

Course: Introduction to Kubernetes

L1:Introduction

In this course, we will explore what kubernetes is; its architecture and building blocks, how it can be run on our local system or in the cloud, different ways we can configure and protect sensitive information, and how we can let external applications access our kubernetes application. We will also learn how to deploy and manage applications and resources with kubernetes.

[Here is a guide](#) on how to create a GCP/Azure account

Before diving into Kubernetes, we need to know some fundamental services that make working with Kubernetes easy and understandable.

Containers

Containers are an application-centric method used to deliver high-performing, scalable applications on any infrastructure of your choice. Containers provide a portable, isolated way of deploying microservices without disturbance from other microservices in our application. These containers install all the other dependencies needed by the microservice to function in a virtual environment. Containers are responsible for running container images, they do not run the microservices, rather, they couple the microservices and their dependencies.

Container Images

An image bundles the application with its runtime, libraries, and other dependencies. This serves as an isolated environment that runs the application. The runtime is the programming language that the application is built with or runs on.

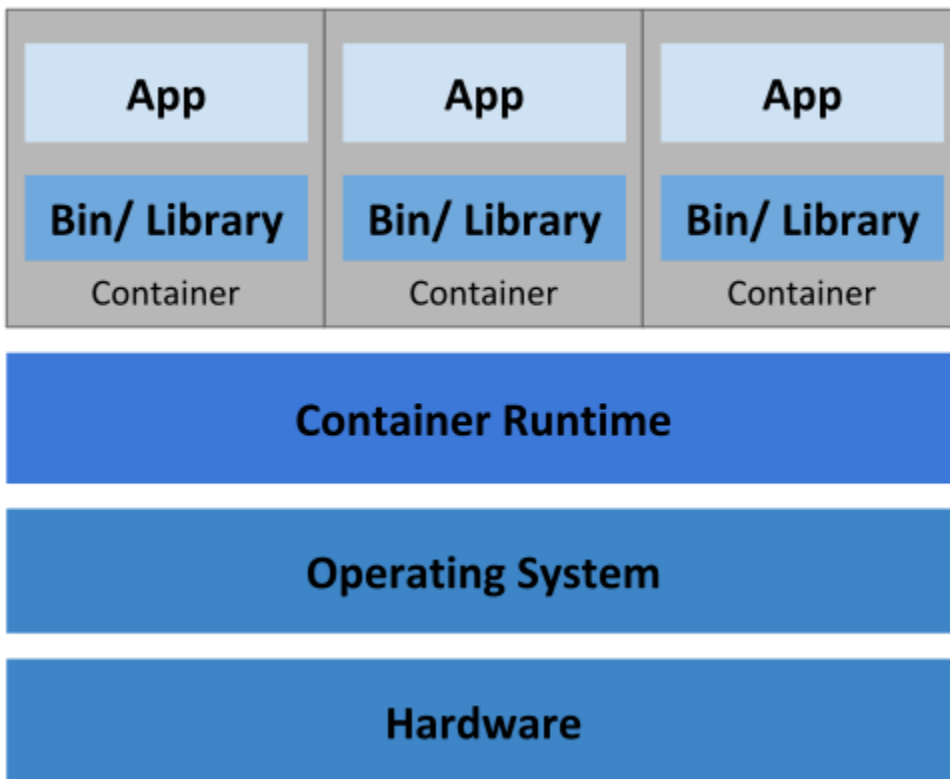
A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment.

You typically create a container image of your application and push it to a registry before referring to it in a pod

Microservices

Microservices are simple, portable, and light applications written in major programming languages such as Rust, Go, Python, etc. Microservices have specific dependencies, libraries, and environmental requirements. Microservices must be deployed with their dependencies before they can successfully run.

In practice, the most important characteristics of a running microservice architecture are ephemerality and scalability. By ephemerality, we mean that individual instances of microservices (containers) will come and go over time. This can be for various reasons (scaling, updates, node failure etc), but the end effect is the same — your containers will die. Scalability (more accurately horizontal scalability) means that if you need to service more traffic, your microservice can easily be scaled up simply by adding more instances.



Container Orchestrators

Container orchestrators are tools which group systems together to form clusters where containers' deployment and management is automated at scale, while meeting the requirements that follow:

1. Fault-tolerance
2. On-demand scalability
3. Optimal resource usage
4. Accessibility from the outside world
5. Seamless updates/rollbacks without any downtime
6. Auto-discovery to automatically discover and communicate with each other.

Importance of container orchestrators

- Group hosts together while creating a cluster
- Schedule containers to run on hosts in the cluster based on resources availability
- Enable containers in a cluster to communicate with each other regardless of the host they are deployed to in the cluster
- Bind containers and storage resources
- Group sets of similar containers and bind them to load-balancing constructs and simplify access to containerized applications by creating a level of abstraction between the containers and the user
- Manage and optimize resource usage
- Allow for implementation of policies to secure access to applications running inside containers.

Deploying Container Orchestrators

Container orchestrators can be deployed in any infrastructure of our choice. We can deploy containers on bare metal, virtual machines, on-premise, on public and hybrid cloud.

They can be installed on top of cloud with resources such as Google compute engine, AWS EC2, Docker Enterprise, IBM Cloud etc. as infrastructure-as-a-service (IAAS).

Also, some cloud services provide managed container orchestration-as-a-service solution, these are container orchestration that are managed and hosted by these cloud providers such as [Amazon Elastic Kubernetes Service](#) (Amazon EKS), [Azure Kubernetes Service](#) (AKS), [DigitalOcean Kubernetes](#), [Google Kubernetes Engine](#) (GKE), [IBM Cloud Kubernetes Service](#), [Oracle Container Engine for Kubernetes](#)

Introduction to Kubernetes Quiz

Question 1

Which is not an importance of container orchestrators?

- Docker**
- Schedule containers to run on hosts in the cluster based on resources availability
- Enable containers in a cluster to communicate with each other regardless of the host they are deployed to in the cluster
- Bind containers and volume resource

Question 2

Container orchestrators can be deployed on bare metal, virtual machines, on-premise, and on-cloud infrastructures

- True**
- False

Question 3

Tools which group systems together to form clusters where containers' deployment and management is automated at scale are called

- a. DEployment
- b. Container
- c. Images
- d. **Container orchestrators**

Question 4

What does a container image encapsulate?

- a. Represents a binary data that encapsulates an application and all its OS dependencies.
- b. Represents a binary data structure that encapsulates an application and all its binary dependencies
- c. **Represents a binary data that encapsulates an application and all its Software dependencies.**
- d. None of the above.

Question 5

Simple, portable, and light applications written in major programming languages that also have specific dependencies, libraries, and environmental requirements are

- a. **Microservices**
- b. Containers
- c. Docker
- d. Images

L2: Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes can be referred to as k8s(pronounced Kate's), 8 refers to the 8 characters between k and s.

Kubernetes is an open source project started by Google and written in the Go programming language. The Kubernetes project was donated to the Cloud Native Computing foundation by Google, and new versions are released every 3 months.

Features of Kubernetes

Kubernetes offer a wide variety of features for orchestrating containers including:

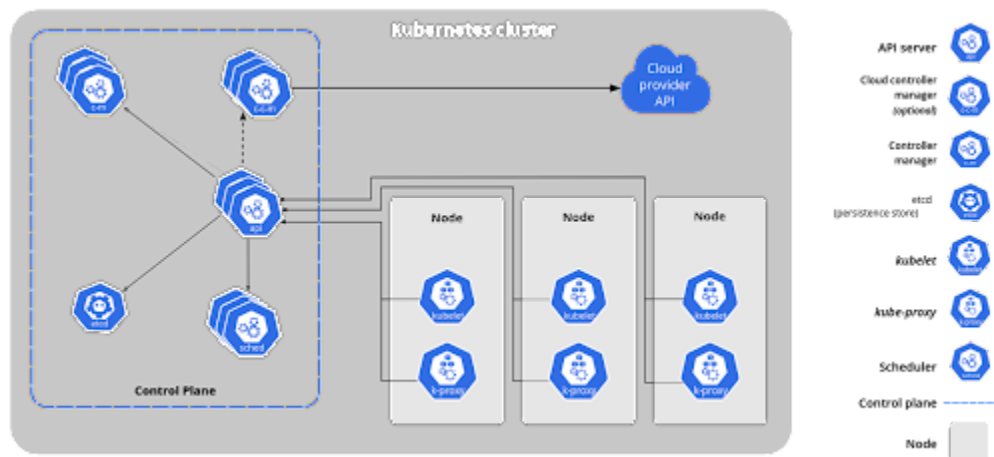
1. Automatic bin packing: Containers are scheduled based on constraints and needed resources, this helps in maximizing utilization without sacrificing availability.
2. Self Healing: Kubernetes ensures a new pod is created to replace an unhealthy or a dead pod.

3. Horizontal scaling: Applications are scaled manually or automatically based on CPU or custom metrics utilization.
4. Service discovery and load balancing: Containers receive their own IP addresses from Kubernetes, while it assigns a single Domain Name System (DNS) name to a set of containers to aid in load-balancing requests across the containers of the set.
5. Automated rollouts and rollbacks: Seamless roll out and roll back application updates and configuration changes, constantly monitoring application health to prevent downtimes.
6. Storage orchestration: Mounts software-defined storage solutions to containers from local storage, external cloud providers, distributed storage, and network storage systems.
7. Batch execution: Kubernetes supports batch execution, long-running jobs, and replacement for failed containers.

