − Docker is required on all the instances of Kubernetes. Following are the steps to install the Docker.

**Step 1** − Log on to the machine with the root user account.

**Step 2** − Update the package information. Make sure that the apt package is working.

**Step 3** − Run the following commands.

$ sudo apt-get update

$ sudo apt-get install apt-transport-https ca-certificates

**Step 4** − Add the new GPG key.

$ sudo apt-key adv \

--keyserver hkp://ha.pool.sks-keyservers.net:80 \

--recv-keys 58118E89F3A912897C070ADBF76221572C52609D

$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" | sudo tee

/etc/apt/sources.list.d/docker.list

**Step 5** − Update the API package image.

$ sudo apt-get update

Once all the above tasks are complete, you can start with the actual installation of the Docker engine. However, before this you need to verify that the kernel version you are using is correct.

## Install Docker Engine

Run the following commands to install the Docker engine.

**Step 1** − Logon to the machine.

**Step 2** − Update the package index.

$ sudo apt-get update

**Step 3** − Install the Docker Engine using the following command.

$ sudo apt-get install docker-engine

**Step 4** − Start the Docker daemon.

$ sudo apt-get install docker-engine

**Step 5** − To very if the Docker is installed, use the following command.

$ sudo docker run hello-world

## Install etcd 2.0

This needs to be installed on Kubernetes Master Machine. In order to install it, run the following commands.

$ curl -L https://github.com/coreos/etcd/releases/download/v2.0.0/etcd

-v2.0.0-linux-amd64.tar.gz -o etcd-v2.0.0-linux-amd64.tar.gz ->1

$ tar xzvf etcd-v2.0.0-linux-amd64.tar.gz ------>2

$ cd etcd-v2.0.0-linux-amd64 ------------>3

$ mkdir /opt/bin ------------->4

$ cp etcd\* /opt/bin ----------->5

In the above set of command −

* First, we download the **etcd**. Save this with specified name.
* Then, we have to un-tar the tar package.
* We make a dir. inside the /opt named bin.
* Copy the extracted file to the target location.

Now we are ready to build Kubernetes. We need to install Kubernetes on all the machines on the cluster.

$ git clone https://github.com/GoogleCloudPlatform/kubernetes.git

$ cd kubernetes

$ make release

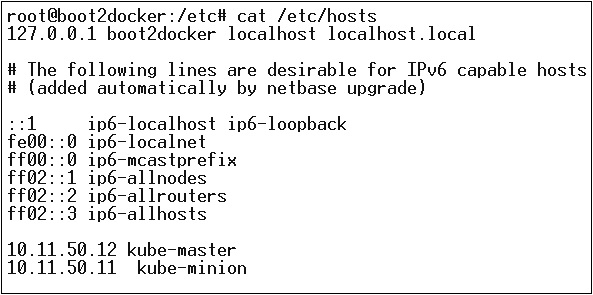
The above command will create a **\_output** dir in the root of the kubernetes folder. Next, we can extract the directory into any of the directory of our choice /opt/bin, etc.

Next, comes the networking part wherein we need to actually start with the setup of Kubernetes master and node. In order to do this, we will make an entry in the host file which can be done on the node machine.

$ echo "<IP address of master machine> kube-master

< IP address of Node Machine>" >> /etc/hosts

Following will be the output of the above command.



Now, we will start with the actual configuration on Kubernetes Master.

First, we will start copying all the configuration files to their correct location.

$ cp <Current dir. location>/kube-apiserver /opt/bin/

$ cp <Current dir. location>/kube-controller-manager /opt/bin/

$ cp <Current dir. location>/kube-kube-scheduler /opt/bin/

$ cp <Current dir. location>/kubecfg /opt/bin/

$ cp <Current dir. location>/kubectl /opt/bin/

$ cp <Current dir. location>/kubernetes /opt/bin/

The above command will copy all the configuration files to the required location. Now we will come back to the same directory where we have built the Kubernetes folder.

$ cp kubernetes/cluster/ubuntu/init\_conf/kube-apiserver.conf /etc/init/

$ cp kubernetes/cluster/ubuntu/init\_conf/kube-controller-manager.conf /etc/init/

$ cp kubernetes/cluster/ubuntu/init\_conf/kube-kube-scheduler.conf /etc/init/

$ cp kubernetes/cluster/ubuntu/initd\_scripts/kube-apiserver /etc/init.d/

$ cp kubernetes/cluster/ubuntu/initd\_scripts/kube-controller-manager /etc/init.d/

$ cp kubernetes/cluster/ubuntu/initd\_scripts/kube-kube-scheduler /etc/init.d/

$ cp kubernetes/cluster/ubuntu/default\_scripts/kubelet /etc/default/

$ cp kubernetes/cluster/ubuntu/default\_scripts/kube-proxy /etc/default/

$ cp kubernetes/cluster/ubuntu/default\_scripts/kubelet /etc/default/

The next step is to update the copied configuration file under /etc. dir.

Configure etcd on master using the following command.

$ ETCD\_OPTS = "-listen-client-urls = http://kube-master:4001"

## Configure kube-apiserver

For this on the master, we need to edit the **/etc/default/kube-apiserver** file which we copied earlier.

$ KUBE\_APISERVER\_OPTS = "--address = 0.0.0.0 \

--port = 8080 \

--etcd\_servers = <The path that is configured in ETCD\_OPTS> \

--portal\_net = 11.1.1.0/24 \

--allow\_privileged = false \

--kubelet\_port = < Port you want to configure> \

--v = 0"

## Configure the kube Controller Manager

We need to add the following content in **/etc/default/kube-controller-manager**.

$ KUBE\_CONTROLLER\_MANAGER\_OPTS = "--address = 0.0.0.0 \

--master = 127.0.0.1:8080 \

--machines = kube-minion \ -----> #this is the kubernatics node

--v = 0

Next, configure the kube scheduler in the corresponding file.

$ KUBE\_SCHEDULER\_OPTS = "--address = 0.0.0.0 \

--master = 127.0.0.1:8080 \

--v = 0"

Once all the above tasks are complete, we are good to go ahead by bring up the Kubernetes Master. In order to do this, we will restart the Docker.

$ service docker restart

## Kubernetes Node Configuration

Kubernetes node will run two services the **kubelet and the kube-proxy**. Before moving ahead, we need to copy the binaries we downloaded to their required folders where we want to configure the kubernetes node.

Use the same method of copying the files that we did for kubernetes master. As it will only run the kubelet and the kube-proxy, we will configure them.

$ cp <Path of the extracted file>/kubelet /opt/bin/

$ cp <Path of the extracted file>/kube-proxy /opt/bin/

$ cp <Path of the extracted file>/kubecfg /opt/bin/

$ cp <Path of the extracted file>/kubectl /opt/bin/

$ cp <Path of the extracted file>/kubernetes /opt/bin/

Now, we will copy the content to the appropriate dir.

$ cp kubernetes/cluster/ubuntu/init\_conf/kubelet.conf /etc/init/

$ cp kubernetes/cluster/ubuntu/init\_conf/kube-proxy.conf /etc/init/

$ cp kubernetes/cluster/ubuntu/initd\_scripts/kubelet /etc/init.d/

$ cp kubernetes/cluster/ubuntu/initd\_scripts/kube-proxy /etc/init.d/

$ cp kubernetes/cluster/ubuntu/default\_scripts/kubelet /etc/default/

$ cp kubernetes/cluster/ubuntu/default\_scripts/kube-proxy /etc/default/

We will configure the **kubelet** and **kube-proxy conf** files.

We will configure the **/etc/init/kubelet.conf**.

$ KUBELET\_OPTS = "--address = 0.0.0.0 \

--port = 10250 \

--hostname\_override = kube-minion \

--etcd\_servers = http://kube-master:4001 \

--enable\_server = true

--v = 0"

/

For kube-proxy, we will configure using the following command.

$ KUBE\_PROXY\_OPTS = "--etcd\_servers = http://kube-master:4001 \

--v = 0"

/etc/init/kube-proxy.conf

Finally, we will restart the Docker service.

$ service docker restart

Now we are done with the configuration. You can check by running the following commands.

$ /opt/bin/kubectl get minions

# Tutorial #0: Quick Start

## Pods

alias k= kubectl

k get pods

kubectl run nginx --image=nginx

k get pods

kubectl run bus --image=busybox

k get pods

k delete pod busy

k get pods

k get pod nginx

k describe pod nginx

k get pod nginx -o wide

k get pod nginx -o yaml

k get pod nginx -o yaml > pod.yaml

Update the yaml to this:

A screen shot of a computer

AI-generated content may be incorrect.

k create -f pod.yaml

k get pod

k delete -f pod.yaml

k get pod

## Deployments

k get deployment

k create deployment -h

k create deployment nginx-dep --image=nginx

OR

k create deploy nginx-dep --image=nginx

k get deploy

k get pod

k delete pod nginx2

k get pod

k get deploy nginx-dep

k get deploy nginx-dep -o wide

k get deploy nginx-dep -o yaml > dep.yaml

Open the dep.yaml file and change no. of replicas:

k edit deploy nginx-dep

A screen shot of a computer

AI-generated content may be incorrect.

k get pod (*see 3 pods running*)

***OR do this:***

kubectl apply -f dep.yaml

copy the name of one of the pods

k delete pod <name of pod>

k get pod -w (*new pod created auto*)

Again edit dep.yaml and change no of replicas to 5 and check no. of running pods. It should show 5.

k get all

This will show 5 replica sets.

Again edit dep.yaml and change no. of replicas to 2 and “k get all” will show 2 repilca sets.

Copy name of replicaset

k get rs <replicaset name>

k get rs <replicaset name> -o yaml > rep.yaml

k expose deploy nginx-dep --port=80

k get svc

k get svc -o wide

k get svc -o yaml

Note the ip address under “clusterIP” and the “Port:targetPort”

k delete deploy nginx-dep

k get all (*shows the service even though the deployment has been deleted, but will no work*)

k delete svc nginx-dep

k get all

# Tutorial #1: Quick Start

Start minikube

**minikube start**

minikube start --memory=4096

Run the hello-minikube app

**~~kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.10 --port=8080~~**

**kubectl create deployment hello-minikube --image=k8s.gcr.io/echoserver:1.10 --port=8080**

Create a service to expose hello-minikube Pod as a Kubernetes service

**kubectl expose deployment hello-minikube --type=NodePort**

***# We have now launched an echoserver pod but we have to wait until the pod is up before curling/accessing it via the exposed service.***

***# To check whether the pod is up and running we can use the following:***

***# can use “kubectl get pods” as well***

**$ kubectl get pod**

View the service

**kubectl get services**

***# We can see the pod is now Running and we will now be able to curl it:***

**$ curl $(minikube service hello-minikube --url)**

Delete the service

**kubectl delete services hello-minikube**

Delete the deployment app

**kubectl delete deployment hello-minikube**

Tutorial 3: T

Stop minikube

**minikube stop**

OR

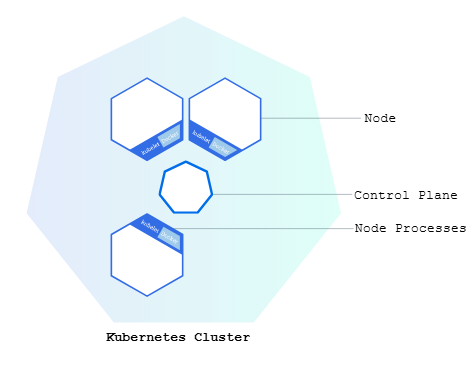
**minikube ssh (docker / tcuser)**

**sudo poweroff**

# Tutorial #2: Deploy, Expose as a Service, Multiple Instances, Rolling Updates

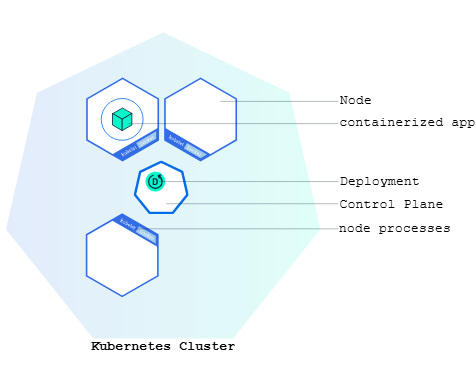
<https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/>

## Create a Cluster



## Using kubectl to Create a Deployment

<https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/>



### Deploy the app

kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1

kubectl get deployments

### View the app

Pods that are running inside Kubernetes are running on a private, isolated network. By default they are visible from other pods and services within the same Kubernetes cluster, but not outside that network. When we use kubectl, we're interacting through an API endpoint to communicate with our application.

The kubectl proxy command can create a proxy that will forward communications into the cluster-wide, private network. The proxy can be terminated by pressing control-C and won't show any output while it's running.

**You need to open a second terminal window to run the proxy.**

kubectl proxy

We now have a connection between our host (the terminal) and the Kubernetes cluster. The proxy enables direct access to the API from these terminals.

You can see all those APIs hosted through the proxy endpoint. For example, we can query the version directly through the API using the curl command:

curl http://localhost:8001/version

The API server will automatically create an endpoint for each pod, based on the pod name, that is also accessible through the proxy.

First we need to get the Pod name, and we'll store it in the environment variable POD\_NAME.

export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')

echo Name of the Pod: $POD\_NAME

You can access the Pod through the proxied API, by running:

curl http://localhost:8001/api/v1/namespaces/default/pods/$POD\_NAME:8080/proxy/

## Using a Service to Expose Your App

<https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/>

Kubernetes [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) are mortal. Pods have a [lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/). When a worker node dies, the Pods running on the Node are also lost. A [Replicaset](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) might then dynamically drive the cluster back to the desired state via the creation of new Pods to keep your application running. As another example, consider an image-processing backend with 3 replicas. Those replicas are exchangeable; the front-end system should not care about backend replicas or even if a Pod is lost and recreated. That said, each Pod in a Kubernetes cluster has a unique IP address, even Pods on the same Node, so there needs to be a way of automatically reconciling changes among Pods so that your applications continue to function.

*A Kubernetes Service is an abstraction layer which defines a logical set of Pods and enables external traffic exposure, load balancing and service discovery for those Pods.*

A [Service](https://kubernetes.io/docs/concepts/services-networking/service/) in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML or JSON, like all Kubernetes object manifests. The set of Pods targeted by a Service is usually determined by a *label selector* (see below for why you might want a Service without including a selector in the spec).

Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic. Services can be exposed in different ways by specifying a type in the spec of the Service:

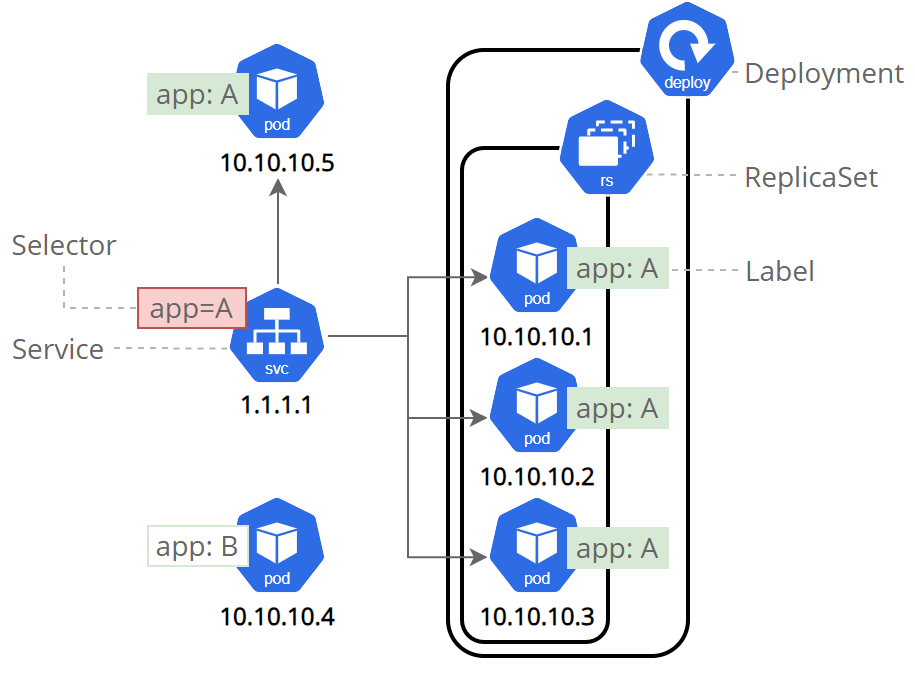
* *ClusterIP* (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.
* *NodePort* - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using NodeIP:NodePort. Superset of ClusterIP.
* *LoadBalancer* - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort.
* *ExternalName* - Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up. This type requires v1.7 or higher of kube-dns, or CoreDNS version 0.0.8 or higher.

### Services and Labels

A Service routes traffic across a set of Pods. Services are the abstraction that allows pods to die and replicate in Kubernetes without impacting your application. Discovery and routing among dependent Pods (such as the frontend and backend components in an application) are handled by Kubernetes Services.

Services match a set of Pods using [labels and selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/), a grouping primitive that allows logical operation on objects in Kubernetes. Labels are key/value pairs attached to objects and can be used in any number of ways:

* Designate objects for development, test, and production
* Embed version tags
* Classify an object using tags



Labels can be attached to objects at creation time or later on. They can be modified at any time. Let's expose our application now using a Service and apply some labels.

### Step 1: Creating a new Service

Let’s verify that our application is running. We’ll use the kubectl get command and look for existing Pods:

kubectl get pods

If no Pods are running then it means the objects from the previous tutorials were cleaned up. In this case, go back and recreate the deployment from the [Using kubectl to create a Deployment](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/#deploy-an-app) tutorial. Please wait a couple of seconds and list the Pods again. You can continue once you see the one Pod running.

Next, let’s list the current Services from our cluster:

kubectl get services

To expose the deployment to external traffic, we'll use the kubectl expose command with the --type=NodePort option:

kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080

We have now a running Service called kubernetes-bootcamp. Here we see that the Service received a unique cluster-IP, an internal port and an external-IP (the IP of the Node).

To find out what port was opened externally (for the type: NodePort Service) we’ll run the describe service subcommand:

kubectl describe services/kubernetes-bootcamp

Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

export NODE\_PORT="$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')"

echo "NODE\_PORT=$NODE\_PORT"

Now we can test that the app is exposed outside of the cluster using curl, the IP address of the Node and the externally exposed port:

curl http://"$(minikube ip):$NODE\_PORT"

**Note:**

If you're running minikube with Docker Desktop as the container driver, a minikube tunnel is needed. This is because containers inside Docker Desktop are isolated from your host computer.

In a separate terminal window, execute:

minikube service kubernetes-bootcamp --url

The output looks like this:

http://127.0.0.1:51082

! Because you are using a Docker driver on darwin, the terminal needs to be open to run it.

Then use the given URL to access the app:

curl 127.0.0.1:51082

And we get a response from the server. The Service is exposed.

### Step 2: Using labels

The Deployment created automatically a label for our Pod. With the describe deployment subcommand you can see the name (the *key*) of that label:

kubectl describe deployment

Let’s use this label to query our list of Pods. We’ll use the kubectl get pods command with -l as a parameter, followed by the label values:

kubectl get pods -l app=kubernetes-bootcamp

You can do the same to list the existing Services:

kubectl get services -l app=kubernetes-bootcamp

Get the name of the Pod and store it in the POD\_NAME environment variable:

export POD\_NAME="$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')"

echo "Name of the Pod: $POD\_NAME"

To apply a new label we use the label subcommand followed by the object type, object name and the new label:

kubectl label pods "$POD\_NAME" version=v1

This will apply a new label to our Pod (we pinned the application version to the Pod), and we can check it with the describe pod command:

kubectl describe pods "$POD\_NAME"

We see here that the label is attached now to our Pod. And we can query now the list of pods using the new label:

kubectl get pods -l version=v1

And we see the Pod.

### Step 3: Deleting a service

To delete Services you can use the delete service subcommand. Labels can be used also here:

kubectl delete service -l app=kubernetes-bootcamp

Confirm that the Service is gone:

kubectl get services

This confirms that our Service was removed. To confirm that route is not exposed anymore you can curl the previously exposed IP and port:

curl http://"$(minikube ip):$NODE\_PORT"

This proves that the application is not reachable anymore from outside of the cluster. You can confirm that the app is still running with a curl from inside the pod:

kubectl exec -ti $POD\_NAME -- curl http://localhost:8080

We see here that the application is up. This is because the Deployment is managing the application. To shut down the application, you would need to delete the Deployment as well.

## Running Multiple Instances of Your App

<https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/>

*Scaling* is accomplished by changing the number of replicas in a Deployment.

**Note:**

If you are trying this after the [previous section](https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/), then you may have deleted the service you created, or have created a Service of type: NodePort. In this section, it is assumed that a service with type: LoadBalancer is created for the kubernetes-bootcamp Deployment.

If you have *not* deleted the Service created in [the previous section](https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/), first delete that Service and then run the following command to create a new Service with its type set to LoadBalancer:

kubectl expose deployment/kubernetes-bootcamp --type="LoadBalancer" --port 8080

Running multiple instances of an application will require a way to distribute the traffic to all of them. Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment. Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods.

Once you have multiple instances of an application running, you would be able to do Rolling updates without downtime. We'll cover that in the next section of the tutorial. Now, let's go to the terminal and scale our application.

### Scaling a Deployment

To list your Deployments, use the get deployments subcommand:

kubectl get deployments

The output should be similar to:

NAME READY UP-TO-DATE AVAILABLE AGE

kubernetes-bootcamp 1/1 1 1 11m

We should have 1 Pod. If not, run the command again. This shows:

* *NAME* lists the names of the Deployments in the cluster.
* *READY* shows the ratio of CURRENT/DESIRED replicas
* *UP-TO-DATE* displays the number of replicas that have been updated to achieve the desired state.
* *AVAILABLE* displays how many replicas of the application are available to your users.
* *AGE* displays the amount of time that the application has been running.

To see the ReplicaSet created by the Deployment, run:

kubectl get rs

Notice that the name of the ReplicaSet is always formatted as [DEPLOYMENT-NAME]-[RANDOM-STRING]. The random string is randomly generated and uses the pod-template-hash as a seed.

Two important columns of this output are:

* *DESIRED* displays the desired number of replicas of the application, which you define when you create the Deployment. This is the desired state.
* *CURRENT* displays how many replicas are currently running. Next, let’s scale the Deployment to 4 replicas. We’ll use the kubectl scale command, followed by the Deployment type, name and desired number of instances:

kubectl scale deployments/kubernetes-bootcamp --replicas=4

To list your Deployments once again, use get deployments:

kubectl get deployments

The change was applied, and we have 4 instances of the application available. Next, let’s check if the number of Pods changed:

kubectl get pods -o wide

There are 4 Pods now, with different IP addresses. The change was registered in the Deployment events log. To check that, use the describe subcommand:

kubectl describe deployments/kubernetes-bootcamp

You can also view in the output of this command that there are 4 replicas now.

### Load Balancing

Let's check that the Service is load-balancing the traffic. To find out the exposed IP and Port we can use describe service as we learned in the previous part of the tutorial:

kubectl describe services/kubernetes-bootcamp

Create an environment variable called NODE\_PORT that has a value as the Node port:

export NODE\_PORT="$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')"

echo NODE\_PORT=$NODE\_PORT

Next, we’ll do a curl to the exposed IP address and port. Execute the command multiple times:

curl http://"$(minikube ip):$NODE\_PORT"

We hit a different Pod with every request. This demonstrates that the load-balancing is working.

**Note:**

If you're running minikube with Docker Desktop as the container driver, a minikube tunnel is needed. This is because containers inside Docker Desktop are isolated from your host computer.

In a separate terminal window, execute:

minikube service kubernetes-bootcamp --url

The output looks like this:

http://127.0.0.1:51082

! Because you are using a Docker driver on darwin, the terminal needs to be open to run it.

Then use the given URL to access the app:

curl 127.0.0.1:51082

### Scale Down

To scale down the Deployment to 2 replicas, run again the scale subcommand:

kubectl scale deployments/kubernetes-bootcamp --replicas=2

List the Deployments to check if the change was applied with the get deployments subcommand:

kubectl get deployments

The number of replicas decreased to 2. List the number of Pods, with get pods:

kubectl get pods -o wide

This confirms that 2 Pods were terminated.

## Performing a Rolling Update

<https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>

Users expect applications to be available all the time, and developers are expected to deploy new versions of them several times a day. In Kubernetes this is done with rolling updates. A **rolling update** allows a Deployment update to take place with zero downtime. It does this by incrementally replacing the current Pods with new ones. The new Pods are scheduled on Nodes with available resources, and Kubernetes waits for those new Pods to start before removing the old Pods.

In the previous module we scaled our application to run multiple instances. This is a requirement for performing updates without affecting application availability. By default, the maximum number of Pods that can be unavailable during the update and the maximum number of new Pods that can be created, is one. Both options can be configured to either numbers or percentages (of Pods). In Kubernetes, updates are versioned and any Deployment update can be reverted to a previous (stable) version.

Similar to application Scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update. An available Pod is an instance that is available to the users of the application.

Rolling updates allow the following actions:

* Promote an application from one environment to another (via container image updates)
* Rollback to previous versions
* Continuous Integration and Continuous Delivery of applications with zero downtime

### Update the version of the app

To list your Deployments, run the get deployments subcommand:

kubectl get deployments

To list the running Pods, run the get pods subcommand:

kubectl get pods

To view the current image version of the app, run the describe pods subcommand and look for the Image field:

kubectl describe pods

To update the image of the application to version 2, use the set image subcommand, followed by the deployment name and the new image version:

kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=docker.io/jocatalin/kubernetes-bootcamp:v2

The command notified the Deployment to use a different image for your app and initiated a rolling update. Check the status of the new Pods, and view the old one terminating with the get pods subcommand:

kubectl get pods

### Verify an update

First, check that the service is running, as you might have deleted it in previous tutorial step, run describe services/kubernetes-bootcamp. If it's missing, you can create it again with:

kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080

Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

export NODE\_PORT="$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')"

echo "NODE\_PORT=$NODE\_PORT"

Next, do a curl to the exposed IP and port:

curl http://"$(minikube ip):$NODE\_PORT"

Every time you run the curl command, you will hit a different Pod. Notice that all Pods are now running the latest version (v2).

You can also confirm the update by running the rollout status subcommand:

kubectl rollout status deployments/kubernetes-bootcamp

To view the current image version of the app, run the describe pods subcommand:

kubectl describe pods

In the Image field of the output, verify that you are running the latest image version (v2).

### Roll back an update

Let’s perform another update, and try to deploy an image tagged with v10:

kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10

Use get deployments to see the status of the deployment:

kubectl get deployments

Notice that the output doesn't list the desired number of available Pods. Run the get pods subcommand to list all Pods:

kubectl get pods

Notice that some of the Pods have a status of ImagePullBackOff.

To get more insight into the problem, run the describe pods subcommand:

kubectl describe pods

In the Events section of the output for the affected Pods, notice that the v10 image version did not exist in the repository.

To roll back the deployment to your last working version, use the rollout undo subcommand:

kubectl rollout undo deployments/kubernetes-bootcamp

The rollout undo command reverts the deployment to the previous known state (v2 of the image). Updates are versioned and you can revert to any previously known state of a Deployment.

Use the get pods subcommand to list the Pods again:

kubectl get pods

To check the image deployed on the running Pods, use the describe pods subcommand:

kubectl describe pods

The Deployment is once again using a stable version of the app (v2). The rollback was successful.

Remember to clean up your local cluster.

kubectl delete deployments/kubernetes-bootcamp services/kubernetes-bootcamp

# Tutorial #3

## Create a cluster

Check that it is properly installed, by running the minikube version command:

minikube version

Start the cluster (*if not already done*), by running the minikube start command:

minikube start

To check if kubectl is installed you can run the kubectl version command:

kubectl version

Let’s view the cluster details. We’ll do that by running kubectl cluster-info:

kubectl cluster-info

To view the nodes in the cluster, run the kubectl get nodes command:

kubectl get nodes

## Deploy an app

Check that kubectl is configured to talk to your cluster, by running the kubectl version command:

kubectl version

To view the nodes in the cluster, run the kubectl get nodescommand:

kubectl get nodes

Let’s run our first app on Kubernetes with the kubectl runcommand. The run command creates a new deployment. We need to provide the deployment name and app image location (include the full repository url for images hosted outside Docker hub). We want to run the app on a specific port so we add the --port parameter:

kubectl run kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1 --port=8080

Great! You just deployed your first application by creating a deployment. This performed a few things for you:

* searched for a suitable node where an instance of the application could be run (we have only 1 available node)
* scheduled the application to run on that Node
* configured the cluster to reschedule the instance on a new Node when needed

To list your deployments, use the get deployments command:

kubectl get deployments

We see that there is 1 deployment running a single instance of your app. The instance is running inside a Docker container on your node.

### View our app

Pods that are running inside Kubernetes are running on a private, isolated network. By default, they are visible from other pods and services within the same Kubernetes cluster, but not outside that network. When we use kubectl, we're interacting through an API endpoint to communicate with our application.

