Kubernetes Administration

-Lecture-

Course ID: KUB201v1.2
Version: 1.2.0
Date: 2021-04-20

SUSE

Proprietary Statement

SUSE LLC, has intellectual property rights relating to technology embodied in the product that is described in this document.

SUSE
Maxfeldstrasse 5
90409 Nuremberg
Germany
www.suse.com

Disclaimer

SUSE LLC, makes no representations or warranties with respect to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose.

Further, SUSE LLC, reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. Further, SUSE LLC, makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, SUSE LLC, reserves the right to make changes to any and all parts of SUSE software, at any time, without any obligation to notify any person or entity of such changes.

Any products or technical information provided under this Agreement may be subject to U.S. export controls and the trade laws of other countries. You agree to comply with all export control regulations and to obtain any required licenses or classification to export, re-export or import deliverables. You agree not to export or re-export to entities on the current U.S. export exclusion lists or to any embargoed or terrorist countries as specified in the U.S. export laws. You agree to not use deliverables for prohibited nuclear, missile, or chemical biological weaponry end uses. SUSE assumes no responsibility for your failure to obtain any necessary export approvals.

This SUSE Training Manual is published solely to instruct students in the use of SUSE networking software. Although third-party application software packages may be used in SUSE training courses, this is for demonstration purposes only and shall not constitute an endorsement of any of these software applications.

Further, SUSE LLC does not represent itself as having any particular expertise in these application software packages and any use by students of the same shall be done at the student's own risk.

Contents

SUSE

Section: 1

Course Introduction

Section Objectives:

1 Course Objectives and Audience

2 Course Lab Environment Overview

3 Certification Options

4 Additional SUSE Training

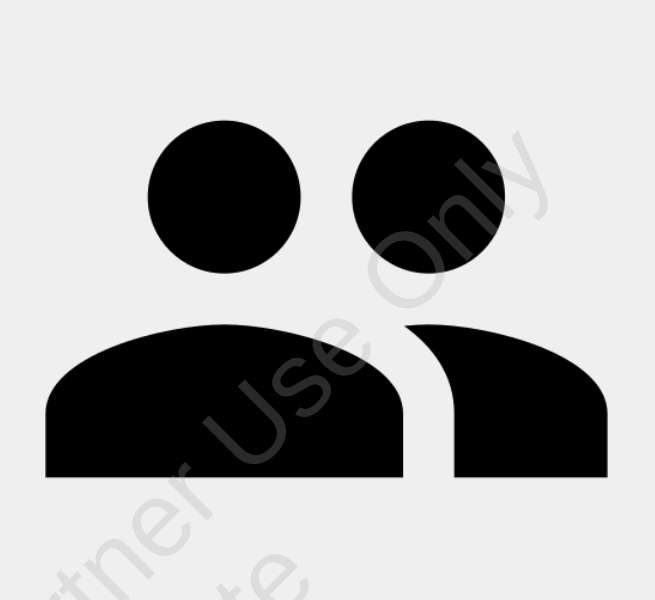
Course Objectives and Audience

Course Overview

- This course is designed for system administrators, DevOps, system engineers and others who need an introduction to Containers, Kubernetes
- The course begins with an introduction to containers and container orchestration
- Students will learn about and explore Kubernetes, including launching applications, configuring networking, storage and security, and using Helm to deploy applications
- The course includes comprehensive presentation content to introduce new concepts and processes and extensive hands-on experience

Audience

- This course is designed for system administrators and others who want to administer Kubernetes

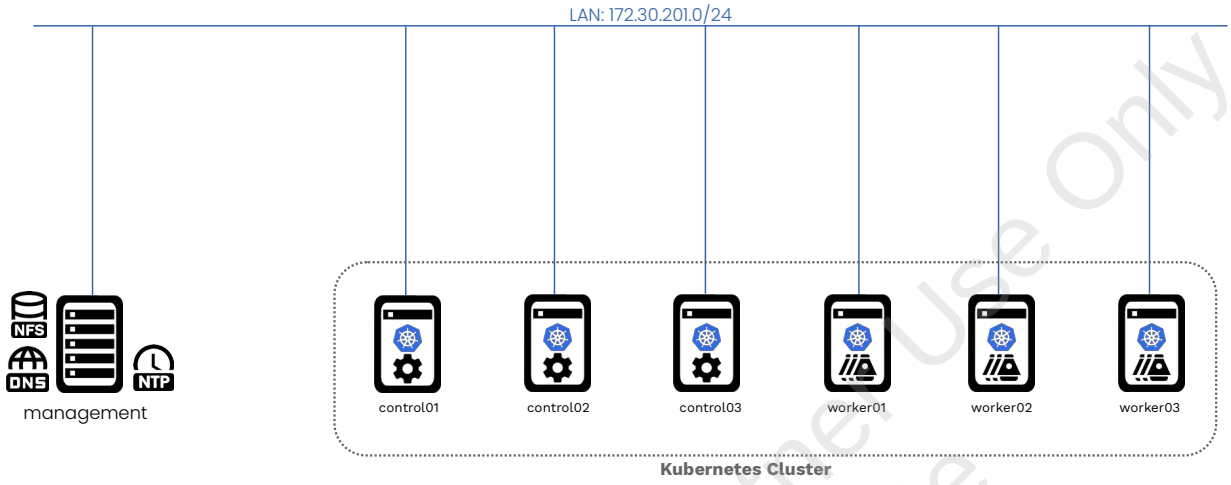


Course Agenda

- Day 1
  - Section 1: Course Introduction
  - Section 2: Introduction to Containers and Container Orchestration
  - Section 3: Kubernetes Administration
- Day 2
  - Section 4: Application Management in Kubernetes with Kustomize
  - Section 5: Application Management in Kubernetes with Helm
  - Section 6: Ingress Networking with an Ingress Controller in Kubernetes
  - Section 7: Storage in Kubernetes
  - Section 8: Resource Usage Control in Kubernetes
  - Section 9: Role Based Access Controls in Kubernetes

Course Lab Environment

Lab Environment Diagram



Required Minimum Product Version

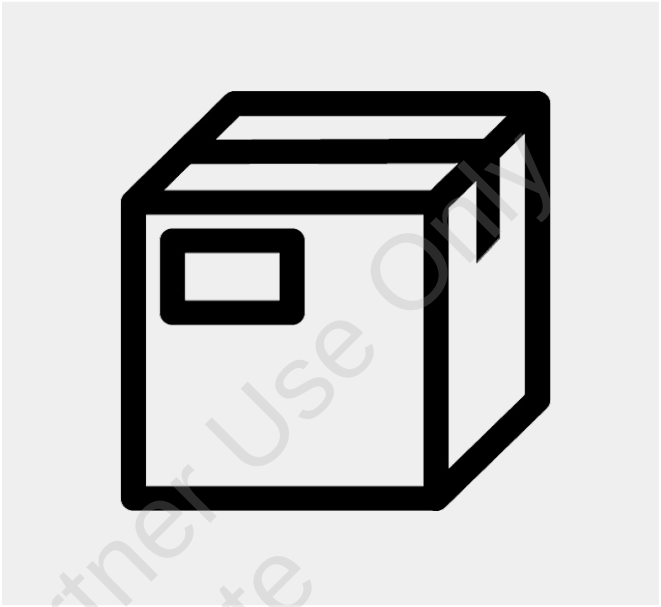
This course is based on the following product version

Product: RKE, K3S and other Kubernetes distros

Version: Kubernetes 1.19, RKE 1.2.5 or comparable

(should work on recent older versions of Kubernetes as well)

This is the minimum version required to run the course.
The material in the course may apply to subsequent versions as well.



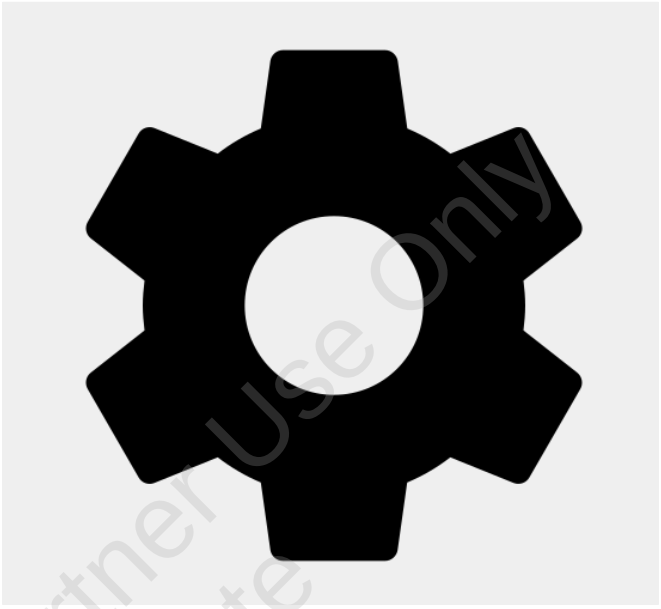
Hardware / Software Requirements

CPU: 4 Core

RAM: 50GB for VMs

Disk: 200GB

Min host OS: openSUSE Leap 15.2, SLES 15 SP2



Certification Options

Associated SUSE Certifications

- There are currently no SUSE Kubernetes Certifications
- For more information:
  https://training.suse.com/certification

Additional SUSE Training

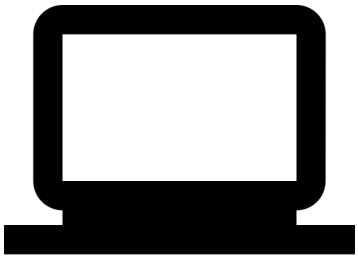
Additional SUSE Training

- SUSE Training is available for:
  - SUSE Linux Enterprise
  - SUSE Linux Enterprise for SAP
  - SUSE Manager
  - SUSE Rancher

- See website for more:

  https://training.suse.com/training

Exercises:



1-1: Start the Lab Environment VMs

SUSE

Section: 2

Introduction to Containers and Container Orchestration

Section Objectives:

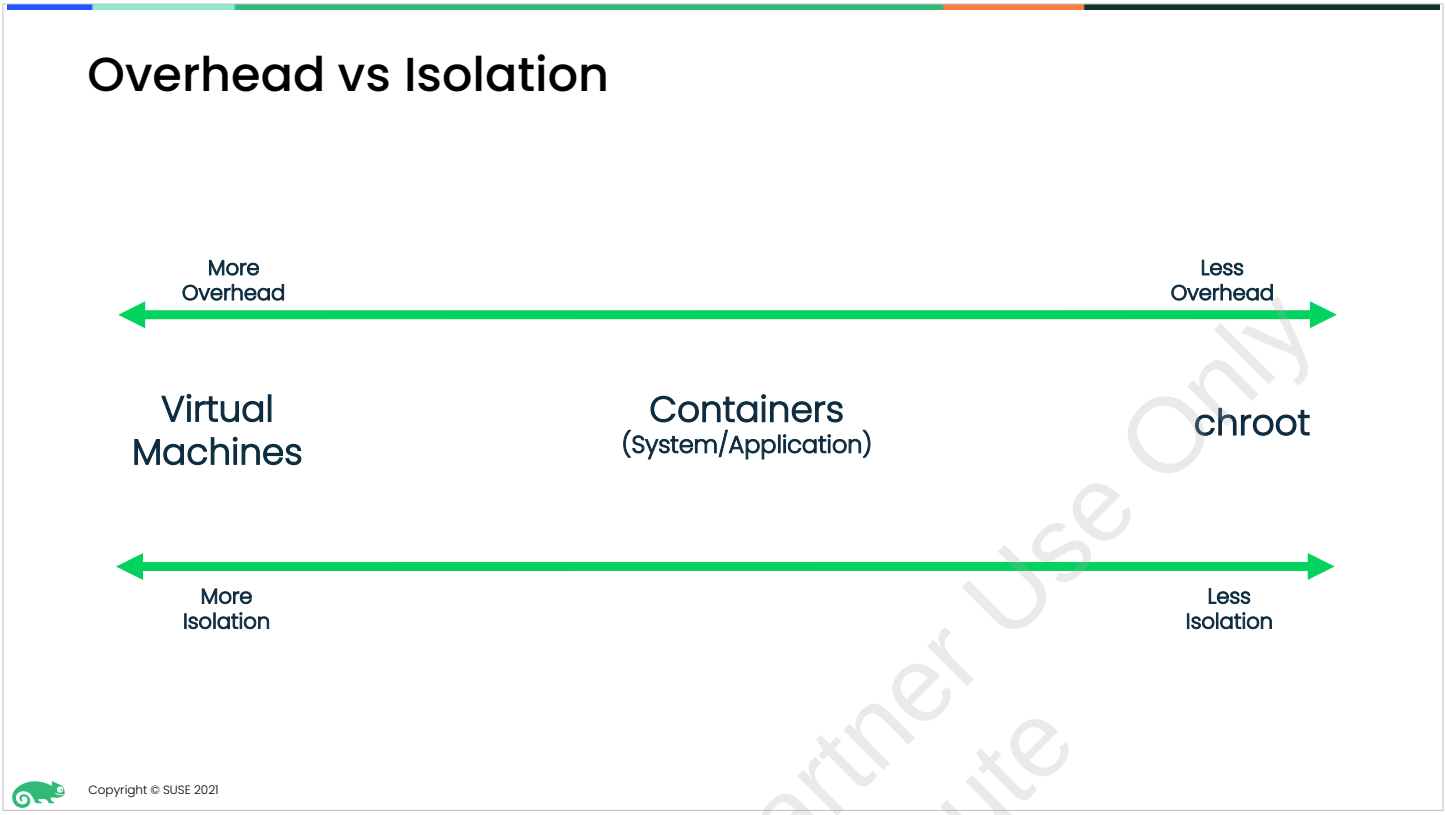
1 Understand Container Concepts

2 Understand Microservice Architecture

3 Understand Kubernetes Concepts

4 Understand SUSE Rancher Kubernetes Offerings

Understand Container Concepts



Virtual machine and containers are similar in many ways but their advantages and disadvantages are different. One of the main goals of virtualization and containers is isolation of workloads from both other workloads and the underlying system. How this isolation is implemented is one of the biggest differences between them.

The most simple way to gain some level of isolation is to run an application in a chroot (change root) environment. A chroot environment allows an application to have filesystem isolation in that the application running in the chroot environment can only see the filesystem directory structure inside the chroot environment. This chroot environment can be as simple as an existing directory in the host system's filesystem or it can be an image of some type that has been mounted into the host system's filesystem. All chroot environments running a host system share the host system's kernel, they only have separate/isolated views of the filesystem. A chroot requires the least overhead but also provide the least isolation.

Containers build upon the concept of chroot environments in that they maintain the filesystem isolation but add process and network isolation. With containers, the filesystems are typically contained in images of some sort. The process and network isolation are provided by kernel cgroups and namespaces. Containers typically come in two flavors: system containers and applications containers. The line between the two flavors can be a bit blurry at times but a simple description is that a system container contains a full OS image minus the kernel where an application container only contains the files/libraries/binaries required to run the desired application. All containers running on a host system share the host system's kernel but have their own filesystem, process space and network stack. Containers require a similar level of overhead to chroot environments but provide a higher level of isolation.

Virtual machines require a special software component to run beyond the requirements for chroot environments and containers. This software component is called a hypervisor and it provides virtual hardware level isolation. This hardware isolation presents a virtualized hardware interface to the virtual machine that makes it think it is running on a separate machine. This provide the highest level of isolation but also requires the highest level of overhead.

In almost all cases a virtual machine uses a disk image as its filesystem. Because the virtual machine thinks it is running on its own hardware it requires its own kernel to be running in addition to all other files libraries and binaries. This means that operating systems that are different from the host OS can be run in a virtual machine.

VM / chroot / Container Charateristics

	Virtual Machines	Containers (System/Application)	chroot
Isolation	• Full hardware isolation provided by Hypervisor and hardware emulation/ abstraction	• Filesystem isolation • Process isolation • Network isolation (cgroups/namespaces)	• Filesystem isolation
Portability	• Completely portable*	• Completely portable*	• Not portable*
Contains	• Full Operating System including own kernel • Can be different OS than the host OS	• Full Operating System minus own kernel or • Only the libraries/apps required	• Whatever is desired
Speed	• Relatively fast to instantiate • Run at close to bare hardware speeds	• Very fast to instantiate • Run at bare hardware speeds	• Very fast to instantiate • Run at bare hardware speeds

This chart demonstrates some of the differences between virtual machines, containers and chroot environments.

What is a Container Image?

- A container image is like the skeleton of an application with just enough of an operating system so that it can work
- It can be thought of as the root filesystem of the application

```
d76d23a13bf5:/ # ls -l
total 0
drwxr-xr-x   1 root root  630 Apr  6 08:00 bin
drwxr-xr-x   5 root root  360 Jul 16 11:08 dev
drwxr-xr-x   1 root root 2008 Jul 16 11:08 etc
drwxr-xr-x   1 root root    0 May  9  2017 home
drwxr-xr-x   1 root root   60 Apr  6 08:00 lib
drwxr-xr-x   1 root root 3424 Apr  6 08:00 lib64
drwxr-xr-x   1 root root    0 May  9  2017 mnt
drwxr-xr-x   1 root root    0 May  9  2017 opt
dr-xr-xr-x 333 root root    0 Jul 16 11:08 proc
drwx------   1 root root   44 Apr  6 08:00 root
drwxr-xr-x   1 root root  118 Apr  6 08:00 run
drwxr-xr-x   1 root root 1222 Apr  6 08:00 sbin
drwxr-xr-x   1 root root    0 May  9  2017 selinux
drwxr-xr-x   1 root root   12 Apr  6 07:59 srv
dr-xr-xr-x  13 root root    0 Jul 16 11:08 sys
drwxrwxrwt   1 root root    0 Apr  6 08:00 tmp
drwxr-xr-x   1 root root  130 Apr  6 07:59 usr
drwxr-xr-x   1 root root   92 Apr  6 07:59 var
```


Images used by CRI-O or Docker in Kubernetes are in the OCI (Open Container Initiative) format. OCI is a vendor neutral standard that has been adopted by many application container vendors such as Docker, SUSE, Red Hat, and others.

Images are usually generated by means of a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Dockerfiles usually begin with a base image that is called with a FROM command. Content is added via RUN commands that are usually used to install applications and libraries needed to make the image do what it is designed to do. A Dockerfile must also contain some kind of command, sometimes denotes with CMD, that will run a specific command when the image is started as a container. A Dockerfile can be as simple as a few lines to install a single application via a package manager like Zypper or can by hundreds of lines long to create complex images with greater simplicity. A good example of a more complex Dockerfile that is used by Wordpress can be found at: https://github.com/docker-library/wordpress/blob/master/php7.4/apache/Dockerfile.

On the right is an example of a filesystem of a basic container. It looks just like any other Linux filesystem and that's because it is. Building images is actually simpler than installing a complete OS in a VM and they only require the specific pieces needed to run a single application.

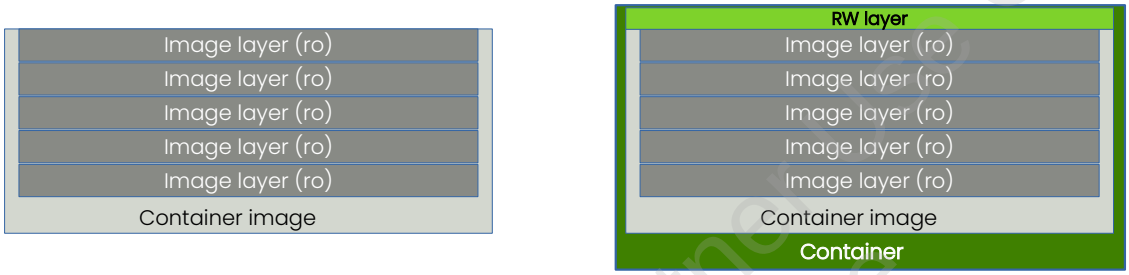
What is a Container?

A container is an “instantiation” of and image, or in other words an image that is put into action.

Images are put into action with container engines. Once it is put into action, it can do what it was meant to do.

Container vs Image in Practice

- An image is a collection of one or more RO layers
- A container is an instantiation of an image
- A container adds a RW layer on top of the RO image layer(s) where all changes are captured



A container is different from and image. Images contain the filesystem that will be used by a container and a container is an “instantiation” of a container image.

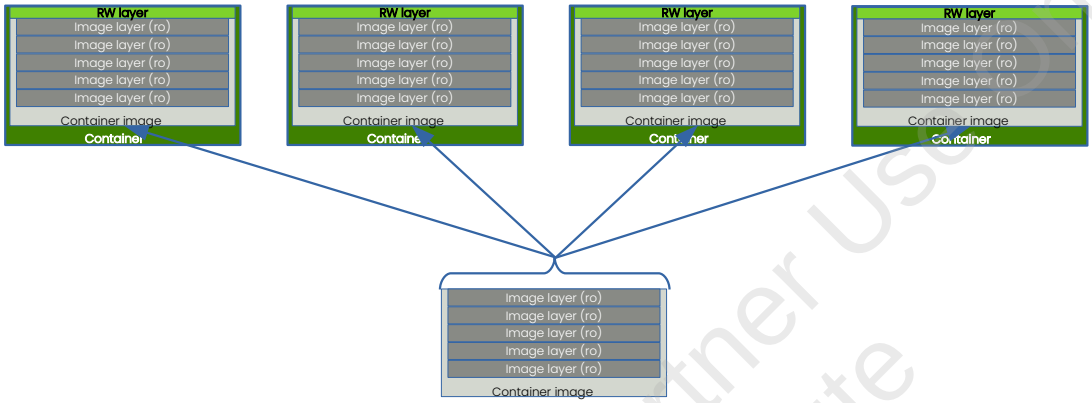
Modern containers are comprised of layers. The base layer is the base collections of files/libraries/binaries that will be used. These base container images are typically designed to be very generic in nature so that that can be used by a wide array of containers. To create an application container image, you start with a base image and then a new layer is added. This layer will contain all of the addition files/libraries/binaries you need for your application. Because these additional layer are copy-on-write, you new layer could contain files that exit in the base (or any other lower) layer but have been changed by you. New container images can be created by either being based on a base image or another container image. The final container image is a collection of the base layer and all additional layers o the image that it was based on.