Architecture

Kubernetes contains the following components at a high level

1. One or more master nodes. This is part of the control plane.
2. One or more worker nodes



Master node: The master node is responsible for providing a running environment for the control plane. This control plane manages the affairs of a Kubernetes cluster. The control plane also contains other agents with separate responsibilities, and they help with cluster management too.

Communication between users and the k8s cluster is done through the following means; the application programming interface (API), command-line interface (CLI), and Web user interface (Web UI). All these act as a channel for the user to send the requests to the control plane managing the k8s clusters.

The control plane should be running at all times because it is an important service. Also, a disruption might result in downtime which may affect businesses as a result of data loss.

Etcd: This is a distributed key-value store which is responsible for holding the k8s cluster state, It can be configured on the master node or on its dedicated host, separating it from other agents in the control plane. This is done to prevent data loss.

Components in the Master Node

API server

Coordinates all administrative tasks. It intercepts calls from the users, operators, and external agents. It reads the Kubernetes cluster current state from the etcd datastore during processing and updates the resulting state of the k8s cluster in the etcd datastore after a call execution. This is the only component in the master node that can communicate to the etcd datastore. It acts as an intermediary for other components in the control plane.

The API server can be configured and customized, it also supports horizontal and additional custom secondary API servers. It is a configuration that transforms the primary API Server into a proxy, to all secondary custom API Servers, and routes all incoming RESTful calls to them based on custom-defined rules.

Scheduler

The scheduler is responsible for assigning workloads in the cluster. It does this by looking at the current state of the Kubernetes cluster and the requirements of the new object. The scheduler, through the API server, receives the cluster state data and new object requirement data, the scheduler uses this new information to find the node that fits the requirements of the new workload. Once this is done, the API server is notified, which then notifies other control plane agents. The scheduler is also configurable and customized using scheduling policies, plugins, and profiles.

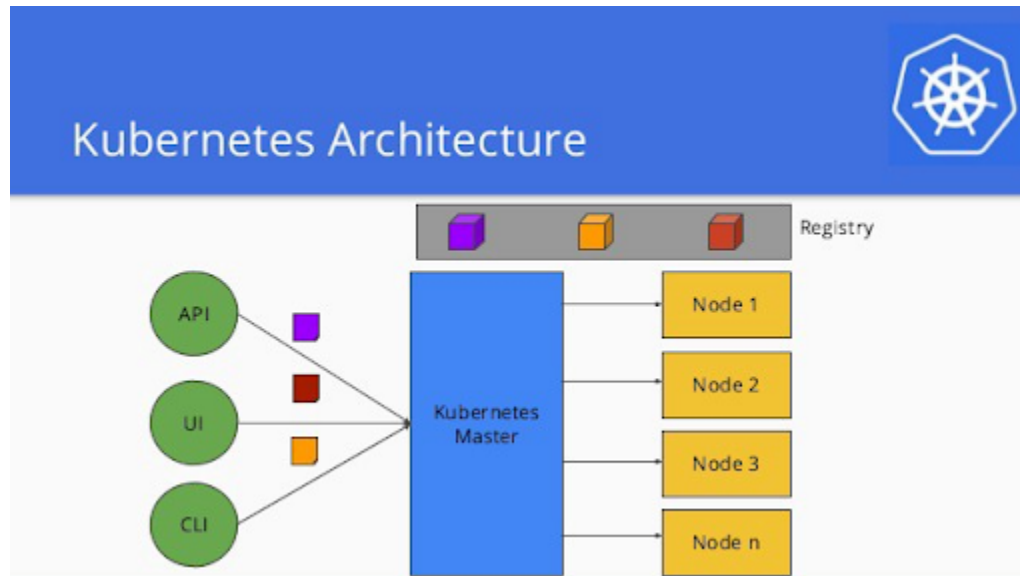
Controller Manager

This controls the state of the Kubernetes cluster. Controllers continuously run, monitor, and compare the cluster's desired state with its current state, from the etcd datastore, via the API server. If there are any differences, they are corrected until they match the desired state.

Worker Node

A worker node provides a running environment for clients application. Worker nodes are controlled by the control plane agents running on the master nodes. Services such as pods run on the master node. A pod is the smallest scheduling unit in Kubernetes. It is the logical collection of one or more containers scheduled together. The collection can be started, stopped, or rescheduled together as a single unit of work. Pods are scheduled on worker nodes, where they find required compute, memory, and storage resources to run, and networking to talk to each other and the outside world. The worker node has the following runtime:

1. Container runtime
2. Node Agent - Kubelet
3. Proxy - Kube-proxy
4. Addons for DNS, Dashboard user interface, cluster-level monitoring and logging.



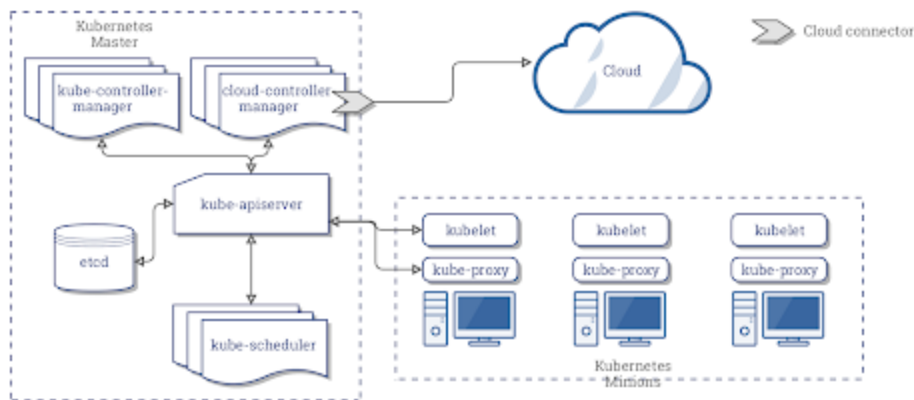
Container Runtime

Kubernetes requires a container runtime to handle the management of a container runtime. The container runtime is located on the node where a pod and its containers are to be scheduled. Kubernetes supports many runtimes including docker, containerd, cri-o etc.

Kubelet

The kubelet is the primary "node agent" that runs on each node. It is a node agent that is located on each worker node in a cluster and has the capability to communicate with the control plane from the master node. It communicates with the API server to receive pod definition and works with the container runtime in a node to run the containers linked to a pod, it also monitors the health and resources of the pods.

It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider.



Kube proxy

This is a network agent that runs on each node and is responsible for dynamically updating the networking rules on the node as well as maintaining the networking rules. It forwards connection requests to the Pods.

DNS

DNS server that assigns DNS records to Kubernetes objects and resources.

Dashboard

A web-based user interface for managing Kubernetes clusters

Monitoring

Responsible for collecting cluster level container metrics and saving them to a central data store

Logging

Collects cluster level container logs and stores them in a central log for later analysis

Kubernetes Quiz

Question 1

The following are features of kubernetes except

- Horizontal scaling**
- Automatic rollout and rollbacks**
- Storage orchestrator**
- Deployment**

Question 2

The kubernetes worker node contains the following runtime except

- a. Node agents
- b. Kubelet
- c. **Container**
- d. Container runtime

Question 3

This is a distributed key-value store which is responsible for holding the k8s cluster state

- a. **Kubelet**
- b. Pods
- c. **Etc**
- d. Volume

Question 4

What is etcd?

- a. **Kubernetes uses etcd as a distributed key-value store for all of its data, including metadata and configuration data, and allows nodes in Kubernetes clusters to read and write data.**
- b. Etcd is a distributed key-value pair for holding the kubernetes cluster stateful metadata.
- c. The etcd is responsible for distributing the key-value pairs across the kubernetes cluster.
- d. All of the above

Question 5

We can collect cluster level container logs and store them in a central log for later analysis with

- a. **Logging**
- b. Monitoring
- c. Trace
- d. Error reporting

L3: Installing Kubernetes and MiniKubes

In this lesson we will learn about the various configurations of Kubernetes, cluster configuration options, infrastructure, tools, and their requirements for running a kubernetes cluster deployment.

Also, we will learn about the requirements for installing minikube, ways to install minikube on different OS, how we can get a kubernetes cluster running on minikube, and how to access the kubernetes dashboard.

R1:Installing Kubernetes

Objective

- Define Kubernetes cluster configuration
- Understand infrastructure where kubernetes is installed

Configuration

There are many different cluster configurations that can be used when installing kubernetes. Some of the configurations are described below:

1. All in One single node installation: Both master and worker nodes components are installed and running on a single node. This is used during learning and testing, it is not advisable for production.
2. Single master and multi worker: Single master node managing multiple worker nodes. The single master node runs a multi-stacked etcd instance.
3. Single master with single node etcd and multi worker nodes: We have a single master node with an external etcd instance. This master node manages the multiple worker nodes.
4. Multi master and multi worker: This configuration is mainly used for high availability. Multi master nodes are configured with stacked etcd instances installed in each master node.

It is recommended to install kubernetes on a multi host environment, with support for high availability control plane setups, and multiple worker nodes for client workload.

Infrastructure

We need to determine the infrastructure for installing kubernetes cluster. This is determined by the environment type e.g learning or production. We need to decide the following:

1. How we should setup kubernetes: On bare metal, public cloud, private or hybrid
2. OS type: Linux or Windows
3. Network solution

Localhost Installation

- Minikube - single-node local kubernetes cluster, recommended for a learning environment deployed on a single host.
- Kind - multi-node kubernetes cluster deployed in docker containers, acting as kubernetes nodes, recommended for a learning environment.
- Docker Desktop - including a local kubernetes cluster for docker users.
- MicroK8s - local and cloud kubernetes cluster, from canonical.
- K3S - lightweight kubernetes cluster for local, cloud, edge, IoT deployments, from Rancher.