We will cover other options on how to expose your application outside the Kubernetes cluster in Module 4.

The kubectl command can create a proxy that will forward communications into the cluster-wide, private network. The proxy can be terminated by pressing control-C and won't show any output while its running.

We will open a second terminal window to run the proxy.

kubectl proxy

We now have a connection between our host (the online terminal) and the Kubernetes cluster. The proxy enables direct access to the API from these terminals.

You can see all those APIs hosted through the proxy endpoint, now available at through [http://localhost:8001](http://localhost:8001/). For example, we can query the version directly through the API using the curl command:

curl http://localhost:8001/version

The API server will automatically create an endpoint for each pod, based on the pod name, that is also accessible through the proxy.

First, we need to get the Pod name, and we'll store in the environment variable POD\_NAME:

export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')

echo Name of the Pod: $POD\_NAME

Now we can make an HTTP request to the application running in that pod:

curl http://localhost:8001/api/v1/namespaces/default/pods/$POD\_NAME/proxy/

The URL is the route to the API of the Pod.

## Viewing Pods and Nodes

Let’s verify that the application we deployed in the previous scenario is running. We’ll use the kubectl get command and look for existing Pods:

kubectl get pods

If no pods are running, list the Pods again.

Next, to view what containers are inside that Pod and what images are used to build those containers we run the describe pods command:

kubectl describe pods

We see here details about the Pod’s container: IP address, the ports used and a list of events related to the lifecycle of the Pod.

### Show app in the terminal

*(Can be taken from previous module)*

Run the proxy:

kubectl proxy

Now again, we'll get the Pod name and query that pod directly through the proxy. To get the Pod name and store it in the POD\_NAME environment variable:

export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')

echo Name of the Pod: $POD\_NAME

To see the output of our application, run a curl request.

curl http://localhost:8001/api/v1/namespaces/default/pods/$POD\_NAME/proxy/

The url is the route to the API of the Pod.

### View Container logs

Anything that the application would normally send to STDOUT becomes logs for the container within the Pod. We can retrieve these logs using the kubectl logs command:

kubectl logs $POD\_NAME

### Executing command on the container

We can execute commands directly on the container once the Pod is up and running. For this, we use the exec command and use the name of the Pod as a parameter. Let’s list the environment variables:

kubectl exec $POD\_NAME env

Again, worth mentioning that the name of the container itself can be omitted since we only have a single container in the Pod.

Next let’s start a bash session in the Pod’s container:

kubectl exec -ti $POD\_NAME bash

We have now an open console on the container where we run our NodeJS application. The source code of the app is in the server.js file:

cat server.js

You can check that the application is up by running a curl command:

curl localhost:8080

*Note: here we used localhost because we executed the command inside the NodeJS container*

To close your container connection type exit.

## Using a Service to expose your app

### Create a new service

Let’s verify that our application is running. We’ll use the kubectl get command and look for existing Pods:

kubectl get pods

Next let’s list the current Services from our cluster:

kubectl get services

We have a Service called kubernetes that is created by default when minikube starts the cluster. To create a new service and expose it to external traffic we’ll use the expose command with NodePort as parameter (minikube does not support the LoadBalancer option yet)

kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080

Let’s run again the get services command:

kubectl get services

We have now a running Service called kubernetes-bootcamp. Here we see that the Service received a unique cluster-IP, an internal port and an external-IP (the IP of the Node).

To find out what port was opened externally (by the NodePort option) we’ll run the describe service command:

kubectl describe services/kubernetes-bootcamp

Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

export NODE\_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')

echo NODE\_PORT=$NODE\_PORT

Now we can test that the app is exposed outside of the cluster using curl, the IP of the Node and the externally exposed port:

curl $(minikube ip):$NODE\_PORT

And we get a response from the server. The Service is exposed.

Determine the IP address and NODE\_PORT values and try the url from a browser.

### Using Labels

The Deployment created automatically a label for our Pod. With describe deployment command you can see the name of the label:

kubectl describe deployment

Let’s use this label to query our list of Pods. We’ll use the kubectl get pods command with -l as a parameter, followed by the label values:

kubectl get pods -l run=kubernetes-bootcamp

You can do the same to list the existing services:

kubectl get services -l run=kubernetes-bootcamp

Get the name of the Pod and store it in the POD\_NAME environment variable:

export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')

echo Name of the Pod: $POD\_NAME

To apply a new label, we use the label command followed by the object type, object name and the new label:

kubectl label pod $POD\_NAME app=v1

This will apply a new label to our Pod (we pinned the application version to the Pod), and we can check it with the describe pod command:

kubectl describe pods $POD\_NAME

We see here that the label is attached now to our Pod. And we can query now the list of pods using the new label:

kubectl get pods -l app=v1

And we see the Pod.

### Deleting a service

To delete Services you can use the delete servicecommand. Labels can be used also here:

kubectl delete service -l run=kubernetes-bootcamp

Confirm that the service is gone:

kubectl get services

This confirms that our Service was removed. To confirm that route is not exposed anymore you can curl the previously exposed IP and port:

curl $(minikube ip):$NODE\_PORT

This proves that the app is not reachable anymore from outside of the cluster. You can confirm that the app is still running with a curl inside the pod:

kubectl exec -ti $POD\_NAME curl localhost:8080

We see here that the application is up.

## Running multiple instances of your app

To list your deployments, use the get deployments command: kubectl get deployments

We should have 1 Pod. If not, run the command again. This shows:

The DESIRED state is showing the configured number of replicas.

The CURRENT state shows how many replicas are running now.

The UP-TO-DATE is the number of replicas that were updated to match the desired (configured) state.

The AVAILABLE state shows how many replicas are actually AVAILABLE to the users.

Next, let’s scale the Deployment to 4 replicas. We’ll use the kubectl scale command, followed by the deployment type, name and desired number of instances:

kubectl scale deployments/kubernetes-bootcamp --replicas=4

To list your Deployments once again, use get deployments:

kubectl get deployments

The change was applied, and we have 4 instances of the application available. Next, let’s check if the number of Pods changed:

kubectl get pods -o wide

There are 4 Pods now, with different IP addresses. The change was registered in the Deployment events log. To check that, use the describe command:

kubectl describe deployments/kubernetes-bootcamp

You can also view in the output of this command that there are 4 replicas now.

### Load balancing

Let’s check that the Service is load-balancing the traffic. To find out the exposed IP and Port we can use the describe service as we learned in the previous Module:

Check if the service exists:

kubectl get services

kubectl describe services/kubernetes-bootcamp

Create the service if it doesn’t exist:

kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080

kubectl describe services/kubernetes-bootcamp

Create an environment variable called NODE\_PORT that has a value as the Node port:

export NODE\_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')

echo NODE\_PORT=$NODE\_PORT

Next, we’ll do a curl to the exposed IP and port. Execute the command multiple times:

curl $(minikube ip):$NODE\_PORT

We hit a different Pod with every request. This demonstrates that the load-balancing is working.

### Scale down

To scale down the Service to 2 replicas, run again the scale command:

kubectl scale deployments/kubernetes-bootcamp --replicas=2

List the Deployments to check if the change was applied with the get deployments command:

kubectl get deployments

The number of replicas decreased to 2. List the number of Pods, with get pods:

kubectl get pods -o wide

This confirms that 2 Pods were terminated.

## Performing a rolling update

### Update the version of the app

To list your deployments, use the get deployments command: kubectl get deployments

To list the running Pods, use the get pods command:

kubectl get pods

To view the current image version of the app, run a describe command against the Pods (look at the Image field):

kubectl describe pods

To update the image of the application to version 2, use the set image command, followed by the deployment name and the new image version:

kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2

The command notified the Deployment to use a different image for your app and initiated a rolling update. Check the status of the new Pods, and view the old one terminating with the get pods command:

kubectl get pods

### Verify an update

First, let’s check that the App is running. To find out the exposed IP and Port we can use describe service:

kubectl describe services/kubernetes-bootcamp

Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

export NODE\_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')

echo NODE\_PORT=$NODE\_PORT

Next, we’ll do a curl to the exposed IP and port:

curl $(minikube ip):$NODE\_PORT

We hit a different Pod with every request and we see that all Pods are running the latest version (v2).

The update can be confirmed also by running a rollout status command:

kubectl rollout status deployments/kubernetes-bootcamp

To view the current image version of the app, run a describe command against the Pods:

kubectl describe pods

We run now version 2 of the app (look at the Image field)

### Rollback an update

Let’s perform another update, and deploy image tagged as v10 :

kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10

Use get deployments to see the status of the deployment:

kubectl get deployments

And something is wrong… We do not have the desired number of Pods available. List the Pods again:

kubectl get pods

A describe command on the Pods should give more insights:

kubectl describe pods

There is no image called v10 in the repository. Let’s roll back to our previously working version. We’ll use the rollout undo command:

kubectl rollout undo deployments/kubernetes-bootcamp

The rollout command reverted the deployment to the previous known state (v2). Updates are versioned and you can revert to any previously known state of a Deployment. List again the Pods:

kubectl get pods

Four Pods are running. Check again the image deployed on the them: kubectl describe pods

We see that the deployment is using a stable version of the app (v2). The Rollback was successful.

## Cleanup

### Delete the service

kubectl get services

kubectl delete services kubernetes-bootcamp

kubectl get services

### Delete the deployment

kubectl get deployments

kubectl delete deployment kubernetes-bootcamp

kubectl get deployments

### Delete the pods

kubectl get pods –o wide

kubectl delete pods --all (or one-by-one)

kubectl get pods –o wide

## Use a Service to Access an Application in a Cluster

<https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/>

### Creating a service for an application running in two pods:

hello-application.yaml:

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: hello-world**

**spec:**

**selector:**

**matchLabels:**

**run: load-balancer-example**

**replicas: 2**

**template:**

**metadata:**

**labels:**

**run: load-balancer-example**

**spec:**

**containers:**

**- name: hello-world**

**image: gcr.io/google-samples/node-hello:1.0**

**ports:**

**- containerPort: 8080**

**protocol: TCP**

1. Run a Hello World application in your cluster: Create the application Deployment using the file above:

**kubectl apply -f https://k8s.io/examples/service/access/hello-application.yaml**

The preceding command creates a Deployment object and an associated ReplicaSet object. The ReplicaSet has two Pods, each of which runs the Hello World application.

1. Display information about the Deployment:

**kubectl get deployments hello-world**

**kubectl describe deployments hello-world**

1. Display information about your ReplicaSet objects:

**kubectl get replicasets**

**kubectl describe replicasets**

1. Create a Service object that exposes the deployment:

**kubectl expose deployment hello-world --type=NodePort --name=example-service**

1. Display information about the Service:

**kubectl describe services example-service**

The output is similar to this:

**Name: example-service**

**Namespace: default**

**Labels: run=load-balancer-example**

**Annotations: <none>**

**Selector: run=load-balancer-example**

**Type: NodePort**

**IP: 10.32.0.16**

**Port: <unset> 8080/TCP**

**TargetPort: 8080/TCP**

**NodePort: <unset> 31496/TCP**

**Endpoints: 10.200.1.4:8080,10.200.2.5:8080**

**Session Affinity: None**

**Events: <none>**

Make a note of the NodePort value for the service. For example, in the preceding output, the NodePort value is 31496.

1. List the pods that are running the Hello World application:

**kubectl get pods --selector="run=load-balancer-example" --output=wide**

The output is similar to this:

**NAME READY STATUS ... IP NODE**

**hello-world-2895499144-bsbk5 1/1 Running ... 10.200.1.4 worker1**

**hello-world-2895499144-m1pwt 1/1 Running ... 10.200.2.5 worker2**

1. Get the public IP address of one of your nodes that is running a Hello World pod. How you get this address depends on how you set up your cluster. For example, if you are using Minikube, you can see the node address by running **kubectl cluster-info**. If you are using Google Compute Engine instances, you can use the **gcloud compute instances list** command to see the public addresses of your nodes.
2. On your chosen node, create a firewall rule that allows TCP traffic on your node port. For example, if your Service has a NodePort value of 31568, create a firewall rule that allows TCP traffic on port 31568. Different cloud providers offer different ways of configuring firewall rules.
3. Use the node address and node port to access the Hello World application:

**curl http://<public-node-ip>:<node-port>**

where **<public-node-ip>** is the public IP address of your node, and **<node-port>** is the NodePort value for your service. The response to a successful request is a hello message:

**Hello Kubernetes!**

# Kubernetes Volumes

<https://medium.com/@muppedaanvesh/a-hand-on-guide-to-kubernetes-volumes-%EF%B8%8F-b59d4d4e347f>

<https://spacelift.io/blog/kubernetes-persistent-volumes>

**Code files**: Kubernetes/Volumes

Kubernetes Persistent Volumes (PVs) provide storage for your application’s Pods. Data written to a volume is managed independently of the Pods that access it, ensuring the data remains available after Pod restarts and failures.

You’ll need to use PVs whenever you deploy stateful applications such as databases and file servers using Kubernetes. Without a PV, your data will be lost when Pods are terminated.

## What are Persistent Volumes?

Persistent Volumes are Kubernetes objects that represent storage resources in your cluster. PVs work in conjunction with Persistent Volume Claims (PVCs), another type of object which permits Pods to request access to PVs. To successfully utilize persistent storage in a cluster, you’ll need a PV and a PVC that connects it to your Pod.

The primary role of PVs is to abstract away the differences between cluster storage implementations. Each PV is assigned a [storage class](https://kubernetes.io/docs/concepts/storage/storage-classes) which defines the type of storage it provisions. Storage classes allow the automatic provisioning of volumes through many different storage providers, including cloud-hosted block storage options such as [AWS Elastic Block Store](https://aws.amazon.com/ebs/) and [GCE Persistent Disk](https://cloud.google.com/persistent-disk).

## What is the difference between Volume and Persistent Volume in Kubernetes?

Kubernetes also supports [Volumes](https://kubernetes.io/docs/concepts/storage/volumes) which can be used instead of, or in addition to, Persistent Volumes.

Volumes exist in the context of specific Pods. A Volume’s lifecycle is coupled to the Pod that owns it. It allows safe use of storage by all the containers in the Pod, but the volume’s data will still be lost when the Pod terminates.

Persistent Volumes build upon the foundations established by Volumes. They provide storage that’s decoupled from Pods, allowing data to persist beyond the lifecycle of individual Pods. Unlike Volumes, Persistent Volumes are first-class objects within your cluster.

Persistent Volumes also distinguish between storage provisioning and utilization. Cluster admins can set up PVs for developers to access using PVCs. This facilitates controlled approaches to storage management.

Plain Volumes can be used when data should be shared between a Pod’s containers but won’t need to be accessible to any other Pods – including replacements of the original. PVs are the solution for most real-world storage use cases where data needs to be persisted independently of Pods.

## Types of Persistent Volume

Kubernetes supports several [Persistent Volume types](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes) that alter where and how your data is stored:

* *local* – Data is stored on devices mounted locally to your cluster’s Nodes.
* *hostPath* – Stores data within a named directory on a Node (this is designed for testing purposes and doesn’t work with multi-Node clusters).
* *nfs* – Used to access Network File System (NFS) mounts.
* *iscsi* – iSCSI (SCSI over IP) storage attachments.
* *csi* – Allows integration with storage providers that support the [Container Storage Interface (CSI)](https://kubernetes.io/docs/concepts/storage/volumes/#csi) specification, such as the block storage services provided by cloud platforms.
* *cephfs* – Allow the use of CephFS volumes.
* *fc* – Fibre Channel (FC) storage attachments.
* *rbd* – Rados Block Device (RBD) volumes.

In addition, there are volume types such as [awsElasticBlockStore](https://kubernetes.io/docs/concepts/storage/volumes/#awselasticblockstore), [azureDisk](https://kubernetes.io/docs/concepts/storage/volumes/#azuredisk), and [gcePersistentDisk](https://kubernetes.io/docs/concepts/storage/volumes/#gcepersistentdisk) that support built-in integrations with specific cloud providers. However, these are all now deprecated in favor of CSI-based volumes.

[Storage classes](https://kubernetes.io/docs/concepts/storage/storage-classes) handle storage provisioning operations for volumes. There are built-in storage classes for each of the supported PV types mentioned above. Additional storage classes can be added by installing plugins that implement the [Kubernetes Volume Provisioning](https://github.com/kubernetes-sigs/sig-storage-lib-external-provisioner) specification.

## Persistent Volume access modes

Persistent Volumes support four different [access modes](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes) that define how they’re mounted to Nodes and Pods:

* **ReadWriteOnce (RWO)** – The volume is mounted with read-write access for a *single* Node in your cluster. Any of the Pods running on that Node can read and write the volume’s contents.
* **ReadOnlyMany (ROX)** – The volume can be concurrently mounted to any of the Nodes in your cluster, with read-only access for any Pod.
* **ReadWriteMany (RWX)** – Similar to ReadOnlyMany, but with read-write access.
* **ReadWriteOncePod (RWOP)** – This new variant, introduced as a beta feature in [Kubernetes v1.27](https://kubernetes.io/blog/2023/04/11/kubernetes-v1-27-release/), enforces that read-write access is provided to a *single* Pod. No other Pods in the cluster will be able to use the volume simultaneously.

The access modes that can be used with a specific PV [depend on the type of storage](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes) that backs it. hostPath volumes support only RWO, for example, while nfs offers RWO, ROX, and RWX. For CSI integrations, the options are defined by the specific storage driver that’s in use, such as AWS or GCE.

## The lifecycles of PVs and PVCs

PVs and PVCs have their own lifecycles, which describe the current status of the volume and whether it’s in use. There are four main stages:

**1. Provisioning**

At the Provisioning stage, the PV is created and its storage is allocated using the selected driver.

Provisioning can occur manually by creating a PersistentVolume object in your cluster, or dynamically, by adding a PVC that refers to an unknown PV. After provisioning, the PV will exist in your cluster, but won’t be actively providing storage.

**2. Binding**

Binding occurs when a cluster user adds a PVC that claims the PV. The PV will enter this state automatically when dynamic provisioning is used, because you’ll have already created the PVC.

Kubernetes automatically watches for new PVCs and binds them to the PVs they reference. Each PV can only be bound to a single PVC at a time. Once a PVC claims a PV, the volume will be Bound, but won’t necessarily be used by a Pod.

**3. Using**

A volume enters use once its PVC is consumed by a Pod. When Pods reference PVCs, Kubernetes automatically mounts the correct volume into the Pod’s filesystem. In this state, the PV is actively providing storage to an application in your cluster.

**4. Reclaiming**

Storage isn’t always required indefinitely. Users can delete the PVC to relinquish access to the PV. When this happens, the storage used by the PV is “reclaimed.”

The reclaim behavior [is customizable](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#reclaiming) and allows you to either delete the provisioned storage, recycle it by emptying its contents, or retain it as-is for future reuse.

## Example: How to create and use a Persistent Volume

Ready to explore Persistent Volumes? Let’s get started. Make sure you’ve got access to a [Kubernetes cluster](https://spacelift.io/blog/kubernetes-cluster) configured in Kubectl before you begin.

**Step 1: Check your cluster’s storage classes**

First, find out which storage classes are available in your cluster:

$ kubectl get storageclass

We’re using a local [Minikube](https://minikube.sigs.k8s.io/docs/start) cluster which includes a single built-in storage class called standard. It provisions volumes on the host machine.

We’ll use the standard storage class in the following examples, but you might need to use an alternative depending on the options available in your cluster. For example, Azure clusters will have azurefile-csi available, while DigitalOcean supports do-block-storage.

**Step 2: Create a Persistent Volume**

Next, copy the following PV [Kubernetes manifest](https://spacelift.io/blog/kubernetes-manifest-file) and save it to pv.yaml:

apiVersion: v1

kind: PersistentVolume

metadata:

name: demo-pv

spec:

accessModes:

- ReadWriteOnce

capacity:

storage: 1Gi

storageClassName: standard

hostPath:

path: /tmp/demo-pv

This defines a 1Gi Persistent Volume that uses the standard storage class. The access mode is set to ReadWriteOnce, which is the only read-write mode supported by the storage class in this tutorial.

The hostPath field configures how the volume will be configured by the storage driver, which in this case uses the hostPath provisioner. It instructs the driver to store the volume’s data at /tmp/demo-pv on the host. You will need to supply different configuration options if you’re using a storage driver with an alternative provisioner.

Run the following command to create your Persistent Volume:

$ kubectl apply -f pv.yaml

See the difference between [kubectl apply vs. kubectl create](https://spacelift.io/blog/kubectl-apply-vs-create).

The PV will now exist in your cluster. It will show as Available as it’s unused and unclaimed:

$ kubectl get pv

The PV will now show as Bound because it’s claimed by the PVC:

$ kubectl get pv

**Step 4: Dynamically provisioning PVs**

Static provisioning of PVs, as shown above, is cumbersome: you must create the PV, then the PVC while ensuring their properties match. You also have to supply the PV configuration data required by the storage provisioner.

Dynamic provisioning is simpler and the more popular option for real-world use. The PV is created for you based on your PVC’s configuration.

Copy the following manifest to pvc-dynamic.yaml:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc-dynamic

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

storageClassName: standard

This manifest creates and claims a new 1Gi PV that will be backed by the standard storage class. Because the volumeName is omitted from the PVC’s spec, the PV is dynamically provisioned:

$ kubectl apply -f pvc-dynamic.yaml

$ kubectl get pv

Dynamically created PVs default to using the Delete retention policy. This means the PV is automatically deleted when the PVC is destroyed.

**Step 5: Attach PVCs to Pods**

Once you’ve provisioned and bound your PV, it’s time to attach the PVC to a Pod. This will mount your PV into the Pod’s filesystem, allowing the Pod to read and write files with full persistence.

Copy the following manifest to pod.yaml:

apiVersion: v1

kind: Pod

metadata:

name: pvc-pod

spec:

containers:

- name: pvc-pod-container

image: nginx:latest

volumeMounts:

- mountPath: /data

name: data

volumes:

- name: data

persistentVolumeClaim:

claimName: pvc-dynamic

What’s happening here?

* First, the **spec.containers.volumeMounts** field sets up a volume for the Pod called data. It’s mounted to /data within the Pod’s containers.
* Next, the **spec.volumes** field defines the data volume as referring to the PVC called pvc-dynamic.
* This results in the PV claimed by pvc-dynamic being mounted to /data inside the container.

Create your Pod, then try writing some files to /data:

$ kubectl apply -f pod.yaml

pod/pvc-pod created

# Get a shell inside the Pod

$ kubectl exec -it pod/pvc-pod -- bash

# No files in the volume yet

root@pvc-pod:/# ls /data

# Write a file to the volume

root@pvc-pod:/# echo "bar" > /data/foo

# The file now shows in the volume

root@pvc-pod:/# ls /data

foo

The files written to the volume are now stored independently of the Pod. You can delete the Pod and recreate it – your data will still be intact within the PV:

$ kubectl delete pod/pvc-pod

pod "pvc-pod" deleted

$ kubectl apply -f pod2.yaml

pod/pvc-pod2 created

$ kubectl exec -it pod/pvc-pod2 -- bash

root@pvc-pod2:/# cat /data/foo

bar

Similarly, you can simultaneously attach the PVC to additional Pods to share files between them.

# Kubernetes Secrets

<https://medium.com/@rajeshmamuddu/kubernetes-secrets-0e4a6413ac69>

<https://spacelift.io/blog/kubernetes-secrets>

**Code files**: Kubernetes/Secrets

## What are secrets in Kubernetes?

Kubernetes Secrets are objects used to store and manage sensitive information such as passwords, OAuth tokens, SSH keys, and API keys. The primary purpose of Secrets is to reduce the risk of exposing sensitive data while deploying applications on Kubernetes.

Instead of embedding sensitive data directly within Pods or configuration files, which could expose it to a wider audience than intended, Secrets allow Kubernetes to store and use sensitive data in a more secure and manageable way.

## Encoding and decoding data for Kubernetes Secrets

Let’s add the following secret to our Kubernetes cluster: “my-awesome-password”. We’ve defined a K8s manifest for creating this secret in a file secret.yaml:

apiVersion: v1

kind: Secret

metadata:

name: my-secret

type: Opaque

data:

password: my-awesome-password

$ kubectl apply -f secret.yaml

Error from server (BadRequest): error when creating "secret.yaml": Secret in version "v1" cannot be handled as a Secret: illegal base64 data at input byte 2

As you can see, we cannot create the secret in our cluster directly, as we need to first base64 encode it. Let’s do that before adding the password there. I will show you how to do it for Linux and MacOS:

echo -n "my-awesome-password" | base64

bXktYXdlc29tZS1wYXNzd29yZA==

Now, we will replace the text in our Secret with its base64 option:

apiVersion: v1

kind: Secret

metadata:

name: my-secret

type: Opaque

data:

password: bXktYXdlc29tZS1wYXNzd29yZA==

kubectl apply -f secret.yaml

secret/my-secret created

We can get the Secret from Kubernetes by running the following command:

kubectl get secret my-secret -o jsonpath='{.data.password}'

bXktYXdlc29tZS1wYXNzd29yZA==

To decode the Secret, we can easily use base64 again:

kubectl get secret my-secret -o jsonpath='{.data.password}' | base64 --decode

my-awesome-password

If you don’t want to encode the data manually as shown before, you have an alternative, by creating the secret imperatively:

kubectl create secret generic my-other-secret --from-literal=password=secret

secret/my-other-secret created

To get the base64 encoded value, we can run:

kubectl get secret my-other-secret -o jsonpath='{.data.password}'

c2VjcmV0%

Also, to get the decoded value, we can run:

kubectl get secret my-other-secret -o jsonpath='{.data.password}' | base64 --decode

## 1. Create Kubernetes Secrets using kubectl

First, create a namespace to store the demo resources for easy cleanup:

$ kubectl create namespace secrets-demo

There are two ways of providing the Secret data to kubectl when creating Secrets using Kubectl, and there are:

* Providing the secret data through a file using the --from-file=<filename> tag or
* Providing the literal secret data using the --from-literal=<key>=<value> tag

To start creating a Secret with kubectl providing the Secret data from a file in any directory of your choice. Create files to store the hypothetical user credentials with the following command:

$ echo -n 'admin' > username.txt

$ echo -n 'password' > password.txt

The -n flag in the above command ensures that no newline character is added at the end of the text. This is crucial since kubectl will encode the extra newline character if present when it reads the file and turns the content into a base64 string.

After running the above commands, you can verify that the password and username were written to the file with the cat command, as in the image below.



Now, create the Kubernetes Secret with the files using the kubectl command below:

kubectl create secret generic database-credentials --from-file=username.txt --from-file=password.txt --namespace=secrets-demo

**Note**: When using the above command, the key of your secret data will be the filename (username.txt and password.txt) by default.  To provide keys for the Secret data, use the following syntax --from-file=[key=]source, for example:

kubectl create secret generic database-credentials \

--from-file=username=username.txt \

--from-file=password=password.txt \

--namespace=secrets-demo

To verify the Secret creation, run the following command:

$ kubectl -n secrets-demo get secrets

## 2. Create Kubernetes Secrets from a YAML manifest file

Before you create a Secret using a manifest file, you must first decide how you want to add the Secret data using the data field and/or the stringData field.

Using the data field, you must encode the secret data using base64.  To convert the username and password to base64, run the following command:

echo -n 'admin' | base64

echo -n 'password' | base64

After running the above command, you will get an output similar to the image below. Copy the base64 values and store them to put in your manifest file.



Now create a demo-secret.yaml manifest file using your preferred method (text editor, vim or nano, etc.) and add the following configuration.

apiVersion: v1

kind: Secret

metadata:

name: demo-secret

type: Opaque

data:

username: YWRtaW4=

password: cGFzc3dvcmQ=

In the above manifest file, the username and password values in the data field are the base64 encoded values of the original credentials.

When using the stringData field, the manifest file will be:

apiVersion: v1

kind: Secret

metadata:

name: demo-secret

type: Opaque

stringData:

username: admin

password: password

To create the Secret, run the following command:

$ kubectl -n secrets-demo apply -f demo-secret.yaml

## List existing Kubernetes Secrets

To list Secrets we can simply run the following command:

kubectl get secrets

To get more details about the Secrets, we can use the *describe* option for kubectl:

kubectl describe secrets

Secrets are namespaced resources, so you can use the *-n*option to get the Secrets from a specific namespace, or you can use *–all-namespaces* to get the secrets from all the namespaces. By default, the *default* namespace will be used.

kubectl get secrets -n secrets-demo

kubectl get secrets --all-namespaces

## View a Kubernetes Secret value with kubectl describe

Using the kubectl describe, you can view some basic information about Kubernetes objects. To use it to view the description of one of the Secrets you’ve created in the article, run:

$ kubectl -n secrets-demo describe secrets/database-credentials

As you can see, the above output doesn’t show the Secret’s contents. This is to protect the Secret from being exposed or logged in the terminal.

To view the Secret data, you will need to decode the secret.