When a container is launched from a container image, a copy of all of the layer that the image is comprised of are downloaded and a copy-on-write read-write layer is added to the top. This layer captures all of the changes made while the container is running. This read-write layer exists for as long as the container exists. When the container is deleted this read-write layer is deleted.

Efficient Use of Disk Space

A single image can be used by multiple containers simultaneously.

The only additional disk space used is in the RW layers or each of the containers.



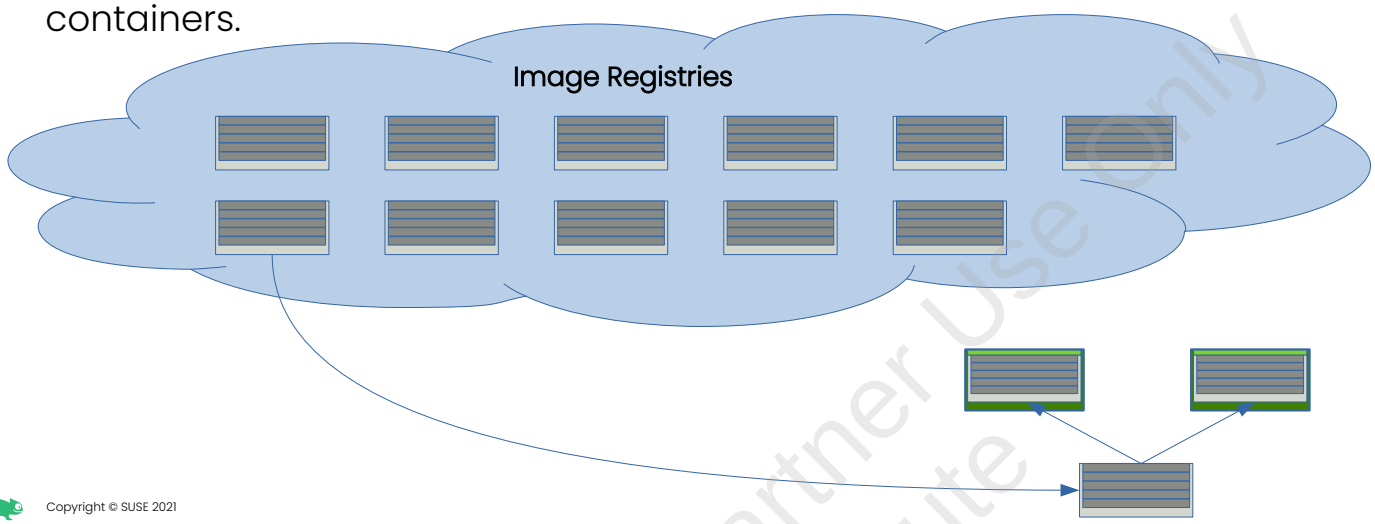
Because of the layering nature of container image and the fact that the layers (including and especially the read-write layer) are copy-on-write, multiple instances of a container can be created and run simultaneously from a single copy of the container image. This provide for very efficient use of storage space when using containers.

Where Do You Get Images to Run As Containers?

Image Registries

Images are stored in repositories called Image Registries.

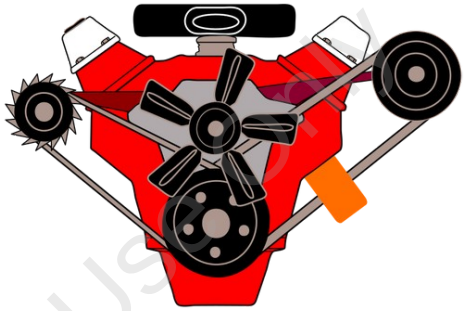
Images can be downloaded (pulled) locally when needed to launch containers.



Where container images need to reside on the local filesystem to be used, all possible images that are available don't always need to reside locally. Image registries can exist both remotely and locally that contain a wide variety of container images for a wide range of applications. When you desire to run a container of a specific application, if the required image does not already exist locally, the container engine will download a copy of the image from a remote registry.

Container Engine

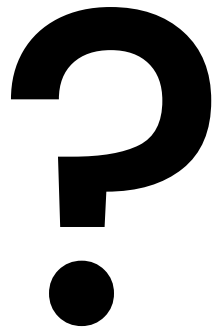
A Container Engine (like containerd) allows a container to run as an independent application directly in an operating system.



Container engines also:

- Provide a network interface
- Separate containers from each other
- Provide a way for containers to access external storage

Questions:



Q. What is a container image?

A. File system image of an application and all its requirements (libraries, etc)

Q. What is a container?

A. A running instance of a container image.

Q. How do containers differ from virtual machines?

A. Contianers share the same kernel as the host OS and VMs run on an abstracted hardware layer with their own OS.

Q. Where are container images stored?

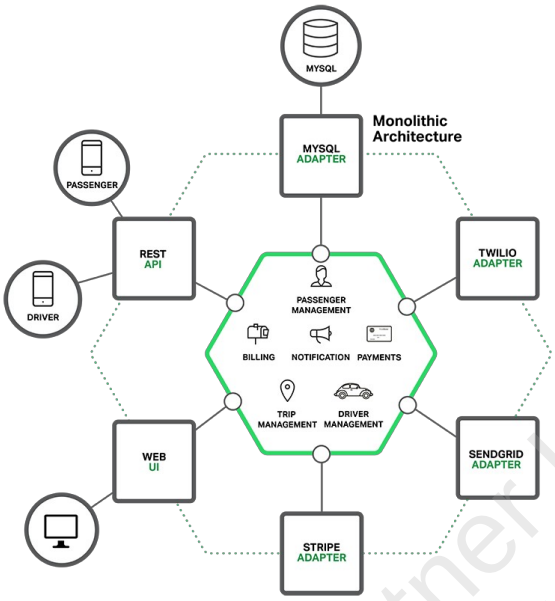
A. Image registries.

Q. What do container engines provide?

A. Environment for a container to run independently and isolated, network connection, connection to storage

Understand Microservice Architecture

Monolithic Application Stack



OLD WAY

Let's say we're designing a new platform to compete with Uber

Problems with monolithic....

Unfortunately, this simple approach has a huge limitation. Successful applications have a habit of growing over time and eventually becoming huge. During each sprint, your development team implements a few more stories, which, of course, means adding many lines of code. After a few years, your small, simple application will have grown into a monstrous monolith. To give an extreme example, I recently spoke to a developer who was writing a tool to analyze the dependencies between the thousands of JARs in their multi-million line of code (LOC) application. I'm sure it took the concerted effort of a large number of developers over many years to create such a beast.

Once your application has become a large, complex monolith, your development organization is probably in a world of pain. Any attempts at agile development and delivery will flounder. One major problem is that the application is overwhelmingly complex. It's simply too large for any single developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming. What's more, this tends to be a downwards spiral. If the codebase is difficult to understand, then changes won't be made correctly. You will end up with a monstrous, incomprehensible big ball of mud.

The sheer size of the application will also slow down development. The larger the application, the longer the start-up time is. For example, in a recent survey some developers reported start-up times as long as 12 minutes. I've also heard anecdotes of applications taking as long as 40 minutes to start up. If developers regularly have to restart the application server, then a large part of their day will be spent waiting around and their productivity will suffer.
Another problem with a large, complex monolithic application is that it is an obstacle to continuous deployment.

Today, the state of the art for SaaS applications is to push changes into production many times a day. This is extremely difficult to do with a complex monolith since you must redeploy the entire application in order to update any one part of it. The lengthy start-up times that I mentioned earlier won't help either. Also, since the impact of a change is usually not very well understood, it is likely that you have to do extensive manual testing. Consequently, continuous deployment is next to impossible to do.

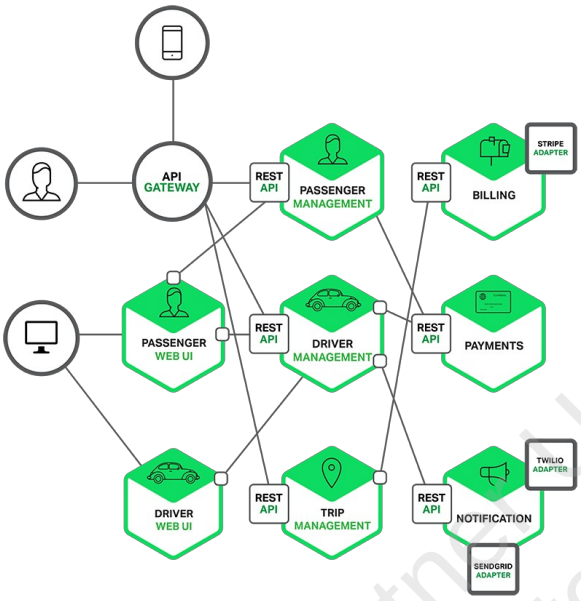
Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements. For example, one module might implement CPU-intensive image processing logic and would ideally be deployed in Amazon EC2 Compute Optimized instances. Another module might be an in-memory database and best suited for EC2 Memory-optimized instances. However, because these modules are deployed together you have to compromise on the choice of hardware.

Another problem with monolithic applications is reliability. Because all modules are running within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.

Last but not least, monolithic applications make it extremely difficult to adopt new frameworks and languages. For example, let's imagine that you have 2 million lines of code written using the XYZ framework. It would be extremely expensive (in both time and cost) to rewrite the entire application to use the newer ABC framework, even if that framework was considerably better. As a result, there is a huge barrier to adopting new technologies. You are stuck with whatever technology choices you made at the start of the project.

To summarize: you have a successful business-critical application that has grown into a monstrous monolith that very few, if any, developers understand. It is written using obsolete, unproductive technology that makes hiring talented developers difficult. The application is difficult to scale and is unreliable. As a result, agile development and delivery of applications is impossible.

Microservice Application



NEW WAY

Instead of building a single monstrous, monolithic application, the idea is to split your application into set of smaller, interconnected services.

A service typically implements a set of distinct features or functionality, such as order management, customer management, etc. Each microservice is a mini-application that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose an API that's consumed by other microservices or by the application's clients. Other microservices might implement a web UI. At runtime, each instance is often a cloud VM or a Docker container.

Each functional area of the application is now implemented by its own microservice. Moreover, the web application is split into a set of simpler web applications (such as one for passengers and one for drivers in our taxi-hailing example). This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases.

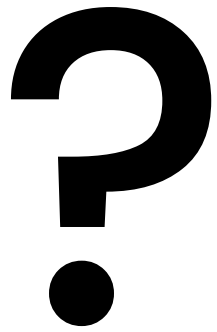
Each backend service exposes a REST API and most services consume APIs provided by other services. For example, Driver Management uses the Notification server to tell an available driver about a potential trip. The UI services invoke the other services in order to render web pages. Services might also use asynchronous, message-based communication. Inter-service communication will be covered in more detail later in this series.

Some REST APIs are also exposed to the mobile apps used by the drivers and passengers. The apps don't, however, have direct access to the backend services. Instead, communication is mediated by an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring, and can be implemented effectively using NGINX.

Benefits of Microservices

- Improves application modularity
- Applications are easier to understand, develop and test
- Support parallel development enabling small autonomous teams to develop, deploy and scale their services independently
- Help enable CI/CD & continuous refactoring
- Produce and ship a better quality product, faster

Questions:



Q. How do microservices differ from monolothic applications?

A. Monolithic apps contain all functionality in a single binary, where each function of a microservice is its own binary. Monolithic apps are coded using a single language, where each microservices app can be coded with different languages.

Q. What are advantages of microservices?

A. Modularity, easier to develop/test, easier to version as each part gets released independently

Understand Kubernetes Concepts

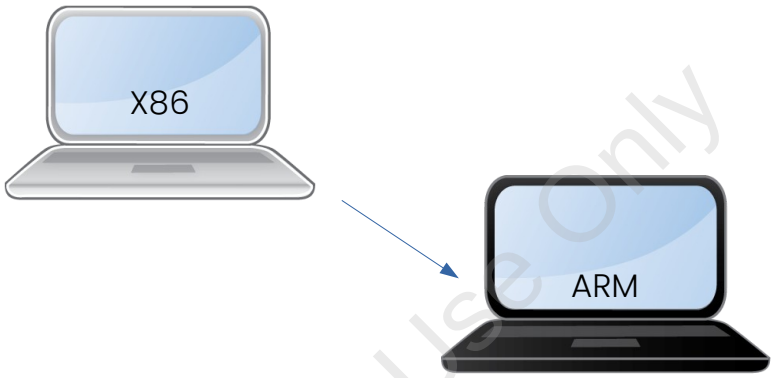
Kubernetes Overview

What is Kubernetes?

- In short, Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services

Kubernetes is:

- Portable



Kubernetes containers are portable across clouds and OS distributions. Images can also be build to be multi-platform so a container that is build for one platform can also be built for another.

Kubernetes is:

- Portable
- Extensible

If Kubernetes doesn't have the functions that you need, it can be extended through the use of network, storage, and other plugins.

Kubernetes is:

- Portable
- Extensible
- Open Souorce

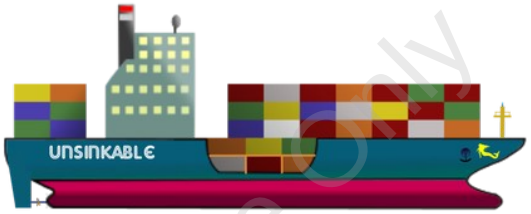
TM

(Apache License 2.0)

Kubernetes is released under an Apache license which allows it to be used and shared with the community instead of owned by only a specific company.

Kubernetes is:

- Portable
- Extensible
- Open Source
- A Framework to Manage Containerized Workloads and Services

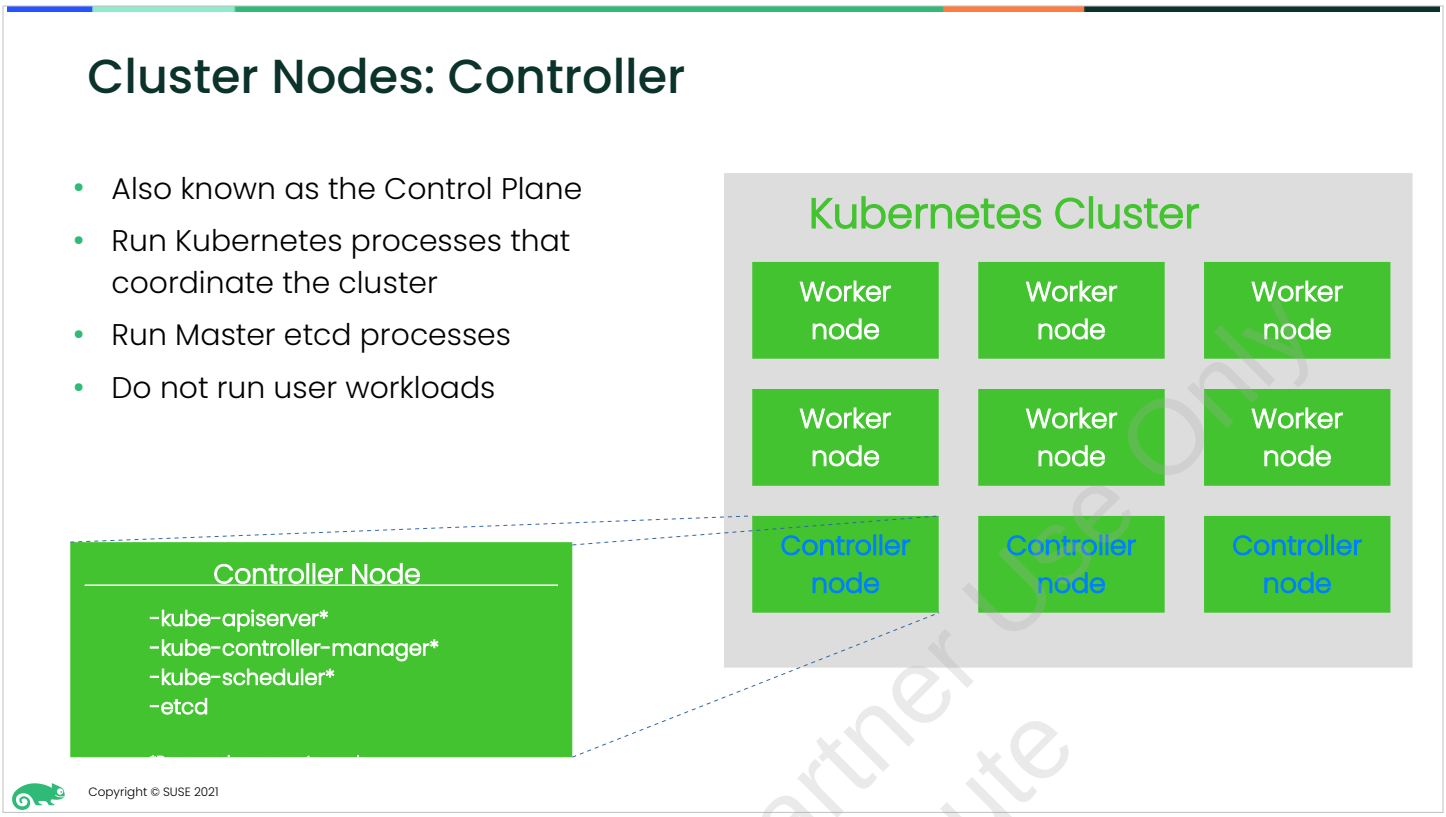


It's not enough to just have containers. Containers and their related workloads must be intelligently managed. Kubernetes is the best platform to do that.

What Does Kubernetes Provide?

- Provides a complete orchestration solution for container based applications
  - Deploy Applications
  - Manage Applications
  - Access Applications
  - Scale Applications
- Provides for scheduling of containers
- Provides a way to consume containers in a developer friendly way
  - Abstracts infrastructure into consumable APIs
  - Lets users manage applications not machines

Kubernetes Infrastructure Architecture



Controller cluster nodes are Kubernetes nodes but they don't run user workloads. These controller nodes run the Kubernetes processes that coordinate the cluster, such as the API server, scheduler and controller manager.

Controller nodes also run the master etcd processes that make up the etcd cluster. Because the controller nodes run these master etcd processes there needs to be either a single controller node and two additional worker nodes, when running a cluster in single-controller mode, or at least three controller nodes when running a cluster in multi-controller mode. The requirement of at least three nodes is due to the quorum requirements of an etcd cluster. If you have a single controller then the two additional etcd master processes will run on worker nodes meaning you would have to have at least three total cluster nodes in the cluster (1 controller and 2 worker). When in multi-controller mode, you must have at least three controller nodes to run the master etcd processes so that they don't have to run on any worker nodes. If you have more that three controller nodes you will still have only three etcd master processes running on three of the controller nodes. Additional controller nodes beyond the three that run the ectd processes will only be used to scale out the Kubernetes cluster coordination tasks. In practice you probably won't need more that three Kubernetes controllers in a cluster unless you have a very large cluster that is changing very often.