On Premise Installation

For on premises, kubernetes can be installed on VMs or bare metal

On-premise VMs: K8s can be installed on VMs created via Vagrant, VMware vSphere, KVM, or a configuration tool in conjunction with a hypervisor.

Bare Metal: K8s can be installed on premise bare metal hosted on top different os.

Cloud Installation

Kubernetes can be installed on almost any cloud production environment. A few of them are listed below.

Hosted Solutions With Hosted Solutions, any given software is completely managed by the provider, while the user pays hosting and management charges. Popular vendors providing hosted solutions for Kubernetes are (listed in alphabetical order):

[Alibaba Cloud Container Service for Kubernetes \(ACK\)](#)

[Amazon Elastic Kubernetes Service \(EKS\)](#)

[Azure Kubernetes Service \(AKS\)](#)

[DigitalOcean Kubernetes](#)

[Google Kubernetes Engine \(GKE\)](#)

[IBM Cloud Kubernetes Service](#)

[Oracle Cloud Container Engine for Kubernetes \(OKE\)](#)

Turnkey Cloud Solutions

Below are only a few of the [Turnkey Cloud Solutions](#) (listed in alphabetical order), to install kubernetes on an underlying IaaS platform such as:

[Alibaba Cloud](#)

[Amazon AWS \(AWS EC2\)](#)

[Google Compute Engine \(GCE\)](#)

[IBM Cloud Private](#)

[Microsoft Azure \(AKS\).](#)

Turnkey On-Premise Solutions The On-Premise solutions install kubernetes on secure internal private clouds:

[GKE On-Prem](#) part of Google Cloud [Anthos](#)

[IBM Private Cloud](#)

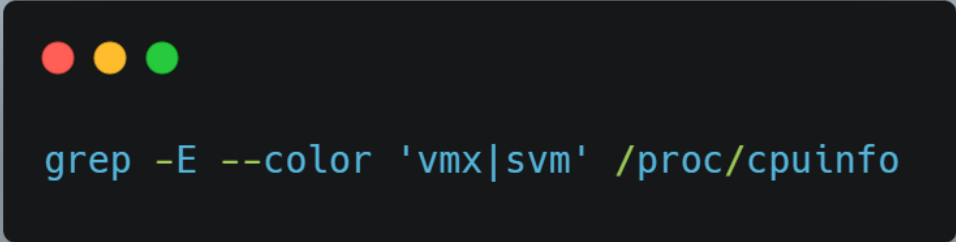
R2:Installing Minikubes

Requirements

1. 2 CPUs or more
2. 2GB free memory
3. 20GB free disk
4. Connection to the Internet
5. A type-2 hypervisor like Docker, Virtual Box, VMWare, Hyper-V, KVM etc.

Linux

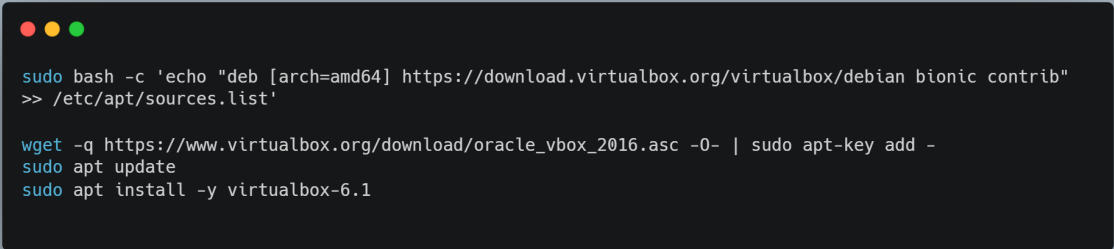
Verify virtualization for linux by running the below command on your terminal



```
grep -E --color 'vmx|svm' /proc/cpuinfo
```

A non empty output indicates support for virtualization

Install VirtualBox hypervisor for linux




```
sudo bash -c 'echo "deb [arch=amd64] https://download.virtualbox.org/virtualbox/debian bionic contrib"
>> /etc/apt/sources.list'

wget -q https://www.virtualbox.org/download/oracle_vbox_2016.asc -O- | sudo apt-key add -
sudo apt update
sudo apt install -y virtualbox-6.1
```

Minikube Installation


Binary download



```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64  
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

or

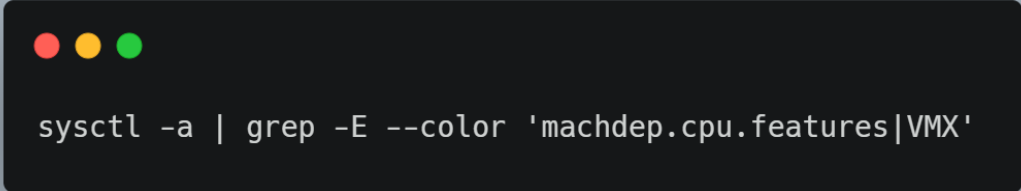
Debian Package



```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb  
sudo dpkg -i minikube_latest_amd64.deb
```

Mac OS

1. Verify virtualization for MacOS. VMX in the output indicates enabled virtualization



```
sysctl -a | grep -E --color 'machdep.cpu.features|VMX'
```

2. Install the [VirtualBox](#) hypervisor for MacOS

3. Download and install the .dmg package.

Minikube Installation

Install using brew package manager



```
brew install minikube
```

Or

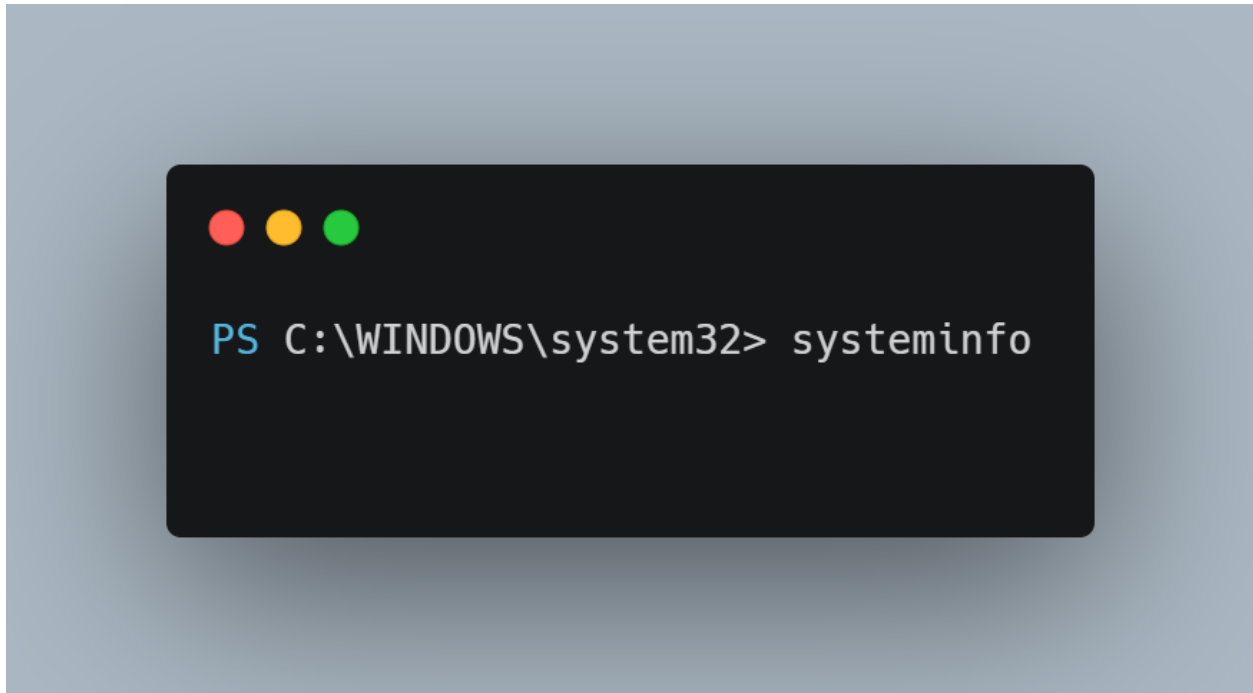
Binary Download