## Decode a Kubernetes Secret

To view the data of the Secret you created, run the following command:

$ kubectl -n secrets-demo get secret database-credentials -o jsonpath='{.data}'

To decode the encoded strings, you can use the following command:

$ echo 'YWRtaW4=' | base64 --decode

$ echo 'cGFzc3dvcmQ=' | base64 --decode

**Note**: If you do the above, you could store the Secret data in your shell history. To avoid that, combine the previous two steps into one command like the one below.

$ kubectl get -n secrets-demo secret database-credentials -o jsonpath='{.data.password}' | base64 --decode

## Delete a Kubernetes Secret

Clean up the entire setup by deleting the namespace, which deletes all the secrets and Pods you created with the following command:

kubectl delete secret <secret-name> -n <namespace>

## Using Secret data as container environment variables

For demo purposes, below is a Pod manifest with the Kubernetes Secret data you created exposed as environment variables. Create a secret-test-env-pod.yaml and paste the configuration in it.

apiVersion: v1

kind: Pod

metadata:

name: env-pod

namespace: secrets-demo

spec:

containers:

- name: secret-test

image: nginx:latest

#command: ['sh', '-c', 'echo "Username: $USER" "Password: $PASSWORD"']

env:

- name: USER

valueFrom:

secretKeyRef:

name: database-credentials

key: username.txt

- name: PASSWORD

valueFrom:

secretKeyRef:

name: database-credentials

key: password.txt

Create the Pod using the following kubectl command:

$ kubectl -n secrets-demo apply -f secret-test-evn-pod.yaml

To verify that Kubernetes mounted the Secret on the Pod, describe the Pod with the following command:

# "env-pod" is the Pod name as in the above manifest file

$ kubectl -n secrets-demo describe pod env-pod

Also, seeing the echo command in the Pod manifest file, you can verify by checking the logs of the Pod with:

$ kubectl -n secrets-demo logs env-pod

To verify if the pod can indeed read the environment variables, bash into the pod and check:

$ kubectl exec -it -n secrets-demo env-pod – bash

# echo $USER

# echo $PASSWORD

# exit

## Using Secret data as files in a volume mounted on a Pod’s container(s)

For demo purposes, below is a Pod manifest with the Kubernetes Secret data you created as files in a volume mounted on the Pod’s containers.

Create a secret-test-volume-pod.yaml and paste the configuration in it.

apiVersion: v1

kind: Pod

metadata:

name: volume-test-pod

spec:

containers:

- name: secret-test

image: nginx

volumeMounts:

- name: secret-volume

mountPath: /etc/config/secret

volumes:

- name: secret-volume

secret:

secretName: database-credentials

Create the Pod using the following kubectl command:

$ kubectl -n secrets-demo apply -f secret-test-volume-pod.yaml

To verify that the Pod can access the Secret data, connect to the container and run the following commands in the volume directory:

$ kubectl -n secrets-demo exec volume-test-pod -- cat /etc/config/secret/username.txt

$ kubectl -n secrets-demo exec volume-test-pod -- cat /etc/config/secret/password.txt

$ kubectl -n secrets-demo exec volume-test-pod -- ls /etc/config/secret

Clean up

$ kubectl -n secrets-demo delete pod <pod name>

$ kubectl -n secrets-demo delete secret<pod name>

$ kubectl delete namespace <namespace>

## More

Kubernetes **secret** objects let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a **secret** is safer and more flexible than putting it verbatim in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) definition or in a [container image](https://kubernetes.io/docs/reference/glossary/?all=true#term-image).

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

Users can create secrets, and the system also creates some secrets.

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in two ways: as files in a [volume](https://kubernetes.io/docs/concepts/storage/volumes/) mounted on one or more of its containers, or used by kubelet when pulling images for the pod.

## Built-in Secrets

Service Accounts Automatically Create and Attach Secrets with API Credentials

Kubernetes automatically creates secrets which contain credentials for accessing the API and it automatically modifies your pods to use this type of secret.

The automatic creation and use of API credentials can be disabled or overridden if desired. However, if all you need to do is securely access the apiserver, this is the recommended workflow.

See the [Service Account](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/) documentation for more information on how Service Accounts work.

## Creating your own Secrets

### Creating a Secret Using kubectl create secret

Say that some pods need to access a database. The username and password that the pods should use is in the files **./username.txt** and **./password.txt** on your local machine.

***# Create files needed for rest of example.***

**echo -n 'admin' > ./username.txt**

**echo -n '1f2d1e2e67df' > ./password.txt**

The **kubectl create secret** command packages these files into a Secret and creates the object on the Apiserver.

**kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt**

**secret "db-user-pass" created**

**Note:** Special characters such as **$**, **\\***, and **!** require escaping. If the password you are using has special characters, you need to escape them using the **\\** character. For example, if your actual password is **S!B\\*d$zDsb**, you should execute the command this way: kubectl create secret generic dev-db-secret --from-literal=username=devuser --from-literal=password=S\!B\\\*d\$zDsb. You do not need to escape special characters in passwords from files (**--from-file**).

You can check that the secret was created like this:

**kubectl get secrets**

**NAME TYPE DATA AGE**

**db-user-pass Opaque 2 51s**

**kubectl describe secrets/db-user-pass**

**Name: db-user-pass**

**Namespace: default**

**Labels: <none>**

**Annotations: <none>**

**Type: Opaque**

**Data**

**====**

**password.txt: 12 bytes**

**username.txt: 5 bytes**

**Note:** **kubectl get** and **kubectl describe** avoid showing the contents of a secret by default. This is to protect the secret from being exposed accidentally to an onlooker, or from being stored in a terminal log.

## Creating a Secret Manually

You can also create a Secret in a file first, in json or yaml format, and then create that object. The Secret contains two maps: data and stringData. The data field is used to store arbitrary data, encoded using base64. The stringData field is provided for convenience, and allows you to provide secret data as unencoded strings.

For example, to store two strings in a Secret using the data field, convert them to base64 as follows:

**echo -n 'admin' | base64**

***YWRtaW4=***

**echo -n '1f2d1e2e67df' | base64**

***MWYyZDFlMmU2N2Rm***

To decode:

**echo "MWYyZDFlMmU2N2Rm" | base64 --decode**

Write a Secret that looks like this:

**apiVersion: v1**

**kind: Secret**

**metadata:**

**name: mysecret**

**type: Opaque**

**data:**

**username: YWRtaW4=**

**password: MWYyZDFlMmU2N2Rm**

Now create the Secret using **kubectl apply**:

**kubectl apply -f ./secret.yaml**

***secret "mysecret" created***

Read secret data:

kubectl get secret mysecret -o jsonpath="{.data.username}" | base64 --decode

For certain scenarios, you may wish to use the stringData field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

If your application uses the following configuration file:

**apiUrl: "https://my.api.com/api/v1"**

**username: "user"**

**password: "password"**

You could store this in a Secret using the following:

**apiVersion: v1**

**kind: Secret**

**metadata:**

**name: mysecret**

**type: Opaque**

**stringData:**

**config.yaml: |-**

**apiUrl: "https://my.api.com/api/v1"**

**username: {{username}}**

**password: {{password}}**

Your deployment tool could then replace the **{{username}}** and **{{password}}** template variables before running **kubectl apply**.

stringData is a write-only convenience field. It is never output when retrieving Secrets. For example, if you run the following command:

**kubectl get secret mysecret -o yaml**

The output will be similar to:

**apiVersion: v1**

**kind: Secret**

**metadata:**

**creationTimestamp: 2018-11-15T20:40:59Z**

**name: mysecret**

**namespace: default**

**resourceVersion: "7225"**

**selfLink: /api/v1/namespaces/default/secrets/mysecret**

**uid: c280ad2e-e916-11e8-98f2-025000000001**

**type: Opaque**

**data:**

**config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5hbWU6IHt7dXNlcm5hbWV9fQpwYXNzd29yZDoge3twYXNzd29yZH19**

If a field is specified in both data and stringData, the value from stringData is used. For example, the following Secret definition:

**apiVersion: v1**

**kind: Secret**

**metadata:**

**name: mysecret**

**type: Opaque**

**data:**

**username: YWRtaW4=**

**stringData:**

**username: administrator**

Results in the following secret:

**apiVersion: v1**

**kind: Secret**

**metadata:**

**creationTimestamp: 2018-11-15T20:46:46Z**

**name: mysecret**

**namespace: default**

**resourceVersion: "7579"**

**selfLink: /api/v1/namespaces/default/secrets/mysecret**

**uid: 91460ecb-e917-11e8-98f2-025000000001**

**type: Opaque**

**data:**

**username: YWRtaW5pc3RyYXRvcg==**

Where **YWRtaW5pc3RyYXRvcg==** decodes to **administrator**.

The keys of data and stringData must consist of alphanumeric characters, ‘-’, ‘\_’ or ‘.’.

**Encoding Note:** The serialized JSON and YAML values of secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the **base64** utility on Darwin/macOS users should avoid using the **-b** option to split long lines. Conversely Linux users should add the option **-w 0** to **base64** commands or the pipeline **base64 | tr -d '\n'**if **-w** option is not available.

## Using Secrets

* <https://v1-13.docs.kubernetes.io/docs/concepts/configuration/secret/>
* <https://www.tutorialspoint.com/kubernetes/kubernetes_secrets.htm>

### Using Secrets with Volumes

UsingSecretVolume.yaml:

apiVersion: v1

kind: Pod

metadata:

  name: mypod

spec:

  containers:

  - name: mypod

    image: gcr.io/google-samples/node-hello:1.0

    volumeMounts:

    - name: foo

      mountPath: "/etc/foo"

      readOnly: true

  volumes:

  - name: foo

    secret:

      secretName: mysecret

**kubectl apply -f ./UsingSecretVolume.yaml**

### Consuming Secret Values from Volumes

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base-64 decoded and stored inside these files.

To connect to the container (pod):

**kubectl exec -it mypod sh**

This is the result of commands executed inside the container from the example above:

**ls /etc/foo/**

***username***

***password***

**cat /etc/foo/username**

***admin***

**cat /etc/foo/password**

***1f2d1e2e67df***

### Using Secrets as Environment Variables

UsingSecretEnv.yaml:

apiVersion: v1

kind: Pod

metadata:

  name: secret-env-pod

spec:

  containers:

  - name: mycontainer

    image: gcr.io/google-samples/node-hello:1.0

    env:

      - name: SECRET\_USERNAME

        valueFrom:

          secretKeyRef:

            name: mysecret

            key: username

      - name: SECRET\_PASSWORD

        valueFrom:

          secretKeyRef:

            name: mysecret

            key: password

  restartPolicy: Never

**kubectl apply -f ./UsingSecretEnv.yaml**

### Consuming Secret Values from Volumes

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base-64 decoded and stored inside these files.

To connect to the container (pod):

**kubectl exec -it secret-env-pod sh**

This is the result of commands executed inside the container from the example above:

**echo $SECRET\_USERNAME**

***admin***

**echo $SECRET\_PASSWORD**

***1f2d1e2e67df***

## Cleanup

**kubectl delete delpoyments/secret-env-pod**

**kubectl delete delpoyments/mypod**

# Kubernetes ConfigMaps

A ConfigMap in Kubernetes is an API object designed to store non-confidential configuration data as key-value pairs. This allows you to decouple configuration artifacts from container images, making it easier to manage and update application settings without rebuilding or redeploying your containers.

ConfigMaps are a fundamental part of Kubernetes application configuration, enabling flexible, environment-specific, and decoupled management of non-sensitive settings

## Key Features

* **Stores non-sensitive data:** ConfigMaps are intended for configuration data that is not confidential. For sensitive data, use Kubernetes Secrets.
* **Key-value pairs:** Data is stored as strings in key-value format. Each key must be a valid environment variable name if you plan to inject it as such.
* **Size limit:** The total size of a ConfigMap object is limited to less than 1 MiB.
* **Decoupling:** Keeps configuration separate from application code and images, making updates and environment changes easier.

## Creating a ConfigMap

You can create a ConfigMap in several ways:

* **From a YAML manifest:**

apiVersion: v1

kind: ConfigMap

metadata:

name: my-configmap

data:

database\_host: "db.example.com"

log\_level: "info"

Save this as my-configmap.yaml and apply with:

kubectl apply -f my-configmap.yaml

* **From the command line:**

kubectl create configmap my-configmap --from-literal=key1=value1 --from-literal=key2=value2

* **From files or directories:**

kubectl create configmap my-configmap --from-file=path/to/file

kubectl create configmap my-configmap --from-file=path/to/dir/

## Using a ConfigMap in Pods

ConfigMaps can be consumed by Pods in several ways:

* **As environment variables:** All or selected keys can be injected as environment variables.

envFrom:

- configMapRef:

name: my-configmap

Or, for a specific key:

env:

- name: CONFIGMAP\_USERNAME

valueFrom:

configMapKeyRef:

name: my-configmap

key: username

* **As command-line arguments:** Reference ConfigMap values in the command section of your Pod spec.
* **As files in a volume:** Mount the ConfigMap as a volume so each key appears as a file in the container.

## Updating and Managing ConfigMaps

* ConfigMaps can be updated independently of Pods. However, running Pods will not automatically pick up changes unless they are restarted or reloaded in a way that supports dynamic config reloading.
* Use kubectl get configmap and kubectl describe configmap <name> to view ConfigMap details.

## Typical Use Cases

* Application configuration (e.g., database URLs, feature flags)
* Environment-specific settings
* Command-line arguments or script parameters for containers

## Summary Table

| **Feature** | **Description** |
| --- | --- |
| Data format | Key-value pairs (strings) |
| Use cases | Non-sensitive config (URLs, flags, etc.) |
| Max size | < 1 MiB per ConfigMap |
| Consumption methods | Environment variables, volumes, command-line args |
| Creation methods | YAML manifest, kubectl CLI, files, directories |

# Kubernetes on Azure (AKS)

<https://learn.microsoft.com/en-us/azure/aks/intro-kubernetes>

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-deploy-cluster?tabs=azure-cli>

<https://medium.com/@kesavan.keshav/deploying-your-first-application-on-aks-hello-world-example-351e0bf7b2c6>

<https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-portal?tabs=azure-cli>

Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance. When you create an AKS cluster, a control plane is automatically created and configured. This control plane is provided at no cost as a managed Azure resource abstracted from the user. You only pay for and manage the nodes attached to the AKS cluster.

You can create an AKS cluster using:

* [Azure CLI](https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-cli)
* [Azure PowerShell](https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-powershell)
* [Azure portal](https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-portal)
* Template-driven deployment options, like [Azure Resource Manager templates](https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-rm-template), [Bicep](https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview), and Terraform.

When you deploy an AKS cluster, you specify the number and size of the nodes, and AKS deploys and configures the Kubernetes control plane and nodes. [Advanced networking](https://learn.microsoft.com/en-us/azure/aks/concepts-network), [Azure Active Directory (Azure AD) integration](https://learn.microsoft.com/en-us/azure/aks/managed-aad), [monitoring](https://learn.microsoft.com/en-us/azure/aks/monitor-aks), and other features can be configured during the deployment process.

For more information on Kubernetes basics, see [Kubernetes core concepts for AKS](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads).

## Access, security, and monitoring

For improved security and management, you can integrate with [Azure AD](https://learn.microsoft.com/en-us/azure/aks/managed-aad) to:

* Use Kubernetes role-based access control (Kubernetes RBAC).

Monitor the health of your cluster and resources.

### Identity and security management

#### Kubernetes RBAC

To limit access to cluster resources, AKS supports [Kubernetes RBAC](https://learn.microsoft.com/en-us/azure/aks/concepts-identity#kubernetes-rbac). Kubernetes RBAC controls access and permissions to Kubernetes resources and namespaces.

#### Azure AD

You can configure an AKS cluster to integrate with Azure AD. With Azure AD integration, you can set up Kubernetes access based on existing identity and group membership. Your existing Azure AD users and groups can be provided with an integrated sign-on experience and access to AKS resources.

For more information on identity, see [Access and identity options for AKS](https://learn.microsoft.com/en-us/azure/aks/concepts-identity).

To secure your AKS clusters, see [Integrate Azure AD with AKS](https://learn.microsoft.com/en-us/azure/aks/azure-ad-integration-cli).

### Integrated logging and monitoring

[Azure Monitor for Container Health](https://learn.microsoft.com/en-us/previous-versions/azure/azure-monitor/containers/containers) collects memory and processor performance metrics from containers, nodes, and controllers within your AKS clusters and deployed applications. You can review both container logs and [the Kubernetes logs](https://learn.microsoft.com/en-us/azure/aks/monitor-aks-reference#resource-logs), which are:

* Stored in an [Azure Log Analytics](https://learn.microsoft.com/en-us/azure/azure-monitor/logs/log-analytics-overview) workspace.
* Available through the Azure portal, Azure CLI, or a REST endpoint.

For more information, see [Monitor AKS container health](https://learn.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-overview).

## Clusters and nodes

AKS nodes run on Azure virtual machines (VMs). With AKS nodes, you can connect storage to nodes and pods, upgrade cluster components, and use GPUs. AKS supports Kubernetes clusters that run multiple node pools to support mixed operating systems and Windows Server containers.

For more information about Kubernetes cluster, node, and node pool capabilities, see [Kubernetes core concepts for AKS](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads).

### Cluster node and pod scaling

As demand for resources change, the number of cluster nodes or pods that run your services automatically scales up or down. You can adjust both the horizontal pod autoscaler or the cluster autoscaler to adjust to demands and only run necessary resources.

For more information, see [Scale an AKS cluster](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-scale).

### Cluster node upgrades

AKS offers multiple Kubernetes versions. As new versions become available in AKS, you can upgrade your cluster using the Azure portal, Azure CLI, or Azure PowerShell. During the upgrade process, nodes are carefully cordoned and drained to minimize disruption to running applications.

To learn more about lifecycle versions, see [Supported Kubernetes versions in AKS](https://learn.microsoft.com/en-us/azure/aks/supported-kubernetes-versions). For steps on how to upgrade, see [Upgrade an AKS cluster](https://learn.microsoft.com/en-us/azure/aks/upgrade-cluster).

### Storage volume support

To support application workloads, you can mount static or dynamic storage volumes for persistent data. Depending on the number of connected pods expected to share the storage volumes, you can use storage backed by:

* [Azure Disks](https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi) for single pod access
* [Azure Files](https://learn.microsoft.com/en-us/azure/aks/azure-files-csi) for multiple, concurrent pod access.

For more information, see [Storage options for applications in AKS](https://learn.microsoft.com/en-us/azure/aks/concepts-storage).

## Virtual networks and ingress

An AKS cluster can be deployed into an existing virtual network. In this configuration, every pod in the cluster is assigned an IP address in the virtual network and can directly communicate with other pods in the cluster and other nodes in the virtual network.

Pods can also connect to other services in a peered virtual network and on-premises networks over ExpressRoute or site-to-site (S2S) VPN connections.

For more information, see the [Network concepts for applications in AKS](https://learn.microsoft.com/en-us/azure/aks/concepts-network).

### Ingress with HTTP application routing

The HTTP application routing add-on helps you easily access applications deployed to your AKS cluster. When enabled, the HTTP application routing solution configures an ingress controller in your AKS cluster.

As applications are deployed, publicly accessible DNS names are auto-configured. The HTTP application routing sets up a DNS zone and integrates it with the AKS cluster. You can then deploy Kubernetes ingress resources as normal.

To get started with Ingress traffic, see [HTTP application routing](https://learn.microsoft.com/en-us/azure/aks/http-application-routing).

## Development tooling integration

Kubernetes has a rich ecosystem of development and management tools that work seamlessly with AKS. These tools include [Helm](https://learn.microsoft.com/en-us/azure/aks/quickstart-helm) and the [Kubernetes extension for Visual Studio Code](https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools).

Azure provides several tools that help streamline Kubernetes.

## Docker image support and private container registry

AKS supports the Docker image format. For private storage of your Docker images, you can integrate AKS with Azure Container Registry (ACR).

To create a private image store, see [Azure Container Registry](https://learn.microsoft.com/en-us/azure/container-registry/container-registry-intro).

## Quotas, virtual machine size restrictions, and region availability in Azure Kubernetes Service (AKS)

**This is for participant’s reference**

<https://learn.microsoft.com/en-us/azure/aks/quotas-skus-regions>

# Autoscaling on AKS

<https://www.techtarget.com/searchcloudcomputing/tutorial/How-to-set-up-autoscaling-in-Azure-Kubernetes-Service>

<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>

<https://learnk8s.io/autoscaling-apps-kubernetes>

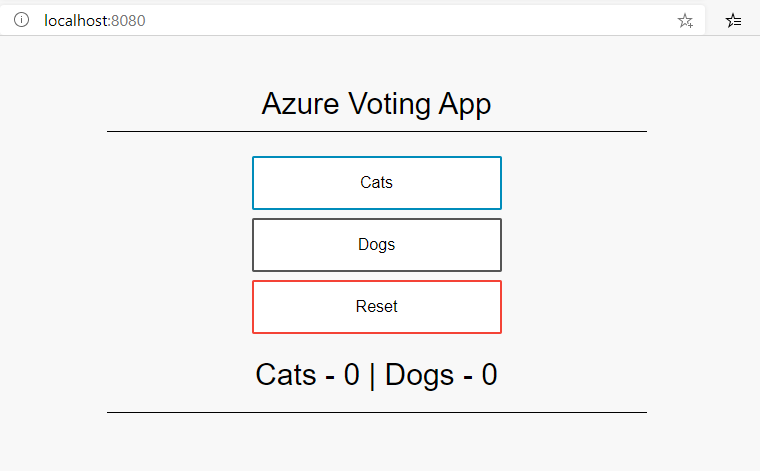
# Tutorial 01: Prepare an application for Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app>

In this tutorial, part one of seven, you prepare a multi-container application to use in Kubernetes. You use existing development tools like Docker Compose to locally build and test the application. You learn how to:

* Clone a sample application source from GitHub
* Create a container image from the sample application source
* Test the multi-container application in a local Docker environment

Once completed, the following application runs in your local development environment:

[](https://learn.microsoft.com/en-us/azure/aks/media/container-service-kubernetes-tutorials/azure-vote-local.png#lightbox)

In later tutorials, you upload the container image to an Azure Container Registry (ACR), and then deploy it into an AKS cluster.

## Before you begin

This tutorial assumes a basic understanding of core Docker concepts such as containers, container images, and docker commands. For a primer on container basics, see [Get started with Docker](https://docs.docker.com/get-started/).

To complete this tutorial, you need a local Docker development environment running Linux containers. Docker provides packages that configure Docker on a [Mac](https://docs.docker.com/docker-for-mac/), [Windows](https://docs.docker.com/docker-for-windows/), or [Linux](https://docs.docker.com/engine/installation/#supported-platforms) system.

**Note**: Azure Cloud Shell does not include the Docker components required to complete every step in these tutorials. Therefore, we recommend using a full Docker development environment.

## Get application code

The [sample application](https://github.com/Azure-Samples/azure-voting-app-redis) used in this tutorial is a basic voting app consisting of a front-end web component and a back-end Redis instance. The web component is packaged into a custom container image. The Redis instance uses an unmodified image from Docker Hub.

Use [git](https://git-scm.com/downloads) to clone the sample application to your development environment.

git clone https://github.com/Azure-Samples/azure-voting-app-redis.git

Change into the cloned directory.

cd azure-voting-app-redis

The directory contains the application source code, a pre-created Docker compose file, and a Kubernetes manifest file. These files are used throughout the tutorial set. The contents and structure of the directory are as follows:

azure-voting-app-redis

│ azure-vote-all-in-one-redis.yaml

│ docker-compose.yaml

│ LICENSE

│ README.md

│

├───azure-vote

│ │ app\_init.supervisord.conf

│ │ Dockerfile

│ │ Dockerfile-for-app-service

│ │ sshd\_config

│ │

│ └───azure-vote

│ │ config\_file.cfg

│ │ main.py

│ │

│ ├───static

│ │ default.css

│ │

│ └───templates

│ index.html

│

└───jenkins-tutorial

config-jenkins.sh

deploy-jenkins-vm.sh

## Create container images

[Docker Compose](https://docs.docker.com/compose/) can be used to automate building container images and the deployment of multi-container applications.

The following command uses the sample docker-compose.yaml file to create the container image, download the Redis image, and start the application.

docker-compose up -d

When completed, use the [docker images](https://docs.docker.com/engine/reference/commandline/images/) command to see the created images. Three images are downloaded or created. The azure-vote-front image contains the front-end application and uses the nginx-flask image as a base. The redis image is used to start a Redis instance.

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

mcr.microsoft.com/azuredocs/azure-vote-front v1 84b41c268ad9 9 seconds ago 944MB

mcr.microsoft.com/oss/bitnami/redis 6.0.8 3a54a920bb6c 2 days ago 103MB

tiangolo/uwsgi-nginx-flask python3.6 a16ce562e863 6 weeks ago 944MB

Run the [docker ps](https://docs.docker.com/engine/reference/commandline/ps/) command to see the running containers.

$ docker ps

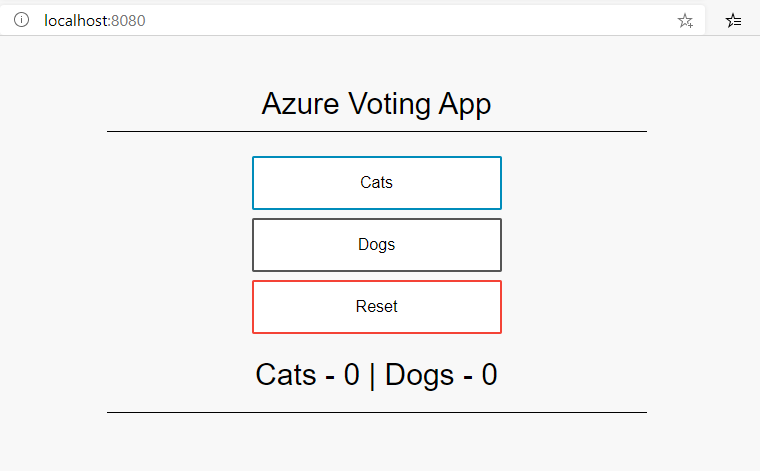
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d10e5244f237 mcr.microsoft.com/azuredocs/azure-vote-front:v1 "/entrypoint.sh /sta…" 3 minutes ago Up 3 minutes 443/tcp, 0.0.0.0:8080->80/tcp azure-vote-front

21574cb38c1f mcr.microsoft.com/oss/bitnami/redis:6.0.8 "/opt/bitnami/script…" 3 minutes ago Up 3 minutes 0.0.0.0:6379->6379/tcp azure-vote-back

## Test application locally

To see your running application, navigate to http://localhost:8080 in a local web browser. The sample application loads, as shown in the following example:

[](https://learn.microsoft.com/en-us/azure/aks/media/container-service-kubernetes-tutorials/azure-vote-local.png#lightbox)

## Clean up resources

Now that the application's functionality has been validated, the running containers can be stopped and removed. ***Do not delete the container images*** - in the next tutorial, you'll upload the azure-vote-front image to an ACR instance.

To stop and remove the container instances and resources, use the [docker-compose down](https://docs.docker.com/compose/reference/down) command.

docker-compose down

When the local application has been removed, you have a Docker image that contains the Azure Vote application, azure-vote-front, to use in the next tutorial.

# Tutorial 02: Deploy and use Azure Container Registry (ACR)

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-acr?tabs=azure-cli>

Azure Container Registry (ACR) is a private registry for container images. A private container registry allows you to securely build and deploy your applications and custom code. In this tutorial, part two of seven, you deploy an ACR instance and push a container image to it. You learn how to:

* Create an ACR instance
* Tag a container image for ACR
* Upload the image to ACR
* View images in your registry

In later tutorials, you integrate your ACR instance with a Kubernetes cluster in AKS, and deploy an application from the image.

## Before you begin

In the [previous tutorial](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app), you created a container image for a simple Azure Voting application. If you haven't created the Azure Voting app image, return to [Tutorial 1: Prepare an application for AKS](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).

## Create an Azure Container Registry

Before creating an ACR, you need a resource group. An Azure resource group is a logical container into which you deploy and manage Azure resources.

1. Create a resource group with the [az group create](https://learn.microsoft.com/en-us/cli/azure/group#az_group_create) command.

az group create --name myResourceGroup --location eastus

1. Create an ACR instance with the [az acr create](https://learn.microsoft.com/en-us/cli/azure/acr#az_acr_create) command and provide your own unique registry name. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. In the rest of this tutorial, <acrName> is used as a placeholder for the container registry name. The Basic SKU is a cost-optimized entry point for development purposes that provides a balance of storage and throughput.

az acr create --resource-group myResourceGroup --name <acrName> --sku Basic

## Log in to the container registry

Log in to your ACR using the [az acr login](https://learn.microsoft.com/en-us/cli/azure/acr#az_acr_login) command and provide the unique name given to the container registry in the previous step.

az acr login --name <acrName>

The command returns a Login Succeeded message once completed.