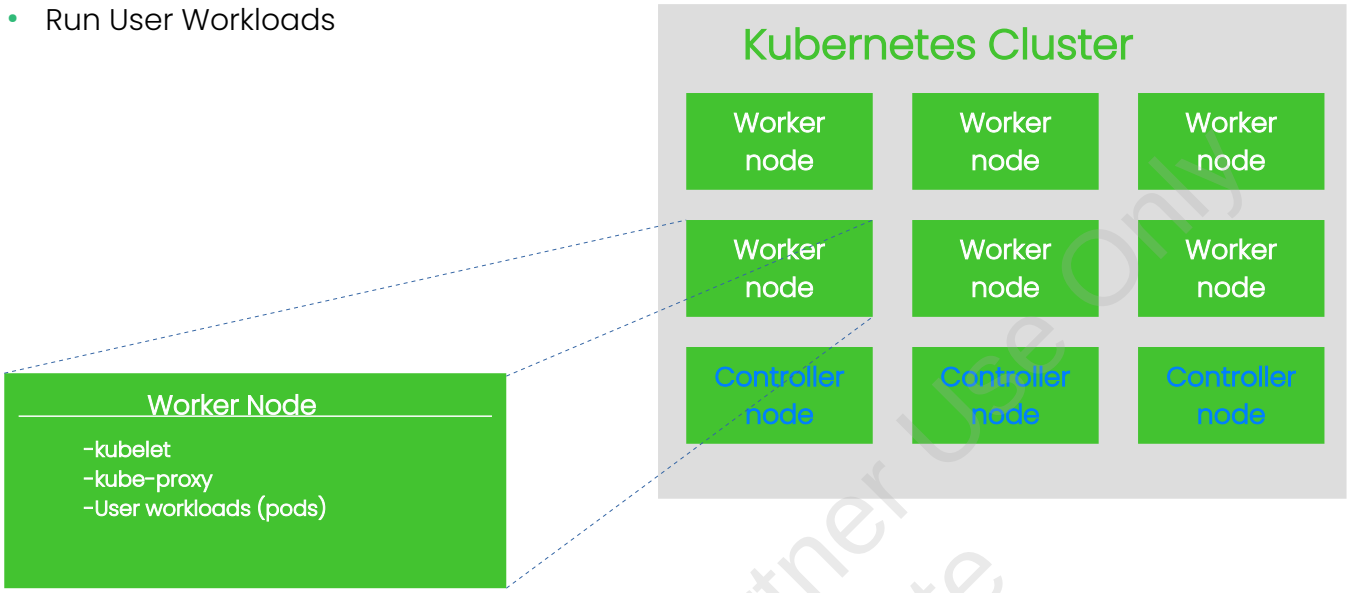
Kube-apiserver validates and configures data for the api objects which include pods, services, replication controllers, and others. It provides the frontend to the cluster's shared state through which all other components interact

Kube-controller-manager watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state

Kube-scheduler schedules workloads for the cluster. It is a smart process that is policy-rich and topology-aware

Cluster Nodes: Worker

- Run User Workloads



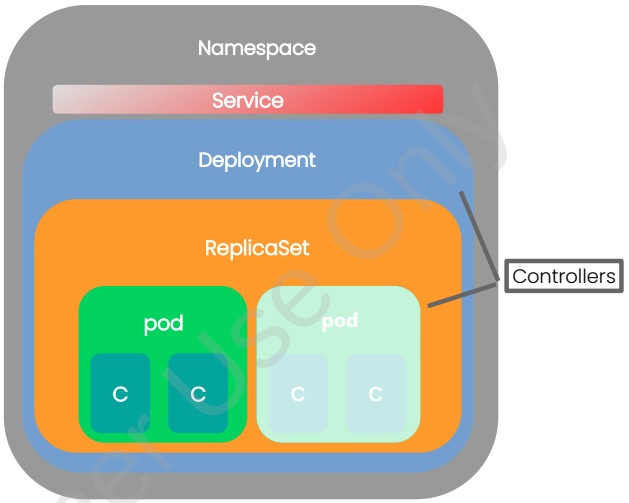
Worker cluster nodes are only used to run infrastructure service workloads and user workloads*. The infrastructure services are workloads that provide additional services for the cluster infrastructure such as Kube_DNS, Dex, etc. These infrastructure workloads can ether be deployed as part of the cluster deployment or as add-ons after the fact.

* Except in the case of a single-controller cluster where two of them will run master etcd processes.

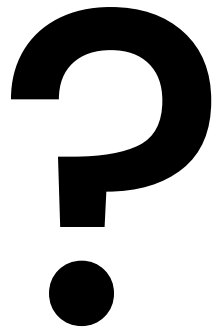
Kubernetes Application Architecture

Logical Hierarchy

Component	Description
container	- a sealed application package
pod	- a small group of tightly coupled containers - the basic unit deployed on Kubernetes
controller	- a reconciliation loop that drives the current state toward the desired state
service	- network/routing policy that directs traffic to a deployed application
namespace	- logically separated groups of resources



Questions:



Q. What is Kubernetes?

A. Platform for orchestrating and managing container workloads.

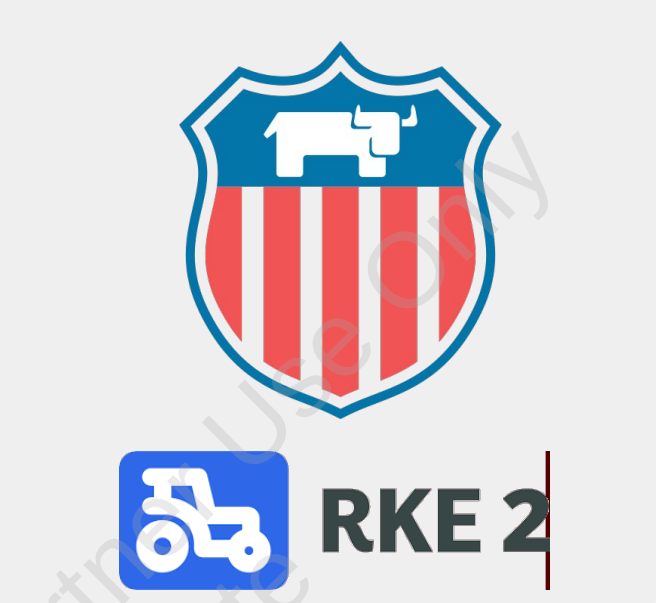
Understand SUSE Rancher Kubernetes Offerings

RKE

- CNCF certified distribution of Kubernetes
- Based on upstream Kubernetes but with 24x7 enterprise support available
- Simplified installation
- Easy, safe, atomic upgrades
- Uses Docker as the container engine

RKE Government (RKE 2)

- CNCF certified version of Kubernetes built for government agencies
- FIPS-enabled alternative to RKE
- Uses Contanerd as the container engine



K3S

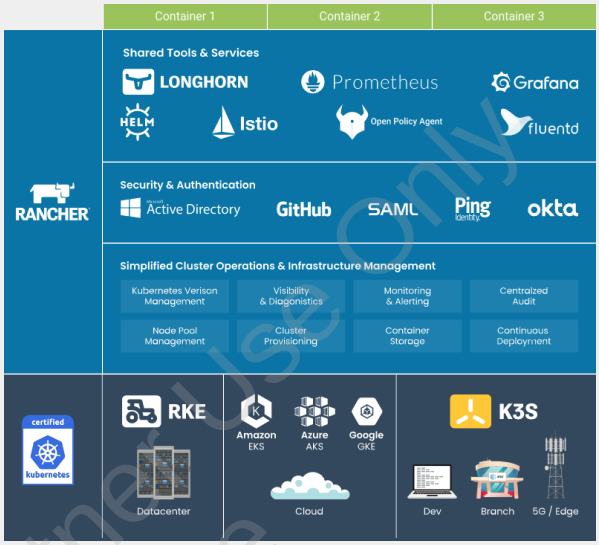
- Lightweight CNCF certified distribution of Kubernetes
- Perfect for deploying on Edge, IoT, CI and ARM
- Packaged as a single binary to reduce dependencies and simplify install and updates

Rancher

- Enterprise ready platform for managing Kubernetes
- Can both deploy and manage Kubernetes clusters locally and in the cloud
- Supports managing any certified Kubernetes distribution
- Provides simplified cluster operations and security, policy and user management

Shared Tools and Services

Rancher provides a rich catalog of services for building, deploying and scaling containerized applications, including app packaging, CI/CD, logging, monitoring and service mesh.



SUSE

Section: 3

Kubernetes Administration

Section Objectives:

1 Understand Basic Kubernetes Commands

2 Work with Namespaces

3 Understand Kubernetes Manifests

4 Understand Multi-pod Deployment

5 Work with Deployments

6 Configure Networking for Applications

7 Use Environment Variables with Applications

8 Use ConfigMaps

9 Work with Secrets

10 Work with Labels and Selectors

11 Configure Node Affinity in Kubernetes

12 Scale Out Applications

Understand Basic Kubernetes Commands

The `kubectl` Command

- Main command for interacting with Kubernetes
- Default configuration file: `~/.kube/config`

Syntax: `kubectl` *VERB RESOURCE* [*OPTIONS*]

Verbs are commands that are used on API Resources.

API Resources are objects that are interacted with in the cluster.

Find more information at: https://kubernetes.io/docs/reference/kubectl/overview/

Basic Commands (Beginner):

create	Create a resource from a file or from stdin.
expose	Take a replication controller, service, deployment or pod and expose it as a new Kubernetes Service
run	Run a particular image on the cluster
set	Set specific features on objects

Basic Commands (Intermediate):

explain	Documentation of resources
get	Display one or many resources
edit	Edit a resource on the server
delete	Delete resources by filenames, stdin, resources and names, or by resources and label selector

Deploy Commands:

rollout	Manage the rollout of a resource
scale	Set a new size for a Deployment, ReplicaSet, Replication Controller, or Job
autoscale	Auto-scale a Deployment, ReplicaSet, or ReplicationController

```
Cluster Management Commands:
  certificate     Modify certificate resources.
  cluster-info    Display cluster info
  top             Display Resource (CPU/Memory/Storage) usage.
  cordon          Mark node as unschedulable
  uncordon        Mark node as schedulable
  drain           Drain node in preparation for maintenance
  taint           Update the taints on one or more nodes

Troubleshooting and Debugging Commands:
  describe        Show details of a specific resource or group of resources
  logs            Print the logs for a container in a pod
  attach                Attach to a running container
  exec            Execute a command in a container
  port-forward    Forward one or more local ports to a pod
  proxy           Run a proxy to the Kubernetes API server
  cp              Copy files and directories to and from containers.
  auth            Inspect authorization

Advanced Commands:
  diff            Diff live version against would-be applied version
  apply           Apply a configuration to a resource by filename or stdin
  patch           Update field(s) of a resource using strategic merge patch
  replace         Replace a resource by filename or stdin
  wait            Experimental: Wait for a specific condition on one or many resources.
  convert         Convert config files between different API versions
  kustomize       Build a kustomization target from a directory or a remote url.

Settings Commands:
  label           Update the labels on a resource
  annotate        Update the annotations on a resource
  completion      Output shell completion code for the specified shell (bash or zsh)

Other Commands:
  api-resources         Print the supported API resources on the server
  api-versions    Print the supported API versions on the server, in the form of "group/version"
  config          Modify kubeconfig files
  plugin                Provides utilities for interacting with plugins.
  version         Print the client and server version information
```

Help Resources

Command	Description
`kubectl --help`	-help with the kubeclt command and verbs
`kubectl api-resources`	-list all API resources with their related verbs
`kubectl explain RESOURCE`	-provides more details about the resource

Listing Commands

The `get` verb is useful for listing resources and information about them.

Syntax: `kubectl get` *RESOURCE*

Command Examples	Description
`kubectl get nodes`	-list all nodes in the cluster
`kubectl get pods`	-list all pods in your current namespace
`kubectl get deployments`	-list all deployments in your current namespace

Option Examples	Description
`-n | --namespace`	-specify a namespace
`-o wide`	-display extra details

`kubectl get` is a listing command. It simply lists all of the api-resources that you tell it to list.

Most api-resources exist in namespaces. -n or `--namespace` will tell the get command which namespace to get the resources from. `--all-namespaces` will get resources from all namespaces.

`-o wide` will provide more verbose output from the command.

Description Commands

The verb `describe` always requires both the type of thing that you are describing and the name of the thing that you are describing.

Syntax: `kubectl describe` *RESOURCE*

Command Examples	Description
`kubectl describe node`	-provide detailed info for a node
`kubectl describe pod`	-provide detailed info for a pod
`kubectl describe deployment`	-provide detailed info for a deployment

Option Examples	Description
`-n | --namespace`	-specify a namespace
`-o wide`	-display extra details

`kubectl describe` shows details of a specific resource or group of resources.

Describe pulls out the most important information about a Resource from the Resource itself and related Resources, and formats and prints this information on multiple lines.

Aggregates data from related Resources
Formats Verbose Output for debugging

The `kubectl` describe command should be the first command used to troubleshoot or get more information about an api-resource.

For example, if a pod is failing, `kubectl describe` that pod will give you the majority of information needed to begin troubleshooting an issue.

Deployment Commands

The `create` and `apply` verbs are used to create and update resources in the cluster.

Syntax: `kubectl create` *RESOURCE*
`kubectl create -f` *MANIFEST*
`kubectl apply -f` *MANIFEST*

Command Examples	Description
`kubectl create namespace`	-create a new namespace
`kubectl create -f pod.yaml`	-create a pod from a manifest
`kubectl apply -f pod.yaml`	-create/update a pod from a manifest

`kubectl create` will create resources directly from the command line or from a manifest for the purposes of development or debugging.

`kubectl apply` is a command that will update an app to match state defined locally in a manifest file. This includes creating a new app.

It is:
Fully declarative - don't need to specify create or update - just manage files
Merges user owned state (e.g. Service selector) with state owned by the cluster (e.g. Service clusterIp)

`kubectl create` and `kubectl apply` seem to be redundant. If you are simply deploying a manifest, they can be used interchangably. However in practice, `kubectl create` should only be used in development environments where it is not so important to be able to keep track of every manifest.

For example:

If you create a new namespace with `kubectl create newproject`, the newproject namespace doesn't necessarily need to be repeated again and again in a development environment.

If you need the newproject namespace deployment to be repeatable then it would be best to create it in a manifest and deployed with: `kubectl apply -f newproject.yaml`

Question: Does it matter if you deploy a manifest with `kubectl create` or `kubectl apply`? Answer: No

However if you need to update any resource with a manifest, then you must always use `kubectl apply` and never `kubectl create` because that isn't something that `kubectl create` can do.

Best practice:

`kubectl create` for dev environments that need resources to be created quickly.

`kubectl apply` for everything else and keep your manifest files safe or in source control so they can be reused or updated as needed.

Delete Commands

The `delete` verb is used to remove resources from the cluster.

Syntax: `kubectl delete` *RESOURCE_TYPE RESOURCE*

Command Examples	Description
`kubectl delete namespace`	-delete a namespace
`kubectl delete pod`	-delete a pod
`kubectl delete deployment`	-delete a deployment

Option Examples	Description
`-f`	-specify a yaml file describing objects to delete
`-n | --namespace`	-specify a namespace

`kubectl delete` will delete a single resource that does not have dependencies it them to another resource.

For example, if you delete a single pod that is not a part of the deployment, then that pod will be deleted. If you delete a pod that is a part of a deployment, the pod will be recreated in a few seconds. This is useful if a pod is non-responsive and you need to recreate it. However, if you want to delete the entire deployment, then you should run:

```
kubectl delete deployment myapp
```

System resources can not be deleted.

For example:

```
> kubectl delete namespace kube-system
Error from server (Forbidden): namespaces "kube-system" is forbidden: this namespace may
not be deleted
```

Basic Troubleshooting Commands

The following commands can be used to do some basic troubleshooting of resources.

Command Examples	Description
`kubectl logs myapp`	-receive app logs from a pod
`kubectl exec -it myapp -- bash`	-launch a shell in a pod and connect to it
`kubectl cp myapp:/var/log/message /home/tux`	-copy a file (/var/log/messages) from a pod to the local filesystem
`kubectl cp testscript.sh myapp:/usr/local/bin`	-copy a file (testscript.sh) into a pod
`kubectl edit service myservice`	-opens the default editor and edits the raw yaml for a service

`kubectl logs` will get the application logs for a specific pod. For example, if it is a mysql pod, it will provide the output of `/var/log/mysql.log`

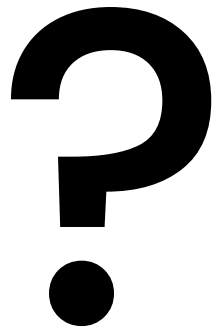
`kubectl exec` will execute a specific binary in a pod
`-i` is interactive
`-t` is in a new terminal
bash is the /bin/bash shell

This is a common way to inspect a running pod. Some pods do not include /bin/bash, some only have the smaller /bin/sh, and some will have no shell application at all.

`kubectl cp` will allow you to copy file into or out of a running pod. This is rarely used in production, but can be a quick fix for developers trying new things.

`kubectl edit` is similar to `kubectl cp` in that it is not usually used in a production environment. It is more of a quick fix for troubleshooting. It also only edits a single instance of a resource. If you edit a single pod's yaml but you have 5 instances of it running, only the single instance will be edited. The rest will not change. In order to change permanently, the edit will need to be in a manifest and redeployed.

Questions:



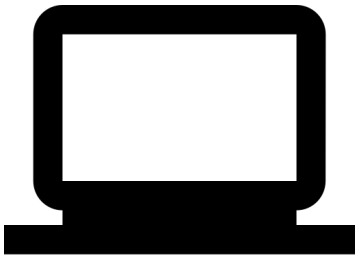
Q. What is the kubectl command used for?

A. The command use to interact with and administer Kubernetes.

Q. What is the default config file for the kubectl command?

A. ~/.kube/config

Exercises:



3-1: Use basic kubectl Commands

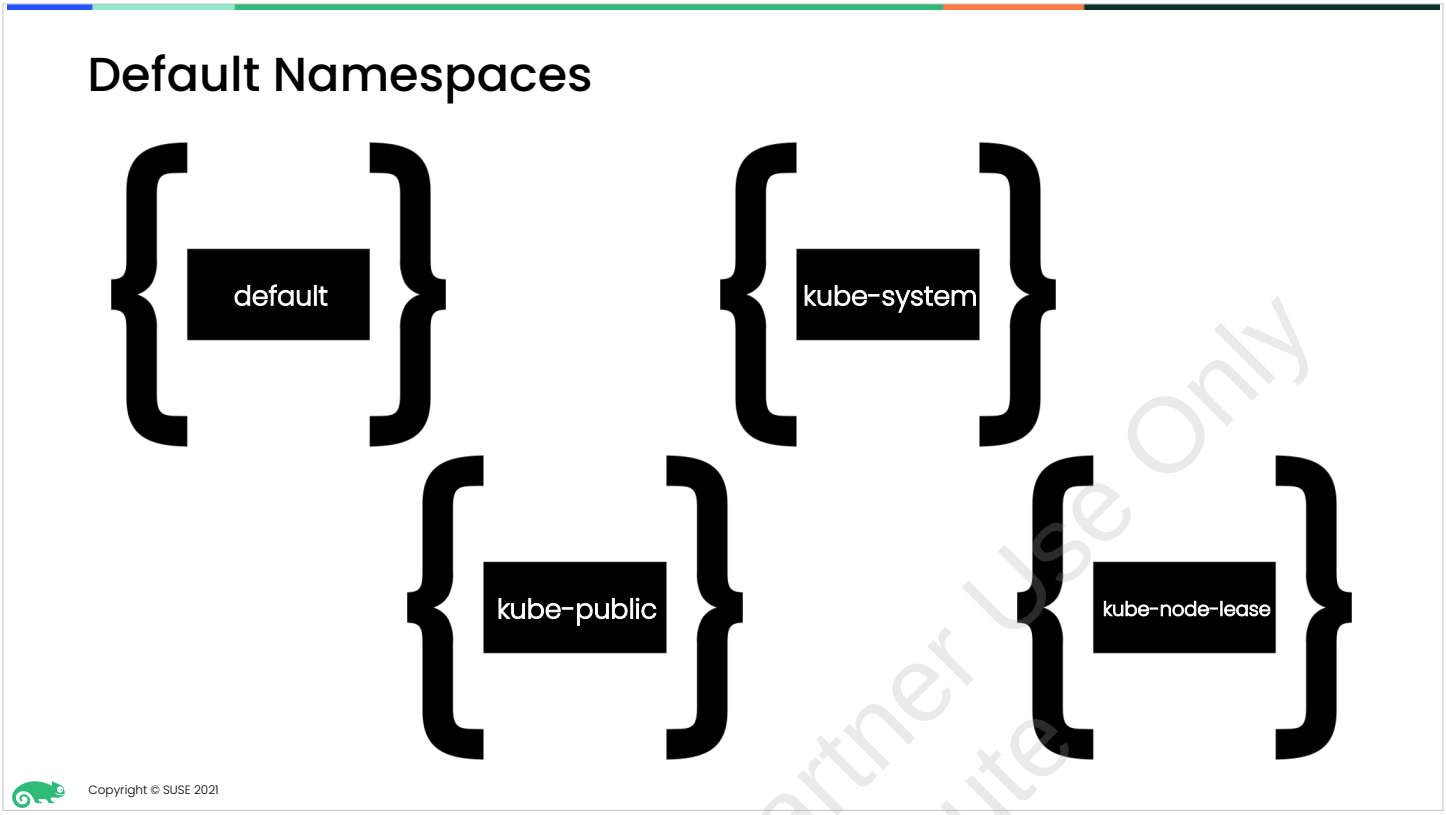
Work with Namespaces

What are Namespaces?

- An abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster

- Organize objects in a cluster and provide a way to divide cluster resources

  (Resources names must be unique within a namespace, but not necessarily across namespaces)



default

The default namespace for objects with no other namespace

kube-system

The namespace for objects created by the Kubernetes system

kube-public

This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

kube-node-lease

This is a placeholder for future functionality.
See: https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/0009-node-heartbeat.md

Is Everything in a Namespace?

Answer: No

Some resources in Kubernetes do not exist in Namespaces

Examples:

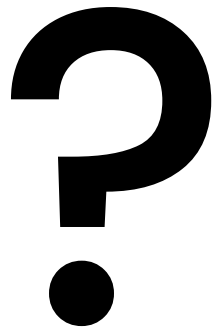
Nodes represent servers or VMs and it doesn't make sense for them to be in a namespace

Persistent Volumes can be used by resources in any namespace and are not limited to just one

The kube-system Namespace

- The namespace for objects created by the Kubernetes system
- Created when the Kubernetes cluster is first created
- All system pods, services, and other resources will be created in this namespace

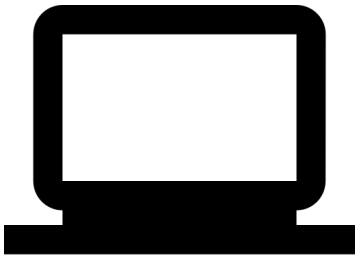
Questions:



Q. What is a Namespace and how is it used in Kubernetes?

A. Namespaces are a way to organize objects in a cluster or divide cluster resources into virtual clusters on a physical cluster.

Exercises:



3-2: Work with Namespaces in Kubernetes

Understand Kubernetes Manifests

Purpose of Manifests

- Files that describe how Kubernetes should configure objects or even the cluster itself
- Simpler than providing each instruction manually via the API or kubectl
- Created/stored in YAML format
- Designed for developers
- Easy to integrate into source control

Although you deploy directly via the API/Kubectl this would become cumbersome for complex deployments and hard to maintain. Having a file that describes an app/deployment/service makes sense.