```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64  
sudo install minikube-darwin-amd64 /usr/local/bin/minikube
```

Windows Installation

1. Verify the virtualization support on your Windows system (multiple output lines ending with 'Yes' indicate supported virtualization)



2. Install the VirtualBox hypervisor for 'Windows hosts'

Download and install the .exe package.

NOTE: You may need to disable Hyper-V on your Windows host (if previously installed and used) while running VirtualBox.

Minikube Installation

We can download the latest release or a specific release from the [Minikube release page](#). Once downloaded, we need to make sure it is added to our PATH.

There are two .exe packages available to download for Windows, found under a Minikube release:

1. minikube-windows-amd64.exe which requires to be added to the PATH: manually
2. minikube-installer.exe which automatically adds the executable to the PATH.

Let's download and install the latest [minikube-installer.exe](#) package.

Test minikube by starting it

Minikube can be started with the minikube start command.

Minikube start

Check the status of minikube with

Minikube status

We can stop minikube with

Minikube stop

Accessing Kubernetes

Kubernetes cluster can be accessed using the following methods

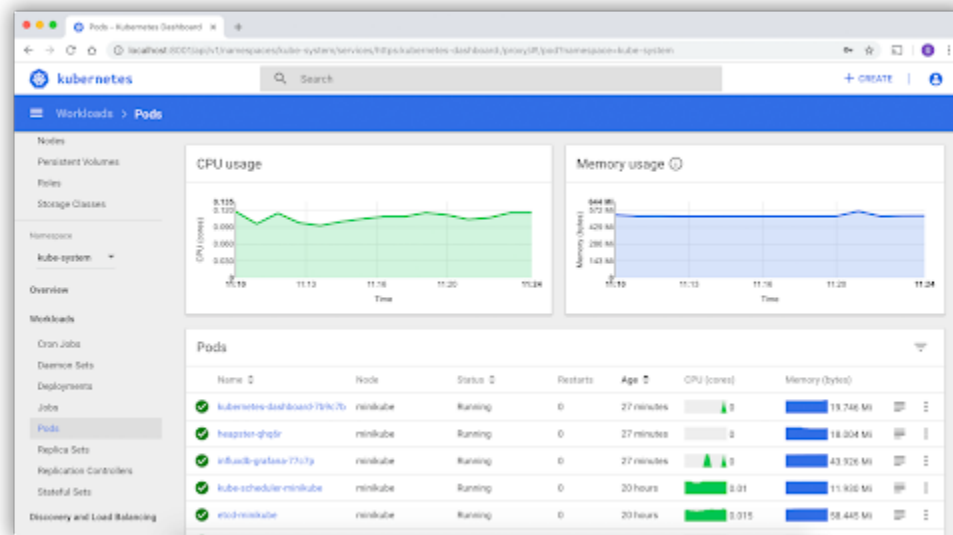
1. Command Line Interface (CLI)
2. Web-based User Interface.
3. APIs from CLI or programmatically.

CLI

Kubectl is the only kubernetes command line interface client that is used to manage cluster resources and applications. It is easily integrated into other systems, can be used in a script or for automation. Kubectl has to be configured with the credentials of the kubernetes cluster before it can be used.

Web-based UI

Kubernetes has a web based UI that is used to manage, monitor, and containerize applications and resources in a cluster.



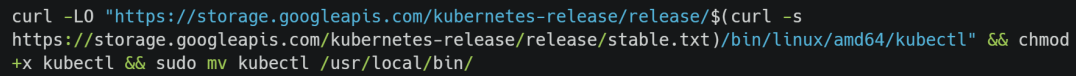
API server

This is the main component of the kubernetes control plane, its responsibility is exposing the kubernetes APIs. These APIs allow users to interact with clusters directly. The CLI tools and Dashboard UI can be used to access the API server and perform various operations on the nodes.

Installing kubectl

Linux Installation

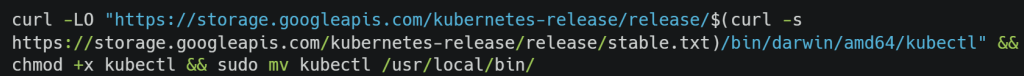
Download latest stable kubectl binary, make it executable and move it to the PATH:



```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl" && chmod
+x kubectl && sudo mv kubectl /usr/local/bin/
```

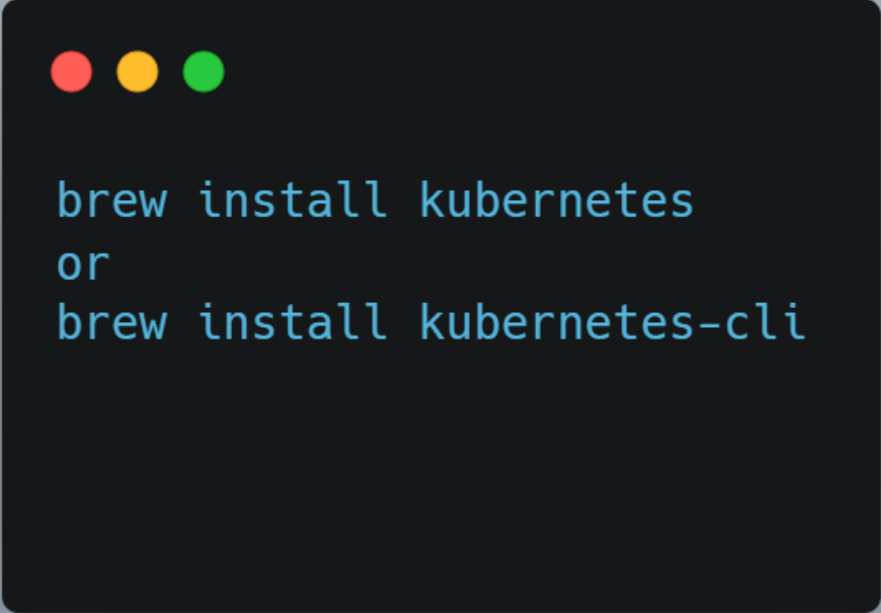
Mac OS installation

Download latest stable kubectl binary, make it executable and move it to the PATH:



```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl" &&
chmod +x kubectl && sudo mv kubectl /usr/local/bin/
```

To install kubectl with [Homebrew package manager](#), issue the following command:



```
brew install kubernetes  
or  
brew install kubernetes-cli
```

Kube config file

In order for kubectl to access the kubernetes cluster, it needs the master node endpoint and other credentials to communicate with the API server running on the master node. To do this, we need a config file stored inside the .kube directory. This is done automatically once minikube is started, but might need to be enabled in other tools. There can also be multiple kubeconfig files connected to a single kubectl client. The kubeconfig detail can be found in either the

```
~/ .kube/config or by using kubectl config view
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/student/.minikube/ca.crt
    server: https://192.168.99.100:8443
    name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
    name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/student/.minikube/profiles/minikube/client.crt
    client-key: /home/student/.minikube/profiles/minikube/client.key
```

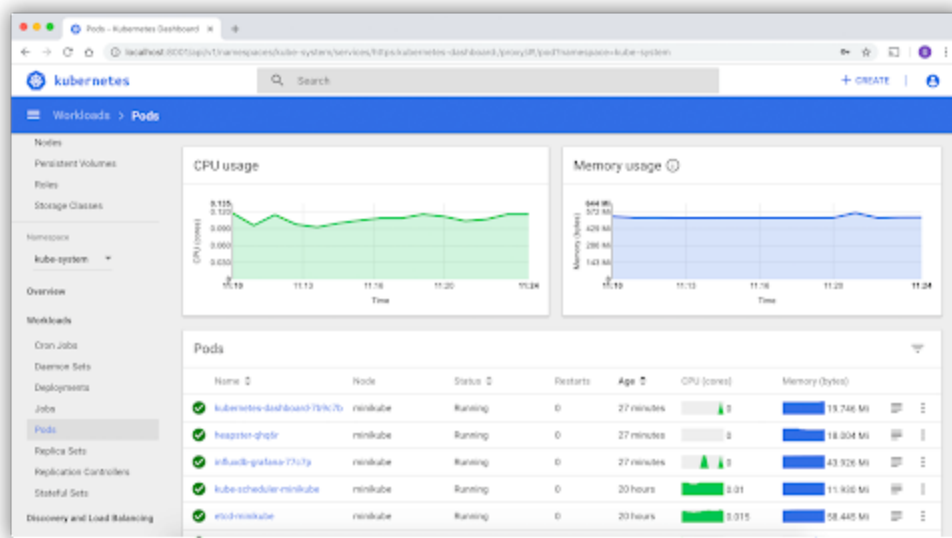
The kubeconfig includes the API server's endpoint server: **https://192.168.99.100:8443** and the minikube user's client authentication key and certificate data.

Once kubectl is installed, we can display information about the Minikube Kubernetes cluster with the kubectl cluster-info command:

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Accessing Dashboard from minikube

We can use the minikube dashboard command to access the kubernetes dashboard



Accessing dashboard with kubectl proxy

The kubectl proxy command authenticates with the API server on the master node to display the Kubernetes dashboard on a different url, usually on the default proxy port 8001. We can do this with the ***kubectl proxy***

We can open another terminal and send requests to API on the default port 8001, without worrying about authentication. We can use the curl command to achieve this. To see all the endpoints for the API server we do this:

```
curl http://localhost:8001/
```

Minikube Practice Project

This tutorial shows you how to run a sample app on Kubernetes using minikube and Katacoda. Katacoda provides a free, in-browser Kubernetes environment.

[Hello minikube](#)

L4: Building Blocks of Kubernetes

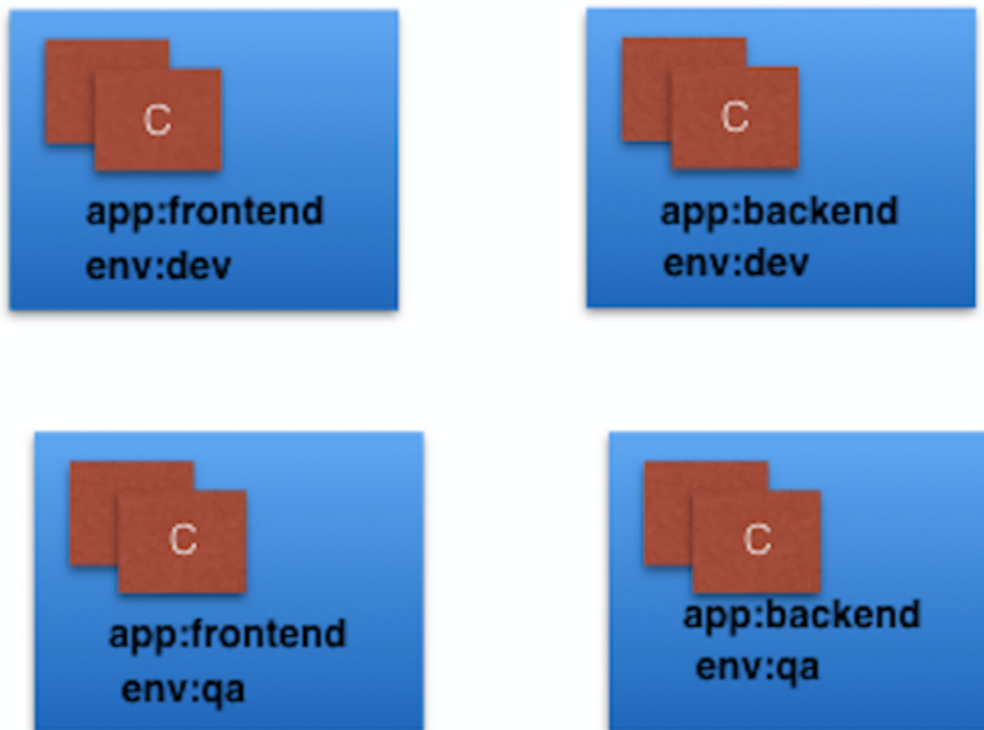
In this lesson we will explore the major building blocks of Kubernetes, discuss Labels and Selectors; their roles in microservices and how they are used to group objects together.

Building blocks of Kubernetes

1. **Pods:** This is the smallest and simplest unit of deployment in Kubernetes that represents a single instance of application. It is a logical collection of one or more containers. Containers in a pod acts as a single entity and can:

- Be scheduled together on the same host
- Share IP address assigned to the Pod
- Mount the same volume

2. **Labels:** these are key-value pairs attached to objects. They are used to monitor and organize objects in a cluster, and select subset of objects based on requirements. Labels are used by controllers to group together objects that have been decoupled.



The image above uses two labels app and env, each of the labels can be used to group two of the pods above, and we can also select one of the pods by mentioning key:value pair.

The image above uses two labels app and env, each of the labels can be used to group two of the pods above, and we can also select one of the pods by mentioning key:value pair.

3. **Label Selectors:** Used by controllers to select a subset of objects. There are two types of selectors supported by kubernetes.

- Equality based Selectors: match objects based on label keys:values, this is achieved by using equality sign =,== for equal or != for not equal.
- Set based selectors: Filters objects based on a set of values, we use in and notion for label values and exist/does not exist for label keys.


4. ReplicaSet: Create replicas of our deployment by running a multiple instance of the same container component. The replica set implements self-healing, manual scaling, or automatic scaling using autoscaler. It supports both equality based selectors, and set based selector. The replica set adds a new pod to a deployment when it detects an unhealthy pod.

5. Deployment: Deployments handles the creation, deletion, and pods update. A deployment creates a ReplicaSet, which then creates a Pod. The Deployment manages the ReplicaSet and Pods. Deployment objects provide declarative updates to Pods and ReplicaSets. The DeploymentController is part of the master node's controller manager, and as a controller it also ensures that the current state always matches the desired state. It allows for seamless application updates and rollbacks through rollouts and rollbacks, and it directly manages its ReplicaSets for application scaling.

6. Namespaces: Namespace is a unique virtual sub cluster in a kubernetes cluster. This namespace can be used to create resources and objects unique to the namespace. Namespaces are unique to teams using a single kubernetes cluster and resources that need to be isolated. Kubernetes creates four namespaces by default:

- Kube-system: Contains objects created by the kubernetes system. Mainly control plane agents
- Default: contains objects created by an administrator or developers, and are assigned objects by default unless another namespace was specified by the developer or administrator.
- Kube-public: An unsecured namespace, readable by anyone and used for exposing public information about the cluster.
- Kube-node-lease: holds node-lease objects used for node heartbeat data.

The command to list all the Namespaces in a cluster is

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The text 'kubectl get namespaces' is displayed in a light blue monospace font.

```
kubectl get namespaces
```

Practice Project

This tutorial provides an introduction to managing applications with [StatefulSets](#). It demonstrates how to create, delete, scale, and update the Pods of StatefulSets.

[stateful basic](#)

Build and deploy a simple, multi-tier web application using Kubernetes and [Docker](#)

L5:Authentication, Authorization and Admission Control

In this lesson we will explore the different layers of access control stages a request will have to pass through before reaching the API server.

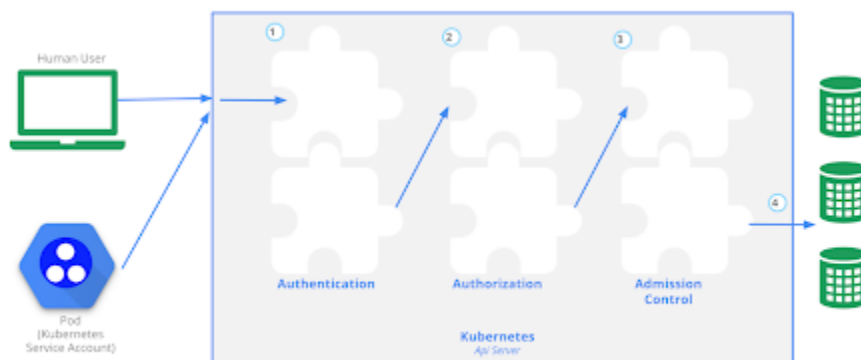
Authentication, Authorization and Admission Control

Objective

- Understand what authentication, authorization, and admission control are
- Differentiate the different kind of kubernetes user
- Understand the various mode of authorization
- Discuss why we need admission control

Access to a Kubernetes cluster resource or object is done by connecting to a specific endpoint on the API server. To access the endpoints, the request passes through several access control stages namely:

1. Authentication
2. Authorization
3. Admission control



Authentication

Kubernetes supports two kinds of users

1. Normal Users: Managed outside the Kubernetes cluster via independent services like google accounts, username/password from a file, etc.

2. **Service accounts:** Allows for communication between the API servers and in cluster processes, they are both manually created and also automatically created through the API server, they are tied to a particular Namespace.

Kubernetes also supports anonymous requests, along with requests from Normal Users and Service accounts.

Some authentication modules are X509 Client certificates, Static token file, Bootstrap tokens, service account tokens, OpenID connect tokens, webhook token authentication, authentication proxy.

Authorization

Successfully authenticated users can now send requests to API servers for different operations, the various requests will have to be authorized by Kubernetes and this will be done using different authorization modules that allow or deny the requests. Kubernetes reviews and evaluates the API request attributes against policies and requests is allowed for successful evaluation.

Authorization has multiple modules, or authorizers and more than one module can be configured for one Kubernetes cluster and each module is checked in sequence.

Mode of Authorization

1. **Node:** This is a special purpose authorization mode that authorizes API requests made by kubelets. It authorizes the Kubelet's read operations for services, endpoints, or nodes, and writes operations for nodes, pods, and events.
2. **Attribute-Based Access Control (ABAC):** Kubernetes grants access to API requests which combine policies with attributes. To enable ABAC mode, we start the API server with the `--authorization-mode=ABAC` option, while specifying the authorization policy with `--authorization-policy-file=PolicyFile.json`
3. **Kubernetes requests authorization decisions to be made by third-party services** which return for successful authorization and false for failure. To enable this mode we need to start the API server with `--authorization-webhook-config-file=SOME_FILENAME` option, where `SOME_FILENAME` is the configuration of the remote authorization service.
4. **Role based access control (RBAC):** regulates access to resources based on the roles of individual users, we can attach multiple roles to subjects like users, service accounts etc. resource access can be restricted by specific operations like create, get, update, and patch. RBAC has two kinds of roles

a. Role: Grants access to resources within a namespace.

b. ClusterRole: Has same permission as namespace, but has a cluster-wide scope

Once a role is created, we can bind it to users with a RoleBinding object. There are 2 kinds

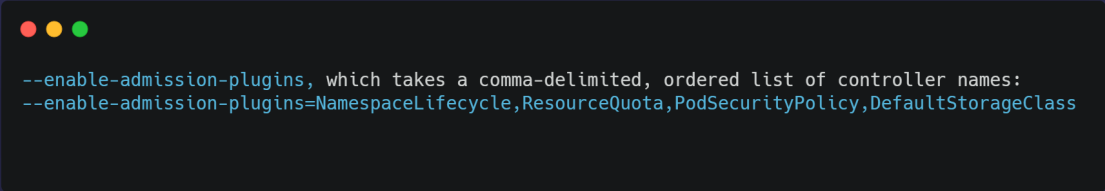
1. RoleBinding: It allows us to bind users to the same namespace as a Role. We could also refer to a ClusterRole in RoleBinding, which would grant permissions to Namespace resources defined in the ClusterRole within the RoleBinding's Namespace.
2. It allows us to grant access to resources at a cluster-level and to all Namespaces.

To enable the RBAC mode, we start the API server with the `--authorization-mode=RBAC` option, allowing us to dynamically configure policies.

Admission Control

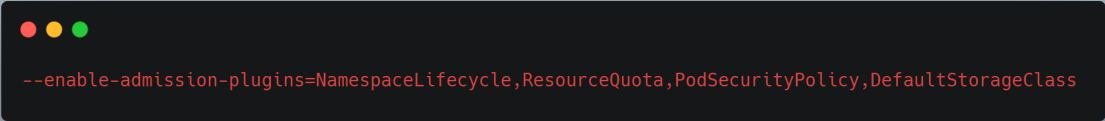
Admission Controllers are used to specify granular access control policies, which include allowing privileged containers, checking on resource quota, etc. We force these policies using different admission controllers, like ResourceQuota, DefaultStorageClass, AlwaysPullImages, etc. They come into effect only after API requests are authenticated and authorized.

To use admission controls, we must start the Kubernetes API server with the