## Tag a container image

To see a list of your current local images, use the [docker images](https://docs.docker.com/engine/reference/commandline/images/) command.

docker images

The following example output shows a list of the current local Docker images:

REPOSITORY TAG IMAGE ID CREATED SIZE

mcr.microsoft.com/azuredocs/azure-vote-front v1 84b41c268ad9 7 minutes ago 944MB

mcr.microsoft.com/oss/bitnami/redis 6.0.8 3a54a920bb6c 2 days ago 103MB

tiangolo/uwsgi-nginx-flask python3.6 a16ce562e863 6 weeks ago 944MB

To use the azure-vote-front container image with ACR, you need to tag the image with the login server address of your registry. The tag is used for routing when pushing container images to an image registry.

To get the login server address, use the [az acr list](https://learn.microsoft.com/en-us/cli/azure/acr#az_acr_list) command and query for the loginServer.

az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table

Then, tag your local azure-vote-front image with the acrLoginServer address of the container registry. To indicate the image version, add :v1 to the end of the image name:

docker tag mcr.microsoft.com/azuredocs/azure-vote-front:v1 <acrLoginServer>/azure-vote-front:v1

To verify the tags are applied, run [docker images](https://docs.docker.com/engine/reference/commandline/images/) again.

docker images

The following example output shows an image tagged with the ACR instance address and a version number:

REPOSITORY TAG IMAGE ID CREATED SIZE

mcr.microsoft.com/azuredocs/azure-vote-front v1 84b41c268ad9 16 minutes ago 944MB

mycontainerregistry.azurecr.io/azure-vote-front v1 84b41c268ad9 16 minutes ago 944MB

mcr.microsoft.com/oss/bitnami/redis 6.0.8 3a54a920bb6c 2 days ago 103MB

tiangolo/uwsgi-nginx-flask python3.6 a16ce562e863 6 weeks ago 944MB

## Push images to registry

Push the azure-vote-front image to your ACR instance using the [docker push](https://docs.docker.com/engine/reference/commandline/push/) command. Make sure to provide your own acrLoginServer address for the image name.

docker push <acrLoginServer>/azure-vote-front:v1

It may take a few minutes to complete the image push to ACR.

## List images in registry

To return a list of images that have been pushed to your ACR instance, use the [az acr repository list](https://learn.microsoft.com/en-us/cli/azure/acr/repository#az_acr_repository_list) command, providing your own <acrName>.

az acr repository list --name <acrName> --output table

The following example output lists the azure-vote-front image as available in the registry:

Result

----------------

azure-vote-front

To see the tags for a specific image, use the [az acr repository show-tags](https://learn.microsoft.com/en-us/cli/azure/acr/repository#az_acr_repository_show_tags) command.

az acr repository show-tags --name <acrName> --repository azure-vote-front --output table

The following example output shows the v1 image tagged in a previous step:

Result

--------

v1

# Tutorial 03: Deploy an Azure Kubernetes Service (AKS) cluster

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-deploy-cluster?tabs=azure-cli>

Kubernetes provides a distributed platform for containerized applications. With AKS, you can quickly create a production ready Kubernetes cluster. In this tutorial, part three of seven, you deploy a Kubernetes cluster in AKS. You learn how to:

* Deploy a Kubernetes AKS cluster that can authenticate to an Azure Container Registry (ACR).
* Install the Kubernetes CLI, kubectl.
* Configure kubectl to connect to your AKS cluster.

In later tutorials, you'll deploy the Azure Vote application to your AKS cluster and scale and update your application.

## Before you begin

In previous tutorials, you created a container image and uploaded it to an ACR instance. If you haven't done these steps and would like to follow along, start with [Tutorial 1: Prepare an application for AKS](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app).

* If you're using Azure CLI, this tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).
* If you're using Azure PowerShell, this tutorial requires that you're running Azure PowerShell version 5.9.0 or later. Run Get-InstalledModule -Name Az to find the version. If you need to install or upgrade, see [Install Azure PowerShell](https://learn.microsoft.com/en-us/powershell/azure/install-az-ps).

## Create a Kubernetes cluster

AKS clusters can use [Kubernetes role-based access control (Kubernetes RBAC)](https://kubernetes.io/docs/reference/access-authn-authz/rbac/), which allows you to define access to resources based on roles assigned to users. If a user is assigned multiple roles, permissions are combined. Permissions can be scoped to either a single namespace or across the whole cluster.

To learn more about AKS and Kubernetes RBAC, see [Control access to cluster resources using Kubernetes RBAC and Azure Active Directory identities in AKS](https://learn.microsoft.com/en-us/azure/aks/azure-ad-rbac).

Create an AKS cluster using [az aks create](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_create). The following example creates a cluster named myAKSCluster in the resource group named myResourceGroup. This resource group was created in the [previous tutorial](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-acr) in the eastus region. The AKS cluster will also be created in the eastus region.

For more information about AKS resource limits and region availability, see [Quotas, virtual machine size restrictions, and region availability in AKS](https://learn.microsoft.com/en-us/azure/aks/quotas-skus-regions).

To allow an AKS cluster to interact with other Azure resources, a cluster identity is automatically created. In this example, the cluster identity is [granted the right to pull images](https://learn.microsoft.com/en-us/azure/aks/cluster-container-registry-integration) from the ACR instance you created in the previous tutorial. To execute the command successfully, you're required to have an **Owner** or **Azure account administrator** role in your Azure subscription.

az aks create \

--resource-group myResourceGroup \

--name myAKSCluster \

--node-count 2 \

--generate-ssh-keys \

--attach-acr <acrName>

To avoid needing an **Owner** or **Azure account administrator** role, you can also manually configure a service principal to pull images from ACR. For more information, see [ACR authentication with service principals](https://learn.microsoft.com/en-us/azure/container-registry/container-registry-auth-service-principal) or [Authenticate from Kubernetes with a pull secret](https://learn.microsoft.com/en-us/azure/container-registry/container-registry-auth-kubernetes). Alternatively, you can use a [managed identity](https://learn.microsoft.com/en-us/azure/aks/use-managed-identity) instead of a service principal for easier management.

After a few minutes, the deployment completes and returns JSON-formatted information about the AKS deployment.

**Note**: To ensure your cluster operates reliably, you should run at least two nodes.

## Install the Kubernetes CLI

Use the Kubernetes CLI, [kubectl](https://kubernetes.io/docs/user-guide/kubectl/), to connect to the Kubernetes cluster from your local computer.

If you use the Azure Cloud Shell, kubectl is already installed. You can also install it locally using the [az aks install-cli](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_install_cli) command.

az aks install-cli

## Connect to cluster using kubectl

To configure kubectl to connect to your Kubernetes cluster, use the [az aks get-credentials](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_get_credentials) command. The following example gets credentials for the AKS cluster named myAKSCluster in myResourceGroup.

az aks get-credentials --resource-group myResourceGroup --name myAKSCluster

To verify connection to your cluster, run [kubectl get nodes](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) to return a list of cluster nodes.

kubectl get nodes

The following example output shows the list of cluster nodes.

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

aks-nodepool1-37463671-vmss000000 Ready agent 2m37s v1.18.10

aks-nodepool1-37463671-vmss000001 Ready agent 2m28s v1.18.10

# Tutorial 04: Run applications in Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-deploy-application?tabs=azure-cli>

Kubernetes provides a distributed platform for containerized applications. You build and deploy your own applications and services into a Kubernetes cluster and let the cluster manage the availability and connectivity. In this tutorial, part four of seven, you deploy a sample application into a Kubernetes cluster. You learn how to:

* Update a Kubernetes manifest file.
* Run an application in Kubernetes.
* Test the application.

In later tutorials, you'll scale out and update your application.

This quickstart assumes you have a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service (AKS)](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads).

**Tip**: AKS clusters can use GitOps for configuration management. GitOp enables declarations of your cluster's state, which are pushed to source control, to be applied to the cluster automatically. To learn how to use GitOps to deploy an application with an AKS cluster, see the [**prerequisites for Azure Kubernetes Service clusters**](https://learn.microsoft.com/en-us/azure/azure-arc/kubernetes/tutorial-use-gitops-flux2?toc=/azure/aks/toc.json#for-azure-kubernetes-service-clusters) in the [**GitOps with Flux v2**](https://learn.microsoft.com/en-us/azure/azure-arc/kubernetes/tutorial-use-gitops-flux2?toc=/azure/aks/toc.json) tutorial.

## Before you begin

In previous tutorials, you packaged an application into a container image, uploaded the image to Azure Container Registry, and created a Kubernetes cluster.

To complete this tutorial, you need the pre-created azure-vote-all-in-one-redis.yaml Kubernetes manifest file. This file download was included with the application source code in a previous tutorial. Verify that you've cloned the repo and that you've changed directories into the cloned repo. If you haven't done these steps and would like to follow along, start with [Tutorial 1: Prepare an application for AKS](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).

## Update the manifest file

In these tutorials, an Azure Container Registry (ACR) instance stores the container image for the sample application. To deploy the application, you must update the image name in the Kubernetes manifest file to include the ACR login server name.

Get the ACR login server name using the [az acr list](https://learn.microsoft.com/en-us/cli/azure/acr) command.

az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table

The sample manifest file from the git repo you cloned in the first tutorial uses the images from Microsoft Container Registry (mcr.microsoft.com). Make sure you're in the cloned azure-voting-app-redis directory, and then open the manifest file with a text editor, such as vi:

vi azure-vote-all-in-one-redis.yaml

Replace mcr.microsoft.com with your ACR login server name. You can find the image name on line 60 of the manifest file. The following example shows the default image name:

containers:

- name: azure-vote-front

image: mcr.microsoft.com/azuredocs/azure-vote-front:v1

Provide your own ACR login server name so your manifest file looks similar to the following example:

containers:

- name: azure-vote-front

image: <acrName>.azurecr.io/azure-vote-front:v1

Save and close the file. In vi, use :wq.

## Deploy the application

To deploy your application, use the [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) command, specifying the sample manifest file. This command parses the manifest file and creates the defined Kubernetes objects.

kubectl apply -f azure-vote-all-in-one-redis.yaml

The following example output shows the resources successfully created in the AKS cluster:

$ kubectl apply -f azure-vote-all-in-one-redis.yaml

deployment "azure-vote-back" created

service "azure-vote-back" created

deployment "azure-vote-front" created

service "azure-vote-front" created

## Test the application

When the application runs, a Kubernetes service exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the [kubectl get service](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command with the --watch argument.

kubectl get service azure-vote-front --watch

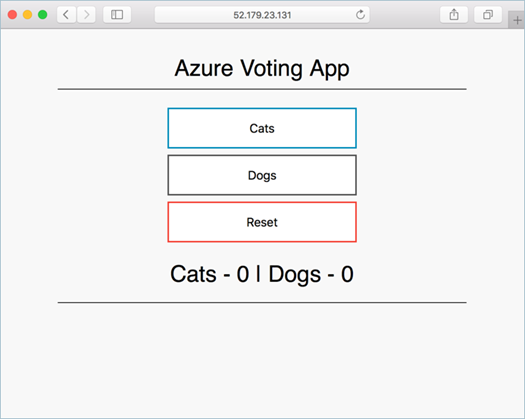
Initially the EXTERNAL-IP for the azure-vote-front service shows as pending.

azure-vote-front LoadBalancer 10.0.34.242 <pending> 80:30676/TCP 5s

When the EXTERNAL-IP address changes from pending to an actual public IP address, use CTRL-C to stop the kubectl watch process. The following example output shows a valid public IP address assigned to the service:

azure-vote-front LoadBalancer 10.0.34.242 52.179.23.131 80:30676/TCP 67s

To see the application in action, open a web browser to the external IP address of your service.

[](https://learn.microsoft.com/en-us/azure/aks/media/container-service-kubernetes-tutorials/azure-vote.png#lightbox)

If the application doesn't load, it might be an authorization problem with your image registry. To view the status of your containers, use the kubectl get pods command. If you can't pull the container images, see [Authenticate with Azure Container Registry from Azure Kubernetes Service](https://learn.microsoft.com/en-us/azure/aks/cluster-container-registry-integration).

# Tutorial 05: Scale applications in Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-scale?tabs=azure-cli>

If you've followed the tutorials, you have a working Kubernetes cluster in AKS and you deployed the sample Azure Voting app. In this tutorial, part five of seven, you scale out the pods in the app and try pod autoscaling. You also learn how to scale the number of Azure VM nodes to change the cluster's capacity for hosting workloads. You learn how to:

* Scale the Kubernetes nodes
* Manually scale Kubernetes pods that run your application
* Configure autoscaling pods that run the app front-end

In later tutorials, the Azure Vote application is updated to a new version.

## Before you begin

In previous tutorials, an application was packaged into a container image. This image was uploaded to Azure Container Registry, and you created an AKS cluster. The application was then deployed to the AKS cluster. If you haven't done these steps, and would like to follow along, start with [Tutorial 1 – Create container images](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).

## Manually scale pods

When the Azure Vote front-end and Redis instance were deployed in previous tutorials, a single replica was created. To see the number and state of pods in your cluster, use the [kubectl get](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command as follows:

kubectl get pods

The following example output shows one front-end pod and one back-end pod:

NAME READY STATUS RESTARTS AGE

azure-vote-back-2549686872-4d2r5 1/1 Running 0 31m

azure-vote-front-848767080-tf34m 1/1 Running 0 31m

To manually change the number of pods in the azure-vote-front deployment, use the [kubectl scale](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#scale) command. The following example increases the number of front-end pods to 5:

kubectl scale --replicas=5 deployment/azure-vote-front

Run [kubectl get pods](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) again to verify that AKS successfully creates the additional pods. After a minute or so, the pods are available in your cluster:

kubectl get pods

READY STATUS RESTARTS AGE

azure-vote-back-2606967446-nmpcf 1/1 Running 0 15m

azure-vote-front-3309479140-2hfh0 1/1 Running 0 3m

azure-vote-front-3309479140-bzt05 1/1 Running 0 3m

azure-vote-front-3309479140-fvcvm 1/1 Running 0 3m

azure-vote-front-3309479140-hrbf2 1/1 Running 0 15m

azure-vote-front-3309479140-qphz8 1/1 Running 0 3m

## Autoscale pods

Kubernetes supports [horizontal pod autoscaling](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/) to adjust the number of pods in a deployment depending on CPU utilization or other select metrics. The [Metrics Server](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/#metrics-server) is used to provide resource utilization to Kubernetes, and is automatically deployed in AKS clusters versions 1.10 and higher. To see the version of your AKS cluster, use the [az aks show](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_show) command, as shown in the following example:

az aks show --resource-group myResourceGroup --name myAKSCluster --query kubernetesVersion --output table

**Note**: If your AKS cluster is less than 1.10, the Metrics Server is not automatically installed. Metrics Server installation manifests are available as a components.yaml asset on Metrics Server releases, which means you can install them via a url. To learn more about these YAML definitions, see the [**Deployment**](https://github.com/kubernetes-sigs/metrics-server/blob/master/README.md#deployment) section of the readme.

Example installation:

kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.6/components.yaml

To use the autoscaler, all containers in your pods and your pods must have CPU requests and limits defined. In the azure-vote-front deployment, the front-end container already requests 0.25 CPU, with a limit of 0.5 CPU.

These resource requests and limits are defined for each container as shown in the following example snippet:

containers:

- name: azure-vote-front

image: mcr.microsoft.com/azuredocs/azure-vote-front:v1

ports:

- containerPort: 80

resources:

requests:

cpu: 250m

limits:

cpu: 500m

The following example uses the [kubectl autoscale](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#autoscale) command to autoscale the number of pods in the azure-vote-front deployment. If average CPU utilization across all pods exceeds 50% of their requested usage, the autoscaler increases the pods up to a maximum of 10 instances. A minimum of 3 instances is then defined for the deployment:

kubectl autoscale deployment azure-vote-front --cpu-percent=50 --min=3 --max=10

Alternatively, you can create a manifest file to define the autoscaler behavior and resource limits. The following is an example of a manifest file named azure-vote-hpa.yaml.

apiVersion: autoscaling/v1

kind: HorizontalPodAutoscaler

metadata:

name: azure-vote-back-hpa

spec:

maxReplicas: 10 # define max replica count

minReplicas: 3 # define min replica count

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: azure-vote-back

targetCPUUtilizationPercentage: 50 # target CPU utilization

---

apiVersion: autoscaling/v1

kind: HorizontalPodAutoscaler

metadata:

name: azure-vote-front-hpa

spec:

maxReplicas: 10 # define max replica count

minReplicas: 3 # define min replica count

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: azure-vote-front

targetCPUUtilizationPercentage: 50 # target CPU utilization

Use kubectl apply to apply the autoscaler defined in the azure-vote-hpa.yaml manifest file.

kubectl apply -f azure-vote-hpa.yaml

To see the status of the autoscaler, use the kubectl get hpa command as follows:

kubectl get hpa

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

azure-vote-front Deployment/azure-vote-front 0% / 50% 3 10 3 2m

After a few minutes, with minimal load on the Azure Vote app, the number of pod replicas decreases automatically to three. You can use kubectl get pods again to see the unneeded pods being removed.

Note: For additional examples on using the horizontal pod autoscaler, see [**HorizontalPodAutoscaler Walkthrough**](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/).

## Manually scale AKS nodes

If you created your Kubernetes cluster using the commands in the previous tutorial, it has two nodes. You can adjust the number of nodes manually if you plan more or fewer container workloads on your cluster.

The following example increases the number of nodes to three in the Kubernetes cluster named myAKSCluster. The command takes a couple of minutes to complete.

az aks scale --resource-group myResourceGroup --name myAKSCluster --node-count 3

When the cluster has successfully scaled, the output is similar to following example:

"agentPoolProfiles": [

{

"count": 3,

"dnsPrefix": null,

"fqdn": null,

"name": "myAKSCluster",

"osDiskSizeGb": null,

"osType": "Linux",

"ports": null,

"storageProfile": "ManagedDisks",

"vmSize": "Standard\_D2\_v2",

"vnetSubnetId": null

}

# Tutorial 06: Update an application in Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-app-update?tabs=azure-cli>

After an application has been deployed in Kubernetes, it can be updated by specifying a new container image or image version. An update is staged so that only a portion of the deployment is updated at the same time. This staged update enables the application to keep running during the update. It also provides a rollback mechanism if a deployment failure occurs.

In this tutorial, part six of seven, the sample Azure Vote app is updated. You learn how to:

* Update the front-end application code
* Create an updated container image
* Push the container image to Azure Container Registry
* Deploy the updated container image

## Before you begin

In previous tutorials, an application was packaged into a container image. This image was uploaded to Azure Container Registry, and you created an AKS cluster. The application was then deployed to the AKS cluster.

An application repository was also cloned that includes the application source code, and a pre-created Docker Compose file used in this tutorial. Verify that you've created a clone of the repo, and have changed directories into the cloned directory. If you haven't completed these steps, and want to follow along, start with [Tutorial 1 – Create container images](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-app).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).

## Update an application

Let's make a change to the sample application, then update the version already deployed to your AKS cluster. Make sure that you're in the cloned azure-voting-app-redis directory. The sample application source code can then be found inside the azure-vote directory. Open the config\_file.cfg file with an editor, such as vi:

vi azure-vote/azure-vote/config\_file.cfg

Change the values for VOTE1VALUE and VOTE2VALUE to different values, such as colors. The following example shows the updated values:

# UI Configurations

TITLE = 'Azure Voting App'

VOTE1VALUE = 'Blue'

VOTE2VALUE = 'Purple'

SHOWHOST = 'false'

Save and close the file. In vi, use :wq.

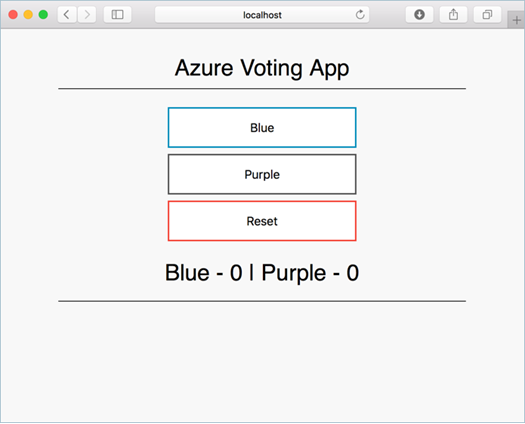
## Update the container image

To re-create the front-end image and test the updated application, use [docker-compose](https://docs.docker.com/compose/). The --build argument is used to instruct Docker Compose to re-create the application image:

docker-compose up --build -d

## Test the application locally

To verify that the updated container image shows your changes, open a local web browser to http://localhost:8080.



The updated values provided in the config\_file.cfg file are displayed in your running application.

## Tag and push the image

To correctly use the updated image, tag the azure-vote-front image with the login server name of your ACR registry. Get the login server name with the [az acr list](https://learn.microsoft.com/en-us/cli/azure/acr#az_acr_list) command:

az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table

Use [docker tag](https://docs.docker.com/engine/reference/commandline/tag/) to tag the image. Replace <acrLoginServer> with your ACR login server name or public registry hostname, and update the image version to :v2 as follows:

docker tag /azure-vote-front:v1 /azure-vote-front:v2

Now use [docker push](https://docs.docker.com/engine/reference/commandline/push/) to upload the image to your registry. Replace <acrLoginServer> with your ACR login server name.

**Note**: If you experience issues pushing to your ACR registry, make sure that you are still logged in. Run the [**az acr login**](https://learn.microsoft.com/en-us/cli/azure/acr#az_acr_login) command using the name of your Azure Container Registry that you created in the [**Create an Azure Container Registry**](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-prepare-acr#create-an-azure-container-registry) step. For example, az acr login --name <azure container registry name>.

docker push <acrLoginServer>/azure-vote-front:v2

## Deploy the updated application

To provide maximum uptime, multiple instances of the application pod must be running. Verify the number of running front-end instances with the [kubectl get pods](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

azure-vote-back-217588096-5w632 1/1 Running 0 10m

azure-vote-front-233282510-b5pkz 1/1 Running 0 10m

azure-vote-front-233282510-dhrtr 1/1 Running 0 10m

azure-vote-front-233282510-pqbfk 1/1 Running 0 10m

If you don't have multiple front-end pods, scale the azure-vote-front deployment as follows:

kubectl scale --replicas=3 deployment/azure-vote-front

To update the application, use the [kubectl set](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#set) command. Update <acrLoginServer> with the login server or host name of your container registry, and specify the v2 application version:

kubectl set image deployment azure-vote-front azure-vote-front=<acrLoginServer>/azure-vote-front:v2

To monitor the deployment, use the [kubectl get pod](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command. As the updated application is deployed, your pods are terminated and re-created with the new container image.

kubectl get pods

The following example output shows pods terminating and new instances running as the deployment progresses:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

azure-vote-back-2978095810-gq9g0 1/1 Running 0 5m

azure-vote-front-1297194256-tpjlg 1/1 Running 0 1m

azure-vote-front-1297194256-tptnx 1/1 Running 0 5m

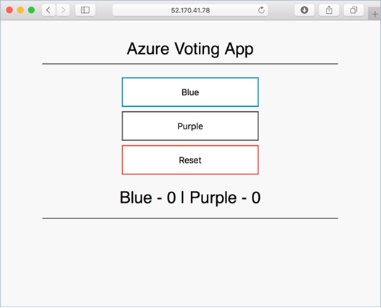
azure-vote-front-1297194256-zktw9 1/1 Terminating 0 1m

## Test the updated application

To view the update application, first get the external IP address of the azure-vote-front service:

kubectl get service azure-vote-front

Now open a web browser to the IP address of your service:



# Tutorial 07: Upgrade Kubernetes in Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-upgrade-cluster?tabs=azure-cli>

## Using Azure CLI

### Get available cluster versions

Before you upgrade a cluster, use the [az aks get-upgrades](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_get_upgrades) command to check which Kubernetes releases are available.

az aks get-upgrades --resource-group myResourceGroup --name myAKSCluster

In the following example output, the current version is 1.18.10, and the available versions are shown under upgrades.

{

"agentPoolProfiles": null,

"controlPlaneProfile": {

"kubernetesVersion": "1.18.10",

...

"upgrades": [

{

"isPreview": null,

"kubernetesVersion": "1.19.1"

},

{

"isPreview": null,

"kubernetesVersion": "1.19.3"

}

]

},

...

}

### Upgrade a cluster

AKS nodes are carefully cordoned and drained to minimize any potential disruptions to running applications. The following steps are performed during this process:

1. The Kubernetes scheduler prevents additional pods from being scheduled on a node that is to be upgraded.
2. Running pods on the node are scheduled on other nodes in the cluster.
3. A new node is created that runs the latest Kubernetes components.
4. When the new node is ready and joined to the cluster, the Kubernetes scheduler begins to run pods on the new node.
5. The old node is deleted, and the next node in the cluster begins the cordon and drain process.

**Note**: If no patch is specified, the cluster will automatically be upgraded to the specified minor version's latest GA patch. For example, setting --kubernetes-version to 1.21 will result in the cluster upgrading to 1.21.9.

When upgrading by alias minor version, only a higher minor version is supported. For example, upgrading from 1.20.x to 1.20 will not trigger an upgrade to the latest GA 1.20 patch, but upgrading to 1.21 will trigger an upgrade to the latest GA 1.21 patch.

Use the [az aks upgrade](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_upgrade) command to upgrade your AKS cluster.

az aks upgrade \

--resource-group myResourceGroup \

--name myAKSCluster \

--kubernetes-version KUBERNETES\_VERSION

**Note**: You can only upgrade one minor version at a time. For example, you can upgrade from 1.14.x to 1.15.x, but you cannot upgrade from 1.14.x to 1.16.x directly. To upgrade from 1.14.x to 1.16.x, you must first upgrade from 1.14.x to 1.15.x, then perform another upgrade from 1.15.x to 1.16.x.

The following example output shows the result of upgrading to 1.19.1. Notice the kubernetesVersion now reports 1.19.1.

{

"agentPoolProfiles": [

{

"count": 3,

"maxPods": 110,

"name": "nodepool1",

"osType": "Linux",

"storageProfile": "ManagedDisks",

"vmSize": "Standard\_DS1\_v2",

}

],

"dnsPrefix": "myAKSClust-myResourceGroup-19da35",

"enableRbac": false,

"fqdn": "myaksclust-myresourcegroup-19da35-bd54a4be.hcp.eastus.azmk8s.io",

"id": "/subscriptions/<Subscription ID>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/managedClusters/myAKSCluster",

"kubernetesVersion": "1.19.1",

"location": "eastus",

"name": "myAKSCluster",

"type": "Microsoft.ContainerService/ManagedClusters"

}

### View the upgrade events

When you upgrade your cluster, the following Kubernetes events may occur on the nodes:

* **Surge**: Create surge node.
* **Drain**: Pods are being evicted from the node. Each pod has a 5 minute timeout to complete the eviction.
* **Update**: Update of a node has succeeded or failed.
* **Delete**: Delete a surge node.

Use kubectl get events to show events in the default namespaces while running an upgrade.

kubectl get events

The following example output shows some of the above events listed during an upgrade.

...

default 2m1s Normal Drain node/aks-nodepool1-96663640-vmss000001 Draining node: [aks-nodepool1-96663640-vmss000001]

...

default 9m22s Normal Surge node/aks-nodepool1-96663640-vmss000002 Created a surge node [aks-nodepool1-96663640-vmss000002 nodepool1] for agentpool %!s(MISSING)

...

### Validate an upgrade

Confirm that the upgrade was successful using the [az aks show](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_show) command.

az aks show --resource-group myResourceGroup --name myAKSCluster --output table

The following example output shows the AKS cluster runs KubernetesVersion 1.19.1:

Name Location ResourceGroup KubernetesVersion CurrentKubernetesVersion ProvisioningState Fqdn

------------ ---------- --------------- ------------------- ------------------------ ------------------- ----------------------------------------------------------------

myAKSCluster eastus myResourceGroup 1.19.1 1.19.1 Succeeded myaksclust-myresourcegroup-19da35-bd54a4be.hcp.eastus.azmk8s.io

### Delete the cluster

As this tutorial is the last part of the series, you may want to delete your AKS cluster. The Kubernetes nodes run on Azure virtual machines and continue incurring charges even if you don't use the cluster.

Use the [az group delete](https://learn.microsoft.com/en-us/cli/azure/group#az_group_delete) command to remove the resource group, container service, and all related resources.

az group delete --name myResourceGroup --yes --no-wait

**Note**: When you delete the cluster, the Azure Active Directory (AAD) service principal used by the AKS cluster isn't removed. For steps on how to remove the service principal, see [**AKS service principal considerations and deletion**](https://learn.microsoft.com/en-us/azure/aks/kubernetes-service-principal#other-considerations). If you used a managed identity, the identity is managed by the platform and it doesn't require that you provision or rotate any secrets.