Manifest Structure

```
apiVersion:
kind:
metadata:
  labels:
spec:
  selector:


  template:
```

Manifest Structure - Examples

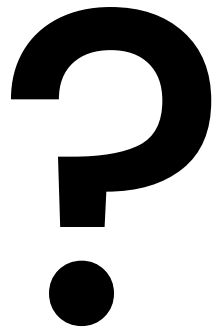
```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    owner: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - conainerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30000
  selector:
      app: nginx
```

Best Practices for Manifests

- One manifest deploys one component
- Use versioned images for pods
- Always use a Deployment (even for 1 pod)
- Define/use environment variables

Questions:



Q. What are manifests in Kubernetes?

A. Yaml files that contain definitions for objects in Kubernetes.

Q. How are manifests used in Kubernetes?

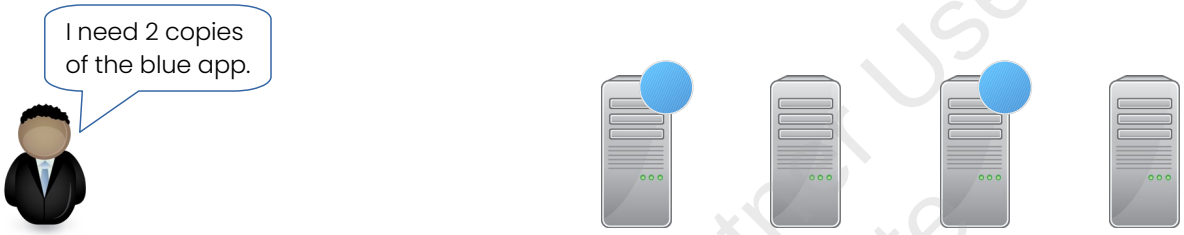
A. Used to crete and modify objects in Kubernetes.

Understand Multi-pod Deployment

ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time.

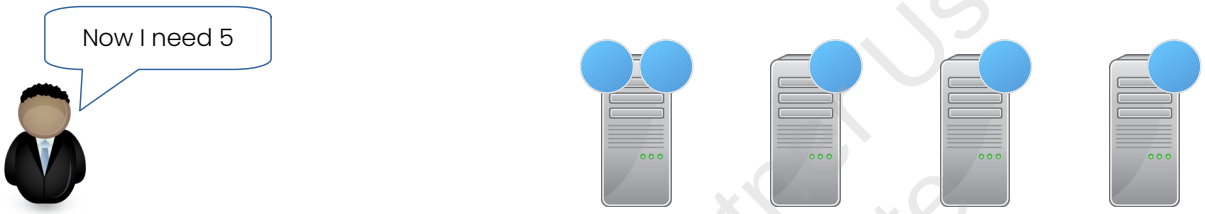
Often used to guarantee the availability of a specified number of identical Pods.



ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time.

Often used to guarantee the availability of a specified number of identical Pods.



When Should a ReplicaSet Be Used?

- When you need to ensure that a specified number of pod replicas are running at a given time
- When you need the ability to scale a certain set of pods

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

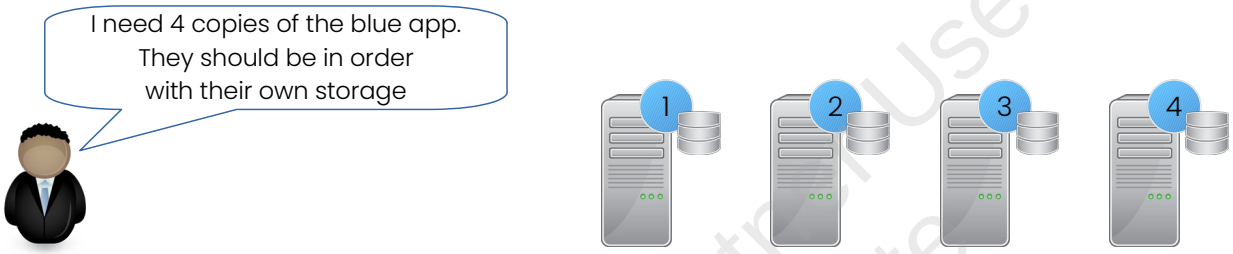
This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section.

StatefulSet

Manages Pods that are based on an identical container spec.

Maintains a sticky identity for each of their Pods.

Pods are created from the same spec, but are not interchangeable.
(Each has a persistent identifier that it maintains across any rescheduling)



When Should a StatefulSet Be Used?

StatefulSets are valuable for applications that require one or more of the following:

- Stable, unique network identifiers
- Stable, persistent storage
- Ordered, graceful deployment and scaling
- Ordered, automated rolling updates

In the above, stable is synonymous with persistence across Pod (re)scheduling. If an application doesn't require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application using a workload object that provides a set of stateless replicas. Deployment or ReplicaSet may be better suited to your stateless needs.

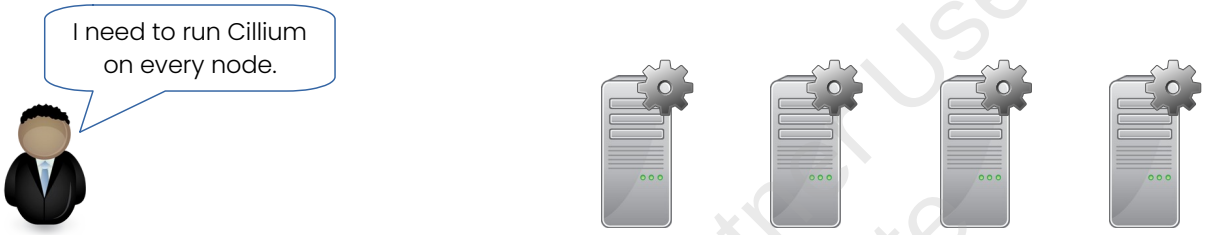
DaemonSet

Ensures that all (or some) Nodes run a copy of a Pod.

As nodes are added to the cluster, Pods are added to them.

As nodes are removed from the cluster, those Pods are garbage collected.

Deleting a DaemonSet will clean up the Pods it created.



If you need a specific application to be run on every node, it would be best to use a DaemonSet. When a new node is created, a new instance of the application will be installed on the node after adding it to the new node.

When Should a DaemonSet Be Used?

DaemonSets are valuable for cluster services that need to be running on every node or a specific subset of nodes.

A DaemonSet ensures that a replica of a service is running on either all cluster nodes or on a specified subset of nodes. This is particularly useful for cluster services such as networking services that need to be present on all nodes where user workloads could be run in order to forward their network traffic. Networking services are not the only types of services that can benefit from this type. Basically any service that needs to be running on a node as part of that node's default set of services should be deployed as a DemonSet. This also ensures that any new nodes of the type that run these services will automatically get an instance of the DaemonSet service when the node is deployed. When a node is removed the instance of the service on that node is removed and garbage collected and not restarted on another node.

Deployment

- A Deployment provides declarative updates for Pods and ReplicaSets
- The use of plain ReplicaSets are being phased out in favor of Deployments as they create ReplicaSets automatically
- Have the ability to use simple and rolling updates
- They can either be Stateful or Stateless

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, the Kubernetes developers suggest using Deployments instead of ReplicaSets.

Stateful vs Stateless Deployments

Stateful

- A stateful application requires permanent storage.
- This storage is usually a network-based solution.

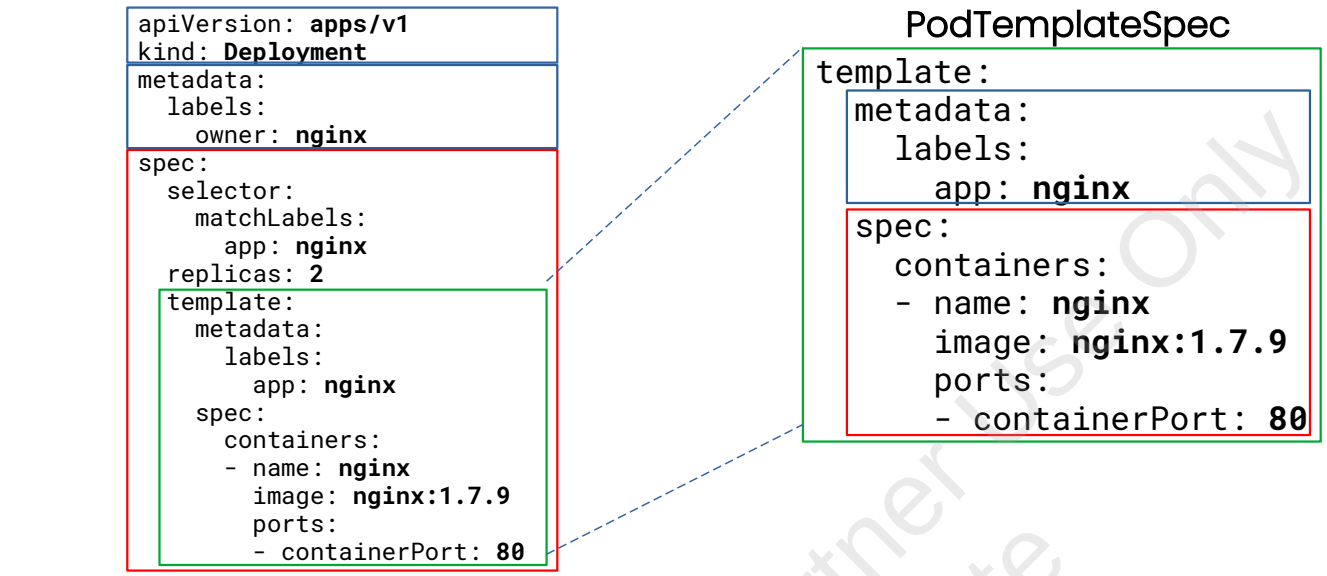
Stateless

- A stateless application does not need a permanent storage solution.
- Any temporary storage is within the pod itself.

Deployment Use Cases

- Create a Deployment to rollout a ReplicaSet
- Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment
- Rollback to an earlier Deployment revision if the current state of the Deployment is not stable
- Scale up the Deployment to facilitate more load
- Pause the Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout
- Use the status of the Deployment as an indicator that a rollout is stuck

Deployment Manifest Layout



Deployment manifests declare the following:

The name of the deployment: `nginx-deployment`
Any labels associated with it: `nginx`
The number of replicas associates with the deployment: 2
The container image that will be used to create the pod: `nginx:1.7.9`
The tcp port that the pod will be expecting network traffic to be incoming on: `80`

It is also Stateless because it is not using any kind of external storage. Once the deployment is deleted, any information associated with it will be removed also.

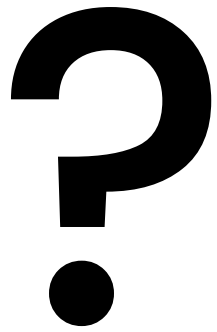
These are just the basics. It is also possible to declare how to update a file, either all at once or via rolling updates.

Notice the selected section of the manifest above is known at the PodTemplateSpec. This part of the manifest is like a manifest inside of a manifest as it has its own `metadata` and `specification` sections. This configuration defines the pod(s) that will be created as part of the Deployment.

The Selector in the Deployment manifest specifies a label (matchLabels) that corresponds with the labels set in the `metadata.labels` section of the PodTemplateSpec that tells the Deployment which pods belong to it.

If a new version of a Deployment manifest is deployed and anything in this section is changed, this will trigger a new state in the Deployment that will need to be rolled out to all pods

Questions:



Q. What are the 4 common controllers used for multi-pod deployment?

A. ReplicaSet, StatefulSet, DaemonSet, Deployment.

Q. What is the difference between a stateless and stateful deployment?

A. Stateful requires persistent storage and stateless do not.

Work with Deployments

Deploy a Simple Stateless Application

Command: **`kubectl apply -f nginx-deployment.yaml`**

Manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    owner: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - conainerPort: 80
```

Display the Status of a Deployment

Command: `kubectl get deployment`

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   2/2     2            2           5m33s
```

`kubectl get deployments` provides:

The number of pods are that in a ready status and the number that have are expected
The number of pods that are currently updated. This is useful if there are a large number of pods that need to be updated to a new version
The number of pods that are available

The `-o wide` option provides in addition:

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES         SELECTOR
nginx-deployment   2/2     2            2           21m   nginx        nginx:1.7.9    app=nginx
```

The name and image of the containers
Any selector that is used with the deployment

Display the Details of a Deployment

Command: `kubectl describe deployment nginx-deployment`

```
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Mon, 24 Feb 2020 12:56:12 +0100
Labels:                 env=app
                        owner=nginx
Annotations:            deployment.kubernetes.io/revision: 1
                        kubectl.kubernetes.io/last-applied-configuration:
                          {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"labels":
{"env":"app","owner":"nginx"},"name":"nginx-deployment",...
Selector:               app=nginx
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        smt.example.com:5000/nginx:1.7.9
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-7db4d6564b (4/4 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  2m16s  deployment-controller  Scaled up replica set nginx-deployment-7db4d6564b to 2
```

`kubectl describe deployment` provides more detail on everything. It also provides the update strategy when moving from one version to another and it provides a list of events that have happened to the Deployment.

Delete a Deployed Application

Command: `kubectl delete deployment nginx-deployment`

or

`kubectl delete -f nginx-deployment.yaml`

As all our objects are linked via the deployment it will delete all of them (ReplicaSets, Service etc).

Note: This will not necessarily delete any persistent storage used depending on how the storage is configured.

Update a Deployed Application

Command: **`kubectl apply -f nginx-deployment.yaml`**

Updated Manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    owner: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.0
        ports:
        - conainerPort: 80
```

Update the existing yaml file

(or create a new one)

You can update any part of the deployment and Kubernetes will ensure the relevant changes are applied automatically when we rerun the apply command.

This is one case where you must use the `kubectl apply` command and never the `kubectl create` command.

You can either update the existing yaml file and rerun or you can create a new one and run it – Kubernetes won't care about the difference. Kubectl has no concept of file management. This gives you power when combined with source control and file versioning.

Rolling Upadate a Deployed Application

Updated Manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    owner: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.0
        ports:
        - conainerPort: 80
```

Update the image and ensure that there is no chance of disruption:

- Only take down 25% of pods at a time for patching
- Have as many as 2 extra pods running temporarily

This deployment manifest will update the image and ensure that there is no chance of disruption. It will only take down 25% of pods at a time for patching and may have as many as 2 extra pods running temporarily.

Advantages of Rolling Updates

- Old pods removed and new updated pods created

  (Note: The pods name will change when you run `kubectl get pods`)

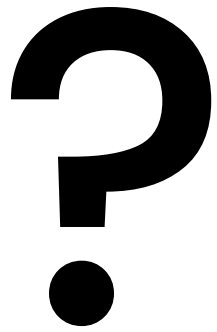
- Updates one pod at a time to ensure the replica set conditions are still met

- Ensure a minimum number of pods are always running

- Temporarily surge above the desired number of replicas

Kubernetes will remove old pods and create the new ones (note the pods name will change when you do kubectl get pods). It will do this one pod at a time to ensure the replica set conditions are still met (always 2 instances running). Always consider this when designing new applications to avoid impact on stateful requirements.

Questions:



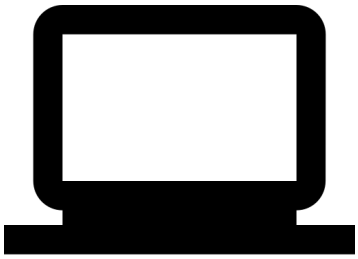
Q. In the `rollingUpdate:` section of the `strategy:` section of a deployment manifest, what does the `maxSurge:` property do?

A. Specify the number of addition pods that will be created duting the rolling update so that service can be maintained.

Q. What command would you use to remove a deployment and its pods?

A. `kubectl delete deployment <deployment _name>`

Exercises:



3-3: Deploy a Simple Pod Using a Deployment

3-4: Delete and Redeploy a Deployment

3-5: Update a Pod in a Deployment

Configure Networking for Applications

What are Services?

Enable apps to be accessed by users, the web, or even from other apps.

Provide a stable interface to your apps.

Services enable applications to be accessed via the network both from inside and outside of the cluster.

ClusterIP

The ClusterIP service type allows traffic inside of Kubernetes to that application.

Each pod has it's own internal IP.

By default, this IP is not available from outside of the cluster.

Not every application needs direct traffic from outside of the cluster.

ClusterIPs are provisioned from each node's CIDR (IP address range) that is set up by Cilium when the node is added to the cluster.

NodePort

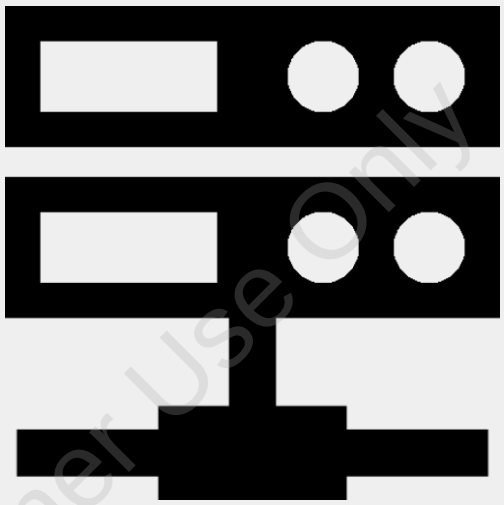
NodePort provides a unique port for an application.

The IP/FQDN of the service would be that of a kubernetes node in your cluster.

The Port is accessible on every node.

Example:

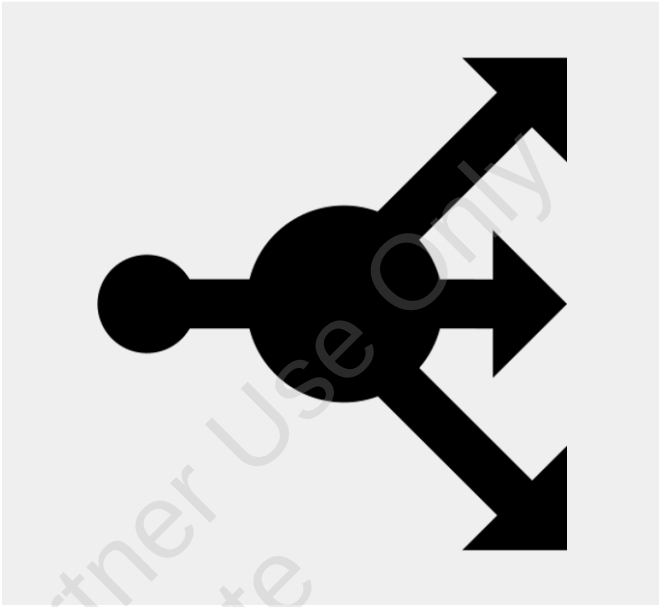
`http://worker01.example.com:31000`



LoadBalancer

Provides unique IP addresses per app.

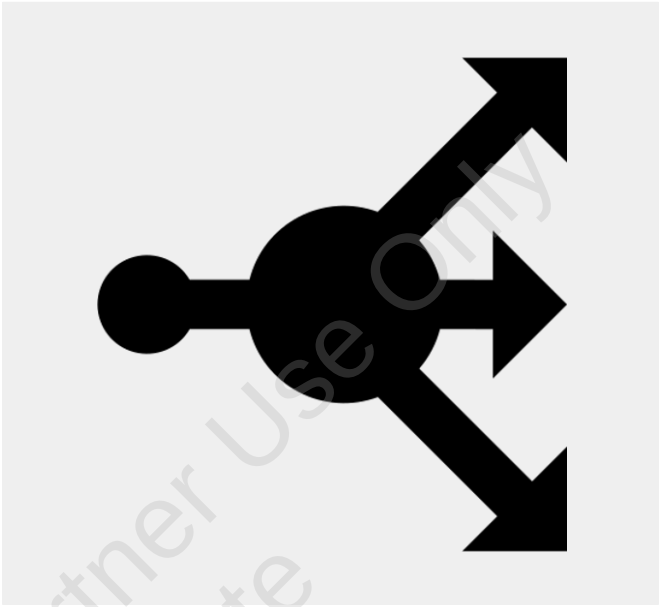
IPs come from a pool of available addresses that you allocate.



LoadBalancer

Public cloud providers have their own LoadBalancers that can be used. Some providers like AWS provide full FQDN instead of IPs for Kubernetes.

The LoadBalancer service type is not built into the Kubernetes cluster at the current time. This must be installed with an application like Metallb.



Metallb (Metal Load balancer) is a project that provides the Kubernetes load balancer service and can be installed in your cluster. However it is not currently supported by SUSE.

Example Service Manifests

ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  type: ClusterIP
  selector:
      app: mysql
```

NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30000
  selector:
      app: nginx
```

LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  type: LoadBalancer
  selector:
      app: tomcat
```

ClusterIP

In applications like databases, it is best to not have direct access from outside but applications inside of Kubernetes may need access to it.

Normally it is ideal for applications to be able to be accessed by users but you don't want a database to be available.

ClusterIP will allow that access from inside of the cluster but not outside.

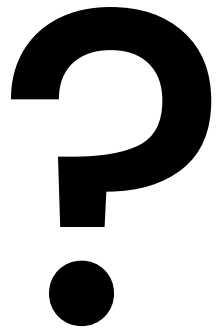
NodePort

If you uses the NodePort service type, you will need to specify the local port that the application is expecting traffic in on. You can also specify which NodePort to use in the 30000-39999 range or Kubernetes will automatically choose one for you.

LoadBalancer

If a load balancer application is installed on your cluster, then this type will cause the load balancer to provision an IP or FQDN to this application.