```
--enable-admission-plugins, which takes a comma-delimited, ordered list of controller names:  
--enable-admission-plugins=NamespaceLifecycle,ResourceQuota,PodSecurityPolicy,DefaultStorageClass
```

Which takes a comma-delimited, ordered list of controller names:



```
--enable-admission-plugins=NamespaceLifecycle,ResourceQuota,PodSecurityPolicy,DefaultStorageClass
```

L6: Services and Volumes

In this lesson we will learn about how service objects are used for communication between different objects in a deployment and cluster.

We will also discuss what volumes are, the different types of volumes, and how they help attach persistent volumes to pods.

R1: Services

Objective

- Define Services.
- Explore Kube proxy and Service Discovery.
- Discuss the different service discovery runtime.
- Discuss the different service type

A service offers a single DNS entry for a containerized application managed by Kubernetes regardless of the number of replicas, by providing a common load balancing access point to a set of pods logically grouped, and managed by a controller such as Deployment, ReplicaSet, or DaemonSet.

Service exposes single Pods, ReplicaSet, Deployments, DaemonSets, and StatefulSets. Service forwards traffic to Pods by using a specified target port or selecting the port which the service is receiving traffic. Service endpoints is a logical set of s Pod's Ip address along with the target points 10.0.0.1.1:5000. These endpoints are created and managed automatically by the Service.

Kube-proxy

Kube-proxy is responsible for implementing the Service configuration on behalf of an administrator or developer, in order to enable traffic routing to an exposed application running in Pods. The kube-proxy keeps an eye on the master node for addition, updates, and removal of Services, and endpoints.

Service Discovery

It is important to be able to discover services at runtime. There are two methods supported by Kubernetes.

1. **Environment Variable:** A new set of environment variables are added in the Pods for all active service by the kubelet daemon running on that node, as soon as the Pod starts on any worker node. Services created after Pods are created will not have the environmental variables set in the Pods.
2. **DNS:** Kubernetes provides an add-on for DNS which creates a DNS record for each service, and has the format my-svc.my-namespace.svc.cluster.local. Services in the same namespace can locate other services with just their names

Service Type

1. **ClusterIP:** This is the default ServiceType, this is a virtual IP address received by a service. It is used to communicate with Services, and can only be accessed from within the cluster.
2. **NodePort:** a high-port, dynamically picked from the default range 30000-32767, is mapped to the respective Service, from all the worker nodes. This is in addition to the ClusterIP. We use this when we want our services to be reached from the external world. The end-user connects to any worker node on the specified high-port, which proxies the request internally to the ClusterIP of the Service, then the request is forwarded to the applications running inside the cluster. The Service is load balancing such requests, and only forwards the request to one of the Pods running the desired application.
3. **LoadBalancer:** in this service type, the external load balancer routes traffic to the NodePort and ClusterIP. The Service is exposed at a static port on each worker node and also exposed externally using an underlying cloud provider's load balancer feature. If there is no support for the automatic creation of load balancer, then this service type won't work. It's field will be populated with the <Pending> status.
4. **ExternalIP:** A Service can be mapped to an ExternalIP address if it can route to one or more of the worker nodes. Traffic that is ingressed into the cluster with the ExternalIP (as destination IP) on the Service port, gets routed to one of the Service endpoints. This type of service requires an external cloud provider such as Google Cloud Platform, or AWS, and a Load Balancer configured on the cloud provider's infrastructure.

Service Practice Project

Applications running in a Kubernetes cluster find and communicate with each other, and the outside world, through the Service abstraction. This document explains what happens to the source IP of packets sent to different types of Services, and how you can toggle this behavior according to your needs.

[Using Source Ip](#)

R2: **Volumes**

Containers running in pods are ephemeral in nature. This means that once a pod crashes, all the data stored in a container are deleted. If a new pod is spun up, it is brought up as a new pod without the old data. This is a challenge solved by the use of volumes.

A volume is a mount point on the container's file system backed by a storage system, the volume type determines the storage medium, content, and access mode. Containers running in a pod share the volume linked to that pod. The volume has the ability to outlive the containers of the pod, allowing for data preservation even though it is deleted once the pod is deleted.

Volume Types

The volume type determines the properties of the directory, this directory is mounted inside a pod and backed by the volume type. The properties can be size, content, access mode etc.

1. `emptyDir`: This is an empty volume created for the pod as soon as it is scheduled on the worker node. If the pod is terminated, everything in `emptyDir` is deleted forever.
2. `hostPath`: Shares a directory between the host and the Pod, if the pod is terminated, this volume type is not deleted with the pod, so everything in the `hostPath` is available to the host.
3. `Nfs`: mount an NFS share to a pod.
4. `Secret`: stores and pass sensitive information like passwords to a pod.
5. `configMap`: This volume type provides configuration data, shell commands, and arguments to a pod.
6. `iscsi`: mount an iscsi share into a pod.
7. `persistenVolumeClaim`: attaches persistent volume to a pod

We also have others like `gcePersistentDisk`, `awsElasticBlockStore`, `azureDisk`, `azureFile`. We mount these volume types to a pod.

PersistentVolumes

Persistent volumes are storage abstractions backed by several storage technologies, local to the host where Pods are deployed with its application containers, network attached storage, cloud storage. An administrator provisions the persistent volume. They are dynamically provisioned based on the `StorageClass` resource which contains predefined provisioners and parameters to create a `PersistentVolume`.

Kubernetes provides a `PersistentVolume` subsystem which provides APIs to users and administrators, so they can manage and consume persistent storage. `PersistentVolume` API resource type is used to manage resources, while `PersistentVolumeClaim` API resource type is used to consume the persistent volume

PersistentVolumeClaims

This is a request for storage by a user. `PersistentVolume` resource request is done based on type, access mode, and size. There are three access modes: `ReadWriteOnce` (read-write by a single node), `ReadOnlyMany` (read-only by many nodes), and `ReadWriteMany` (read-write by many nodes). Once a suitable `PersistentVolume` is found, it is bound to a `PersistentVolumeClaim`. Once the user is done with the volume, the

PersistentVolume will be released and used according to the persistentVolumeReclaimPolicy.

PersistentVolume Practice Project

MySQL and Wordpress each require a PersistentVolume to store data. Their PersistentVolumeClaims will be created at the deployment step.

[wordpress stateful app](#)

L7: ConfigMaps and Secrets

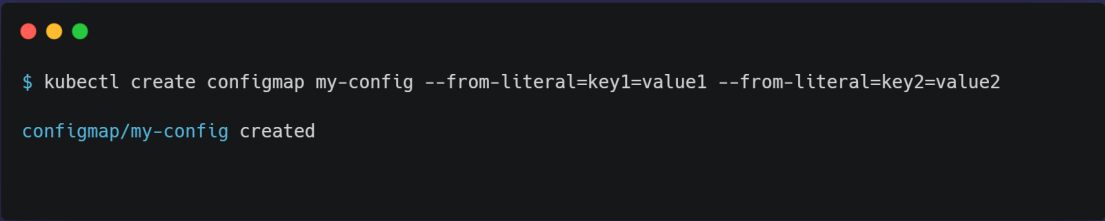
In this lesson, we will learn about how we can pass configuration details and sensitive information, why configMaps and Secrets are important, and different ways we can create configMaps and Secrets.

ConfigMaps and Secrets

ConfigMaps allow us to decouple the configuration details from the container image. Using ConfigMaps, we pass configuration data as key-value pairs which are consumed by Pods or any other system components and controllers, in the form of environment variables, sets of commands and arguments, or volumes. We can create ConfigMaps from literal values, from configuration files, from one or more files or directories.

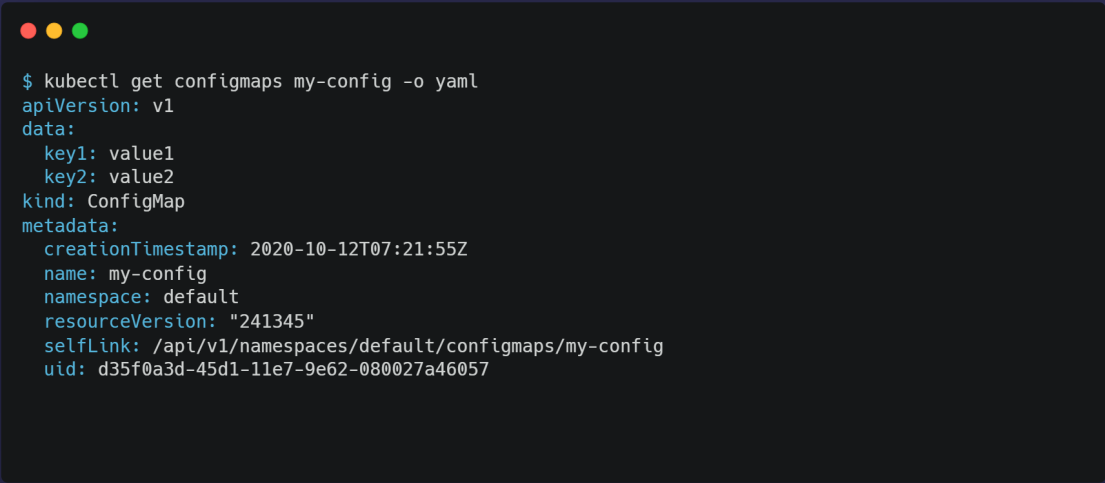
A ConfigMap can be created with the `kubectl create` command, and we can display its details using the **kubectl** `get` command.

Create the ConfigMap

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The terminal shows a command to create a ConfigMap from literal values and the resulting output.