## Using Azure Portal

<https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-upgrade-cluster?tabs=azure-portal>

### Get available cluster versions

To check which Kubernetes releases are available for your cluster:

1. Sign in to the [Azure portal](https://portal.azure.com/).
2. Navigate to your AKS cluster.
3. Under **Settings**, select **Cluster configuration**.
4. In **Kubernetes version**, select **Upgrade version**. This will redirect you to a new page.
5. In **Kubernetes version**, select the version to check for available upgrades.

If no upgrades are available, create a new cluster with a supported version of Kubernetes and migrate your workloads from the existing cluster to the new cluster. It's not supported to upgrade a cluster to a newer Kubernetes version when no upgrades are available.

### Upgrade a cluster

AKS nodes are carefully cordoned and drained to minimize any potential disruptions to running applications. The following steps are performed during this process:

1. The Kubernetes scheduler prevents additional pods from being scheduled on a node that is to be upgraded.
2. Running pods on the node are scheduled on other nodes in the cluster.
3. A new node is created that runs the latest Kubernetes components.
4. When the new node is ready and joined to the cluster, the Kubernetes scheduler begins to run pods on the new node.
5. The old node is deleted, and the next node in the cluster begins the cordon and drain process.

**Note**: If no patch is specified, the cluster will automatically be upgraded to the specified minor version's latest GA patch. For example, setting --kubernetes-version to 1.21 will result in the cluster upgrading to 1.21.9.

When upgrading by alias minor version, only a higher minor version is supported. For example, upgrading from 1.20.x to 1.20 will not trigger an upgrade to the latest GA 1.20 patch, but upgrading to 1.21 will trigger an upgrade to the latest GA 1.21 patch.

To upgrade your AKS cluster:

1. In the Azure portal, navigate to your AKS cluster.
2. Under **Settings**, select **Cluster configuration**.
3. In **Kubernetes version**, select **Upgrade version**. This will redirect you to a new page.
4. In **Kubernetes version**, select your desired version and then select **Save**.

It takes a few minutes to upgrade the cluster, depending on how many nodes you have.

### View the upgrade events

When you upgrade your cluster, the following Kubernetes events may occur on the nodes:

* **Surge**: Create surge node.
* **Drain**: Pods are being evicted from the node. Each pod has a 5 minute timeout to complete the eviction.
* **Update**: Update of a node has succeeded or failed.
* **Delete**: Delete a surge node.

Use kubectl get events to show events in the default namespaces while running an upgrade.

kubectl get events

The following example output shows some of the above events listed during an upgrade.

...

default 2m1s Normal Drain node/aks-nodepool1-96663640-vmss000001 Draining node: [aks-nodepool1-96663640-vmss000001]

...

default 9m22s Normal Surge node/aks-nodepool1-96663640-vmss000002 Created a surge node [aks-nodepool1-96663640-vmss000002 nodepool1] for agentpool %!s(MISSING)

...

### Validate an upgrade

To confirm that the upgrade was successful, navigate to your AKS cluster in the Azure portal. On the **Overview** page, select the **Kubernetes version** and ensure it's the latest version you installed in the previous step.

### Delete the cluster

As this tutorial is the last part of the series, you may want to delete your AKS cluster. The Kubernetes nodes run on Azure virtual machines and continue incurring charges even if you don't use the cluster.

To delete your AKS cluster:

1. In the Azure portal, navigate to your AKS cluster.
2. On the **Overview** page, select **Delete**.
3. A popup will appear that asks you to confirm the deletion of the cluster. Select **Yes**.

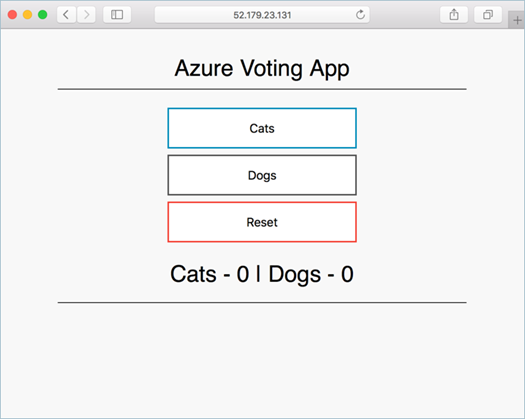
**Note**: When you delete the cluster, the Azure Active Directory (AAD) service principal used by the AKS cluster isn't removed. For steps on how to remove the service principal, see [**AKS service principal considerations and deletion**](https://learn.microsoft.com/en-us/azure/aks/kubernetes-service-principal#other-considerations). If you used a managed identity, the identity is managed by the platform and it doesn't require that you provision or rotate any secrets.

# Quickstart: Deploy an Azure Kubernetes Service cluster using the Azure CLI

<https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-cli>

Azure Kubernetes Service (AKS) is a managed Kubernetes service that lets you quickly deploy and manage clusters. In this quickstart, you will:

* Deploy an AKS cluster using the Azure CLI.
* Run a sample multi-container application with a web front-end and a Redis instance in the cluster.



This quickstart assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service (AKS)](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads).

If you don't have an [Azure subscription](https://learn.microsoft.com/en-us/azure/guides/developer/azure-developer-guide#understanding-accounts-subscriptions-and-billing), create an [Azure free account](https://azure.microsoft.com/free/?ref=microsoft.com&utm_source=microsoft.com&utm_medium=docs&utm_campaign=visualstudio) before you begin.

To learn more about creating a Windows Server node pool, see [Create an AKS cluster that supports Windows Server containers](https://learn.microsoft.com/en-us/azure/aks/learn/quick-windows-container-deploy-cli).

## Prerequisites

* Use the Bash environment in [Azure Cloud Shell](https://learn.microsoft.com/en-us/azure/cloud-shell/overview). For more information, see [Quickstart for Bash in Azure Cloud Shell](https://learn.microsoft.com/en-us/azure/cloud-shell/quickstart).

[](https://shell.azure.com/)

* If you prefer to run CLI reference commands locally, [install](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](https://learn.microsoft.com/en-us/cli/azure/run-azure-cli-docker).
  + If you're using a local installation, sign in to the Azure CLI by using the [az login](https://learn.microsoft.com/en-us/cli/azure/reference-index#az-login) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](https://learn.microsoft.com/en-us/cli/azure/authenticate-azure-cli).
  + When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](https://learn.microsoft.com/en-us/cli/azure/azure-cli-extensions-overview).
  + Run [az version](https://learn.microsoft.com/en-us/cli/azure/reference-index?#az-version) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](https://learn.microsoft.com/en-us/cli/azure/reference-index?#az-upgrade).
* This article requires version 2.0.64 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.
* The identity you are using to create your cluster has the appropriate minimum permissions. For more details on access and identity for AKS, see [Access and identity options for Azure Kubernetes Service (AKS)](https://learn.microsoft.com/en-us/azure/aks/concepts-identity).
* If you have multiple Azure subscriptions, select the appropriate subscription ID in which the resources should be billed using the [az account](https://learn.microsoft.com/en-us/cli/azure/account) command.
* Verify Microsoft.OperationsManagement and Microsoft.OperationalInsights providers are registered on your subscription. These are Azure resource providers required to support [Container insights](https://learn.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-overview). To check the registration status, run the following commands:

az provider show -n Microsoft.OperationsManagement -o table

az provider show -n Microsoft.OperationalInsights -o table

If they are not registered, register Microsoft.OperationsManagement and Microsoft.OperationalInsights using the following commands:

az provider register --namespace Microsoft.OperationsManagement

az provider register --namespace Microsoft.OperationalInsights

**Note**: Run the commands with administrative privileges if you plan to run the commands in this quickstart locally instead of in Azure Cloud Shell.

## Create a resource group

An [Azure resource group](https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/overview) is a logical group in which Azure resources are deployed and managed. When you create a resource group, you are prompted to specify a location. This location is:

* The storage location of your resource group metadata.
* Where your resources will run in Azure if you don't specify another region during resource creation.

The following example creates a resource group named myResourceGroup in the eastus location.

Create a resource group using the [az group create](https://learn.microsoft.com/en-us/cli/azure/group#az-group-create) command.

az group create --name myResourceGroup --location eastus

The following output example resembles successful creation of the resource group:

JSON:

{

"id": "/subscriptions/<guid>/resourceGroups/myResourceGroup",

"location": "eastus",

"managedBy": null,

"name": "myResourceGroup",

"properties": {

"provisioningState": "Succeeded"

},

"tags": null

}

## Create AKS cluster

Create an AKS cluster using the [az aks create](https://learn.microsoft.com/en-us/cli/azure/aks#az-aks-create) command with the --enable-addons monitoring and --enable-msi-auth-for-monitoring parameter to enable [Azure Monitor Container insights](https://learn.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-overview) with managed identity authentication (preview). The following example creates a cluster named myAKSCluster with one node and enables a system-assigned managed identity:

az aks create -g myResourceGroup -n myAKSCluster --enable-managed-identity --node-count 1 --enable-addons monitoring --enable-msi-auth-for-monitoring --generate-ssh-keys

After a few minutes, the command completes and returns JSON-formatted information about the cluster.

**Note**: When you create an AKS cluster, a second resource group is automatically created to store the AKS resources. For more information, see [**Why are two resource groups created with AKS?**](https://learn.microsoft.com/en-us/azure/aks/faq#why-are-two-resource-groups-created-with-aks)

## Connect to the cluster

To manage a Kubernetes cluster, use the Kubernetes command-line client, [kubectl](https://kubernetes.io/docs/user-guide/kubectl/). kubectl is already installed if you use Azure Cloud Shell.

1. Install kubectl locally using the [az aks install-cli](https://learn.microsoft.com/en-us/cli/azure/aks#az-aks-install-cli) command:

az aks install-cli

1. Configure kubectl to connect to your Kubernetes cluster using the [az aks get-credentials](https://learn.microsoft.com/en-us/cli/azure/aks#az-aks-get-credentials) command. The following command:

* Downloads credentials and configures the Kubernetes CLI to use them.
* Uses ~/.kube/config, the default location for the [Kubernetes configuration file](https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/). Specify a different location for your Kubernetes configuration file using --file argument.

az aks get-credentials --resource-group myResourceGroup --name myAKSCluster

1. Verify the connection to your cluster using the [kubectl get](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command. This command returns a list of the cluster nodes.

kubectl get nodes

The following output example shows the single node created in the previous steps. Make sure the node status is Ready:

Output:

NAME STATUS ROLES AGE VERSION

aks-nodepool1-31718369-0 Ready agent 6m44s v1.12.8

## Deploy the application

A [Kubernetes manifest file](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads#deployments-and-yaml-manifests) defines a cluster's desired state, such as which container images to run.

In this quickstart, you will use a manifest to create all objects needed to run the [Azure Vote application](https://github.com/Azure-Samples/azure-voting-app-redis.git). This manifest includes two [Kubernetes deployments](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads#deployments-and-yaml-manifests):

* The sample Azure Vote Python applications.
* A Redis instance.

Two [Kubernetes Services](https://learn.microsoft.com/en-us/azure/aks/concepts-network#services) are also created:

* An internal service for the Redis instance.
* An external service to access the Azure Vote application from the internet.

1. Create a file named azure-vote.yaml and copy in the following manifest.

* If you use the Azure Cloud Shell, this file can be created using code, vi, or nano as if working on a virtual or physical system.

YAML:

apiVersion: apps/v1

kind: Deployment

metadata:

name: azure-vote-back

spec:

replicas: 1

selector:

matchLabels:

app: azure-vote-back

template:

metadata:

labels:

app: azure-vote-back

spec:

nodeSelector:

"kubernetes.io/os": linux

containers:

- name: azure-vote-back

image: mcr.microsoft.com/oss/bitnami/redis:6.0.8

env:

- name: ALLOW\_EMPTY\_PASSWORD

value: "yes"

resources:

requests:

cpu: 100m

memory: 128Mi

limits:

cpu: 250m

memory: 256Mi

ports:

- containerPort: 6379

name: redis

---

apiVersion: v1

kind: Service

metadata:

name: azure-vote-back

spec:

ports:

- port: 6379

selector:

app: azure-vote-back

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: azure-vote-front

spec:

replicas: 1

selector:

matchLabels:

app: azure-vote-front

template:

metadata:

labels:

app: azure-vote-front

spec:

nodeSelector:

"kubernetes.io/os": linux

containers:

- name: azure-vote-front

image: mcr.microsoft.com/azuredocs/azure-vote-front:v1

resources:

requests:

cpu: 100m

memory: 128Mi

limits:

cpu: 250m

memory: 256Mi

ports:

- containerPort: 80

env:

- name: REDIS

value: "azure-vote-back"

---

apiVersion: v1

kind: Service

metadata:

name: azure-vote-front

spec:

type: LoadBalancer

ports:

- port: 80

selector:

app: azure-vote-front

For a breakdown of YAML manifest files, see [Deployments and YAML manifests](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads#deployments-and-yaml-manifests).

1. Deploy the application using the [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) command and specify the name of your YAML manifest:

kubectl apply -f azure-vote.yaml

The following example resembles output showing the successfully created deployments and services:

Output:

deployment "azure-vote-back" created

service "azure-vote-back" created

deployment "azure-vote-front" created

service "azure-vote-front" created

## Test the application

When the application runs, a Kubernetes service exposes the application front-end to the internet. This process can take a few minutes to complete.

Monitor progress using the [kubectl get service](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command with the --watch argument.

kubectl get service azure-vote-front --watch

The **EXTERNAL-IP** output for the azure-vote-front service will initially show as pending.

Output:

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

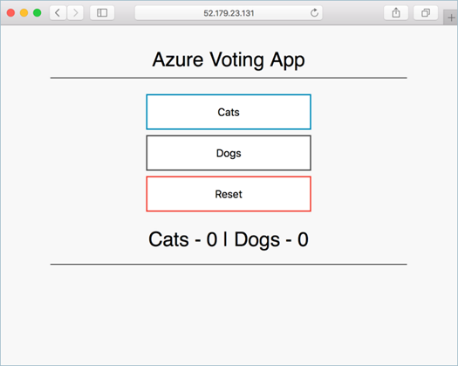
azure-vote-front LoadBalancer 10.0.37.27 <pending> 80:30572/TCP 6s

Once the **EXTERNAL-IP** address changes from pending to an actual public IP address, use CTRL-C to stop the kubectl watch process. The following example output shows a valid public IP address assigned to the service:

Output:

azure-vote-front LoadBalancer 10.0.37.27 52.179.23.131 80:30572/TCP 2m

To see the Azure Vote app in action, open a web browser to the external IP address of your service.



## Delete the cluster

To avoid Azure charges, if you don't plan on going through the tutorials that follow, clean up your unnecessary resources. Use the [az group delete](https://learn.microsoft.com/en-us/cli/azure/group#az-group-delete) command to remove the resource group, container service, and all related resources.

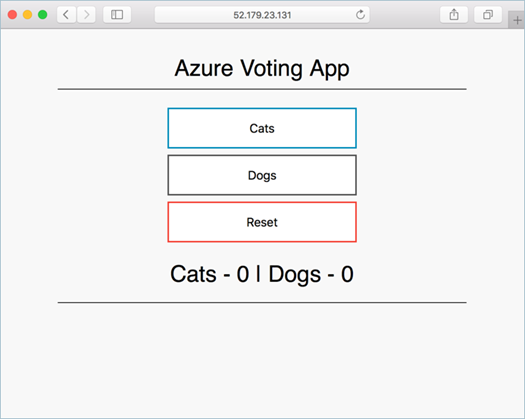
az group delete --name myResourceGroup --yes --no-wait

**Note**: The AKS cluster was created with system-assigned managed identity (default identity option used in this quickstart), the identity is managed by the platform and does not require removal.

# Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using the Azure portal

Azure Kubernetes Service (AKS) is a managed Kubernetes service that lets you quickly deploy and manage clusters. In this quickstart, you will:

* Deploy an AKS cluster using the Azure portal.
* Run a sample multi-container application with a web front-end and a Redis instance in the cluster.



This quickstart assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service (AKS)](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads).

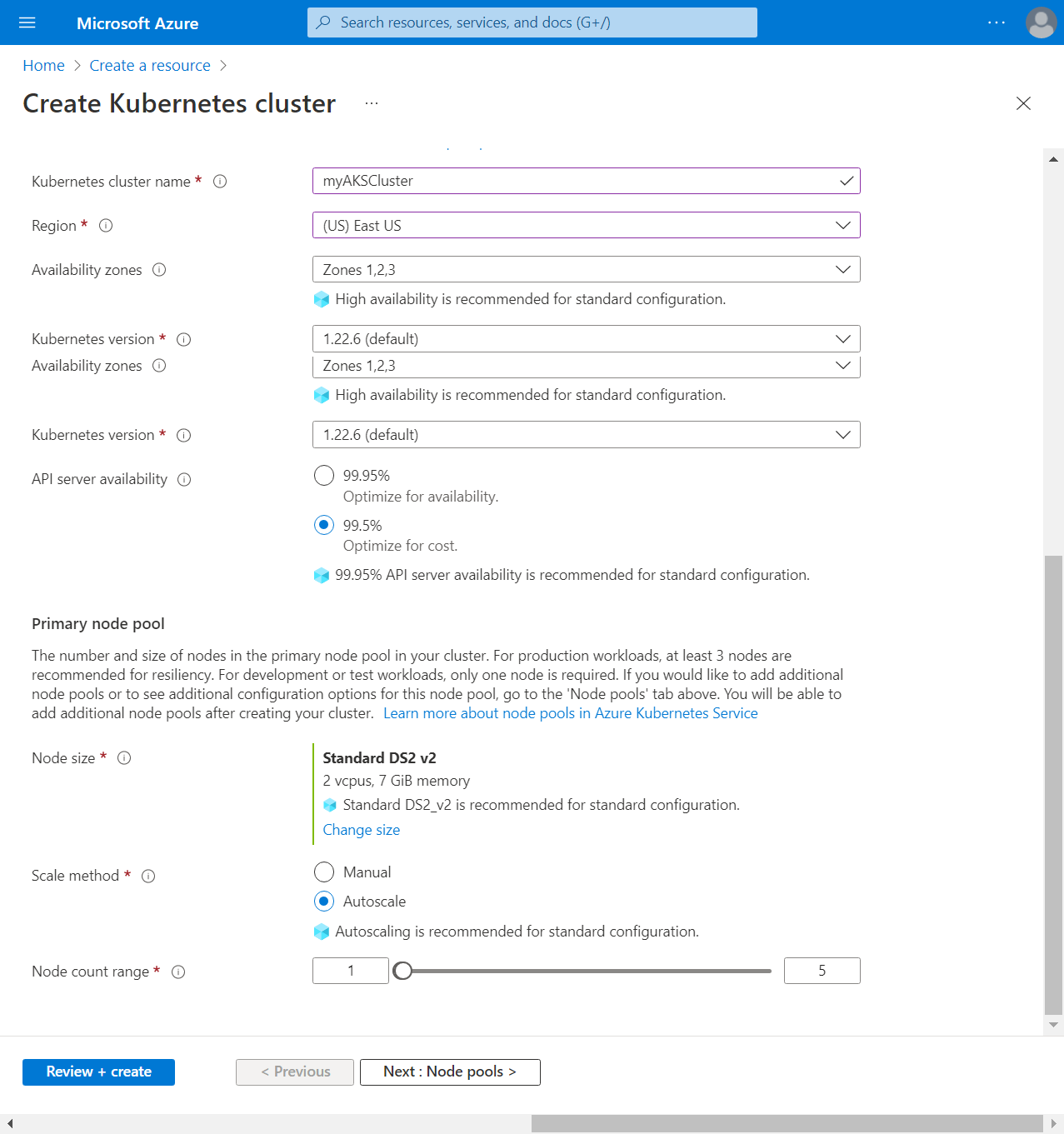
## Prerequisites

If you don't have an [Azure subscription](https://learn.microsoft.com/en-us/azure/guides/developer/azure-developer-guide#understanding-accounts-subscriptions-and-billing), create an [Azure free account](https://azure.microsoft.com/free/?ref=microsoft.com&utm_source=microsoft.com&utm_medium=docs&utm_campaign=visualstudio) before you begin.

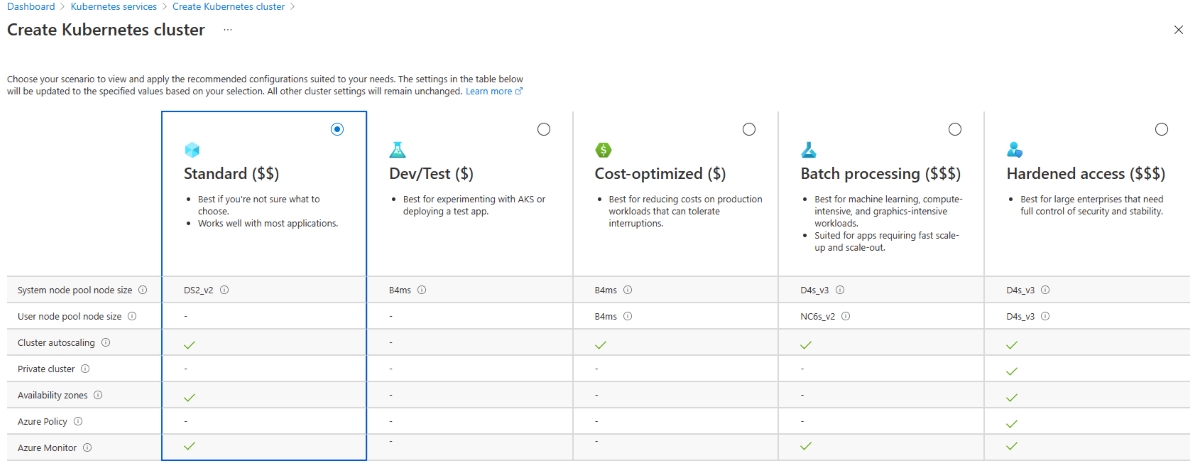
* If you're unfamiliar with the Azure Cloud Shell, review [Overview of Azure Cloud Shell](https://learn.microsoft.com/en-us/azure/cloud-shell/overview).
* The identity you're using to create your cluster has the appropriate minimum permissions. For more details on access and identity for AKS, see [Access and identity options for Azure Kubernetes Service (AKS)](https://learn.microsoft.com/en-us/azure/aks/concepts-identity).

## Create an AKS cluster

1. Sign in to the [Azure portal](https://portal.azure.com/).
2. On the Azure portal menu or from the **Home** page, select **Create a resource**.
3. Select **Containers** > **Kubernetes Service**.
4. On the **Basics** page, configure the following options:
   * **Project details**:
     + Select an Azure **Subscription**.
     + Select or create an Azure **Resource group**, such as myResourceGroup.
   * **Cluster details**:
     + Ensure the the **Preset configuration** is Standard ($$). For more details on preset configurations, see [Cluster configuration presets in the Azure portal](https://learn.microsoft.com/en-us/azure/aks/quotas-skus-regions#cluster-configuration-presets-in-the-azure-portal).
     + Enter a **Kubernetes cluster name**, such as myAKSCluster.
     + Select a **Region** for the AKS cluster, and leave the default value selected for **Kubernetes version**.
     + Select **99.5%** for **API server availability**.
   * **Primary node pool**:
     + Leave the default values selected.



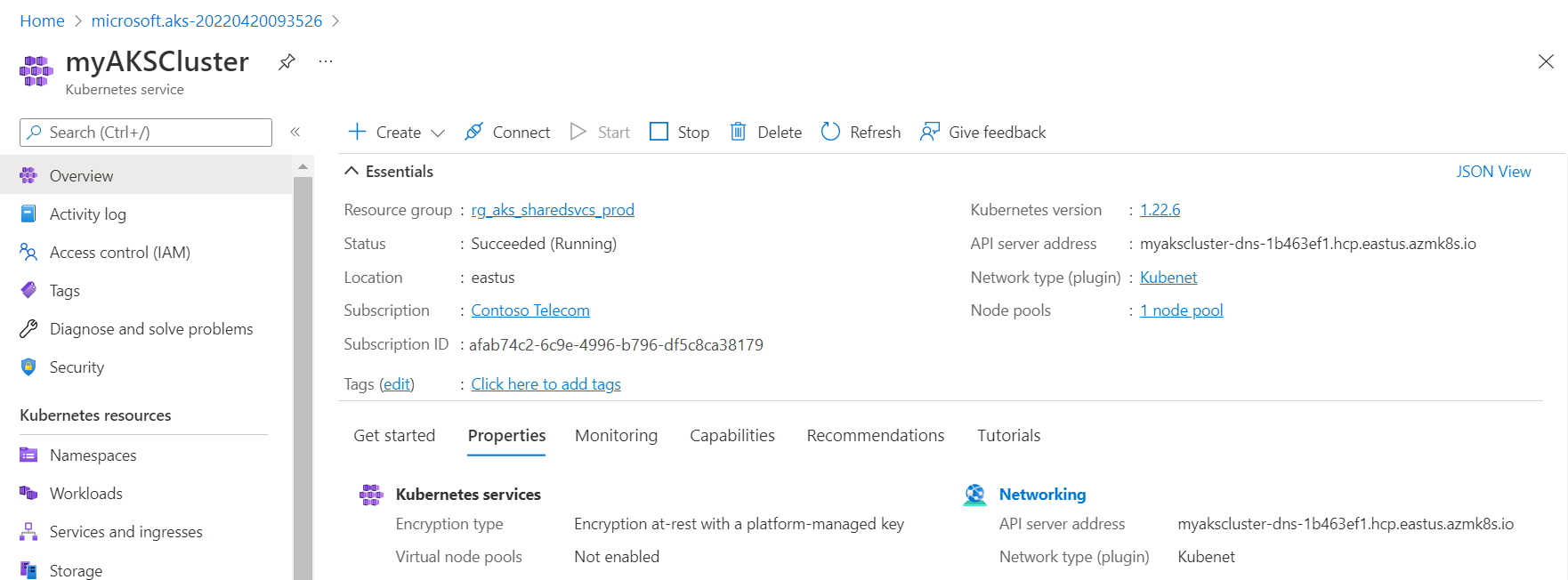
**Note**: You can change the preset configuration when creating your cluster by selecting Learn more and compare presets and choosing a different option.



1. Select **Next: Node pools** when complete.
2. Keep the default **Node pools** options. At the bottom of the screen, click **Next: Access**.
3. On the **Access** page, configure the following options:
   * The default value for **Resource identity** is **System-assigned managed identity**. Managed identities provide an identity for applications to use when connecting to resources that support Azure Active Directory (Azure AD) authentication. For more details about managed identities, see [What are managed identities for Azure resources?](https://learn.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview).
   * The Kubernetes role-based access control (RBAC) option is the default value to provide more fine-grained control over access to the Kubernetes resources deployed in your AKS cluster.

By default, Basic networking is used, and [Container insights](https://learn.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-overview) is enabled.

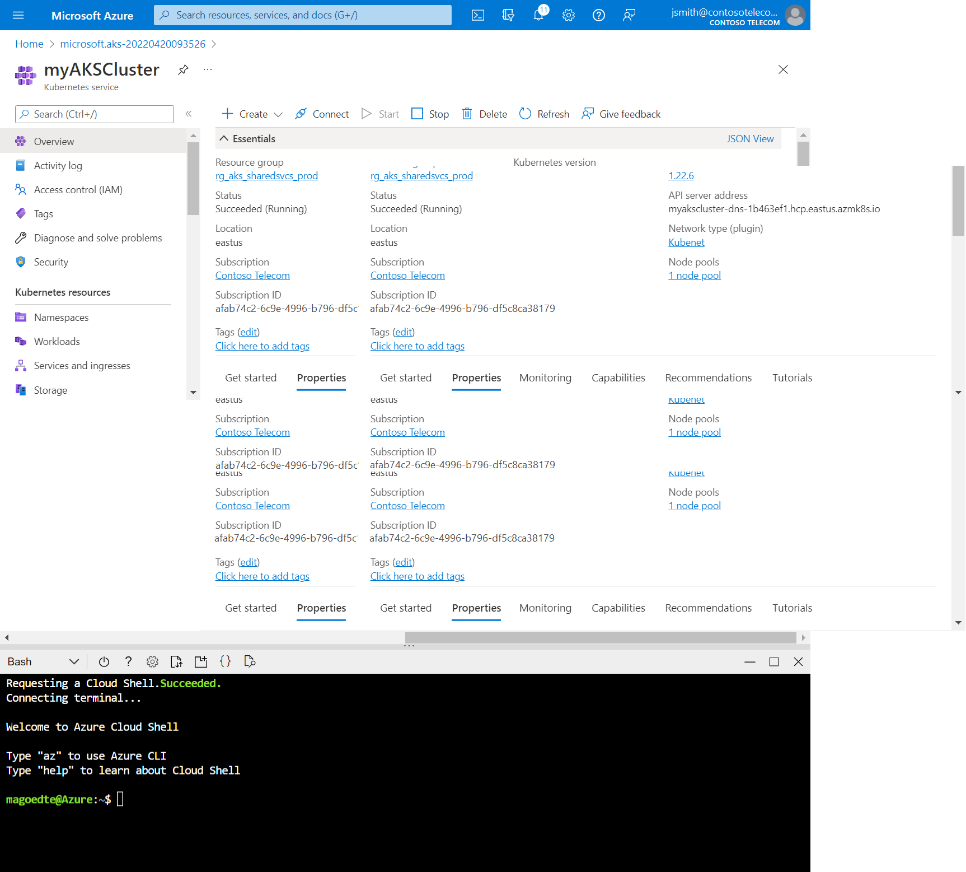
1. Click **Review + create**. When you navigate to the **Review + create** tab, Azure runs validation on the settings that you have chosen. If validation passes, you can proceed to create the AKS cluster by selecting **Create**. If validation fails, then it indicates which settings need to be modified.
2. It takes a few minutes to create the AKS cluster. When your deployment is complete, navigate to your resource by either:
   * Selecting **Go to resource**, or
   * Browsing to the AKS cluster resource group and selecting the AKS resource. In this example you browse for myResourceGroup and select the resource myAKSCluster.