Questions:



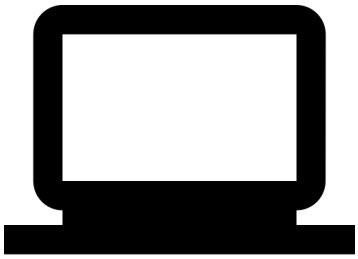
Q. What is a Service in Kubernetes and how is one used?

A. An object created in kuberenetes that allows an application to be accessed via the network.

Q. What are the 3 different types of services?

A. ClusterIP, NodePort, LoadBalancer.

Exercises:



3-6: Create and Edit a Service for an Application

Use Environment Variables with Applications

Environment Variables

- Environment variables can be set in pods when they are deployed
- Containers commonly use environment variables to configure their application at runtime

  (Check the container image description in the container image registry to determine which variables a container can/must use before creating a manifest)

When you create a Pod, you can set environment variables for the containers that run in the Pod. To set environment variables, include the env or envFrom field in the configuration file.

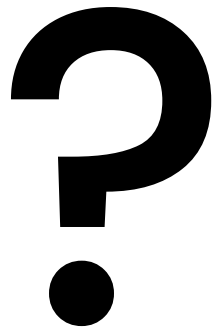
Environment Variable Example

Manifest:

```
kind: Pod
apiVersion: v1
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envars
spec:
  containers:
  - name: envar-demo-container
    image: gcr.io/google-samples/node-
hello:1.0
    env:
    - name: DEMO_GREETING
      value: "SUSE Rocks!"
```

In this pod, a new environment variable will be added **DEMO_GREETING** with the value of "**SUSE Rocks!**"

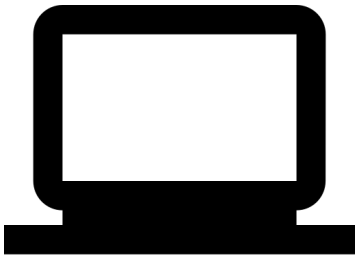
Questions:



Q. How are environment vairables used in Kubernetes?

A. They are used to set environmet variables inside of containers in a pod. These variables can then be used by applications running in the pod.

Exercises:



3-7: Use Environment Variables in a Pod

Use ConfigMaps

ConfigMaps

- Environment variables can be set in pods via ConfigMaps when they are deployed
- Containers commonly use environment variables to configure their application at runtime
- ConfigMaps allow for environment variables to bet set independently from the pod's manifest

When you create a Pod, you can set environment variables for the containers that run in the Pod. TO allow for a wider degree of flexibility these environment variables can be set in separate ConfigMaps rather theembedded in the pod spec. To set environment variables via ConfigMaps, include the envFrom field in the configuration file and then reference the name of the ConfigMap that contains the key:value pairs..

ConfigMap Example

Manifest:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: my-configmap
data:
    CONFIGMAP_VAR1: configmap_value_1
    CONFIGMAP_VAR2: configmap_value_2
```

In this ConfigMap, two new environment variables will be added:

- CONFIGMAP_VAR1 with the value of "configmap_value_1"
- CONFIGMAP_VAR2 with the value of "configmap_value_2"

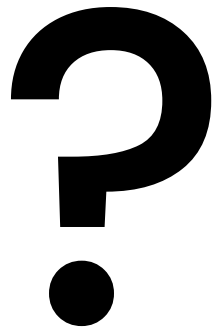
Example Pod Using a ConfigMap

Manifest:

```
kind: Pod
apiVersion: v1
metadata:
  name: configmap-demo
  labels:
    purpose: demonstrate-configmaps
spec:
  containers:
  - name: configmap-demo-container
    image: gcr.io/google-samples/node-
hello:1.0
    envFrom:
      - configMapRef:
          name: my-configmap
```

In this pod, new environment variables will be added by reading them from a ConfigMap named **my-configmap**

Questions:



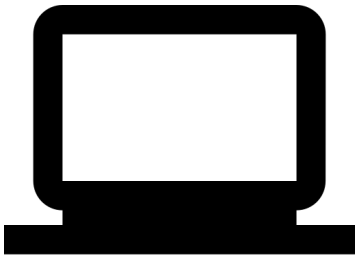
Q. What are ConfigMaps and how are they used in Kubernetes?

A. Objects in Kubernets that contain key-value pairs that can be exposed to pods resulting in environment variables being set in the containers.

Q. How are ConfigMaps different from environment variables set in pods?

A. ConfigMaps allow environment variables to be set independently from pods.

Exercises:



3-8: Use ConfigMaps with a Pod

Work with Secrets

Secrets in a Kubernetes Cluster

- Secrets allow you to securely store values in a Kubernetes cluster such as:
  - usernames/passwords
  - encryption keys
  - certificates
  - (etc)

- Secrets can be accessed by pods when needed

Kubernetes Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Storing confidential information in a Secret is safer and more flexible than putting it verbatim in a Pod definition or in a container image.

How are Secrets Accessed?

To use a secret, a Pod needs to reference the secret.

A secret can be used with a Pod in different ways:

- As files in a volume mounted on one or more of its containers
- As environment variables set in the pod
- By the kubelet via the imagePullSecret field when pulling images for the Pod

Define a Secret from a File

Any values you wish to store in the secret are entered into a file.

Example command for username/password:

```
mysecret1.txt: { username: user1
                 password: password1
```

```
kubectl create secret generic mysecret1 \
  --from-file=mysecret1.txt
```

Example command for public ssh key:

```
kubectl create secret generic sshpubkey \
  --from-file=~/.ssh/id_rsa.pub
```

When storing a file as a secret the entire file is stored as the data in the secret. The content of the file can be formatted in any way that would be needed by the application. The secret file in this example is very simple for the sake of demonstration. It simply contains a username and password that could be accessed by an app that needs them.

Command breakdown:

```
kubectl create secret generic mysecret1 –from-file=mysecret1.txt
```

`kubectl create secret`	Create a secret using specified subcommand
`generic`	Create a secret from a local file, directory or literal value. Other secret types include docker-registry and tls
`mysecret`	The name of the secret that Kubernetes will recognize. This can be anything
`--from-file=mysecret1.txt`	The file that contains the secret

Define a Secret as Key/Value Pairs from the CLI

Values defined from the command line are plain text.

Command:
```
kubectl create secret generic mysecret2 \
        --from-literal=username=user2 \
        --from-literal=password=password2
```

The values provided on the command line in the --from-literal flag are plain text.. When the secret is defined the key/value pairs are the data that is stored in the secret.

Define a Secret as Key/Value Pairs in YAML

Values defined in the yaml file must be base64 encoded

Commands: `echo -n "user2" | base64`

`echo -n "password2" | base64`

Command: `kubectl apply -f mysecret2.yaml`

Manifest:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret2
type: Opaque
data:
  username: dXNlcjI=
  password: cGFzc3dvcmQy
```

Base64 encoded values

The values stored in a yaml file must first be base64 encoded. When the secret is defined the key value pairs are the data that is stored in the secret.

List Secrets

List the defined secrets.

Syntax: `kubectl get secrets`

```
> kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-hkf9w   kubernetes.io/service-account-token   3      9d
mysecret1             Opaque                                1      55m13s
sshpubkey             Opaque                                1      55m42s
mysecret2             Opaque                                2      56m25s
```

A service-account-token is a secret that is assigned to a service account.

Service accounts automatically create and attach Secrets with API credentials
Kubernetes automatically creates secrets which contain credentials for accessing the API and 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 API server, this is the recommended workflow.

See the ServiceAccount documentation for more information on how service accounts work.

Describe Secrets

Display details of the secret.

Syntax: `kubectl describe secrets mysecret1`
`kubectl describe secrets mysecret2`

```
> kubectl describe secrets mysecret1
Name:         mysecret1
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
mysecret1.txt:  100 bytes
```

```
> kubectl describe secrets mysecret2
Name:         mysecret2
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  9 bytes
username:  5 bytes
```

Use Secrets

Secret Stored as a File

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-file-secret
spec:
  containers:
  - name: opensusepod
    image: opensuse/leap
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: secretmnt
        mountPath: "/mnt/secret"
  volumes:
  - name: secretmnt
    secret:
      secretName: mysecret1
```

Secret Stored as Key/Value Pairs

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-env-secret
spec:
  containers:
  - name: mypod
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret2
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret2
            key: password
```

Command breakdown:

Option 1:

```
kubectl apply -f mysecret.yaml
```

`mysecret.yaml`	The YAML file containing the secret definition

Option 2:

```
kubectl create secret generic supersecretsauce –from-file=secret.txt
```

`kubectl create secret`	Create a secret using specified subcommand
`generic`	Create a secret from a local file, directory or literal value. Other secret types include docker-registry and tls
`mysecret`	The name of the secret that Kubernetes will recognize. This can be anything
`--from-file=secret.txt`	The file that contains the secret

This secret file is very simple for the sake of demonstration. It simply contains a usernname and password that could be accessed by an app that needs them.

Access Secrets

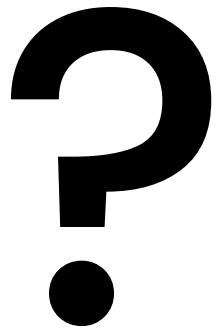
Secret Stored as a File

```
> kubectl exec -it pod-file-secret -- bash
pod-file-secret:/ # cat /mnt/secret/mysecret1.txt

username:user1
password:password1
```

Secret Stored as Key/Value Pairs

```
> kubectl exec -it pod-env-secret -- bash
pod-env-secret:/ # echo ${SECRET_USERNAME}
user2
pod-env-secret:/ # echo ${SECRET_PASSWORD}
password2
```

Questions:



Q. How are secrets used in Kubernetes?

A. The securily store values in the Kubernetes cluster.

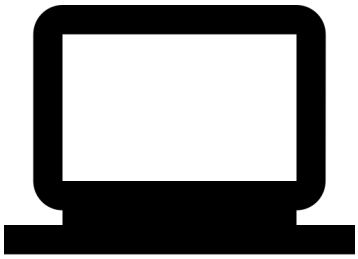
Q. In what ways can secrets be accessed in a pod?

A. Environment variables key value pairs, files.

Q. What typs of things can be stored as secrets?

A. usernames:passwords, ssh keys, certificates.

Exercises:



3-9: Define and Access Secrets as Volumes

3-10: Define and Access Secrets as Environment Variables

Work with Labels and Selectors

What are Labels and Selectors?

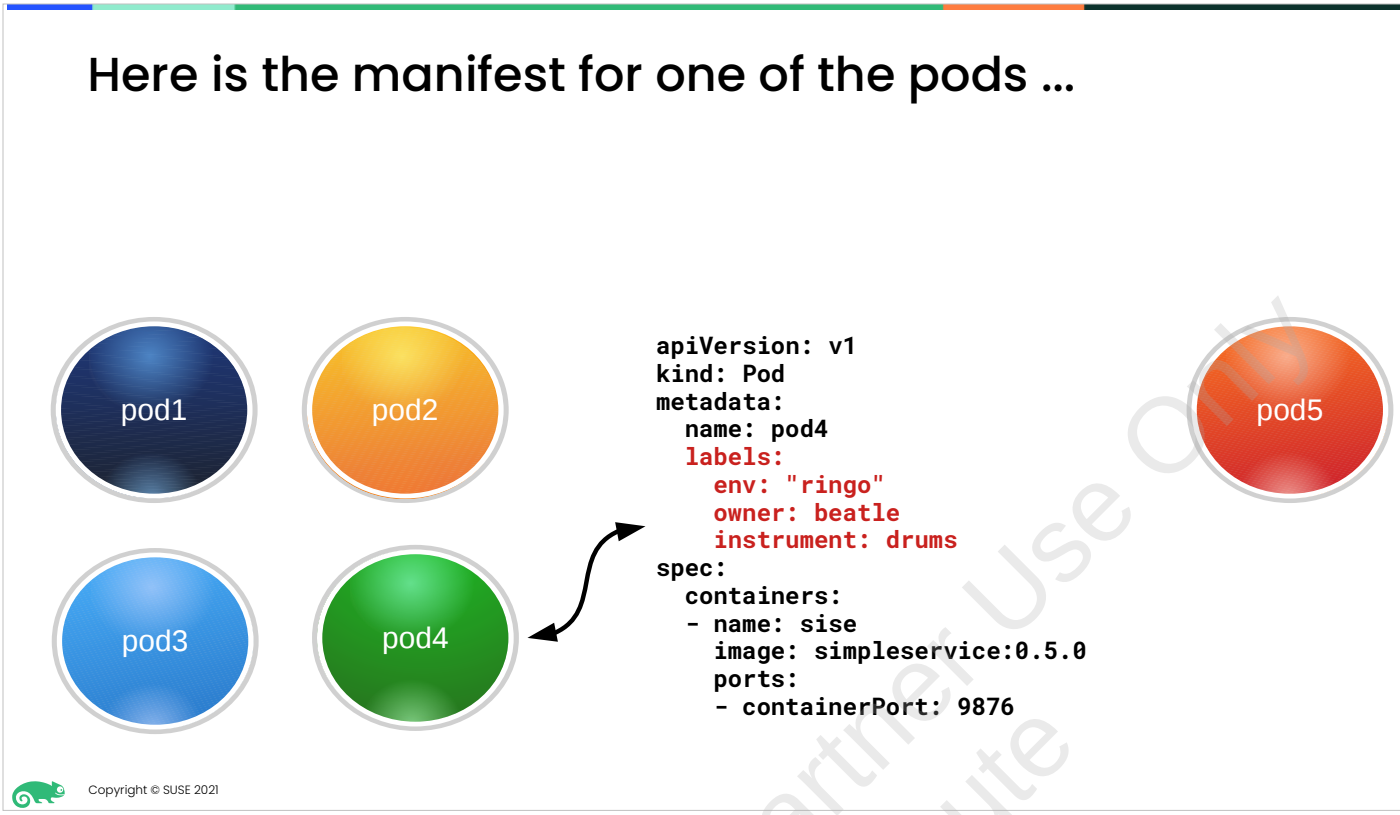
Labels

- Metadata that can be attached to API objects such as pods that generally represent identity
- They can be attributes in manifests or assigned manually

Selectors

- Functions in kubectl that can query API objects that use labels
- Queries can be a simple `get` command or it can be an action such as `delete` that applies only to the labels that match the query

Here are some identical pods ...

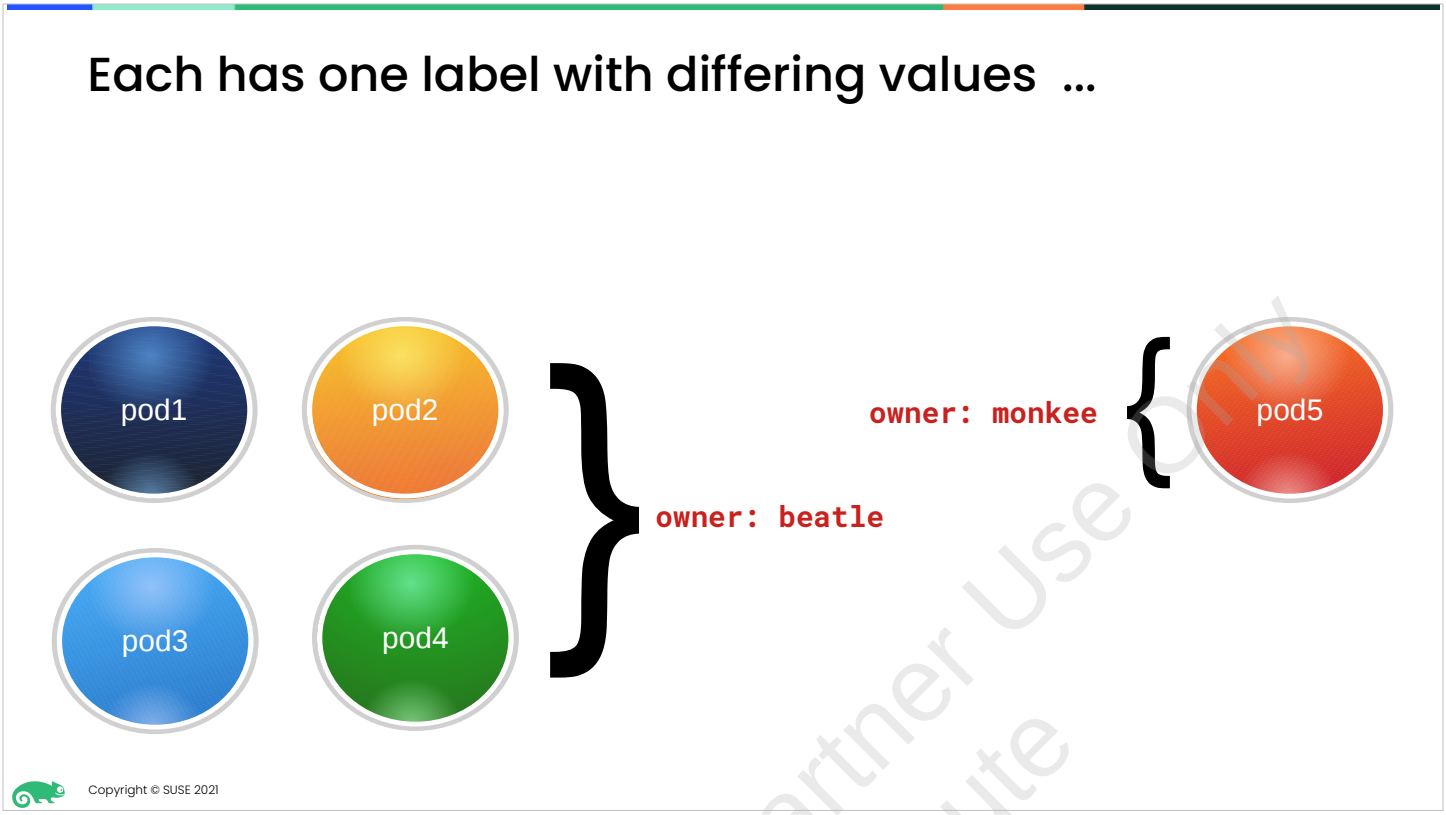


This is the manifest for one of the pods. In this manifest, the name of the pod is pod3. This isn't very descriptive, but it does have some more attributes.

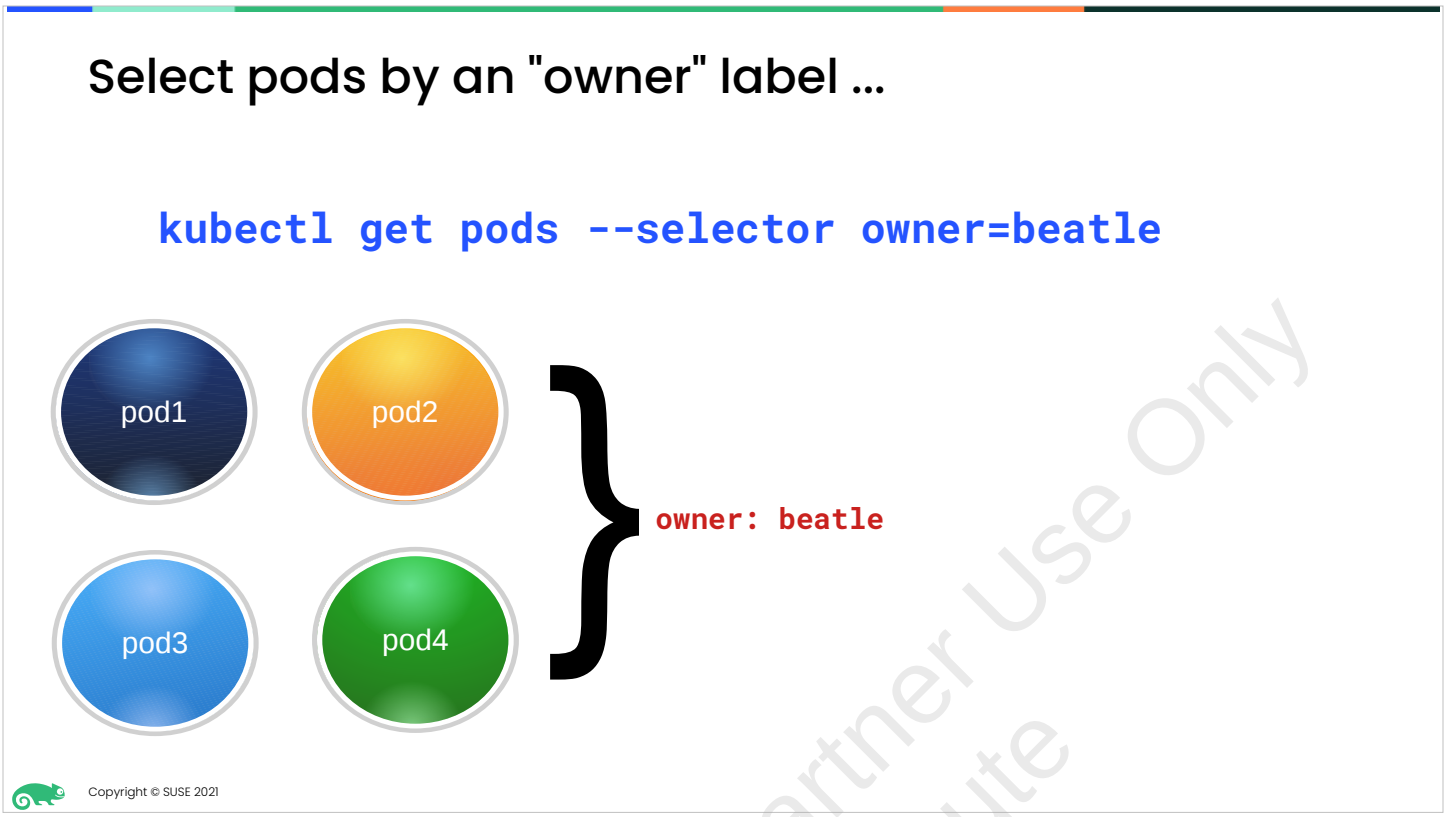
It has 3 labels:

```
labels:
   env: "george"
   owner: beatle
   guitar: rhythm
```

These labels can be anything. Both the labels and the values that they represent are open ended. This allows you the flexibility to use labels in the way that works best for your project.