```
$ kubectl create configmap my-config --from-literal=key1=value1 --from-literal=key2=value2
configmap/my-config created
```

Display the ConfigMap Details for my-config



```
$ kubectl get configmaps my-config -o yaml
apiVersion: v1
data:
  key1: value1
  key2: value2
kind: ConfigMap
metadata:
  creationTimestamp: 2020-10-12T07:21:55Z
  name: my-config
  namespace: default
  resourceVersion: "241345"
  selfLink: /api/v1/namespaces/default/configmaps/my-config
  uid: d35f0a3d-45d1-11e7-9e62-080027a46057
```

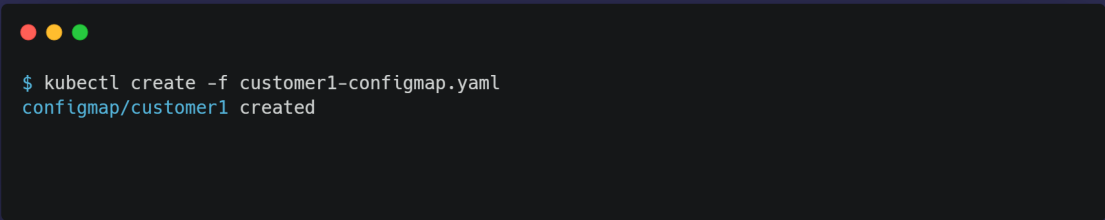
With the `-o yaml` option, we are requesting the `kubectl` command to produce the output in the YAML format. As we can see, the object has the `ConfigMap` kind, and it has the key-value pairs inside the `data` field. The name of `ConfigMap` and other details are part of the `metadata` field.

Create a configMap from a Configuration File

First, we need to create a configuration file with the following content

Where we specify the kind, metadata, and data fields, targeting the `v1` endpoint of the API server.

If we name the file with the configuration above as `customer1-configmap.yaml`, we can then create the `ConfigMap` with the following command:



```
$ kubectl create -f customer1-configmap.yaml
configmap/customer1 created
```

Create a ConfigMap from a File

First, we need to create a file `permission-reset.properties` with the following configuration data:

```
permission=read-only
allowed="true"
resetCount=3
```

We can then create the ConfigMap with the following command:

```
$ kubectl create configmap permission-config --from-file=<path/to/>permission-reset.properties
configmap/permission-config created
```

Use ConfigMap inside Pods

As Environment Variables

Inside a Container, we can retrieve the key-value data of an entire ConfigMap or the values of specific ConfigMap keys as environment variables.

In the following example, all the myapp-full-container Container's environment variables receive the values of the full-config-map ConfigMap keys:

```
...
containers:
- name: myapp-full-container
  image: myapp
  envFrom:
  - configMapRef:
    name: full-config-map
...
```

In the following example, the myapp-specific-container Container's environment variables receive their values from specific key-value pairs from two separate ConfigMaps, config-map-1 and config-map-2:

```

...
containers:
- name: myapp-specific-container
  image: myapp
  env:
  - name: SPECIFIC_ENV_VAR1
    valueFrom:
      configMapKeyRef:
        name: config-map-1
        key: SPECIFIC_DATA
  - name: SPECIFIC_ENV_VAR2
    valueFrom:
      configMapKeyRef:
        name: config-map-2
        key: SPECIFIC_INFO
...

```

With the above, we will get the **SPECIFIC_ENV_VAR1** environment variable set to the value of **SPECIFIC_DATA** key from config-map-1 ConfigMap, and **SPECIFIC_ENV_VAR2** environment variable set to the value of **SPECIFIC_INFO** key from config-map-2 ConfigMap.

As Volumes

We can mount a vol-config-map ConfigMap as a Volume inside a Pod. For each key in the ConfigMap, a file gets created in the mount path (where the file is named with the key name) and the content of that file becomes the respective key's value:...

```

...
containers:
- name: myapp-vol-container
  image: myapp
  volumeMounts:
  - name: config-volume
    mountPath: /etc/config
volumes:
- name: config-volume
  configMap:
    name: vol-config-map
...

```

Secrets

[Secret](#) objects can help by allowing us to encode the sensitive information before sharing it. With Secrets, we can share sensitive information like passwords, tokens, or keys in the form of key-value pairs, similar to ConfigMaps. Thus, we can control how the information in a Secret is used, reducing the risk for accidental exposures. In Deployments or other resources, the Secret object is referenced, without exposing its content.

It is important to keep in mind that by default, the Secret data is stored as plain text inside etcd. Therefore administrators must limit access to the API server and etcd. However, Secret data can be encrypted at rest while it is stored in etcd, but this feature needs to be enabled at the API server level.

Create a Secret from Literal and Display its Details

To create a Secret, we can use the `kubectl create secret` command:

```
$ kubectl create secret generic my-password --from-literal=password=mysqlpassword
```

The above command would create a secret called `my-password`, which has the value of the `password` key set to `mysqlpassword`.


After successfully creating a secret we can analyze it with the `get` and `describe` commands. They do not reveal the content of the Secret. The type is listed as `Opaque`.

```
$ kubectl get secret my-password
NAME          TYPE      DATA   AGE
my-password   Opaque    1       8m

$ kubectl describe secret my-password
Name:         my-password
Namespace:    default
Labels:       <none>
Annotations:  <none>
Type          Opaque
Data
====
password:    13 bytes
```

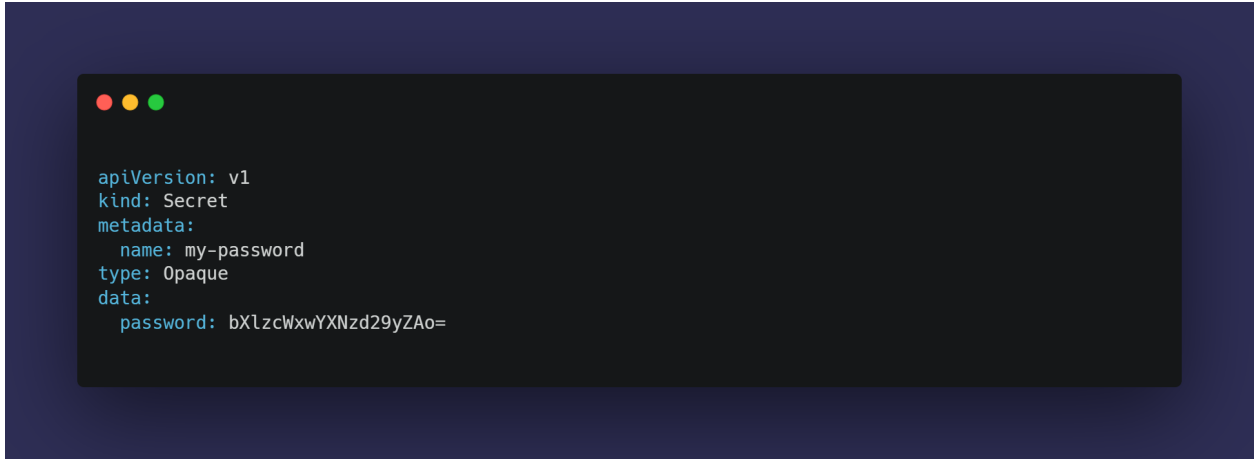
We can create a Secret manually from a YAML configuration file. The example file below is named mypass.yaml. There are two types of maps for sensitive information inside a Secret: data and stringData.

With data maps, each value of a sensitive information field must be encoded using base64. If we want to have a configuration file for our Secret, we must first create the base64 encoding for our password:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal shows a command to echo a password and pipe it to base64, followed by the resulting encoded string.

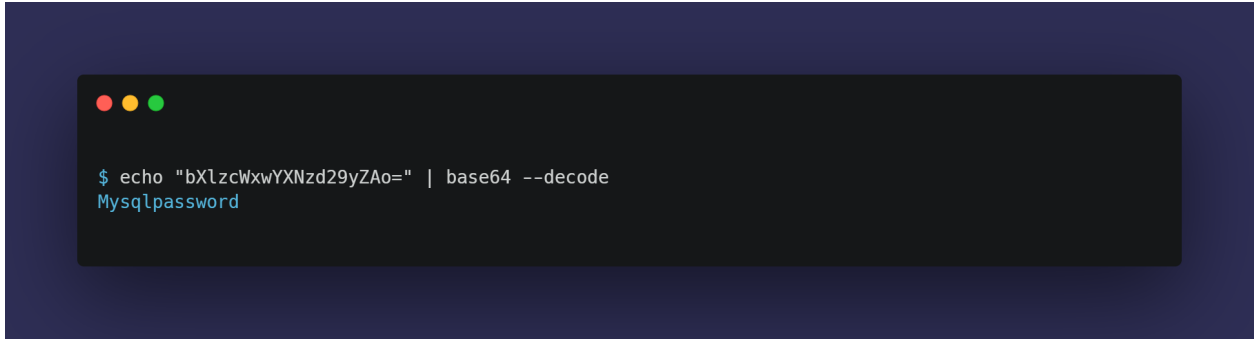
```
$ echo mysqlpassword | base64
bXlzcWxwYXNzd29yZAo=
```

and then use it in the configuration file:

A terminal window with a dark background and three colored window control buttons in the top-left corner. The terminal displays a YAML configuration for a Kubernetes Secret, including apiVersion, kind, metadata, type, and a base64-encoded password.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-password
type: Opaque
data:
  password: bXlzcWxwYXNzd29yZAo=
```