## Connect to the cluster

To manage a Kubernetes cluster, use the Kubernetes command-line client, [kubectl](https://kubernetes.io/docs/user-guide/kubectl/). kubectl is already installed if you use Azure Cloud Shell. If you're unfamiliar with the Cloud Shell, review [Overview of Azure Cloud Shell](https://learn.microsoft.com/en-us/azure/cloud-shell/overview).

1. Open Cloud Shell using the >\_ button on the top of the Azure portal.



**Note**: To perform these operations in a local shell installation:

* 1. Verify Azure CLI or Azure PowerShell is installed.
  2. Connect to Azure via the az login or Connect-AzAccount command.

1. Configure kubectl to connect to your Kubernetes cluster using the [az aks get-credentials](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_get_credentials) command. The following command downloads credentials and configures the Kubernetes CLI to use them.

az aks get-credentials --resource-group myResourceGroup --name myAKSCluster

1. Verify the connection to your cluster using kubectl get to return a list of the cluster nodes.

kubectl get nodes

Output shows the single node created in the previous steps. Make sure the node status is Ready:

Output:

NAME STATUS ROLES AGE VERSION

aks-agentpool-12345678-vmss000000 Ready agent 23m v1.19.11

aks-agentpool-12345678-vmss000001 Ready agent 24m v1.19.11

## Deploy the application

A Kubernetes manifest file defines a cluster's desired state, like which container images to run.

In this quickstart, you will use a manifest to create all objects needed to run the Azure Vote application. This manifest includes two Kubernetes deployments:

* The sample Azure Vote Python applications.
* A Redis instance.

Two Kubernetes Services are also created:

* An internal service for the Redis instance.
* An external service to access the Azure Vote application from the internet.

1. In the Cloud Shell, use an editor to create a file named azure-vote.yaml, such as:

* code azure-vote.yaml
* nano azure-vote.yaml, or
* vi azure-vote.yaml.

1. Copy in the following YAML definition:

YAML

apiVersion: apps/v1

kind: Deployment

metadata:

name: azure-vote-back

spec:

replicas: 1

selector:

matchLabels:

app: azure-vote-back

template:

metadata:

labels:

app: azure-vote-back

spec:

nodeSelector:

"kubernetes.io/os": linux

containers:

- name: azure-vote-back

image: mcr.microsoft.com/oss/bitnami/redis:6.0.8

env:

- name: ALLOW\_EMPTY\_PASSWORD

value: "yes"

resources:

requests:

cpu: 100m

memory: 128Mi

limits:

cpu: 250m

memory: 256Mi

ports:

- containerPort: 6379

name: redis

---

apiVersion: v1

kind: Service

metadata:

name: azure-vote-back

spec:

ports:

- port: 6379

selector:

app: azure-vote-back

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: azure-vote-front

spec:

replicas: 1

selector:

matchLabels:

app: azure-vote-front

template:

metadata:

labels:

app: azure-vote-front

spec:

nodeSelector:

"kubernetes.io/os": linux

containers:

- name: azure-vote-front

image: mcr.microsoft.com/azuredocs/azure-vote-front:v1

resources:

requests:

cpu: 100m

memory: 128Mi

limits:

cpu: 250m

memory: 256Mi

ports:

- containerPort: 80

env:

- name: REDIS

value: "azure-vote-back"

---

apiVersion: v1

kind: Service

metadata:

name: azure-vote-front

spec:

type: LoadBalancer

ports:

- port: 80

selector:

app: azure-vote-front

For a breakdown of YAML manifest files, see [Deployments and YAML manifests](https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads#deployments-and-yaml-manifests).

1. Deploy the application using the kubectl apply command and specify the name of your YAML manifest:

kubectl apply -f azure-vote.yaml

Output shows the successfully created deployments and services:

Output:

deployment "azure-vote-back" created

service "azure-vote-back" created

deployment "azure-vote-front" created

service "azure-vote-front" created

## Test the application

When the application runs, a Kubernetes service exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the kubectl get service command with the --watch argument.

kubectl get service azure-vote-front --watch

The **EXTERNAL-IP** output for the azure-vote-front service will initially show as pending.

Output:

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

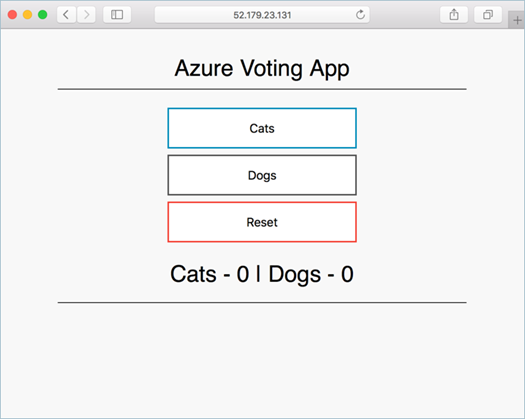
azure-vote-front LoadBalancer 10.0.37.27 <pending> 80:30572/TCP 6s

Once the **EXTERNAL-IP** address changes from pending to an actual public IP address, use CTRL-C to stop the kubectl watch process. The following example output shows a valid public IP address assigned to the service:

Output:

azure-vote-front LoadBalancer 10.0.37.27 52.179.23.131 80:30572/TCP 2m

To see the Azure Vote app in action, open a web browser to the external IP address of your service.



## Delete cluster

To avoid Azure charges, if you don't plan on going through the tutorials that follow, clean up your unnecessary resources. Select the **Delete** button on the AKS cluster dashboard. You can also use the [az group delete](https://learn.microsoft.com/en-us/cli/azure/group#az-group-delete) command or the [Remove-AzResourceGroup](https://learn.microsoft.com/en-us/powershell/module/az.resources/remove-azresourcegroup) cmdlet to remove the resource group, container service, and all related resources.

az group delete --name myResourceGroup --yes --no-wait

**Note**: The AKS cluster was created with a system-assigned managed identity. This identity is managed by the platform and doesn't require removal.

# Kubernetes core concepts for Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/concepts-clusters-workloads>

Application development continues to move toward a container-based approach, increasing our need to orchestrate and manage resources. As the leading platform, Kubernetes provides reliable scheduling of fault-tolerant application workloads. Azure Kubernetes Service (AKS), a managed Kubernetes offering, further simplifies container-based application deployment and management.

This article introduces:

* Core Kubernetes infrastructure components:
  + control plane
  + nodes
  + node pools
* Workload resources:
  + pods
  + deployments
  + sets
* How to group resources into namespaces.

## What is Kubernetes?

Kubernetes is a rapidly evolving platform that manages container-based applications and their associated networking and storage components. Kubernetes focuses on the application workloads, not the underlying infrastructure components. Kubernetes provides a declarative approach to deployments, backed by a robust set of APIs for management operations.

You can build and run modern, portable, microservices-based applications, using Kubernetes to orchestrate and manage the availability of the application components. Kubernetes supports both stateless and stateful applications as teams progress through the adoption of microservices-based applications.

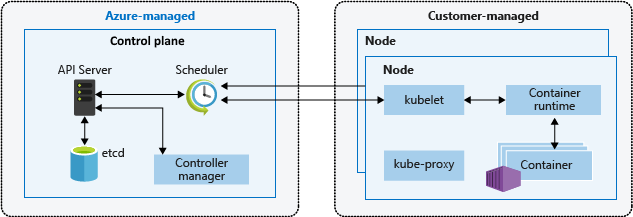
As an open platform, Kubernetes allows you to build your applications with your preferred programming language, OS, libraries, or messaging bus. Existing continuous integration and continuous delivery (CI/CD) tools can integrate with Kubernetes to schedule and deploy releases.

AKS provides a managed Kubernetes service that reduces the complexity of deployment and core management tasks, like upgrade coordination. The Azure platform manages the AKS control plane, and you only pay for the AKS nodes that run your applications.

## Kubernetes cluster architecture

A Kubernetes cluster is divided into two components:

* Control plane: provides the core Kubernetes services and orchestration of application workloads.
* Nodes: run your application workloads.



## Control plane

When you create an AKS cluster, a control plane is automatically created and configured. This control plane is provided at no cost as a managed Azure resource abstracted from the user. You only pay for the nodes attached to the AKS cluster. The control plane and its resources reside only on the region where you created the cluster.

The control plane includes the following core Kubernetes components:

| **Component** | **Description** |
| --- | --- |
| kube-apiserver | The API server is how the underlying Kubernetes APIs are exposed. This component provides the interaction for management tools, such as kubectl or the Kubernetes dashboard. |
| etcd | To maintain the state of your Kubernetes cluster and configuration, the highly available etcd is a key value store within Kubernetes. |
| kube-scheduler | When you create or scale applications, the Scheduler determines what nodes can run the workload and starts them. |
| kube-controller-manager | The Controller Manager oversees a number of smaller Controllers that perform actions such as replicating pods and handling node operations. |

AKS provides a single-tenant control plane, with a dedicated API server, scheduler, etc. You define the number and size of the nodes, and the Azure platform configures the secure communication between the control plane and nodes. Interaction with the control plane occurs through Kubernetes APIs, such as kubectl or the Kubernetes dashboard.

While you don't need to configure components (like a highly available etcd store) with this managed control plane, you can't access the control plane directly. Kubernetes control plane and node upgrades are orchestrated through the Azure CLI or Azure portal. To troubleshoot possible issues, you can review the control plane logs through Azure Monitor logs.

To configure or directly access a control plane, deploy a self-managed Kubernetes cluster using [Cluster API Provider Azure](https://github.com/kubernetes-sigs/cluster-api-provider-azure).

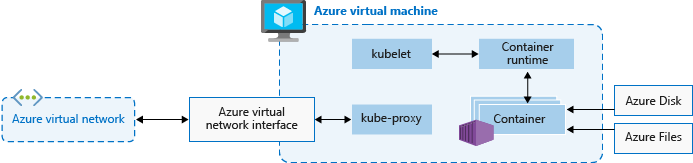
For associated best practices, see [Best practices for cluster security and upgrades in AKS](https://learn.microsoft.com/en-us/azure/aks/operator-best-practices-cluster-security).

For AKS cost management information, see [AKS cost basics](https://learn.microsoft.com/en-us/azure/architecture/aws-professional/eks-to-aks/cost-management#aks-cost-basics) and [Pricing for AKS](https://azure.microsoft.com/pricing/details/kubernetes-service/#pricing).

## Nodes and node pools

To run your applications and supporting services, you need a Kubernetes node. An AKS cluster has at least one node, an Azure virtual machine (VM) that runs the Kubernetes node components and container runtime.

| **Component** | **Description** |
| --- | --- |
| kubelet | The Kubernetes agent that processes the orchestration requests from the control plane along with scheduling and running the requested containers. |
| kube-proxy | Handles virtual networking on each node. The proxy routes network traffic and manages IP addressing for services and pods. |
| container runtime | Allows containerized applications to run and interact with additional resources, such as the virtual network and storage. AKS clusters using Kubernetes version 1.19+ for Linux node pools use containerd as their container runtime. Beginning in Kubernetes version 1.20 for Windows node pools, containerd can be used in preview for the container runtime, but Docker is still the default container runtime. AKS clusters using prior versions of Kubernetes for node pools use Docker as their container runtime. |



The Azure VM size for your nodes defines CPUs, memory, size, and the storage type available (such as high-performance SSD or regular HDD). Plan the node size around whether your applications may require large amounts of CPU and memory or high-performance storage. Scale out the number of nodes in your AKS cluster to meet demand. For more information on scaling, see [Scaling options for applications in AKS](https://learn.microsoft.com/en-us/concepts-scale.md).

In AKS, the VM image for your cluster's nodes is based on Ubuntu Linux, [Mariner Linux](https://learn.microsoft.com/en-us/azure/aks/use-mariner), or Windows Server 2019. When you create an AKS cluster or scale out the number of nodes, the Azure platform automatically creates and configures the requested number of VMs. Agent nodes are billed as standard VMs, so any VM size discounts (including [Azure reservations](https://learn.microsoft.com/en-us/azure/cost-management-billing/reservations/save-compute-costs-reservations)) are automatically applied.

For managed disks, the default disk size and performance will be assigned according to the selected VM SKU and vCPU count. For more information, see [Default OS disk sizing](https://learn.microsoft.com/en-us/azure/aks/cluster-configuration#default-os-disk-sizing).

If you need advanced configuration and control on your Kubernetes node container runtime and OS, you can deploy a self-managed cluster using [Cluster API Provider Azure](https://github.com/kubernetes-sigs/cluster-api-provider-azure).

### Resource reservations

AKS uses node resources to help the node function as part of your cluster. This usage can create a discrepancy between your node's total resources and the allocatable resources in AKS. Remember this information when setting requests and limits for user deployed pods.

To find a node's allocatable resources, run:

kubectl describe node [NODE\_NAME]

To maintain node performance and functionality, AKS reserves resources on each node. As a node grows larger in resources, the resource reservation grows due to a higher need for management of user-deployed pods.

**Note**: Using AKS add-ons such as Container Insights (OMS) will consume additional node resources.

Two types of resources are reserved:

* **CPU**

Reserved CPU is dependent on node type and cluster configuration, which may cause less allocatable CPU due to running additional features.

| **CPU cores on host** | **1** | **2** | **4** | **8** | **16** | **32** | **64** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Kube-reserved (millicores) | 60 | 100 | 140 | 180 | 260 | 420 | 740 |

* **Memory**

Memory utilized by AKS includes the sum of two values.

1. **kubelet daemon**  
   The kubelet daemon is installed on all Kubernetes agent nodes to manage container creation and termination.

By default on AKS, kubelet daemon has the memory.available<750Mi eviction rule, ensuring a node must always have at least 750 Mi allocatable at all times. When a host is below that available memory threshold, the kubelet will trigger to terminate one of the running pods and free up memory on the host machine.

1. **A regressive rate of memory reservations** for the kubelet daemon to properly function (kube-reserved).
   * 25% of the first 4 GB of memory
   * 20% of the next 4 GB of memory (up to 8 GB)
   * 10% of the next 8 GB of memory (up to 16 GB)
   * 6% of the next 112 GB of memory (up to 128 GB)
   * 2% of any memory above 128 GB

**Note**: AKS reserves an additional 2GB for system process in Windows nodes that are not part of the calculated memory.

Memory and CPU allocation rules:

* Keep agent nodes healthy, including some hosting system pods critical to cluster health.
* Cause the node to report less allocatable memory and CPU than it would if it were not part of a Kubernetes cluster.

The above resource reservations can't be changed.

For example, if a node offers 7 GB, it will report 34% of memory not allocatable including the 750Mi hard eviction threshold.

0.75 + (0.25\*4) + (0.20\*3) = 0.75GB + 1GB + 0.6GB = 2.35GB / 7GB = 33.57% reserved

In addition to reservations for Kubernetes itself, the underlying node OS also reserves an amount of CPU and memory resources to maintain OS functions.

For associated best practices, see [Best practices for basic scheduler features in AKS](https://learn.microsoft.com/en-us/azure/aks/operator-best-practices-scheduler).

### Node pools

Nodes of the same configuration are grouped together into node pools. A Kubernetes cluster contains at least one node pool. The initial number of nodes and size are defined when you create an AKS cluster, which creates a default node pool. This default node pool in AKS contains the underlying VMs that run your agent nodes.

**Note**: To ensure your cluster operates reliably, you should run at least two (2) nodes in the default node pool.

You scale or upgrade an AKS cluster against the default node pool. You can choose to scale or upgrade a specific node pool. For upgrade operations, running containers are scheduled on other nodes in the node pool until all the nodes are successfully upgraded.

For more information about how to use multiple node pools in AKS, see [Create and manage multiple node pools for a cluster in AKS](https://learn.microsoft.com/en-us/azure/aks/use-multiple-node-pools).

### Node selectors

In an AKS cluster with multiple node pools, you may need to tell the Kubernetes Scheduler which node pool to use for a given resource. For example, ingress controllers shouldn't run on Windows Server nodes.

Node selectors let you define various parameters, like node OS, to control where a pod should be scheduled.

The following basic example schedules an NGINX instance on a Linux node using the node selector "kubernetes.io/os": linux:

kind: Pod

apiVersion: v1

metadata:

name: nginx

spec:

containers:

- name: myfrontend

image: mcr.microsoft.com/oss/nginx/nginx:1.15.12-alpine

nodeSelector:

"kubernetes.io/os": linux

For more information on how to control where pods are scheduled, see [Best practices for advanced scheduler features in AKS](https://learn.microsoft.com/en-us/azure/aks/operator-best-practices-advanced-scheduler).

## Pods

Kubernetes uses pods to run an instance of your application. A pod represents a single instance of your application.

Pods typically have a 1:1 mapping with a container. In advanced scenarios, a pod may contain multiple containers. Multi-container pods are scheduled together on the same node, and allow containers to share related resources.

When you create a pod, you can define resource requests to request a certain amount of CPU or memory resources. The Kubernetes Scheduler tries to meet the request by scheduling the pods to run on a node with available resources. You can also specify maximum resource limits to prevent a pod from consuming too much compute resource from the underlying node. Best practice is to include resource limits for all pods to help the Kubernetes Scheduler identify necessary, permitted resources.

For more information, see [Kubernetes pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) and [Kubernetes pod lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/).

A pod is a logical resource, but application workloads run on the containers. Pods are typically ephemeral, disposable resources. Individually scheduled pods miss some of the high availability and redundancy Kubernetes features. Instead, pods are deployed and managed by Kubernetes Controllers, such as the Deployment Controller.

## Deployments and YAML manifests

A deployment represents identical pods managed by the Kubernetes Deployment Controller. A deployment defines the number of pod replicas to create. The Kubernetes Scheduler ensures that additional pods are scheduled on healthy nodes if pods or nodes encounter problems.

You can update deployments to change the configuration of pods, container image used, or attached storage. The Deployment Controller:

* Drains and terminates a given number of replicas.
* Creates replicas from the new deployment definition.
* Continues the process until all replicas in the deployment are updated.

Most stateless applications in AKS should use the deployment model rather than scheduling individual pods. Kubernetes can monitor deployment health and status to ensure that the required number of replicas run within the cluster. When scheduled individually, pods aren't restarted if they encounter a problem, and aren't rescheduled on healthy nodes if their current node encounters a problem.

You don't want to disrupt management decisions with an update process if your application requires a minimum number of available instances. Pod Disruption Budgets define how many replicas in a deployment can be taken down during an update or node upgrade. For example, if you have five (5) replicas in your deployment, you can define a pod disruption of 4 (four) to only allow one replica to be deleted or rescheduled at a time. As with pod resource limits, best practice is to define pod disruption budgets on applications that require a minimum number of replicas to always be present.

Deployments are typically created and managed with kubectl create or kubectl apply. Create a deployment by defining a manifest file in the YAML format.

The following example creates a basic deployment of the NGINX web server. The deployment specifies three (3) replicas to be created, and requires port 80 to be open on the container. Resource requests and limits are also defined for CPU and memory.

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: mcr.microsoft.com/oss/nginx/nginx:1.15.2-alpine

ports:

- containerPort: 80

resources:

requests:

cpu: 250m

memory: 64Mi

limits:

cpu: 500m

memory: 256Mi

A breakdown of the deployment specifications in the YAML manifest file is as follows:

| **Specification** | **Description** |
| --- | --- |
| .apiVersion | Specifies the API group and API resource you want to use when creating the resource. |
| .kind | Specifies the type of resource you want to create. |
| .metadata.name | Specifies the name of the deployment. This file will run the nginx image from Docker Hub. |
| .spec.replicas | Specifies how many pods to create. This file will create three deplicated pods. |
| .spec.selector | Specifies which pods will be affected by this deployment. |
| .spec.selector.matchLabels | Contains a map of {key, value} pairs that allows the deployment to find and manage the created pods. |
| .spec.selector.matchLabels.app | Has to match .spec.template.metadata.labels. |
| .spec.template.labels | Specifies the {key, value} pairs attached to the object. |
| .spec.template.app | Has to match .spec.selector.matchLabels. |
| .spec.spec.containers | Specifies the list of containers belonging to the pod. |
| .spec.spec.containers.name | Specifies the name of the container specified as a DNS label. |
| .spec.spec.containers.image | Specifies the container image name. |
| .spec.spec.containers.ports | Specifies the list of ports to expose from the container. |
| .spec.spec.containers.ports.containerPort | Specifies the number of port to expose on the pod's IP address. |
| .spec.spec.resources | Specifies the compute resources required by the container. |
| .spec.spec.resources.requests | Specifies the minimum amount of compute resources required. |
| .spec.spec.resources.requests.cpu | Specifies the minimum amount of CPU required. |
| .spec.spec.resources.requests.memory | Specifies the minimum amount of memory required. |
| .spec.spec.resources.limits | Specifies the maximum amount of compute resources allowed. This limit is enforced by the kubelet. |
| .spec.spec.resources.limits.cpu | Specifies the maximum amount of CPU allowed. This limit is enforced by the kubelet. |
| .spec.spec.resources.limits.memory | Specifies the maximum amount of memory allowed. This limit is enforced by the kubelet. |

More complex applications can be created by including services (such as load balancers) within the YAML manifest.

For more information, see [Kubernetes deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/).

### Package management with Helm

[Helm](https://helm.sh/) is commonly used to manage applications in Kubernetes. You can deploy resources by building and using existing public Helm charts that contain a packaged version of application code and Kubernetes YAML manifests. You can store Helm charts either locally or in a remote repository, such as an [Azure Container Registry Helm chart repo](https://learn.microsoft.com/en-us/azure/container-registry/container-registry-helm-repos).

To use Helm, install the Helm client on your computer, or use the Helm client in the [Azure Cloud Shell](https://shell.azure.com/). Search for or create Helm charts, and then install them to your Kubernetes cluster. For more information, see [Install existing applications with Helm in AKS](https://learn.microsoft.com/en-us/azure/aks/kubernetes-helm).

## StatefulSets and DaemonSets

Using the Kubernetes Scheduler, the Deployment Controller runs replicas on any available node with available resources. While this approach may be sufficient for stateless applications, The Deployment Controller is not ideal for applications that require:

* A persistent naming convention or storage.
* A replica to exist on each select node within a cluster.

Two Kubernetes resources, however, let you manage these types of applications:

* StatefulSets maintain the state of applications beyond an individual pod lifecycle, such as storage.
* DaemonSets ensure a running instance on each node, early in the Kubernetes bootstrap process.

### StatefulSets

Modern application development often aims for stateless applications. For stateful applications, like those that include database components, you can use StatefulSets. Like deployments, a StatefulSet creates and manages at least one identical pod. Replicas in a StatefulSet follow a graceful, sequential approach to deployment, scale, upgrade, and termination. The naming convention, network names, and storage persist as replicas are rescheduled with a StatefulSet.

Define the application in YAML format using kind: StatefulSet. From there, the StatefulSet Controller handles the deployment and management of the required replicas. Data is written to persistent storage, provided by Azure Managed Disks or Azure Files. With StatefulSets, the underlying persistent storage remains, even when the StatefulSet is deleted.

For more information, see [Kubernetes StatefulSets](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/).

Replicas in a StatefulSet are scheduled and run across any available node in an AKS cluster. To ensure at least one pod in your set runs on a node, you use a DaemonSet instead.

### DaemonSets

For specific log collection or monitoring, you may need to run a pod on all, or selected, nodes. You can use DaemonSet deploy on one or more identical pods, but the DaemonSet Controller ensures that each node specified runs an instance of the pod.

The DaemonSet Controller can schedule pods on nodes early in the cluster boot process, before the default Kubernetes scheduler has started. This ability ensures that the pods in a DaemonSet are started before traditional pods in a Deployment or StatefulSet are scheduled.

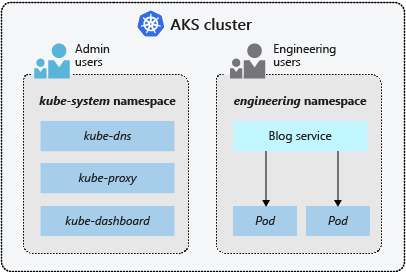
Like StatefulSets, a DaemonSet is defined as part of a YAML definition using kind: DaemonSet.

For more information, see [Kubernetes DaemonSets](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/).

**Note**: If using the [**Virtual Nodes add-on**](https://learn.microsoft.com/en-us/azure/aks/virtual-nodes-cli#enable-virtual-nodes-addon), DaemonSets will not create pods on the virtual node.

## Namespaces

Kubernetes resources, such as pods and deployments, are logically grouped into a namespace to divide an AKS cluster and restrict create, view, or manage access to resources. For example, you can create namespaces to separate business groups. Users can only interact with resources within their assigned namespaces.



When you create an AKS cluster, the following namespaces are available:

| **Namespace** | **Description** |
| --- | --- |
| default | Where pods and deployments are created by default when none is provided. In smaller environments, you can deploy applications directly into the default namespace without creating additional logical separations. When you interact with the Kubernetes API, such as with kubectl get pods, the default namespace is used when none is specified. |
| kube-system | Where core resources exist, such as network features like DNS and proxy, or the Kubernetes dashboard. You typically don't deploy your own applications into this namespace. |
| kube-public | Typically not used, but can be used for resources to be visible across the whole cluster, and can be viewed by any user. |

For more information, see [Kubernetes namespaces](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/).

# Kubernetes Ingress Controller

<https://www.ibm.com/cloud/blog/kubernetes-ingress>

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

# Security: AKS-managed Azure Active Directory integration

<https://learn.microsoft.com/en-us/azure/aks/managed-aad#enable-aks-managed-azure-ad-integration-on-your-existing-cluster>

AKS-managed Azure Active Directory (Azure AD) integration simplifies the Azure AD integration process. Previously, you were required to create a client and server app, and the Azure AD tenant had to grant Directory Read permissions. Now, the AKS resource provider manages the client and server apps for you.

## Azure AD authentication overview

Cluster administrators can configure Kubernetes role-based access control (Kubernetes RBAC) based on a user's identity or directory group membership. Azure AD authentication is provided to AKS clusters with OpenID Connect. OpenID Connect is an identity layer built on top of the OAuth 2.0 protocol. For more information on OpenID Connect, see the [Open ID connect documentation](https://learn.microsoft.com/en-us/azure/active-directory/develop/v2-protocols-oidc).

Learn more about the Azure AD integration flow in the [Azure AD documentation](https://learn.microsoft.com/en-us/azure/aks/concepts-identity#azure-ad-integration).

## Limitations

* AKS-managed Azure AD integration can't be disabled.
* Changing an AKS-managed Azure AD integrated cluster to legacy Azure AD is not supported.
* Clusters without Kubernetes RBAC enabled aren't supported with AKS-managed Azure AD integration.

## Prerequisites

Before getting started, make sure you have the following prerequisites:

* Azure CLI version 2.29.0 or later.
* kubectl, with a minimum version of [1.18.1](https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.18.md#v1181) or [kubelogin](https://github.com/Azure/kubelogin).
* If you're using [helm](https://github.com/helm/helm), you need a minimum version of helm 3.3.

**Important**: You must use kubectl with a minimum version of 1.18.1 or kubelogin. The difference between the minor versions of Kubernetes and kubectl shouldn't be more than 1 version. You'll experience authentication issues if you don't use the correct version.

Use the following commands to install kubectl and kubelogin:

sudo az aks install-cli

kubectl version --client

kubelogin --version

## Before you begin

You need an Azure AD group for your cluster. This group will be registered as an admin group on the cluster to grant cluster admin permissions. You can use an existing Azure AD group or create a new one. Make sure to record the object ID of your Azure AD group.