All of the pods have labels for that owner. That would be similar to which version that they belonged to. If you need to specify which version of a pod is 1.0 and be able to act on it, it would be as simple as assigning it a label for 1.0. An older version of the pod could be another version.



Labels can be acted upon with selectors. The selector for `owner=beatle` will not include pods with a label of `owner=monkey`.

Select pods by a different "owner" label ...

```
kubectl get pods --selector owner=monkee
```

Similarly a selector for owner=monkey would not include any with an `owner=beatle` label.

Add a label to a pod ...

```
kubectl label pods pod5 surname=jones
```

Pods can also receive labels after they have been created. In this example, pod5 has been given a new label `surname=jones`. The new label surname can have a value that can be used to differentiate pod5 from other pods that share the label monkey.

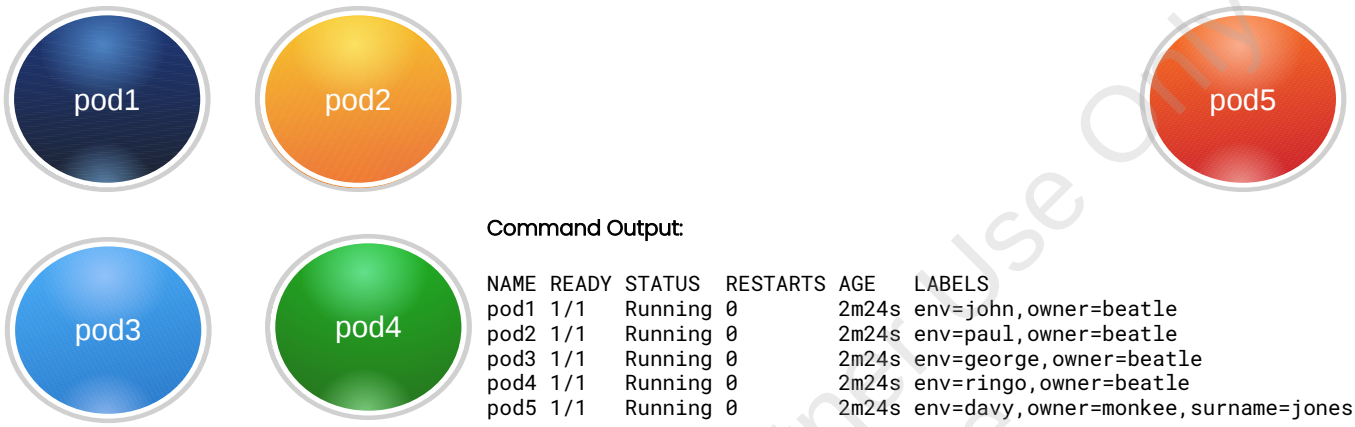
Test the new label ...

```
kubectl get pods --selector surname=jones
```

Selecting on the new label `surname=jones` should return the appropriat pod..

Get all labels ...

```
kubectl get pods --show-labels
```



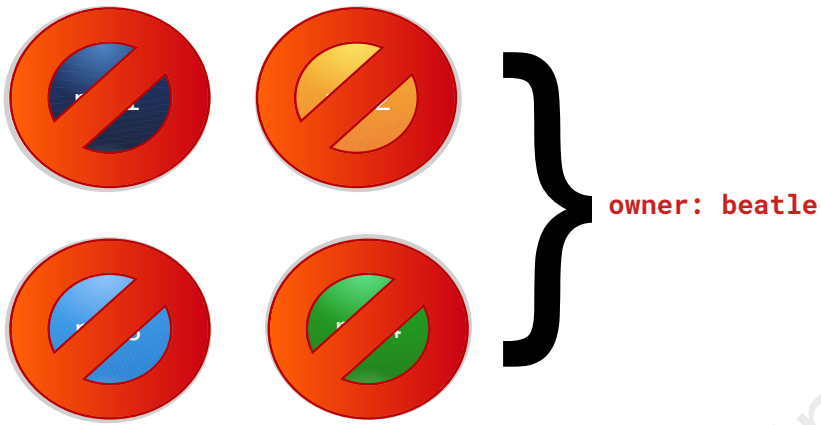
Command Output:

```
NAME READY STATUS  RESTARTS AGE   LABELS
pod1 1/1   Running 0        2m24s env=john,owner=beatle
pod2 1/1   Running 0        2m24s env=paul,owner=beatle
pod3 1/1   Running 0        2m24s env=george,owner=beatle
pod4 1/1   Running 0        2m24s env=ringo,owner=beatle
pod5 1/1   Running 0        2m24s env=davy,owner=monkee,surname=jones
```


Kubectl get pod can also list all labels using the `--list-labels` flag. This will help if there are multiple resources in a directory with similar names. By remembering to add labels, we can make working with pods much easier.

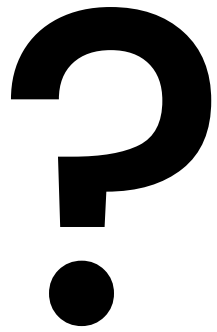
Delete pods using a selector ...

```
kubectl delete pods --selector owner=beatle
```



By deleting pods with the `owner=beatle` selector, any other pods will remain untouched. Listing and deleting aren't the only functions that can be used with labels and selectors, but they are some of the most common.

Questions:



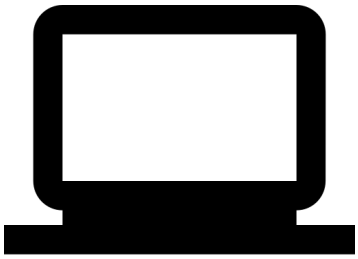
Q. What are labels in Kubernetes?

A. A way to tag objects in Kubernetes cluster.

Q. How are labels used in Kubernetes?

A. Selectors are used to select objects based on labels so that can be acted upon.

Exercises:



3-11: Work with Labels and Selectors

Configure Node Affinity in Kubernetes

Options for Pod/Node Affinity

- Kubernetes provides a couple of options to create pod/node affinity
- Each option's approach is slightly different though the end results are similar
- Options:
  - Node Selectors
  - Taints and Tolerations

NodeSelectors

- Labels are applied to Nodes
- Pods specs are updated with NodeSelectors for the corresponding labels
- Pods will prefer to be scheduled on nodes with labels that match their nodeSelctor
- Approach: Attraction vs Rejection

Command:

```
> kubectl label nodes worker01 disktype=ssd
```

Example Manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Taints and Tolerations

- Taints are applied to Nodes
- Pods specs are updated with tolerations for the corresponding taints
- The nodes will only allow pods with tolerations for their nodes' taints to be scheduled on them
- Approach: Rejection vs Attraction

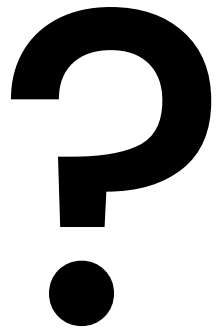
Command:

```
> kubectl taint nodes worker01 disktype=ssd:NoSchedule
```

Example Manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "disktype"
    operator: "Equal"
    value: "ssd"
    effect: "NoSchedule"
```

Questions:



Q. NodeSelectors are used to attract pods to nodes. (True or False)

A. True

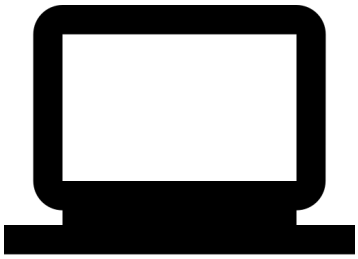
Q. Taints are used to repel pods away from nodes. (True or False)

A. True

Q. Tolerations are used to attract pods to nodes. (True or False)

A. False

Exercises:



3-12: Work with Node Selectors

3-13: Work with Taints and Tolerations

Scale Out Applications

Horizontally Scale an Application

- When deploying or redeploying an application, the replica spec can be defined
- If the number is changed and the manifest is redeployed, then the matching number of pod replicas will change also

Manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    owner: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.0
        ports:
      - conainerPort: 80
```

Horizontal scaling is adding more pods. Vertical scaling is adding more nodes or a new Kubernetes cluster.

The number of pods that you need is entirely dependent on the workload that you are expecting. For example, a background service such as for handling backups may only be needed once a day and have minimum requirements or a production web service may get a standard amount of traffic 46 weeks out of the month but the 6 weeks before the end of the year, it might rise 100-200% or more. If you know how much to expect depending on the time of the year, you can manually change the number of application pods. If you don't, you can opt for automatic scaling.

Horizontally Scale an Application

Apps can be scaled using the command line or by updating a manifest.

Command:

```
kubectl apply -f nginx-deployment.yaml
```

or

```
kubectl scale deployments nginx-deployment --replicas=4
```

The second command will scale a deployment to 4 pods without editing the manifest.

HorizontalPodAutoscaler

Monitors metrics of the cluster and uses these to automatically scale pod replicas out and back.

Example:

- App to be scaled: php-apache
- The minimum number of pods: 1
- The maximum number: 10
- If the CPU Utilization of the node is ≥ 50%, then more pods will be created

Note: Other metrics can also be used with an autoscaler

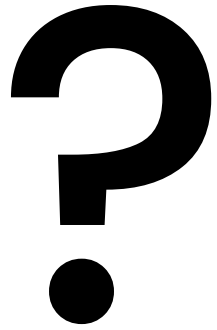
Manifest:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Requirements for autoscaling:

-Metrics server deployed
-Autoscaler
-An application that can be scaled

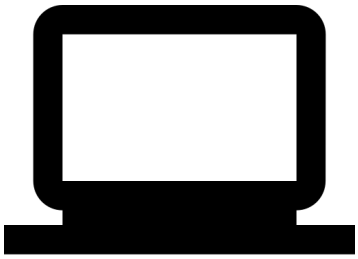
Questions:



Q. Individual pods can be scaled out on Kubernetes without a Deployment. (True or False)

A. False

Exercises:



3-14: Scale a Deployment

3-15: Configure Horizontal Pod Autoscaling

SUSE

Section: 4

Application Management on Kubernetes with Kustomize

Section Objectives:

1 Understand Kustomize Concepts

2 Use Kustomize to Deploy Applications

Understand Kustomize Concepts

What is Kustomize?

Simply said, Kustomize is:

- Native Kubernetes Configuration Management
- A declarative tool that works directly with yaml
- A template-free way to customize application configuration

How is Kustomize Kubernetes Native?

- Built into kubectl as of v 1.14 (`kubectl -k`)
- Operates on standard Kubernetes objects
- Uses plain yaml and standard manifest file structure
- Matches the same "declarative" ideology as Kubernetes

How is Kustomize Declarative?

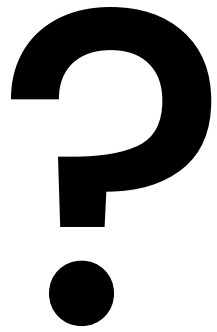
- You declare exactly what you want in standard yaml manifests and Kustomize generates the final manifest

How is Kustomize Template Free?

- You do not provide a templatized version of the manifest
- You do provide standard, valid, independently deployable Kubernetes manifests
- Acts as a yaml patching system rather than a template engine

  (It acts as a "stream editor" like sed to add/delete/update the final manifest)
- What you provide can be "stackable" in that each can be applied to the final manifest in layers

Questions:



Q. What is Kustomize and how is it used?

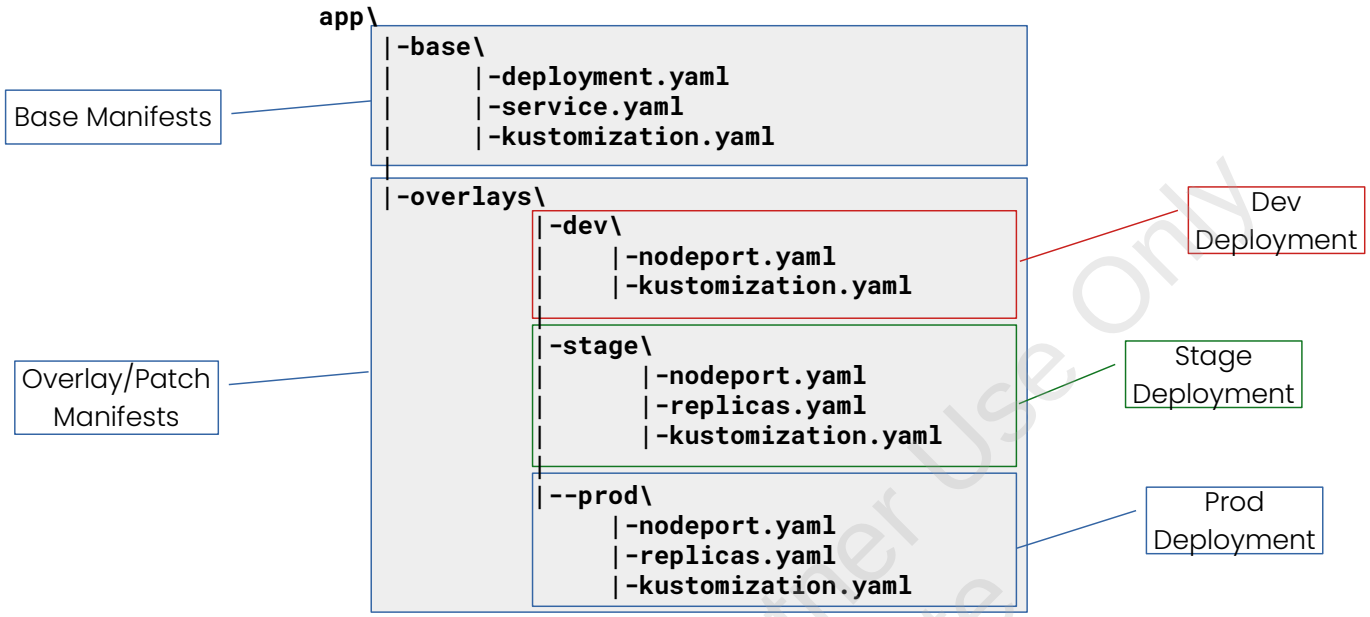
A. A native Kubernetes, template free way to customize application configuration.

Q. How is Kustomize "native" to Kubernetes?

A. As of Kubernetes 1.14 kubectl supports management of Kubernetes object using Kustomize.

Use Kustomize to Deploy Applications

Example Directory Structure



Example "Base" Manifests

deployment

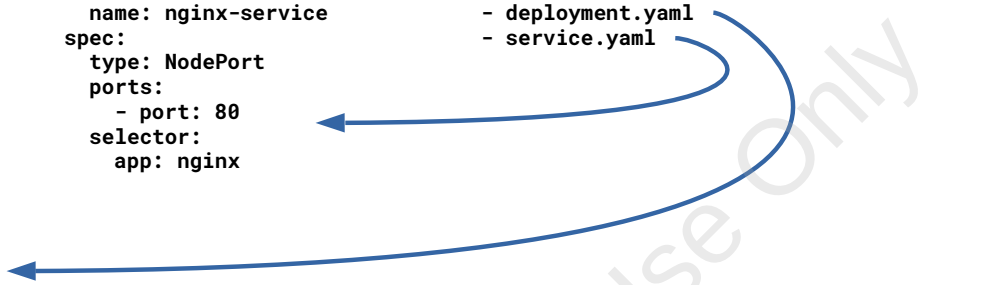
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    env: "apps"
    owner: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

service

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  ports:
    - port: 80
  selector:
    app: nginx
```

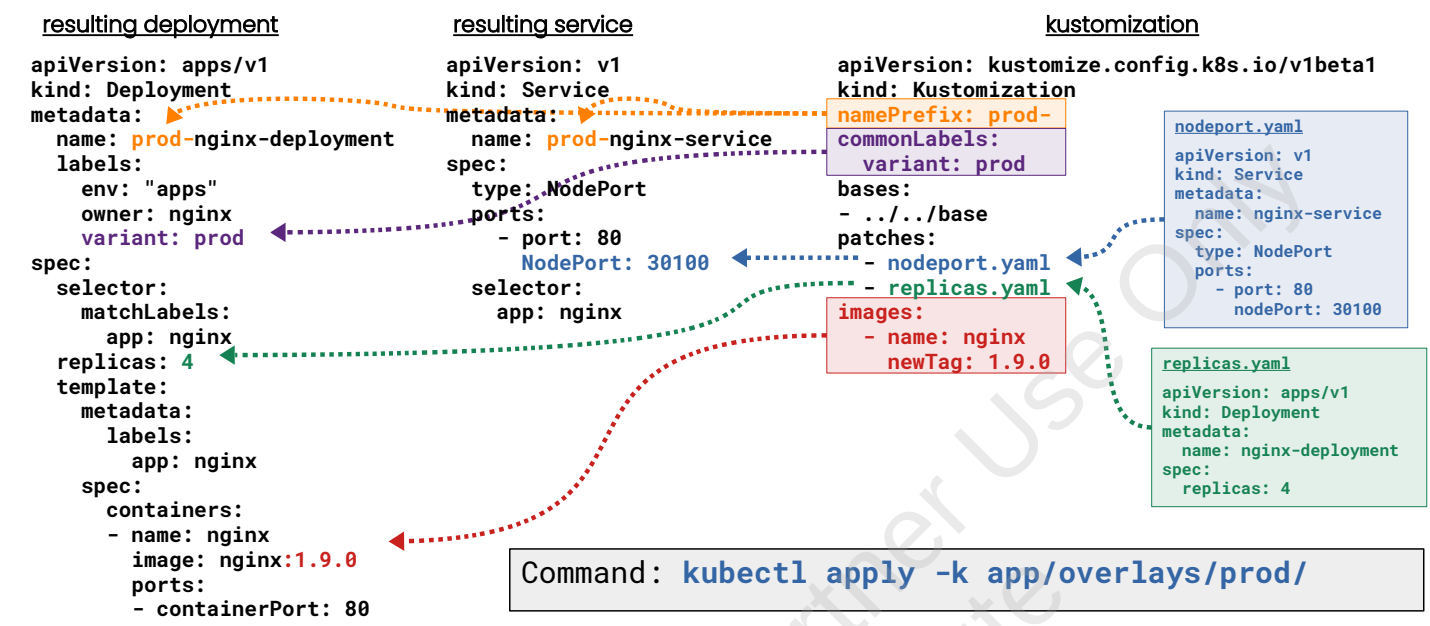
kustomization

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - deployment.yaml
  - service.yaml
```

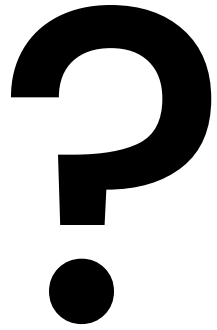


Command: **kubectl apply -k app/base/**

Example "Base" Manifests



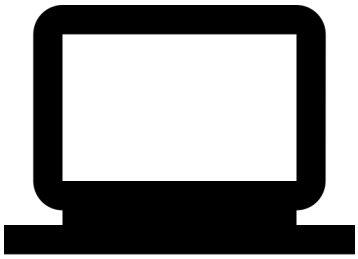
Questions:



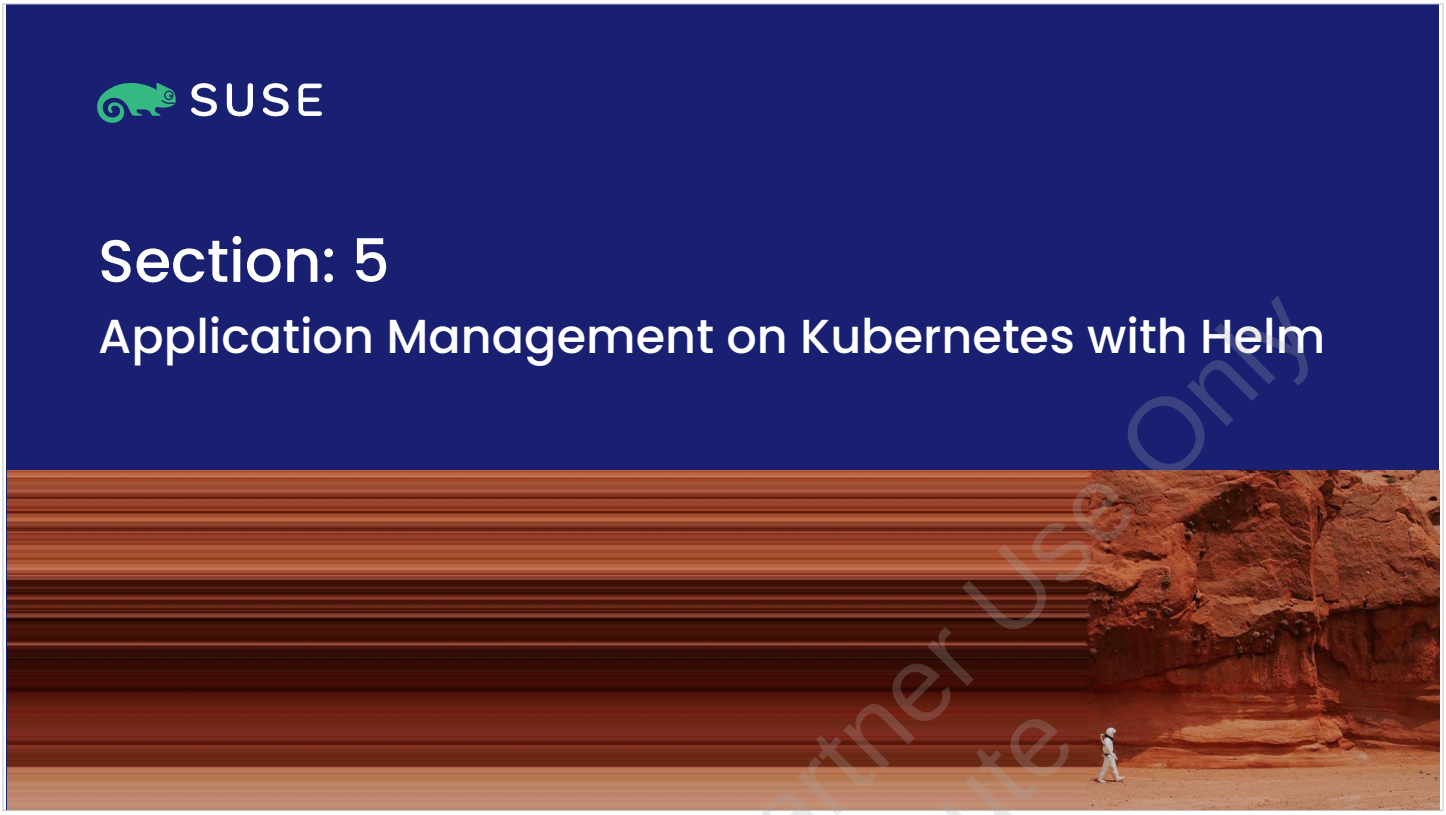
Q. What is the name of the files used to tell Kustomize what it is supposed to do?