Please note that base64 encoding does not mean encryption, and anyone can easily decode our encoded data:

A terminal window with a dark background and three colored window control buttons in the top-left corner. The terminal shows a command to echo the base64-encoded password and pipe it to base64 --decode, resulting in the original password.

```
$ echo "bXlzcWxwYXNzd29yZAo=" | base64 --decode
Mysqlpassword
```

Therefore, make sure you do not commit a Secret's configuration file in the source code.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-password
type: Opaque
stringData:
  password: mysqlpassword
```

With stringData maps, there is no need to encode the value of each sensitive information field. The value of the sensitive field will be encoded when the my-password Secret is created:

Using the mypass.yaml configuration file we can now create a secret with kubectl create command:

```
$ kubectl create -f mypass.yaml
secret/my-password created
```

Use Secrets Inside Pods

Secrets are consumed by Containers in Pods as mounted data volumes, or as environment variables, and are referenced in their entirety or specific key-values.

Using Secrets as Environment Variables

Below, we reference only the password key of the my-password Secret and assign its value to the WORDPRESS_DB_PASSWORD environment variable:

```

....
spec:
  containers:
  - image: wordpress:4.7.3-apache
    name: wordpress
    env:
    - name: WORDPRESS_DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-password
          key: password
....

```

Using Secrets as Files from a Pod

We can also mount a Secret as a Volume inside a Pod. The following example creates a file for each my-password Secret key (where the files are named after the names of the keys), the files containing the values of the Secret:

```

....
spec:
  containers:
  - image: wordpress:4.7.3-apache
    name: wordpress
    volumeMounts:
    - name: secret-volume
      mountPath: "/etc/secret-data"
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: my-password
....

```

For more details, you can explore the documentation on managing [Secrets](#).

ConfigMaps Practice Project

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.

Configure a Pod to Use a ConfigMap

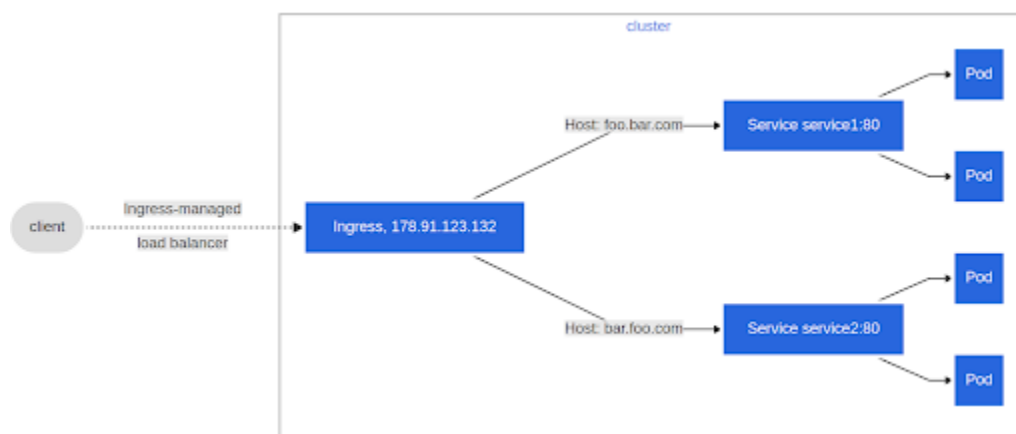
L8: Ingress

In this lesson we will discuss Ingress, and how it can be configured to let our applications be accessed from external services.

Ingress

An ingress is a collection of rules that allow inbound connections to reach the cluster services. Ingress configures a layer 7 HTTP/HTTPS load balancer for services and provides the following

- TLS (Transport Layer Security)
- Name-based virtual hosting
- Fanout routing
- Load Balancing
- Custom rules.



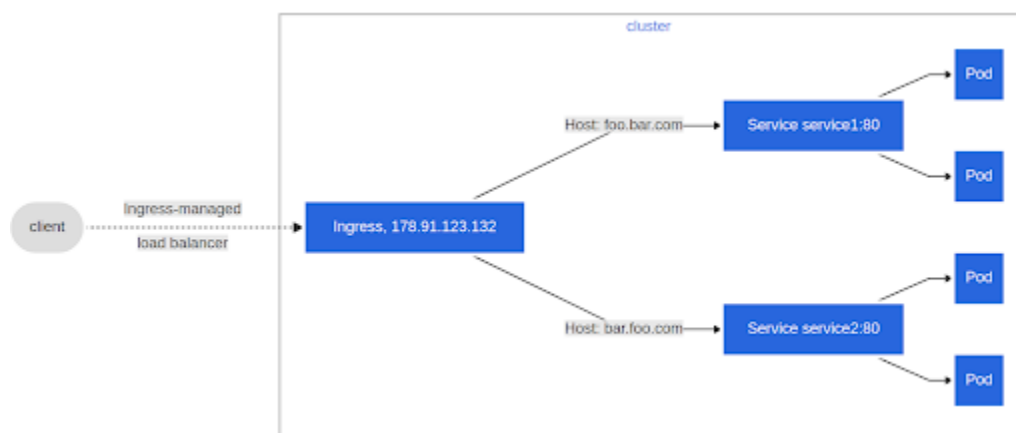
With Ingress, users do not connect directly to a Service. Users reach the Ingress endpoint, and, from there, the request is forwarded to the desired Service. You can see an example of a sample Ingress definition below:

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: virtual-host-ingress
  namespace: default
spec:
  rules:
    - host: blue.example.com
      http:
        paths:
          - backend:
              service:
                name: webserver-blue-svc
                port:
                  number: 80
            path: /
            pathType: ImplementationSpecific
    - host: green.example.com
      http:
        paths:
          - backend:
              service:
                name: webserver-green-svc
                port:
                  number: 80
            path: /
            pathType: ImplementationSpecific

```

In the example above, user requests to both blue.example.com and green.example.com would go to the same Ingress endpoint, and, from there, they would be forwarded to webserver-blue-svc, and webserver-green-svc, respectively. This is an example of a Name-Based Virtual Hosting Ingress rule.



Name-Based Virtual Hosting Ingress

We can also define Fanout Ingress rules, when requests to `example.com/blue` and `example.com/green` would be forwarded to `webserver-blue-svc` and `webserver-green-svc`, respectively:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: fan-out-ingress
  namespace: default
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /blue
        backend:
          service:
            name: webserver-blue-svc
            port:
              number: 80
        pathType: ImplementationSpecific
      - path: /green
        backend:
          service:
            name: webserver-green-svc
            port:
              number: 80
        pathType: ImplementationSpecific
```


The Ingress resource does not do any request forwarding by itself, it merely accepts the definitions of traffic routing rules. The ingress is fulfilled by an Ingress Controller, which is a reverse proxy responsible for traffic routing, based on rules defined in the Ingress resource.

Ingress Controller

An [Ingress Controller](#) is an application watching the Master Node's API server for changes in the Ingress resources and updates the Layer 7 Load Balancer accordingly. Ingress Controllers are also known as Controllers, Ingress Proxy, Service Proxy, Revers Proxy, etc. Kubernetes supports an array of Ingress Controllers, and, if needed, we can also build our own. [GCE L7 Load Balancer Controller](#) and [Nginx Ingress Controller](#) are commonly used Ingress Controllers. Other controllers are [Contour](#), [HAProxy Ingress](#), [Istio](#), [Kong](#), [Traefik](#), etc.

Start the Ingress Controller with Minikube

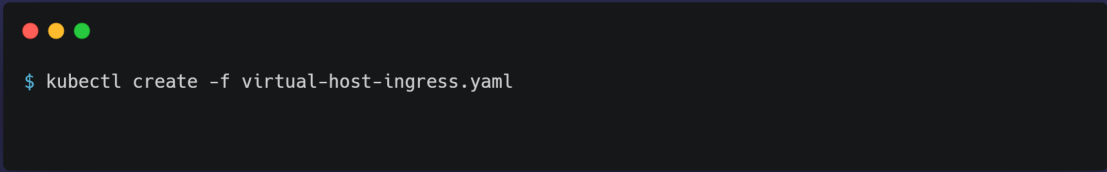
Minikube ships with the [Nginx Ingress Controller](#) setup as an add-on, disabled by default. It can be easily enabled by running the following command:



```
$ minikube addons enable ingress
```

Deploy an Ingress Controller


Once the Ingress Controller is deployed, we can create an Ingress resource using the `kubectl create` command. For example, if we create a `virtual-host-ingress.yaml` file with the Name-Based Virtual Hosting Ingress rule definition that we saw in the Ingress II section, then we use the following command to create an Ingress resource:



```
$ kubectl create -f virtual-host-ingress.yaml
```

Access Services Using Ingress

As our current setup is on Minikube, we will need to update the host configuration file (`/etc/hosts` on Mac and Linux) on our workstation to the Minikube IP for those URLs. After the update, the file should look similar to:



```
$ cat /etc/hosts
127.0.0.1    localhost
::1         localhost
192.168.99.100 blue.example.com green.example.com
```

Now we can open `blue.example.com` and `green.example.com` on the browser, and access each application.

Ingress Project

Create a Kubernetes Service object that exposes an external IP address.

[Expose external Ip address](#)