# List existing groups in the directory

az ad group list --filter "displayname eq '<group-name>'" -o table

Use the following command to create a new Azure AD group for your cluster administrators:

# Create an Azure AD group

az ad group create --display-name myAKSAdminGroup --mail-nickname myAKSAdminGroup

## Create an AKS cluster with Azure AD enabled

1. Create an Azure resource group.

# Create an Azure resource group

az group create --name myResourceGroup --location centralus

1. Create an AKS cluster and enable administration access for your Azure AD group.

# Create an AKS-managed Azure AD cluster

az aks create -g myResourceGroup -n myManagedCluster --enable-aad --aad-admin-group-object-ids <id> [--aad-tenant-id <id>]

A successful creation of an AKS-managed Azure AD cluster has the following section in the response body:

"AADProfile": {

"adminGroupObjectIds": [

"5d24\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*afa27aed"

],

"clientAppId": null,

"managed": true,

"serverAppId": null,

"serverAppSecret": null,

"tenantId": "72f9\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*d011db47"

}

## Access an Azure AD enabled cluster

Before you access the cluster using an Azure AD defined group, you'll need the [Azure Kubernetes Service Cluster User](https://learn.microsoft.com/en-us/azure/role-based-access-control/built-in-roles#azure-kubernetes-service-cluster-user-role) built-in role.

1. Get the user credentials to access the cluster.

az aks get-credentials --resource-group myResourceGroup --name myManagedCluster

1. Follow the instructions to sign in.
2. Use the kubectl get nodes command to view nodes in the cluster.

kubectl get nodes

NAME STATUS ROLES AGE VERSION

aks-nodepool1-15306047-0 Ready agent 102m v1.15.10

aks-nodepool1-15306047-1 Ready agent 102m v1.15.10

aks-nodepool1-15306047-2 Ready agent 102m v1.15.10

1. Configure [Azure role-based access control (Azure RBAC)](https://learn.microsoft.com/en-us/azure/aks/azure-ad-rbac) to configure other security groups for your clusters.

## Troubleshooting access issues with Azure AD

**Important**: The steps described in this section bypass the normal Azure AD group authentication. Use them only in an emergency.

If you're permanently blocked by not having access to a valid Azure AD group with access to your cluster, you can still obtain the admin credentials to access the cluster directly.

To do these steps, you need to have access to the [Azure Kubernetes Service Cluster Admin](https://learn.microsoft.com/en-us/azure/role-based-access-control/built-in-roles#azure-kubernetes-service-cluster-admin-role) built-in role.

az aks get-credentials --resource-group myResourceGroup --name myManagedCluster --admin

## Enable AKS-managed Azure AD integration on your existing cluster

You can enable AKS-managed Azure AD integration on your existing Kubernetes RBAC enabled cluster. Make sure to set your admin group to keep access on your cluster.

az aks update -g MyResourceGroup -n MyManagedCluster --enable-aad --aad-admin-group-object-ids <id-1> [--aad-tenant-id <id>]

A successful activation of an AKS-managed Azure AD cluster has the following section in the response body:

"AADProfile": {

"adminGroupObjectIds": [

"5d24\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*afa27aed"

],

"clientAppId": null,

"managed": true,

"serverAppId": null,

"serverAppSecret": null,

"tenantId": "72f9\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*d011db47"

}

Download user credentials again to access your cluster by following the steps [here](#_Access_an_Azure).

# Security: Use Azure role-based access control for Kubernetes Authorization

<https://learn.microsoft.com/en-us/azure/aks/manage-azure-rbac>

When you leverage [integrated authentication between Azure Active Directory (Azure AD) and AKS](https://learn.microsoft.com/en-us/azure/aks/managed-aad), you can use Azure AD users, groups, or service principals as subjects in [Kubernetes role-based access control (Kubernetes RBAC)](https://learn.microsoft.com/en-us/concepts-identity#azure-rbac-for-kubernetes-authorization). This feature frees you from having to separately manage user identities and credentials for Kubernetes. However, you still have to set up and manage Azure RBAC and Kubernetes RBAC separately.

This article covers how to use Azure RBAC for Kubernetes Authorization, which allows for the unified management and access control across Azure resources, AKS, and Kubernetes resources. For more information, see [Azure RBAC for Kubernetes Authorization](https://learn.microsoft.com/en-us/concepts-identity#azure-rbac-for-kubernetes-authorization).

## Before you begin

* You need the Azure CLI version 2.24.0 or later installed and configured. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).
* You need kubectl, with a minimum version of [1.18.3](https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.18.md#v1183).
* You need managed Azure AD integration enabled on your cluster before you can add Azure RBAC for Kubernetes authorization. If you need to enable managed Azure AD integration, see [Use Azure AD in AKS](#_Security:_AKS-managed_Azure) (*AKS-managed Azure Active Directory integration*).
* If you have CRDs and are making custom role definitions, the only way to cover CRDs today is to use Microsoft.ContainerService/managedClusters/\*/read. For the remaining objects, you can use the specific API groups, such as Microsoft.ContainerService/apps/deployments/read.
* New role assignments can take up to five minutes to propagate and be updated by the authorization server.
* This article requires that the Azure AD tenant configured for authentication is same as the tenant for the subscription that holds your AKS cluster.

## Create a new AKS cluster with managed Azure AD integration and Azure RBAC for Kubernetes Authorization

Create an Azure resource group using the [az group create](https://learn.microsoft.com/en-us/cli/azure/group#az_group_create) command.

az group create --name myResourceGroup --location westus2

Create an AKS cluster with managed Azure AD integration and Azure RBAC for Kubernetes Authorization using the [az aks create](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_create) command.

az aks create -g myResourceGroup -n myManagedCluster --enable-aad --enable-azure-rbac

The output will look similar to the following example output:

"AADProfile": {

"adminGroupObjectIds": null,

"clientAppId": null,

"enableAzureRbac": true,

"managed": true,

"serverAppId": null,

"serverAppSecret": null,

"tenantId": "\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*-\*\*\*\*"

}

## Enable Azure RBAC on an existing AKS cluster

Add Azure RBAC for Kubernetes Authorization into an existing AKS cluster using the [az aks update](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_update) command with the enable-azure-rbac flag.

az aks update -g myResourceGroup -n myAKSCluster --enable-azure-rbac

## Disable Azure RBAC for Kubernetes Authorization from an AKS cluster

Remove Azure RBAC for Kubernetes Authorization from an existing AKS cluster using the [az aks update](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_update) command with the disable-azure-rbac flag.

az aks update -g myResourceGroup -n myAKSCluster --disable-azure-rbac

## Create role assignments for users to access the cluster

AKS provides the following built-in roles:

| **Role** | **Description** |
| --- | --- |
| Azure Kubernetes Service RBAC Reader | Allows read-only access to see most objects in a namespace. It doesn't allow viewing roles or role bindings. This role doesn't allow viewing Secrets, since reading the contents of Secrets enables access to ServiceAccount credentials in the namespace, which would allow API access as any ServiceAccount in the namespace (a form of privilege escalation). |
| Azure Kubernetes Service RBAC Writer | Allows read/write access to most objects in a namespace. This role doesn't allow viewing or modifying roles or role bindings. However, this role allows accessing Secrets and running Pods as any ServiceAccount in the namespace, so it can be used to gain the API access levels of any ServiceAccount in the namespace. |
| Azure Kubernetes Service RBAC Admin | Allows admin access, intended to be granted within a namespace. Allows read/write access to most resources in a namespace (or cluster scope), including the ability to create roles and role bindings within the namespace. This role doesn't allow write access to resource quota or to the namespace itself. |
| Azure Kubernetes Service RBAC Cluster Admin | Allows super-user access to perform any action on any resource. It gives full control over every resource in the cluster and in all namespaces. |

Roles assignments scoped to the **entire AKS cluster** can be done either on the Access Control (IAM) blade of the cluster resource on Azure portal or by using the following Azure CLI commands:

Get your AKS resource ID using the [az aks show](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_show) command.

AKS\_ID=$(az aks show -g myResourceGroup -n myManagedCluster --query id -o tsv)

Create a role assignment using the [az role assignment create](https://learn.microsoft.com/en-us/cli/azure/role/assignment#az_role_assignment_create) command. <AAD-ENTITY-ID> can be a username or the client ID of a service principal.

az role assignment create --role "Azure Kubernetes Service RBAC Admin" --assignee <AAD-ENTITY-ID> --scope $AKS\_ID

**Note**: You can create the Azure Kubernetes Service RBAC Reader and Azure Kubernetes Service RBAC Writer role assignments scoped to a specific namespace within the cluster using the [**az role assignment create**](https://learn.microsoft.com/en-us/cli/azure/role/assignment#az_role_assignment_create) command and setting the scope to the desired namespace.

az role assignment create --role "Azure Kubernetes Service RBAC Reader" --assignee <AAD-ENTITY-ID> --scope $AKS\_ID/namespaces/<namespace-name>

## Create custom roles definitions

The following example custom role definition allows a user to only read deployments and nothing else. For the full list of possible actions, see [Microsoft.ContainerService operations](https://learn.microsoft.com/en-us/azure/role-based-access-control/resource-provider-operations#microsoftcontainerservice).

To create your own custom role definitions, copy the following file, replacing <YOUR SUBSCRIPTION ID> with your own subscription ID, and then save it as deploy-view.json.

{

"Name": "AKS Deployment Reader",

"Description": "Lets you view all deployments in cluster/namespace.",

"Actions": [],

"NotActions": [],

"DataActions": [

"Microsoft.ContainerService/managedClusters/apps/deployments/read"

],

"NotDataActions": [],

"assignableScopes": [

"/subscriptions/<YOUR SUBSCRIPTION ID>"

]

}

Create the role definition using the [az role definition create](https://learn.microsoft.com/en-us/cli/azure/role/definition#az_role_definition_create) command, setting the --role-definition to the deploy-view.json file you created in the previous step.

az role definition create --role-definition @deploy-view.json

Assign the role definition to a user or other identity using the [az role assignment create](https://learn.microsoft.com/en-us/cli/azure/role/assignment#az_role_assignment_create) command.

az role assignment create --role "AKS Deployment Reader" --assignee <AAD-ENTITY-ID> --scope $AKS\_ID

## Use Azure RBAC for Kubernetes Authorization with kubectl

Make sure you have the [Azure Kubernetes Service Cluster User](https://learn.microsoft.com/en-us/azure/role-based-access-control/built-in-roles#azure-kubernetes-service-cluster-user-role) built-in role, and then get the kubeconfig of your AKS cluster using the [az aks get-credentials](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_get-credentials) command.

az aks get-credentials -g myResourceGroup -n myManagedCluster

Now, you can use kubectl manage your cluster. For example, you can list the nodes in your cluster using kubectl get nodes. The first time you run it, you'll need to sign in, as shown in the following example:

kubectl get nodes

To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code AAAAAAAAA to authenticate.

NAME STATUS ROLES AGE VERSION

aks-nodepool1-93451573-vmss000000 Ready agent 3h6m v1.15.11

aks-nodepool1-93451573-vmss000001 Ready agent 3h6m v1.15.11

aks-nodepool1-93451573-vmss000002 Ready agent 3h6m v1.15.11

## Use Azure RBAC for Kubernetes Authorization with kubelogin

AKS created the [kubelogin](https://github.com/Azure/kubelogin) plugin to help unblock additional scenarios, such as non-interactive logins, older kubectl versions, or leveraging SSO across multiple clusters without the need to sign in to a new cluster.

You can use the kubelogin plugin by running the following command:

export KUBECONFIG=/path/to/kubeconfig

kubelogin convert-kubeconfig

Similar to kubectl, you need to log in the first time you run it, as shown in the following example:

kubectl get nodes

To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code AAAAAAAAA to authenticate.

NAME STATUS ROLES AGE VERSION

aks-nodepool1-93451573-vmss000000 Ready agent 3h6m v1.15.11

aks-nodepool1-93451573-vmss000001 Ready agent 3h6m v1.15.11

aks-nodepool1-93451573-vmss000002 Ready agent 3h6m v1.15.11

## Clean up resources

### Delete role assignment

# List role assignments

az role assignment list --scope $AKS\_ID --query [].id -o tsv

# Delete role assignments

az role assignment delete --ids <LIST OF ASSIGNMENT IDS>

### Delete role definition

az role definition delete -n "AKS Deployment Reader"

### Delete resource group and AKS cluster

az group delete -n myResourceGroup

# Security: Use Kubernetes role-based access control with Azure Active Directory in Azure Kubernetes Service

<https://learn.microsoft.com/en-us/azure/aks/azure-ad-rbac?tabs=portal>

Azure Kubernetes Service (AKS) can be configured to use Azure Active Directory (Azure AD) for user authentication. In this configuration, you sign in to an AKS cluster using an Azure AD authentication token. Once authenticated, you can use the built-in Kubernetes role-based access control (Kubernetes RBAC) to manage access to namespaces and cluster resources based on a user's identity or group membership.

This article shows you how to:

* Control access using Kubernetes RBAC in an AKS cluster based on Azure AD group membership.
* Create example groups and users in Azure AD.
* Create Roles and RoleBindings in an AKS cluster to grant the appropriate permissions to create and view resources.

## Before you begin

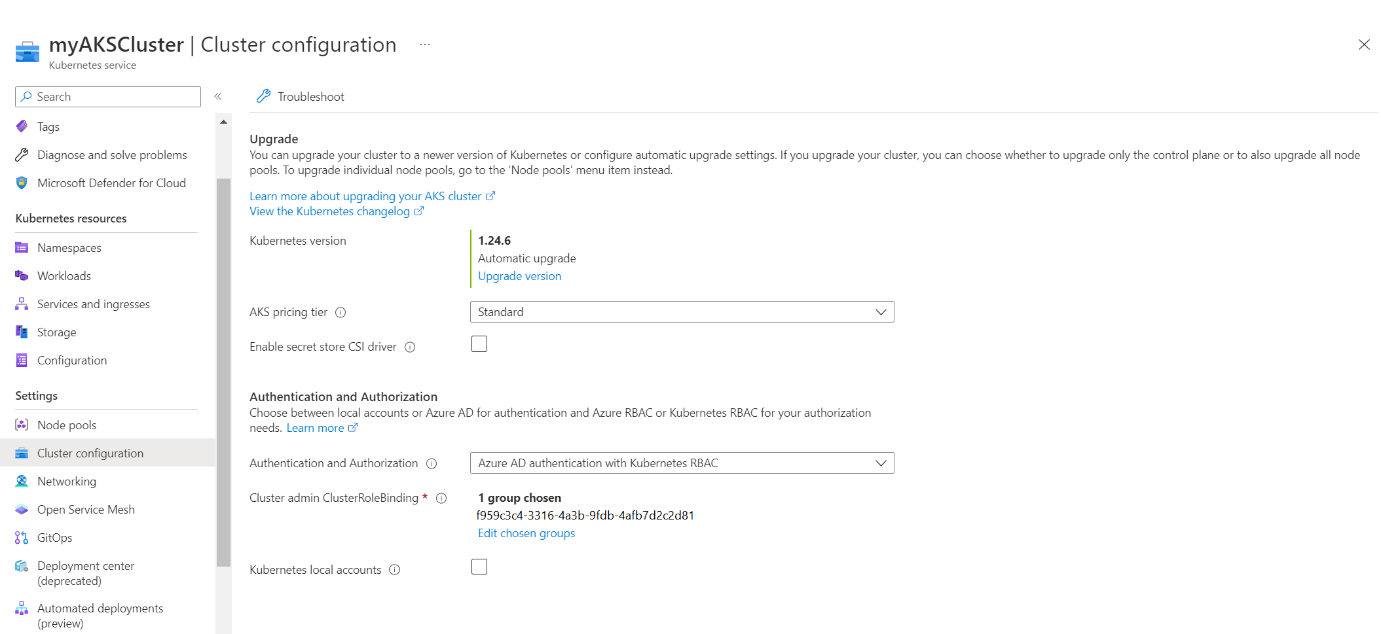
* You have an existing AKS cluster with Azure AD integration enabled. If you need an AKS cluster with this configuration, see [Integrate Azure AD with AKS](https://learn.microsoft.com/en-us/azure/aks/managed-aad).
* Kubernetes RBAC is enabled by default during AKS cluster creation. To upgrade your cluster with Azure AD integration and Kubernetes RBAC, [Enable Azure AD integration on your existing AKS cluster](https://learn.microsoft.com/en-us/azure/aks/managed-aad#enable-aks-managed-azure-ad-integration-on-your-existing-cluster).
* Make sure that Azure CLI version 2.0.61 or later is installed and configured. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).
* If using Terraform, install [Terraform](https://learn.microsoft.com/en-us/azure/developer/terraform/overview) version 2.99.0 or later.

Use the Azure portal or Azure CLI to verify Azure AD integration with Kubernetes RBAC is enabled.

[Azure portal](https://learn.microsoft.com/en-us/azure/aks/azure-ad-rbac?tabs=portal#tabpanel_1_portal)

To verify using the Azure portal:

* From your browser, sign in to the [Azure portal](https://portal.azure.com/).
* Navigate to **Kubernetes services**, and from the left-hand pane select **Cluster configuration**.
* Under the **Authentication and Authorization** section, verify the **Azure AD authentication with Kubernetes RBAC** option is selected.

[](https://learn.microsoft.com/en-us/azure/aks/media/azure-ad-rbac/rbac-portal.png#lightbox)

## Create demo groups in Azure AD

In this article, we'll create two user roles to show how Kubernetes RBAC and Azure AD control access to cluster resources. The following two example roles are used:

* **Application developer**
  + A user named aksdev that's part of the appdev group.
* **Site reliability engineer**
  + A user named akssre that's part of the opssre group.

In production environments, you can use existing users and groups within an Azure AD tenant.

1. First, get the resource ID of your AKS cluster using the [az aks show](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_show) command. Then, assign the resource ID to a variable named AKS\_ID so it can be referenced in other commands.

AKS\_ID=$(az aks show \

--resource-group myResourceGroup \

--name myAKSCluster \

--query id -o tsv)

1. Create the first example group in Azure AD for the application developers using the [az ad group create](https://learn.microsoft.com/en-us/cli/azure/ad/group#az_ad_group_create) command. The following example creates a group named appdev:

APPDEV\_ID=$(az ad group create --display-name appdev --mail-nickname appdev --query Id -o tsv)

1. Create an Azure role assignment for the appdev group using the [az role assignment create](https://learn.microsoft.com/en-us/cli/azure/role/assignment#az_role_assignment_create) command. This assignment lets any member of the group use kubectl to interact with an AKS cluster by granting them the Azure Kubernetes Service Cluster User Role.

az role assignment create \

--assignee $APPDEV\_ID \

--role "Azure Kubernetes Service Cluster User Role" \

--scope $AKS\_ID

**Tip**: If you receive an error such as Principal 35bfec9328bd4d8d9b54dea6dac57b82 doesn't exist in the directory a5443dcd-cd0e-494d-a387-3039b419f0d5., wait a few seconds for the Azure AD group object ID to propagate through the directory then try the az role assignment create command again.

1. Create a second example group for SREs named opssre.

OPSSRE\_ID=$(az ad group create --display-name opssre --mail-nickname opssre --query objectId -o tsv)

1. Create an Azure role assignment to grant members of the group the Azure Kubernetes Service Cluster User Role.

az role assignment create \

--assignee $OPSSRE\_ID \

--role "Azure Kubernetes Service Cluster User Role" \

--scope $AKS\_ID

## Create demo users in Azure AD

Now that we have two example groups created in Azure AD for our application developers and SREs, we'll create two example users. To test the Kubernetes RBAC integration at the end of the article, you'll sign in to the AKS cluster with these accounts.

### Set the user principal name and password for application developers

Set the user principal name (UPN) and password for the application developers. The UPN must include the verified domain name of your tenant, for example aksdev@contoso.com.

The following command prompts you for the UPN and sets it to AAD\_DEV\_UPN so it can be used in a later command:

echo "Please enter the UPN for application developers: " && read AAD\_DEV\_UPN

The following command prompts you for the password and sets it to AAD\_DEV\_PW for use in a later command:

echo "Please enter the secure password for application developers: " && read AAD\_DEV\_PW

### Create the user accounts

1. Create the first user account in Azure AD using the [az ad user create](https://learn.microsoft.com/en-us/cli/azure/ad/user#az_ad_user_create) command. The following example creates a user with the display name AKS Dev and the UPN and secure password using the values in AAD\_DEV\_UPN and AAD\_DEV\_PW:

AKSDEV\_ID=$(az ad user create \

--display-name "AKS Dev" \

--user-principal-name $AAD\_DEV\_UPN \

--password $AAD\_DEV\_PW \

--query objectId -o tsv)

1. Add the user to the appdev group created in the previous section using the [az ad group member add](https://learn.microsoft.com/en-us/cli/azure/ad/group/member#az_ad_group_member_add) command.

az ad group member add --group appdev --member-id $AKSDEV\_ID

1. Set the UPN and password for SREs. The UPN must include the verified domain name of your tenant, for example akssre@contoso.com. The following command prompts you for the UPN and sets it to AAD\_SRE\_UPN for use in a later command:

echo "Please enter the UPN for SREs: " && read AAD\_SRE\_UPN

1. The following command prompts you for the password and sets it to AAD\_SRE\_PW for use in a later command:

echo "Please enter the secure password for SREs: " && read AAD\_SRE\_PW

1. Create a second user account. The following example creates a user with the display name AKS SRE and the UPN and secure password using the values in AAD\_SRE\_UPN and AAD\_SRE\_PW:

# Create a user for the SRE role

AKSSRE\_ID=$(az ad user create \

--display-name "AKS SRE" \

--user-principal-name $AAD\_SRE\_UPN \

--password $AAD\_SRE\_PW \

--query objectId -o tsv)

# Add the user to the opssre Azure AD group

az ad group member add --group opssre --member-id $AKSSRE\_ID

## Create AKS cluster resources for app devs

We have our Azure AD groups, users, and Azure role assignments created. Now, we'll configure the AKS cluster to allow these different groups access to specific resources.

1. Get the cluster admin credentials using the [az aks get-credentials](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_get_credentials) command. In one of the following sections, you get the regular user cluster credentials to see the Azure AD authentication flow in action.

az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --admin

1. Create a namespace in the AKS cluster using the [kubectl create namespace](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#create) command. The following example creates a namespace name dev:

kubectl create namespace dev

**Note**: In Kubernetes, Roles define the permissions to grant, and RoleBindings apply them to desired users or groups. These assignments can be applied to a given namespace, or across the entire cluster. For more information, see [**Using Kubernetes RBAC authorization**](https://learn.microsoft.com/en-us/azure/aks/concepts-identity#kubernetes-rbac).

If the user you grant the Kubernetes RBAC binding for is in the same Azure AD tenant, assign permissions based on the userPrincipalName (UPN). If the user is in a different Azure AD tenant, query for and use the objectId property instead.

1. Create a Role for the dev namespace, which grants full permissions to the namespace. In production environments, you can specify more granular permissions for different users or groups. Create a file named role-dev-namespace.yaml and paste the following YAML manifest:

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: dev-user-full-access

namespace: dev

rules:

- apiGroups: ["", "extensions", "apps"]

resources: ["\*"]

verbs: ["\*"]

- apiGroups: ["batch"]

resources:

- jobs

- cronjobs

verbs: ["\*"]

1. Create the Role using the [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) command and specify the filename of your YAML manifest.

kubectl apply -f role-dev-namespace.yaml

1. Get the resource ID for the appdev group using the [az ad group show](https://learn.microsoft.com/en-us/cli/azure/ad/group#az_ad_group_show) command. This group is set as the subject of a RoleBinding in the next step.

az ad group show --group appdev --query id -o tsv

1. Create a RoleBinding for the appdev group to use the previously created Role for namespace access. Create a file named rolebinding-dev-namespace.yaml and paste the following YAML manifest. On the last line, replace groupObjectId with the group object ID output from the previous command.

kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: dev-user-access

namespace: dev

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: Role

name: dev-user-full-access

subjects:

- kind: Group

namespace: dev

name: groupObjectId

**Tip**: If you want to create the RoleBinding for a single user, specify kind: User and replace groupObjectId with the user principal name (UPN) in the above sample.

1. Create the RoleBinding using the [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) command and specify the filename of your YAML manifest:

kubectl apply -f rolebinding-dev-namespace.yaml

## Create the AKS cluster resources for SREs

Now, we'll repeat the previous steps to create a namespace, Role, and RoleBinding for the SREs.

1. Create a namespace for sre using the [kubectl create namespace](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#create) command.

kubectl create namespace sre

1. Create a file named role-sre-namespace.yaml and paste the following YAML manifest:

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: sre-user-full-access

namespace: sre

rules:

- apiGroups: ["", "extensions", "apps"]

resources: ["\*"]

verbs: ["\*"]

- apiGroups: ["batch"]

resources:

- jobs

- cronjobs

verbs: ["\*"]

1. Create the Role using the [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) command and specify the filename of your YAML manifest.

kubectl apply -f role-sre-namespace.yaml

1. Get the resource ID for the opssre group using the [az ad group show](https://learn.microsoft.com/en-us/cli/azure/ad/group#az_ad_group_show) command.

az ad group show --group opssre --query id -o tsv

1. Create a RoleBinding for the opssre group to use the previously created Role for namespace access. Create a file named rolebinding-sre-namespace.yaml and paste the following YAML manifest. On the last line, replace groupObjectId with the group object ID output from the previous command.

kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: sre-user-access

namespace: sre

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: Role

name: sre-user-full-access

subjects:

- kind: Group

namespace: sre

name: groupObjectId

1. Create the RoleBinding using the [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) command and specify the filename of your YAML manifest.

kubectl apply -f rolebinding-sre-namespace.yaml

## Interact with cluster resources using Azure AD identities

Now, we'll test that the expected permissions work when you create and manage resources in an AKS cluster. In these examples, we'll schedule and view pods in the user's assigned namespace, and try to schedule and view pods outside of the assigned namespace.

1. Reset the kubeconfig context using the [az aks get-credentials](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_get_credentials) command. In a previous section, you set the context using the cluster admin credentials. The admin user bypasses Azure AD sign-in prompts. Without the --admin parameter, the user context is applied that requires all requests to be authenticated using Azure AD.

az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --overwrite-existing

1. Schedule a basic NGINX pod using the [kubectl run](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#run) command in the dev namespace.

kubectl run nginx-dev --image=mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine --namespace dev

1. Enter the credentials for your own appdev@contoso.com account created at the start of the article as the sign-in prompt. Once you're successfully signed in, the account token is cached for future kubectl commands. The NGINX is successfully schedule, as shown in the following example output:

$ kubectl run nginx-dev --image=mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine --namespace dev

To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code B24ZD6FP8 to authenticate.

pod/nginx-dev created

1. Use the [kubectl get pods](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command to view pods in the dev namespace.

kubectl get pods --namespace dev

1. Ensure the status of the NGINX pod is Running. The output will look similar to the following output:

$ kubectl get pods --namespace dev

NAME READY STATUS RESTARTS AGE

nginx-dev 1/1 Running 0 4m

### Create and view cluster resources outside of the assigned namespace

Try to view pods outside of the dev namespace. Use the [kubectl get pods](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command again, this time to see --all-namespaces.

kubectl get pods --all-namespaces

The user's group membership doesn't have a Kubernetes Role that allows this action, as shown in the following example output:

Error from server (Forbidden): pods is forbidden: User "aksdev@contoso.com" cannot list resource "pods" in API group "" at the cluster scope

In the same way, try to schedule a pod in a different namespace, such as the sre namespace. The user's group membership doesn't align with a Kubernetes Role and RoleBinding to grant these permissions, as shown in the following example output:

$ kubectl run nginx-dev --image=mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine --namespace sre

Error from server (Forbidden): pods is forbidden: User "aksdev@contoso.com" cannot create resource "pods" in API group "" in the namespace "sre"

### Test the SRE access to the AKS cluster resources

To confirm that our Azure AD group membership and Kubernetes RBAC work correctly between different users and groups, try the previous commands when signed in as the opssre user.

1. Reset the kubeconfig context using the [az aks get-credentials](https://learn.microsoft.com/en-us/cli/azure/aks#az_aks_get_credentials) command that clears the previously cached authentication token for the aksdev user.

az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --overwrite-existing

1. Try to schedule and view pods in the assigned sre namespace. When prompted, sign in with your own opssre@contoso.com credentials created at the start of the article.

kubectl run nginx-sre --image=mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine --namespace sre

kubectl get pods --namespace sre

As shown in the following example output, you can successfully create and view the pods:

$ kubectl run nginx-sre --image=mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine --namespace sre

To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code BM4RHP3FD to authenticate.

pod/nginx-sre created

$ kubectl get pods --namespace sre

NAME READY STATUS RESTARTS AGE

nginx-sre 1/1 Running 0

1. Try to view or schedule pods outside of assigned SRE namespace.

kubectl get pods --all-namespaces

kubectl run nginx-sre --image=mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine --namespace dev

These kubectl commands fail, as shown in the following example output. The user's group membership and Kubernetes Role and RoleBindings don't grant permissions to create or manager resources in other namespaces.