A. Manifests.

Exercises:



4-1: Manage Applications with Kustomize



Section Objectives:

1 Understand Basic Helm Concepts

2 Manage Applications with Helm

Understand Basic Helm Concepts

What is Helm?

- Helm is a tool for managing packages of pre-configured Kubernetes resources
- Use Helm to:
  - Find and use popular software packages
  - Share your own applications
  - Create reproducible builds of your Kubernetes applications
  - Intelligently manage your Kubernetes manifest files
  - Manage releases of Helm packages

To quote the Helm documentation:

"Helm installs charts into Kubernetes, creating a new release for each installation. And to find new charts, you search Helm chart repositories."

It is important to understand that even though Helm is often compared to package managers such as RPM and Dpkg there is a significant difference between helm "packages" and traditional software packages.

With traditional software packages, everything required to install and run the package is included inside the software package itself. These things include all required files, scripts, configuration, documentation and the metadata that describes how the package is installed (i.e. what goes where). A software repository for these software packages is comprised of two parts: the software packages themselves and a catalog that contains the metadata of each of the packages in the repository.

A Helm package, or chart, only contains the metadata that describes how to deploy the package (i.e. templatized Kubernetes manifests). This metadata is similar to the metadata that is contained in the header of a RPM package. A Helm repository contains this metadata for all charts in the repository. The "files" that are installed when the chart is deployed are similar to the files/scrips/config/etc in a RPM package. With Helm, these files are containers that reside in one or more separate container registries.

Three Central Concepts to Helm

To understand Helm you must understand 3 main concepts:

Chart

- A Chart is a Helm package
- Charts contain all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster

Repository

- A Repository is the place where charts can be collected and shared

Release

- A Release is an instance of a chart running in a Kubernetes cluster
- One chart can be installed multiple times into the same cluster
- Each time it is installed, a new Release is created

Simple Helm Explanation

Helm installs charts into Kubernetes ...

Creating a new release for each installation ...

To find new charts, you can search Helm chart repositories.

Helm and Templates

- Helm uses templating to build the final manifest by replacing values in the template manifests with values provided on the CLI from the values file
- Templates are created using Go's templating syntax

  (i.e. lots of curly brackets in a yaml file)

Example Template File

```
apiVersion: v1
kind: Service
metadata:
  name: {{ template "wordpress.fullname" . }}
  labels: {{- include "wordpress.labels" . | nindent 4 }}
  {{- if .Values.service.annotations }}
  annotations: {{- include "wordpress.tplValue" (dict
"value" .Values.service.annotations "context" $) | nindent 4 }}
  {{- end }}
spec:
  type: {{ .Values.service.type }}
  {{- if (or (eq .Values.service.type "LoadBalancer")
(eq .Values.service.type "NodePort")) }}
  externalTrafficPolicy: {{ .Values.service.externalTrafficPolicy |
quote }}
  {{- end }}
  {{- if (and (eq .Values.service.type
"LoadBalancer") .Values.service.loadBalancerSourceRanges) }}
  loadBalancerSourceRanges:
  {{- with .Values.service.loadBalancerSourceRanges }}
{{ toYaml . | indent 4 }}
  {{- end }}
  {{- end }}
  ports:
    - name: http
      port: {{ .Values.service.port }}
      targetPort: http
      {{- if (and (or (eq .Values.service.type "NodePort")
(eq .Values.service.type "LoadBalancer")) (not
(empty .Values.service.nodePorts.http))) }}
      nodePort: {{ .Values.service.nodePorts.http }}
      {{- else if eq .Values.service.type "ClusterIP" }}
      nodePort: null
      {{- end }}

...
```

Helm Chart Structure

- Helm charts are a directory structure stored in an archive file
- `Chart.yaml` provide description of the helm chart
- `values.yaml` contains all possible values used in the templates
- `templates` directory contains template yaml files used to generate the final yaml files

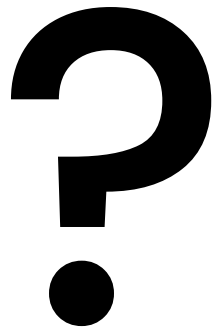
Example Chart Filename

```
<chart_name>.<version>.tgz
```

Example Expanded Chart Directory

```
<chart_name>/
            |-Chart.yaml
            |-README.md
            |-values.yaml
            |-templates/
                        |-<template_file>.yaml
                        |-<template_file>.yaml
                        |-...
```

Questions:



Q. What is Helm and how is it used?

A. A tool for managing packages of preconfigured Kubernetes resources.

Q. What is a Helm chart?

A. A Helm package that contains all of the definitions required to run an application on a Kubernetes cluster.

Q. What is a Release?

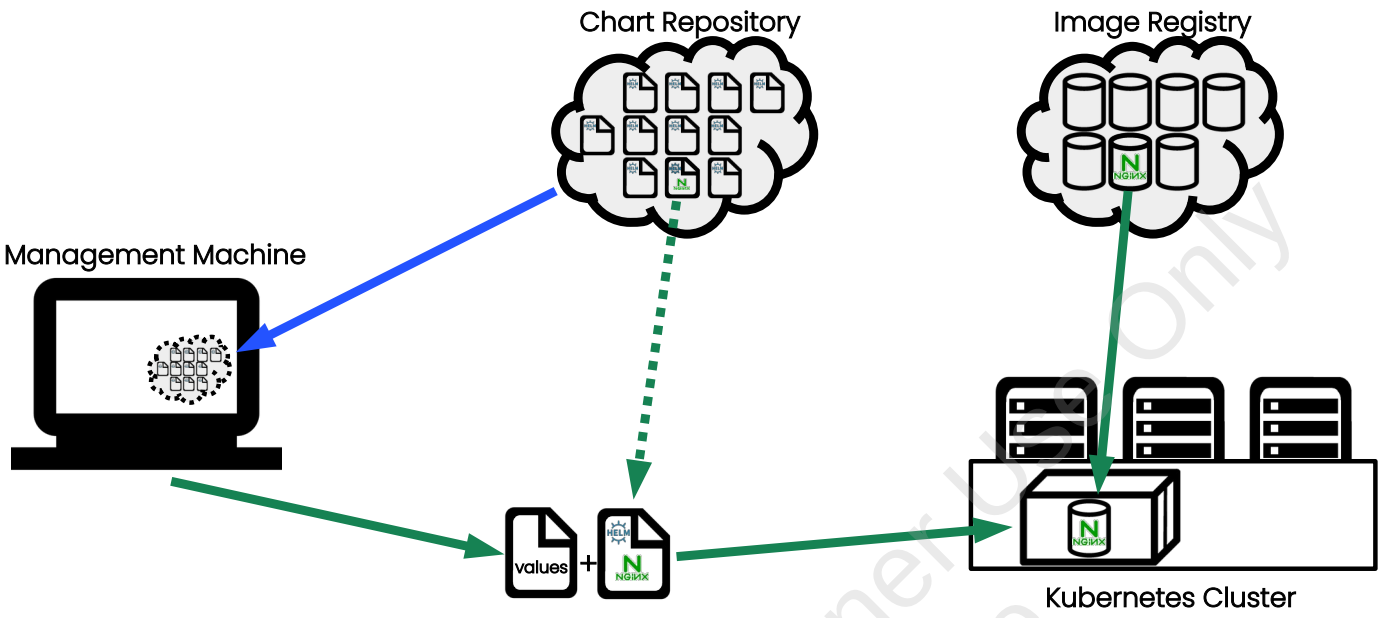
A. An instance of a deployed (installed) helm chart

Q. Where are Helm charts stored?

A. Chart repository.

Manage Applications with Helm

Helm Workflow

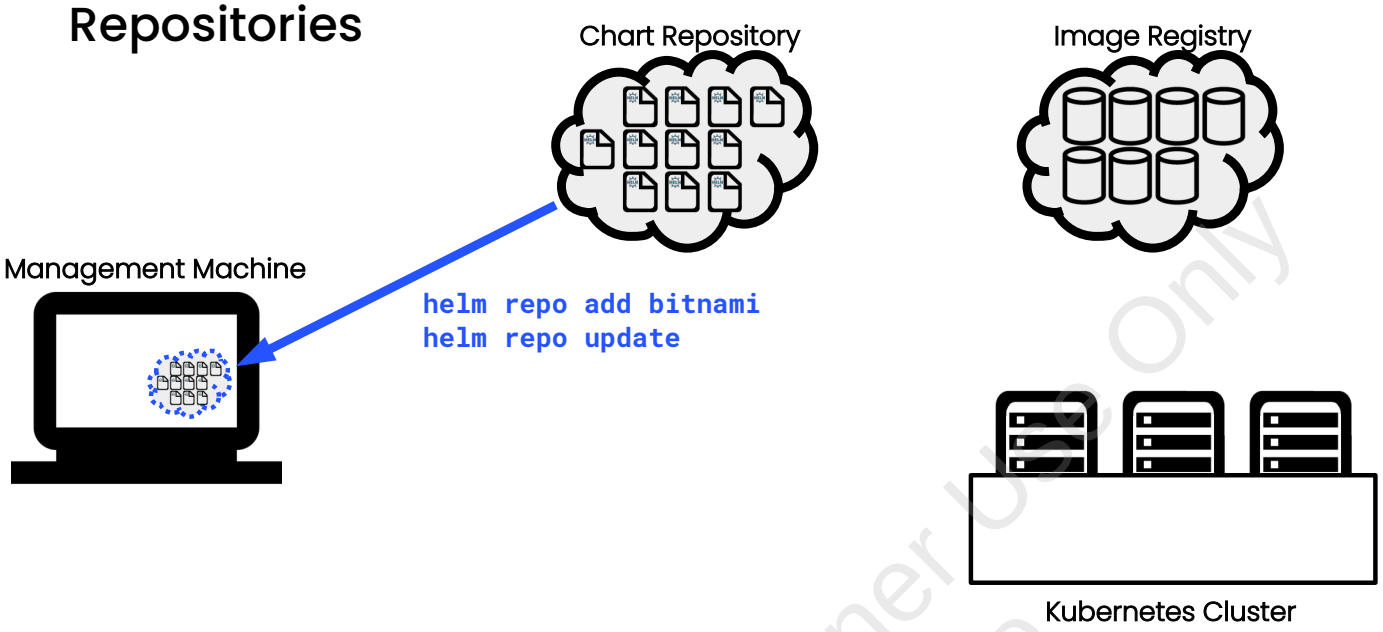


Work with Chart Repositories

Syntax: `helm repo` *MODE OPTIONS*

Mode/Option	Description
`add`	-add a chart repository
`remove`	-remove a chart repository
`list`	-list chart repositories
`update`	-update info of available charts

Chart Repositories



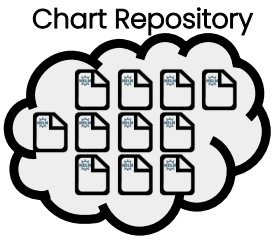
Search for Helm Charts

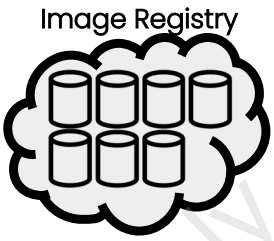
Starting with Helm v3 you can search for a chart in either Helm Hub or repositories.

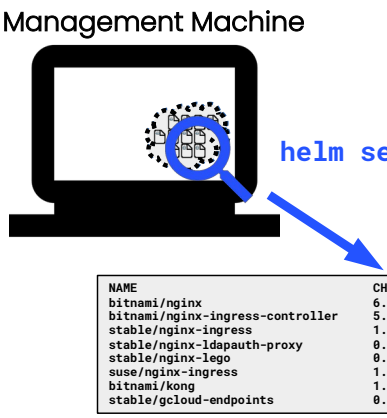
Syntax: `helm search` *MODE OPTIONS*

Mode/Option	Description
`hub <chart>`	-search the Helm Hub for a chart
`repo <chart>`	-search the added repositories for a chart

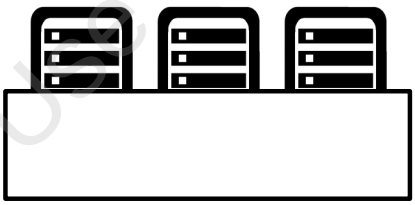
Chart Repositories








```
NAME                                    CHART VERSION   APP VERSION     DESCRIPTION
bitnami/nginx                           6.0.2           1.19.1          Chart for the nginx server
bitnami/nginx-ingress-controller        5.4.1           0.34.1          Chart for the nginx Ingress controller
stable/nginx-ingress                    1.41.2          v0.34.1         An nginx Ingress controller that uses ConfigMap...
stable/nginx-ldapauth-proxy             0.1.4           1.13.5          nginx proxy with ldapauth
stable/nginx-lego                       0.3.1                           Chart for nginx-ingress-controller and kube-lego
suse/nginx-ingress                      1.41.2          0.15.0          An nginx Ingress controller that uses ConfigMap...
bitnami/kong                            1.2.5           2.0.5           Kong is a scalable, open source API layer (aka ...
stable/gcloud-endpoints                 0.1.2           1               DEPRECATED Develop, deploy, protect and monitor...
```



Display Information about Charts

Syntax: `helm show` *SUBCOMMAND REPO/CHART OPTIONS*

Subcommand/Option	Description
`all`	-show all info for a chart
`chart`	-show chart's definition
`readme`	-show chart's README
`values`	-show chart's values

Determine Values for a Chart

The values used for a chart can be displayed using the following:

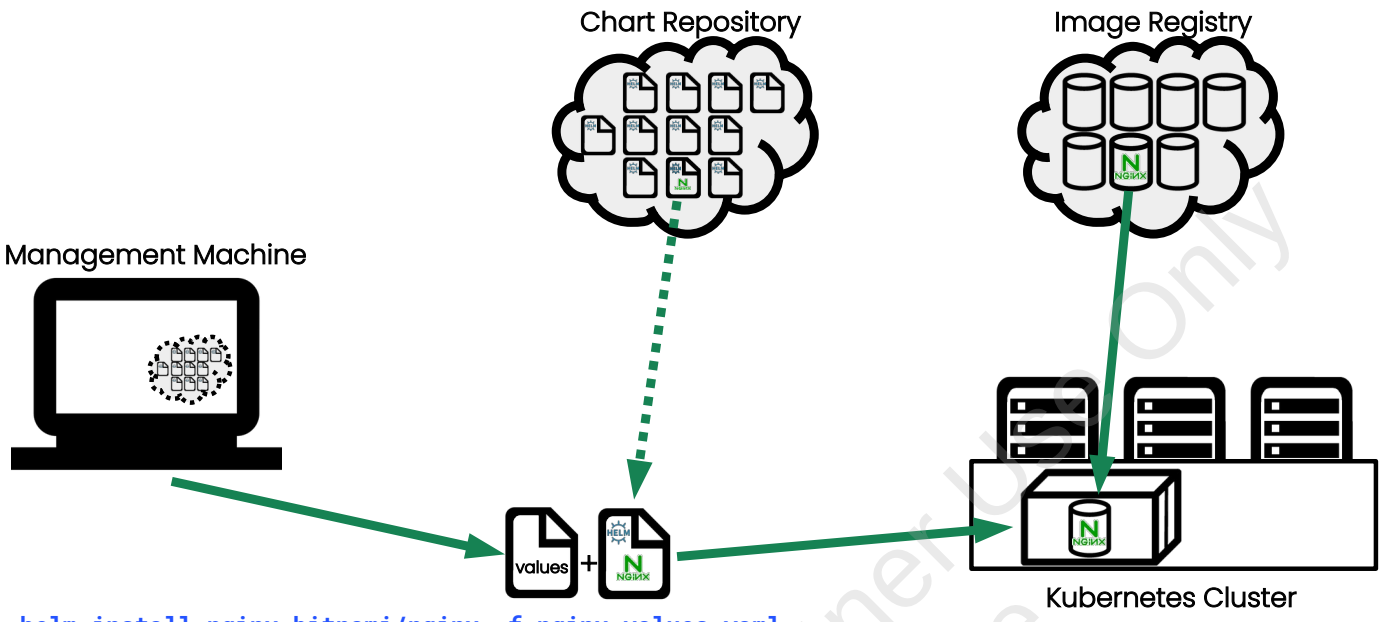
```
> helm show values stable/nginx
## Bitnami NGINX image version
## ref: https://hub.docker.com/r/bitnami/nginx/tags/
##
image:
  registry: docker.io
  repository: bitnami/nginx
  tag: 1.19.1-debian-10-r0
  ## Specify a imagePullPolicy
  ## Defaults to 'Always' if image tag is 'latest', else set to 'IfNotPresent'
  ## ref: http://kubernetes.io/docs/user-guide/images/#pre-pulling-images
  ##
  pullPolicy: IfNotPresent
  ## Optionally specify an array of imagePullSecrets.
  ## Secrets must be manually created in the namespace.
  ## ref: https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/
  ##
  # pullSecrets:
  #   - myRegistryKeySecretName

## String to partially override nginx.fullname template (will maintain the release name)
##
# nameOverride:

...
```

Install Charts

Syntax: `helm install` *NAME REPO/CHART OPTIONS*

Mode/Option	Description
`-f / --values <file>`	-specify values for the chart in a values file
`--set <value>`	-specify values for the chart on the CLI format: key=value[,key=value]
`--generate-name`	-generate a release name if on is not provided
`--dry-run`	-simulate an install
`-n / --namespace`	-specify the namespace to install into

Install a Chart




```
helm install nginx bitnami/nginx -f nginx-values.yaml
```

Display Status of a Release

The state of a release can be displayed using the following:

Syntax: `helm status` *NAME OPTIONS*

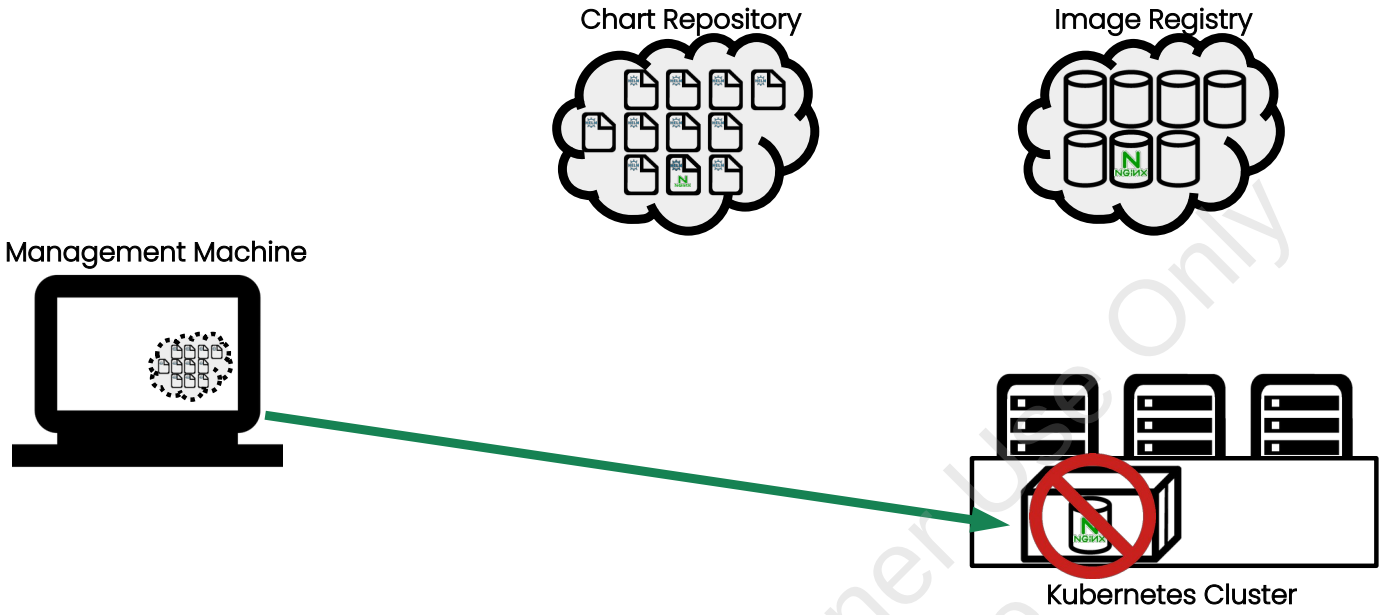
```
> helm status nginx
NAME: nginx
LAST DEPLOYED: Mon Aug  3 21:10:00 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Get the NGINX URL:

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

  export SERVICE_IP=$(kubectl get svc --namespace default nginx --template "{{ range
(index .status.loadBalancer.ingress 0) }}{{.}}{{ end }}")
  echo "NGINX URL: http://$SERVICE_IP/"
```

Uninstall a Release

Syntax: `helm uninstall` *NAME OPTIONS*

Option	Description
`--keep-history`	-remove all resources and mark as deleted but retain the release history
`--dry-run`	-simulate an uninstall
`-n / --namespace`	-specify the namespace the release is in

Uninstall a Chart



Display Release History

The release history for a chart can be displayed using the following:

Syntax: `helm history` *NAME OPTIONS*

```
> helm history nginx
REVISION   UPDATED                   STATUS       CHART          APP VERSION   DESCRIPTION
1          Mon Aug  3 21:10:00 2020   superseded   nginx-6.0.2    1.19.1        Install complete
2          Mon Aug  3 21:17:33 2020   deployed     nginx-6.0.2    1.19.1        Upgrade complete
```

Upgrade a Release

Syntax: `helm upgrade` *RELEASE REPO/CHART OPTIONS*

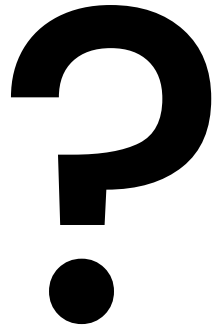
Option	Description
`-f / --values <file>`	-specify values for the chart
`--history-max`	-limit number of revisions saved per release (default=10)
`--dry-run`	-simulate an install
`-n / --namespace`	-specify the namespace to install into

Roll Back a Release

Syntax: `helm rollback` *RELEASE REVISION OPTIONS*

Option	Description
`--cleanup-on-fail`	-allow deletion of new resource created in this rollback when rollback fails
`--dry-run`	-simulate an uninstall
`-n / --namespace`	-specify the namespace the release is in

Questions:



Q. What command is used to deploy a Helm chart?

A. `helm install`

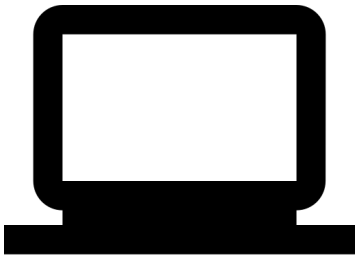
Q. What command is used to remove a release?