$ kubectl get pods --all-namespaces

Error from server (Forbidden): pods is forbidden: User "akssre@contoso.com" cannot list pods at the cluster scope

$ kubectl run nginx-sre --image=mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine --namespace dev

Error from server (Forbidden): pods is forbidden: User "akssre@contoso.com" cannot create pods in the namespace "dev"

## Clean up resources

In this article, you created resources in the AKS cluster and users and groups in Azure AD. To clean up all of the resources, run the following commands:

# Get the admin kubeconfig context to delete the necessary cluster resources.

az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --admin

# Delete the dev and sre namespaces. This also deletes the pods, Roles, and RoleBindings.

kubectl delete namespace dev

kubectl delete namespace sre

# Delete the Azure AD user accounts for aksdev and akssre.

az ad user delete --upn-or-object-id $AKSDEV\_ID

az ad user delete --upn-or-object-id $AKSSRE\_ID

# Delete the Azure AD groups for appdev and opssre. This also deletes the Azure role assignments.

az ad group delete --group appdev

az ad group delete --group opssre

# Storage: Kubernetes Storage by Example

<https://codeburst.io/kubernetes-storage-by-example-part-1-27f44ae8fb8b>

## ****Writable Container Layer****

As containers are ephemeral, it is proper to think of containers as immutable. At the same time, we remind ourselves that they themselves do have a writable container layer that is our simplest form of storage.

*When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the “container layer”. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.*

— Docker — [About Storage Drivers](https://docs.docker.com/storage/storagedriver/)

This writable container layer is typically used for temporary files and such; not suitable for writing either persistent or large volumes of data.

## ****Volumes (Temporary)****

*On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers. First, when a Container crashes, kubelet will restart it, but the files will be lost — the Container starts with a clean state. Second, when running Containers together in a Pod it is often necessary to share files between those Containers. The Kubernetes Volume abstraction solves both of these problems.*

— Kubernetes — [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/)

We start our exploration of Kubernetes Volumes with the emptyDir Volume type in much the same way we used writable container layers, i.e., creating temporary files (neither persistent or large volumes of data).

*An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node. As the name says, it is initially empty. Containers in the Pod can all read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each Container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted forever.*

— Kubernetes — [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/)

Here we use an emptyDir Volume to share files between two containers in a Pod.

empty-dir/pod.yaml

|  |
| --- |
| apiVersion: v1 |
| kind: Pod |
| metadata: |
| name: empty-dir |
| spec: |
| containers: |
| - name: busybox-a |
| command: ['tail', '-f', '/dev/null'] |
| image: busybox |
| volumeMounts: |
| - name: cache |
| mountPath: /cache |
| - name: busybox-b |
| command: ['tail', '-f', '/dev/null'] |
| image: busybox |
| volumeMounts: |
| - name: cache |
| mountPath: /cache |
| volumes: |
| - name: cache |
| emptyDir: {} |

Unless otherwise noted, all the examples in this article are provided as Kubernetes configuration files organized into folders; one per section. For example, we can apply this section’s configuration with the following command.

kubectl apply -f empty-dir

**note**: We can use this same command, replacing apply with delete, to clean up after ourselves.

We write a file to the Volume from the busybox-a container:

kubectl exec empty-dir --container busybox-a -- sh -c "echo \"Hello World\" > /cache/hello.txt"

And read from that same file from the busybox-b container:

kubectl exec empty-dir --container busybox-b -- cat /cache/hello.txt

## **Volumes (Persistent)**

Not to be confused with a PersistentVolume (covered in the next section), here we explore connecting Volumes to Pods with persistent data. A representative example of this is an awsElasticBlockStore Volume:

*An awsElasticBlockStore volume mounts an Amazon Web Services (AWS) EBS Volume into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of an EBS volume are preserved and the volume is merely unmounted. This means that an EBS volume can be pre-populated with data, and that data can be “handed off” between Pods.*

— Kubernetes — [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/)

For simplicity, however, in this article we will use a hostPath Volume:

*A hostPath volume mounts a file or directory from the host node’s filesystem into your Pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.*

— Kubernetes — [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/)

In preparation for this example, we need to create a folder named data in the root of our development machine with a file named hello.txt in it. With this in place, we create our configuration:

host-path/pod.yaml

|  |
| --- |
| apiVersion: v1 |
| kind: Pod |
| metadata: |
| name: host-path |
| spec: |
| containers: |
| - name: busybox |
| command: ['tail', '-f', '/dev/null'] |
| image: busybox |
| volumeMounts: |
| - name: data |
| mountPath: /data |
| volumes: |
| - name: data |
| hostPath: |
| path: /data |

After applying this configuration, we can see that we can read the hello.txt file:

kubectl apply -f host-path

kubectl exec host-path -- cat /data/hello.txt

## ****PersistentVolume and PersistentVolumeClaim****

In the previous example, there were three distinct tasks:

* Create the data folder: had to be done using the root user of our development machine; this is an **infrastructure** task.
* Create the Kubernetes Volume: requires knowledge of the development machine (folder location); this is an **infrastructure** task.
* Create the Kubernetes Pod; this is a **development** task.

The problem, however, with the previous example is that the creation of the Kubernetes Volume and Pod are tightly coupled (defined in the Pod configuration) and yet these are different types (infrastructure and development) of tasks that are often managed by different teams of people.

We can decouple this configuration using a Kubernetes PersistentVolume (infrastructure task) and PersistentVolumeClaim (development task).

*A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.*

*A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted once read/write or many times read-only).*

— Kubernetes — [Persistent Volumes](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

**note**: While it is not demonstrated in this article, PersistentVolumeClaims (and their bound PersistentVolumes) exist separately from their connected Pods, i.e., one can disconnect them and reconnect them to other Pods.

In this configuration, the infrastructure team provisions a PersistentVolume with a storageClassName (development) identifier.

persistent-volume/pv.yaml

|  |
| --- |
| apiVersion: v1 |
| kind: PersistentVolume |
| metadata: |
| name: data |
| spec: |
| accessModes: |
| - ReadWriteOnce |
| capacity: |
| storage: 1Gi |
| hostPath: |
| path: /data |
| storageClassName: development |

The development team then creates a PersistentVolumeClaim using the development storageClassName; thus binding it to the provided PersistentVolume. **The development team required no knowledge of how the infrastructure team provisioned the volume;**to the development team it is just a ReadWriteOnce 1Gi volume.

persistent-volume/pvc.yaml

|  |
| --- |
| apiVersion: v1 |
| kind: PersistentVolumeClaim |
| metadata: |
| name: data |
| spec: |
| accessModes: |
| - ReadWriteOnce |
| resources: |
| requests: |
| storage: 1Gi |
| storageClassName: development |

Finally, the development team connects the PersistentVolumeClaim (bound to the PersistentVolume) to the Pod.

persistent-volume/pod.yaml

|  |
| --- |
| apiVersion: v1 |
| kind: Pod |
| metadata: |
| name: persistent-volume |
| spec: |
| containers: |
| - name: busybox |
| command: ['tail', '-f', '/dev/null'] |
| image: busybox |
| volumeMounts: |
| - name: data |
| mountPath: /data |
| volumes: |
| - name: data |
| persistentVolumeClaim: |
| claimName: data |

After applying this configuration, we can see that we can read the hello.txt file as before:

kubectl apply -f persistent-volume

kubectl exec persistent-volume -- cat /data/hello.txt

## ****StorageClass****

Using the previous pattern, the infrastructure team is required to create a PersistentVolume for each PersistentVolumeClaim the development team requires; this is called static provisioning.

Another approach is the infrastructure team can create a Kubernetes StorageClass that dynamically provisions PersistentVolumes on demand.

*A StorageClass provides a way for administrators to describe the “classes” of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent. This concept is sometimes called “profiles” in other storage systems.*

— Kubernetes — [Storage Classes](https://kubernetes.io/docs/concepts/storage/storage-classes/)

Here the infrastructure team creates a StorageClass (identified by the name development-dynamic) that automatically provisions PersistentVolumes using the microk8s.io/hostpath provisioner. This provisioner, specific to microk8s, uses folders and files on the development workstation.

storage-class/sc.yaml

|  |
| --- |
| apiVersion: storage.k8s.io/v1 |
| kind: StorageClass |
| metadata: |
| name: development-dynamic |
| provisioner: microk8s.io/hostpath |

As before, the development team only needs to specify the storageClassName, e.g., development-dynamic, using a PersistentVolumeClaim to bind it to a dynamically provisioned PersistentVolume.

**note**: Observe that, here, the storageClassName resolves to a StorageClass; in the previous example it resolves to a PersistentVolume.

storage-class/pvc.yaml

|  |
| --- |
| apiVersion: v1 |
| kind: PersistentVolumeClaim |
| metadata: |
| name: data-dynamic |
| spec: |
| accessModes: |
| - ReadWriteOnce |
| resources: |
| requests: |
| storage: 1Gi |
| storageClassName: development-dynamic |

Finally, the development team connects the PersistentVolumeClaim (bound to the PersistentVolume) to the Pod.

**storage-class/pod.yaml**

|  |
| --- |
| apiVersion: v1 |
| kind: Pod |
| metadata: |
| name: storage-class |
| spec: |
| containers: |
| - name: busybox |
| command: ['tail', '-f', '/dev/null'] |
| image: busybox |
| volumeMounts: |
| - name: data |
| mountPath: /data |
| volumes: |
| - name: data |
| persistentVolumeClaim: |
| claimName: data-dynamic |

After applying this configuration, we can see that we can first write and then read the hello.txt file as expected:

kubectl apply -f persistent-volume

kubectl exec storage-class -- sh -c "echo \"Hello World\" > /data/hello.txt"

kubectl exec storage-class -- cat /data/hello.txt

# Storage: Container Storage Interface (CSI) drivers on Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/csi-storage-drivers>

The Container Storage Interface (CSI) is a standard for exposing arbitrary block and file storage systems to containerized workloads on Kubernetes. By adopting and using CSI, Azure Kubernetes Service (AKS) can write, deploy, and iterate plug-ins to expose new or improve existing storage systems in Kubernetes without having to touch the core Kubernetes code and wait for its release cycles.

The CSI storage driver support on AKS allows you to natively use:

* [**Azure Disks**](https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi) can be used to create a Kubernetes DataDisk resource. Disks can use Azure Premium Storage, backed by high-performance SSDs, or Azure Standard Storage, backed by regular HDDs or Standard SSDs. For most production and development workloads, use Premium Storage. Azure Disks are mounted as ReadWriteOnce and are only available to one node in AKS. For storage volumes that can be accessed by multiple pods simultaneously, use Azure Files.
* [**Azure Files**](https://learn.microsoft.com/en-us/azure/aks/azure-files-csi) can be used to mount an SMB 3.0/3.1 share backed by an Azure storage account to pods. With Azure Files, you can share data across multiple nodes and pods. Azure Files can use Azure Standard storage backed by regular HDDs or Azure Premium storage backed by high-performance SSDs.
* [**Azure Blob storage**](https://learn.microsoft.com/en-us/azure/aks/azure-blob-csi) can be used to mount Blob storage (or object storage) as a file system into a container or pod. Using Blob storage enables your cluster to support applications that work with large unstructured datasets like log file data, images or documents, HPC, and others. Additionally, if you ingest data into [Azure Data Lake storage](https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction), you can directly mount and use it in AKS without configuring another interim filesystem.

**Important**: Starting with Kubernetes version 1.26, in-tree persistent volume types kubernetes.io/azure-disk and kubernetes.io/azure-file are deprecated and will no longer be supported. Removing these drivers following their deprecation is not planned, however you should migrate to the corresponding CSI drivers disks.csi.azure.com and file.csi.azure.com. To review the migration options for your storage classes and upgrade your cluster to use Azure Disks and Azure Files CSI drivers, see [Migrate from in-tree to CSI drivers][migrate-from-in-tree-to-csi-drivers].

In-tree drivers refers to the storage drivers that are part of the core Kubernetes code opposed to the CSI drivers, which are plug-ins.

**Note**: Azure Disks CSI driver v2 (preview) improves scalability and reduces pod failover latency. It uses shared disks to provision attachment replicas on multiple cluster nodes and integrates with the pod scheduler to ensure a node with an attachment replica is chosen on pod failover. Azure Disks CSI driver v2 (preview) also provides the ability to fine tune performance. If you're interested in participating in the preview, submit a request: [**https://aka.ms/DiskCSIv2Preview**](https://aka.ms/DiskCSIv2Preview). This preview version is provided without a service level agreement, and you can occasionally expect breaking changes while in preview. The preview version isn't recommended for production workloads. For more information, see [**Supplemental Terms of Use for Microsoft Azure Previews**](https://azure.microsoft.com/support/legal/preview-supplemental-terms/).

## Prerequisites

* You need the Azure CLI version 2.42 or later installed and configured. Run az --version to find the version. If you need to install or upgrade, see [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli).
* If the open-source CSI Blob storage driver is installed on your cluster, uninstall it before enabling the Azure Blob storage driver.

## Enable CSI storage drivers on an existing cluster

To enable CSI storage drivers on a new cluster, include one of the following parameters depending on the storage system:

* --enable-disk-driver allows you to enable the [Azure Disks CSI driver](https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi).
* --enable-file-driver allows you to enable the [Azure Files CSI driver](https://learn.microsoft.com/en-us/azure/aks/azure-files-csi).
* --enable-blob-driver allows you to enable the [Azure Blob storage CSI driver](https://learn.microsoft.com/en-us/azure/aks/azure-blob-csi).
* --enable-snapshot-controller allows you to enable the [snapshot controller](https://kubernetes-csi.github.io/docs/snapshot-controller.html).

az aks update -n myAKSCluster -g myResourceGroup --enable-disk-driver --enable-file-driver --enable-blob-driver --enable-snapshot-controller

It may take several minutes to complete this action. Once it's complete, you should see in the output the status of enabling the driver on your cluster. The following example resembles the section indicating the results when enabling the Blob storage CSI driver:

"storageProfile": {

"blobCsiDriver": {

"enabled": true

},

## Disable CSI storage drivers on a new or existing cluster

To disable CSI storage drivers on a new cluster, include one of the following parameters depending on the storage system:

* --disable-disk-driver allows you to disable the [Azure Disks CSI driver](https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi).
* --disable-file-driver allows you to disable the [Azure Files CSI driver](https://learn.microsoft.com/en-us/azure/aks/azure-files-csi).
* --disable-blob-driver allows you to disable the [Azure Blob storage CSI driver](https://learn.microsoft.com/en-us/azure/aks/azure-blob-csi).
* --disable-snapshot-controller allows you to disable the [snapshot controller](https://kubernetes-csi.github.io/docs/snapshot-controller.html).

az aks create -n myAKSCluster -g myResourceGroup --disable-disk-driver --disable-file-driver --disable-blob-driver --disable-snapshot-controller

To disable CSI storage drivers on an existing cluster, use one of the parameters listed earlier depending on the storage system:

az aks update -n myAKSCluster -g myResourceGroup --disable-disk-driver --disable-file-driver --disable-blob-driver --disable-snapshot-controller

# Storage: Use the Azure Disks Container Storage Interface (CSI) driver in Azure Kubernetes Service (AKS)

<https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi>

The Azure Disks Container Storage Interface (CSI) driver is a [CSI specification](https://github.com/container-storage-interface/spec/blob/master/spec.md)-compliant driver used by Azure Kubernetes Service (AKS) to manage the lifecycle of Azure Disks.

The CSI is a standard for exposing arbitrary block and file storage systems to containerized workloads on Kubernetes. By adopting and using CSI, AKS now can write, deploy, and iterate plug-ins to expose new or improve existing storage systems in Kubernetes. Using CSI drivers in AKS avoids having to touch the core Kubernetes code and wait for its release cycles.

To create an AKS cluster with CSI driver support, see [Enable CSI driver on AKS](#_Storage:_Container_Storage). This article describes how to use the Azure Disks CSI driver version 1.

## Azure Disks CSI driver features

In addition to in-tree driver features, Azure Disks CSI driver supports the following features:

* Performance improvements during concurrent disk attach and detach
  + In-tree drivers attach or detach disks in serial, while CSI drivers attach or detach disks in batch. There's significant improvement when there are multiple disks attaching to one node.
* Premium SSD v1 and v2 are supported.
* Zone-redundant storage (ZRS) disk support
  + Premium\_ZRS, StandardSSD\_ZRS disk types are supported. ZRS disk could be scheduled on the zone or non-zone node, without the restriction that disk volume should be co-located in the same zone as a given node. For more information, including which regions are supported, see [Zone-redundant storage for managed disks](https://learn.microsoft.com/en-us/azure/virtual-machines/disks-redundancy).
* [Snapshot](https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi#volume-snapshots)
* [Volume clone](https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi#clone-volumes)
* [Resize disk PV without downtime](https://learn.microsoft.com/en-us/azure/aks/azure-disk-csi#resize-a-persistent-volume-without-downtime)

**Note**: Depending on the VM SKU that's being used, the Azure Disks CSI driver might have a per-node volume limit. For some powerful VMs (for example, 16 cores), the limit is 64 volumes per node. To identify the limit per VM SKU, review the **Max data disks** column for each VM SKU offered. For a list of VM SKUs offered and their corresponding detailed capacity limits, see [**General purpose virtual machine sizes**](https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-general).

## Use CSI persistent volumes with Azure Disks

A [persistent volume](https://learn.microsoft.com/en-us/azure/aks/concepts-storage#persistent-volumes) (PV) represents a piece of storage that's provisioned for use with Kubernetes pods. A PV can be used by one or many pods and can be dynamically or statically provisioned. This article shows you how to dynamically create PVs with Azure disk for use by a single pod in an AKS cluster. For static provisioning, see [Create a static volume with Azure Disks](https://learn.microsoft.com/en-us/azure/aks/azure-csi-disk-storage-provision#statically-provision-a-volume).

For more information on Kubernetes volumes, see [Storage options for applications in AKS](https://learn.microsoft.com/en-us/azure/aks/concepts-storage).

## Dynamically create Azure Disks PVs by using the built-in storage classes

A storage class is used to define how a unit of storage is dynamically created with a persistent volume. For more information on Kubernetes storage classes, see [Kubernetes storage classes](https://kubernetes.io/docs/concepts/storage/storage-classes/).

When you use the Azure Disks CSI driver on AKS, there are two more built-in StorageClasses that use the Azure Disks CSI storage driver. The other CSI storage classes are created with the cluster alongside the in-tree default storage classes.

* managed-csi: Uses Azure Standard SSD locally redundant storage (LRS) to create a managed disk.
* managed-csi-premium: Uses Azure Premium LRS to create a managed disk.

The reclaim policy in both storage classes ensures that the underlying Azure Disks are deleted when the respective PV is deleted. The storage classes also configure the PVs to be expandable. You just need to edit the persistent volume claim (PVC) with the new size.

To use these storage classes, create a [PVC](https://learn.microsoft.com/en-us/azure/aks/concepts-storage#persistent-volume-claims) and respective pod that references and uses them. A PVC is used to automatically provision storage based on a storage class. A PVC can use one of the pre-created storage classes or a user-defined storage class to create an Azure-managed disk for the desired SKU and size. When you create a pod definition, the PVC is specified to request the desired storage.

Create an example pod and respective PVC by running the [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) command:

kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/azuredisk-csi-driver/master/deploy/example/pvc-azuredisk-csi.yaml

kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/azuredisk-csi-driver/master/deploy/example/nginx-pod-azuredisk.yaml

The output of the command resembles the following example:

persistentvolumeclaim/pvc-azuredisk created

pod/nginx-azuredisk created

After the pod is in the running state, run the following command to create a new file called test.txt.

kubectl exec nginx-azuredisk -- touch /mnt/azuredisk/test.txt

To validate the disk is correctly mounted, run the following command and verify you see the test.txt file in the output:

kubectl exec nginx-azuredisk -- ls /mnt/azuredisk

lost+found

outfile

test.txt

# Create a Kubernetes dev space with Azure Dev Spaces (.NET Core and VS Code)

**Pre-reqs**

* Azure subscription
* VS Code
* Azure CLI ver 2.0.43 or higher
* Kubernetes cluster running Kubernetes 1.9.6 or later, in the EastUS, EastUS2, CentralUS, WestUS2, WestEurope, SoutheastAsia, CanadaCentral, or CanadaEast region, with **Http Application Routing** enabled
* **Steps**:
* Create RG

az group create --name MyResourceGroup --location eastus

* Create AKS

az aks create -g MyResourceGroup -n MyAKS --location eastus --kubernetes-version 1.10.9 --enable-addons http\_application\_routing

**Setup Azure DevSpaces**

* Set up DevSpaces on your AKS cluster

az aks use-dev-spaces -g MyResourceGroup -n MyAKS

* Download the Azure Dev Spaces extension for VS Code (<https://marketplace.visualstudio.com/items?itemName=azuredevspaces.azds>)
* Click Install once on the extension's Marketplace page, and again in VS Code
* Download the Azure Dev Spaces CLI from <https://aka.ms/get-azds-windows>

**Build and run the code**

* Download sample code from GitHub: <https://github.com/Azure/dev-spaces>
* Change directory to the webfrontend folder:

cd dev-spaces/samples/dotnetcore/getting-started/webfrontend

* Generate Docker and Helm chart assets:

azds prep --public

* Select the default Azure Dev Space

azds space select

* Enter “1” to select the default Dev Space to continue and then enter “y” to continue
* Build and run your code in AKS. In the terminal window from the webfrontend folder, run this command:

azds up

* Scan the console output for information about the URL that was created by the up command. It will be in the form:

(pending registration) Service 'webfrontend' port 'http' will be available at <url>\r\nService 'webfrontend' port 80 (TCP) is available at http://localhost:<port>

Open this URL in a browser window, and you should see the web app load.

**Update a content file**

1. Locate a file, such as ./Views/Home/Index.cshtml, and make an edit to the HTML. For example, change line 7 that reads <h2>Application uses</h2> to something like: <h2>Hello k8s in Azure!</h2>
2. Save the file. Moments later, in the Terminal window you'll see a message saying a file in the running container was updated.
3. Go to your browser and refresh the page. You should see the web page display the updated HTML.

What happened? Edits to content files, like HTML and CSS, don't require recompilation in a .NET Core web app, so an active azds up command automatically syncs any modified content files into the running container in Azure, so you can see your content edits right away.

**Update a code file**

Updating code files requires a little more work, because a .NET Core app needs to rebuild and produce updated application binaries.

1. In the terminal window, press Ctrl+C (to stop azds up).
2. Open the code file named Controllers/HomeController.cs, and edit the message that the About page will display: ViewData["Message"] = "Your application description page.";
3. Save the file.
4. Run azds up in the terminal window.

This command rebuilds the container image and redeploys the Helm chart. To see your code changes take effect in the running application, go to the About menu in the web app.

## Debug a container in Kubernetes

In this section, you'll use VS Code to directly debug your container running in Azure. You'll also learn how to get a faster edit-run-test loop.

**Initialize debug assets with the VS Code extension**

You first need to configure your code project so VS Code will communicate with the dev space in Azure. The VS Code extension for Azure Dev Spaces provides a helper command to set up debug configuration.

Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command:

Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces

This adds debug configuration for Azure Dev Spaces under the .vscode folder. This command is not to be confused with the azds prep command, which configures the project for deployment.

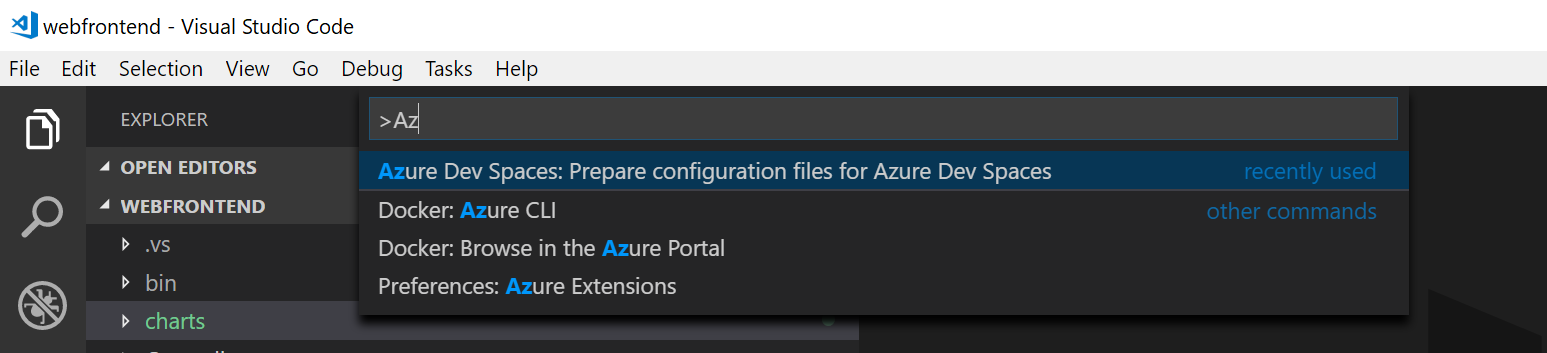
**Initialize debug assets with the VS Code extension**

You first need to configure your code project so VS Code will communicate with the dev space in Azure. The VS Code extension for Azure Dev Spaces provides a helper command to set up debug configuration.

Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command:

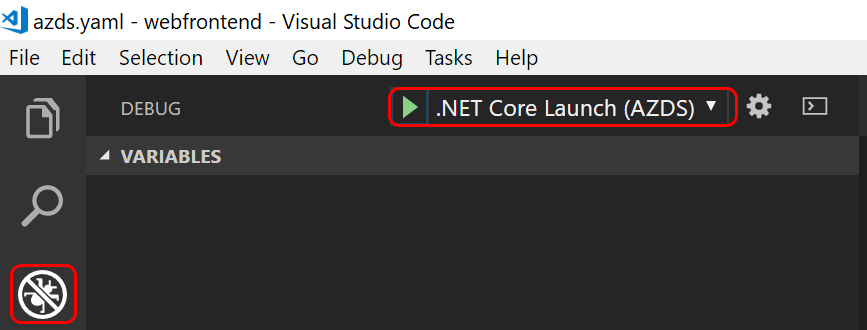
Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces

This adds debug configuration for Azure Dev Spaces under the .vscode folder. This command is not to be confused with the azds prep command, which configures the project for deployment.



**Select the AZDS debug configuration**

1. To open the Debug view, click on the Debug icon in the **Activity Bar** on the side of VS Code.
2. Select **.NET Core Launch (AZDS)** as the active debug configuration.



**Debug the container in Kubernetes**

Hit **F5** to debug your code in Kubernetes.

As with the up command, code is synced to the dev space, and a container is built and deployed to Kubernetes. This time, of course, the debugger is attached to the remote container.

*The VS Code status bar will display a clickable URL.*

Set a breakpoint in a server-side code file, for example within the Index() function in the Controllers/HomeController.cs source file. Refreshing the browser page causes the breakpoint to be hit.

You have full access to debug information just like you would if the code was executing locally, such as the call stack, local variables, exception information, etc.

**Edit code and refresh**

With the debugger active, make a code edit. For example, modify the About page's message in Controllers/HomeController.cs.

public IActionResult About()

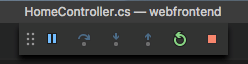
{

ViewData["Message"] = "My custom message in the About page.";

return View();

}

Save the file, and in the **Debug actions pane**, click the **Refresh** button.



Instead of rebuilding and redeploying a new container image each time code edits are made, which will often take considerable time, Azure Dev Spaces will incrementally recompile code within the existing container to provide a faster edit/debug loop.

Refresh the web app in the browser, and go to the About page. You should see your custom message appear in the UI.

# Join a new node to the master

sudo kubeadm token create --print-join-command

kubeadm join 172.20.37.25:6443 --token <token> --discovery-token-ca-cert-hash sha256:a968c697abcae8beafe51738ede08f2888003e01e1aa5cca1a7eb8b69ef79767

# Expose Deployment as a Service

## Example 1: Direct Run

**kubectl run hello-world --replicas=5 --labels="run=load-balancer-example" --image=gcr.io/google-samples/node-hello:1.0 --port=8080**

**kubectl get deployments hello-world**

**kubectl describe deployments hello-world**

**kubectl get replicasets**

**kubectl describe replicasets**

**kubectl expose deployment hello-world --type=NodePort --name=my-service**

**kubectl get services my-service**

**kubectl describe services my-service**

Note the NodePort value

**kubectl cluster-info**

Note the ip address of the k8s cluster

**curl http://<ip>:<port>**

OR in browser:

**http://< ip>:<port>**

## Example 2: With a YAML file

nginx-deployment.yaml

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: nginx-deployment**

**labels:**

**app: nginx**

**spec:**

**replicas: 3**

**selector:**

**matchLabels:**

**app: nginx**

**template:**

**metadata:**

**labels:**

**app: nginx**

**spec:**

**containers:**

**- name: nginx**

**image: nginx:1.7.9**

**ports:**

**- containerPort: 80**

**kubectl apply -f nginx-deployment.yaml**

OR

**kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml**

**kubectl get deployments**

**kubectl get rs**

**kubectl expose deployment nginx-deployment --type=NodePort --name=nginx-service**

**kubectl get services nginx-service**

**kubectl describe services nginx-service**

Note the NodePort value

**kubectl cluster-info**

Note the ip address of the k8s cluster

**curl http://<ip>:<port>**

OR in browser:

**http://< ip>:<port>**