A. `helm uninstall`

Q. What command is used to display information about a release?

A. `helm info`

Exercises:



5-1: Add a Repository to Helm

5-2: Manage Applications with Helm

SUSE

Section: 6

Ingress Networking with an Ingress Controller in Kubernetes

Section Objectives:

1 Understand Ingress Networking for Applications

2 Work with an Ingress Controller

Understand Ingress Networking for Applications

What is an Ingress Controller?

- Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster
- Traffic routing is controlled by rules defined on the Ingress resource
- Ingress can provide load balancing, SSL termination and name-based virtual hosting
- You have the choice of either using NodePort or an External IP address for your ingress
- SUSE RKE provides an Ingress controller based on the NGINX ingress controller and K3s is based on the Traefik ingress controller

Work with an Ingress Controller

NodePort Controller Values

The NodePort Values specify the NodePort port rather than letting a random one get chosen.

(In this file, port 30443 is selected)

```
# Enable the creation of pod security policy
podSecurityPolicy:
  enabled: false

# Create a specific service account
serviceAccount:
  create: true
  name: nginx-ingress

# Publish services on port HTTPS/30443
# These services are exposed on each node
controller:
  service:
    enableHttp: false
    type: NodePort
    nodePorts:
      https: 30443
```


There are a few things to know about the service section of the values file:

`EnableHttp: false` is usually a best practice This means that the application will be expecting incoming traffic to be on https port 443 and to have a certificate. If no actual certificate is available, a self-signed certificate is possible. No plain http traffic on port 80 will be accepted.

`NodePort` will actually be listening for traffic on a worker node at port 30443. For example: https://192.168.111.10:30443

External IP Controller Values

The External IP Values allow you to define the external IP that you would like the application to be able to use.

This is a great resource for users with smaller installations that want to assign a single IP to a group of applications in a production environment.

```
# Enable the creation of pod security policy
podSecurityPolicy:
  enabled: false

# Create a specific service account
serviceAccount:
  create: true
  name: nginx-ingress

# Publish services on port HTTPS/443
# These services are exposed on the node with
IP 10.86.4.158
controller:
  service:
    enableHttp: false
    externalIPs:
    - 10.86.4.158
```

Similar to the NodePort file, here the service section is declaring a specific IP to use with this service. In this case, the service will be listening for port https://10.86.4.158

Internal Port Values

When configuring an app that will use an ingress, it is important that the service related to that app be configured to listen on a specific internal port.

(In this example, it will be expecting internal traffic on port 1234)

```
kind: Service
apiVersion: v1
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
  - port: 1234
```

Each Kubernetes service must listen for a traffic on an internal port. This port isn't regulated, but of course it would not be a good idea to use a well known port such as 80, 443, 22, etc.

Ingress Rules

After the ingress controller has been successfully deployed, the ingress rules can be defined in a separate manifest.

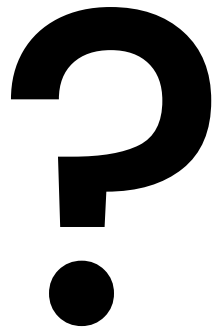
(Each service specified in this manifest will send traffic on that internal port 1234 on which the application services will be listening.)

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: caasp-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
        - path: /red
          backend:
            serviceName: red-service
            servicePort: 1234
        - path: /green
          backend:
            serviceName: green-service
            servicePort: 1234
        - path: /blue
          backend:
            serviceName: blue-service
            servicePort: 1234
```


The Ingress rules manifest but set out the path for traffic to flow through as it comes in.

For example: traffic that comes in on **https://mywebsite.com/red** will flow to a service called **red-service** on port **1234**. The **red-service** is connected to an app called **red-app**. The same thing happens if a user goes to **https://mywebsite.com/blue** and they are connect to the **blue-app** via the **blue-service**.

Questions:



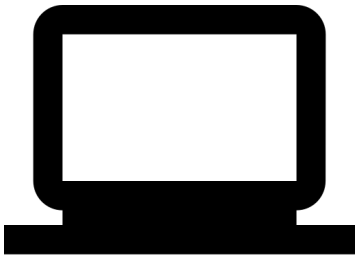
Q. What is an Ingress Controller and how is it used?

A. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster.

Q. What Ingress Controller is used on SUSE Rancher by default?

A. Nginx

Exercises:



6-1: Configure Ingress for an Application

SUSE

Section: 7

Storage in Kubernetes

Section Objectives:

1 Understand Kubernetes Storage Concepts

2 Work with Persistent Storage in Storage Classes

Understand Kubernetes Storage Concepts

Persistent Volume Storage

- With persistent volume storage you can attach persistent storage to your pods
- Multiple storage back-end types are supported by K8s (i.e. NFS, Longhorn, Ceph, etc)
- A persistent volume must first be defined before that volume can be claimed for use
- Claimed volumes can be attached to pods

What Are Volumes?

A **volume** is storage that is ready to be used

For example:

- A database needs data for its data
- An application needs a place to store logs
- A web server needs content to display

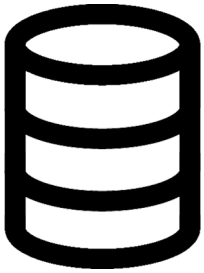
A **Persistent Volume** should be available as long as the application needs it.

A **Persistent Volume Claim** is used by an application to reach out to the Persistent Volume.

Persistent Volumes and Persistent Volume Claims

Persistent Volumes

The persistent volume is storage that is ready to be used.

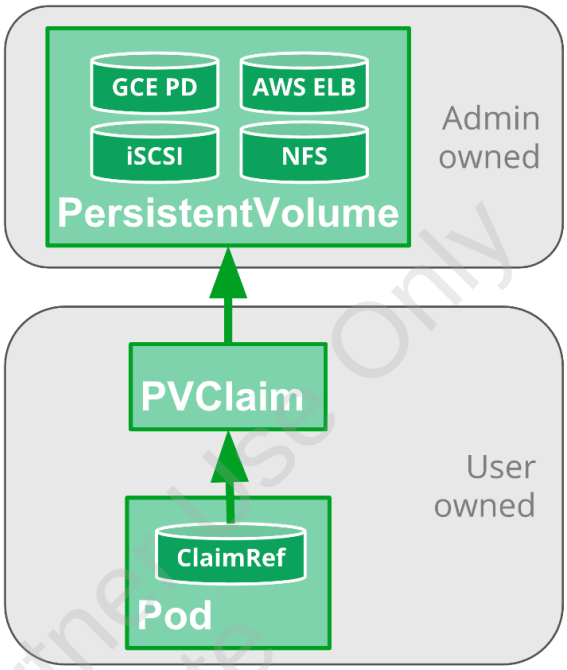


Persistent Volume Claim

The persistent volume claim is how the storage is connected to the application. It is the hand that utilizes the storage.

Persistent Volumes

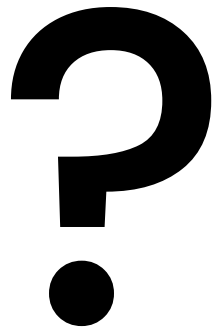
- Admin/Storage Class provisions them
- Users/Pods claim them
- Volumes have an Independent lifetime and fate
- Can be handed-off between pods
- Lives until user is done with it
- Dynamically "scheduled" and managed (like nodes and pods)



Storage Classes

- Storage classes can be defined that could map to quality-of-service levels or other policies defined by administrators
- When combined with backend provisioners they provide for dynamic creation of persistent storage volumes
- Storage volumes are automatically provisioned on-demand when a persistent volume claim is made
- A provisioner must exist for each specific storage backend type

Questions:



Q. Why is persistent storage needed in Kubernetes?

A. By default data is stored in the container image of an instantiated container and is only accessible to that container instance and will be deleted when that instance of the container is deleted.

Q. What is a PersistentVolume in Kubernetes?

A. An object in Kubernetes that represents a chunk of storage that is ready to be used but not yet accessible to a pod.

Q. What is a PersistentVolumeClaim in Kubernetes?

A. A "requests" that connects a PersistentVolume to a pod.

Work with Persistent Storage in Storage Classes

Persistent Volume

- Define a storage volume that already exists that can be used by cluster resources

Assign a label

Optionally assign a storageClassName

Path to the storage volume

Manifest:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-vol-01
  labels:
    volname: vol-01
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    server: 172.30.201.2
    path: "/srv/nfs/vol-01"
```

Commands: **kubectl apply -f persistentvolume.yaml**
kubectl get pv

Requirements for manually created volumes:

-Volumes must be pre-created on the storage back end
-All worker nodes must be able to access the storage back-end(s)

PersistentVolumeClaim

- Request to use storage
- Can request specific properties such as size, access modes, etc.

Manifest:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-claim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
  storageClassName: ""
  selctor:
    matchLabels:
      volname: "vol-01"
```

Limit to a specific storageClassName

Map request to a specific label

Commands: **kubectl apply -f persistentvolumeclaim.yaml**
kubectl get pvc

Requirements for persistent volume claims with manually created volumes:

-Volume must already exist on the storage back-end
-The Persistent Volume object must have already been created in the cluster

PersistentVolumeClaim (storaegClass)

- Can request a volume from a specific storageClass
- Volume could be dynamically created volume using a storageClass provisioner

Manifest:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-sc-claim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
  storageClassName: "nfs-client"
```

Limit to a specific storageClassName

Commands: **kubectl apply -f persistentvolumeclaim.yaml**
kubectl get pvc

Requirements when using a Storage Class with a Storage Class Provisioiner:

-A storage class provisioner that is compatible with the storage back-end
-All worker nodes must be able to access the storage back-end

Usa a Persistent Volume

- When using a persistent volume you specify both the mountpoint and the claim that is associated with the volume

Manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-web
spec:
  selector:
   matchLabels:
     app: nfs-web
  replicas: 1
  template:
    metadata:
      labels:
        app: nfs-web
    spec:
      containers:
      - name: web
        image: nginx:1.9.0
        ports:
          - name: web
            containerPort: 80
        volumeMounts:
          - name: nfs-vol-01
            mountPath: "/usr/share/nginx/html"
      volumes:
      - name: nfs-vol
        persistentVolumeClaim:
          claimName: nfs-claim
```

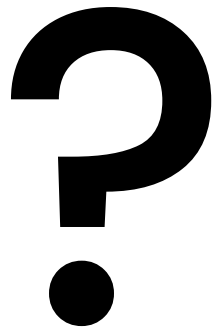
Specify the mountpoint inside the container

Reference the persistentVolumeClaim

When connecting a persistent volume to an application you reference the Persistent Volume Claim object not the Persistent Volume object.

You must also specify a mount point where you want the volume to be mounted and accessed inside of the container's filesystem.

Questions:



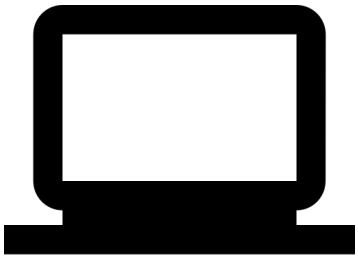
Q. What is a StorageClass in Kubernetes?

A. A Kubernetes object that can provide QOS for access to storage or when combined with a provisioner dynamically create a PersistentVolume on demand.

Q. What is required for PersistentVolumes and PersistentVolumeClaims to be created on-demand in Kubernetes?

A. StorageClass with a Provisioner

Exercises:



7-1: Configure Persistent Storage with NFS

7-2: Configure Persistent Storage with a NFS StorageClass

Section Objectives:

1 Understand Resource Usage Control in Kubernetes

2 Work with Limit Ranges

3 Work with Resource Quotas

Understand Resource Usage Control in Kubernetes

The Problem

By default containers run with unbounded compute resources.

Meaning ...

A pod or even a single container could potentially monopolize all available resources.

At the end of the day a container is simply a process running on an OS. If precautions are not put into place the process is free to consume as much of the capacity of the underlying hardware as it wants. Out of control processes can cause the system performance to degrade of even cause the system to hang. As there is no hardware level isolation between the container processes other OS level components such as cgroups must be leveraged to provide constraints.

Solutions to the Problem

Limit Ranges

- Set at the namespace level but applied at the pod/container level
- Enforce minimum/maximum compute resources usage per Pod/Container
- Enforce minimum/maximum storage request per PersistentVolumeClaim
- Set default request/limit for compute resources in a namespace

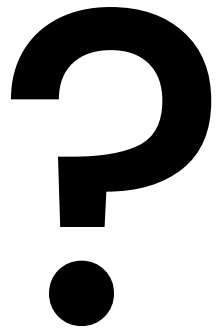
Resource Quotas

- Set and applied at the namespace level
- Constrain aggregate resource consumption per namespace
- Can limit both number of objects and total amount of resource consumed

The most important concept to understand is that, while Limit Ranges and Quotas are created at the namespace layer, Limits/Requests and Limit Ranges are applied at the pod/container level where quotas apply to the aggregate of all pods/containers in a namespace.

Limit Ranges and Quotas both use Limits/Requests as the mechanism to restrict/control resource consumption.

Questions:



Q. By default ontainers can consume as many resources as they want by. (True or False)

A. True

Q. The amount of resources a container can consume can be controlled by what two things?

A. Limits, Quotas.

Work with Limit Ranges

Limits vs Requests

- When a pod/container is created, by default it is unconstrained with regards to resource consumption
- Pods/containers can Request resources and Limits can be placed on resource consumption
- Pods/Containers can specify both Requests and Limits

Limit	Request
Upper/lower bound on the amount of a resource a pod/container can consume	The specific amount of a resource a pod wants or needs to run

Kubernetes provides a way to not only limit how much of the underlying resources a container can consume (**Limits**), it also provides a way for the containers themselves to tell it how many resources they need, at minimum, to run (**Requests**). These **Limits** and **Requests** are then used to control resource usage by the workloads running on the Kubernetes cluster

LimitRanges

- A LimitRange is created in a namespace
- A LimitRange can constrain resources such as CPU, memory and storage
- A LimitRange can define:
  - Maximum and Minimum bounds for Limits
  - Maximum and minimum bounds for Requests
  - Default Limits and Requests values

Manifest:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-and-memory-limits
spec:
  limits:
  - default:
      cpu: "1"
      memory: 200Mi
    defaultRequest:
      cpu: 500m
      memory: 100Mi
    max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 200m
      memory: 3Mi
    type: Container
```


LimitRanges are created at the namespace level and apply to each pod/container running in that namespace. However, the **Limits/Requests** defined in a **LimitRange** are applied to pods/containers that are created after the **LimitRange** itself is created. They are not applied retroactively to pods/containers already running in the namespace unless one attempts to change the **Limits/Requests** of the already running pods/containers.

Minimum and **Maximum** Limits/Requests can be specified. These values can prevent pods/containers to be created that specify **Limits/Requests** outside of these values. **Default Limits/Requests** can also be specified. The defaults will apply to any pod/container that is created in the namespace that doesn't specify its own Limits/Requests.

Limits/Requests in Pods

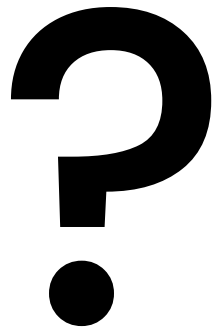
- Pods can specify either/both Limits and Requests
- If neither Limits nor Requests are specified the default Limits/Requests defined in the LimitRange will be applied

Manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx-pod
    image: nginx
    resources:
      limits:
        cpu: "4"
        memory: 500Mi
      requests:
        cpu: "1"
        memory: 100Mi
```


Pods/containers can specify their own Limits/Requests in their manifests. This allows them to not only be "good citizens" by proactively limiting the resources they can consume but also communicate to the scheduler any minimum level of resources (CPU/Memory/Storage) that they require to run.

Questions:



Q. What are Limits with regard to resource usage in Kubernetes?

A. Mechanisms to limit the minimum/maximum amount of resources that can be consumed by pods.

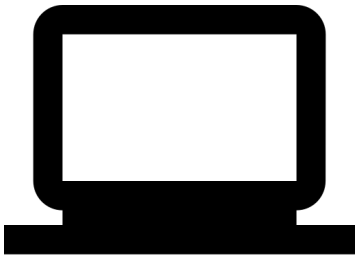
Q. What are Requests with regard to resource usage in Kubernetes?

A. Mechanisms for pods to requist specific amounts of resources.

Q. What is a LimitRange in Kubernetes and how is it used?

A. A setting applied to a namespace to set and enforce limits.

Exercises:



8-1: Define Default Limits for Pods in a Namespace

8-2: Define Limits for Containers and Pods

Work with Resource Quotas

Limit Ranges vs Resource Quotas

Limit Ranges

- Created in a namespace but enforced on a pod/container basis
- Enforced only on objects created after LimitRange is created
- Apply only to specific resources (CPU, Memory, Storage, etc)

Resource Quotas

- Enforced on the aggregate resources consumed in a namespace
- Enforced only on objects created after Quota is created
- Apply to both amount of resources and number of objects
- Can be used in conjunction with LimitRanges
- Can be set for both resource limits and requests

Quotas are created at the namespace level and apply to the aggregate resources consumed by all pods/containers running in the namespace. Like with **LimitRanges**, quotas only apply to pods/containers created in the namespace after they Quotas themselves have been created. They do not apply retroactively to any pod/container already running in the namespace unless one attempts to change the **Limits/Requests** of the already running pods/containers.

Generally Scoped Quotas in a Namespace

- Quotas can be scoped generally within a namespace
- Apply to all pods/containers in a namespace

Manifest:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-quota
spec:
  hard:
    cpu: "10"
    memory: 200Gi
    pods: 10
```

```
> kubectl describe quota
Name:       pod-quota
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     10
memory      0     20Gi
pods        0     10
```


Quotas can be created in a namespace that are generally applied to all pods/container that are created despite any priority request that may be requested by the pods/containers. These would be considered as **Generally Scoped Quotas** rather then specifically scoped quotas in the namespace.

Priority Scoped Quotas in a Namespace

- Quotas can be scoped based on priority
- Only apply to pods assigned that priority

```
> kubectl describe quota
Name:       pods-high
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     10
memory      0     20Gi
pods        0     10

Name:       pods-low
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     5
memory      0     10Gi
pods        0     5
```

Manifest:

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-high
  spec:
    hard:
      cpu: "10"
      memory: 200Gi
      pods: 10
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["high"]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-low
  spec:
    hard:
      cpu: "5"
      memory: 100Gi
      pods: 5
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["low"]
```

Priority Scoped Pod vs Quota

Pod Manifest:

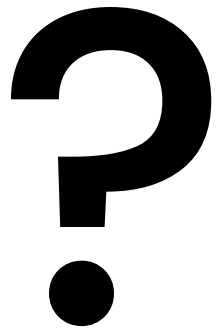
```
apiVersion: v1
kind: Pod
metadata:
  name: high-priority
spec:
  containers:
  - name: high-priority
    image: opensuse/leap
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello; sleep 10;done"]
    resources:
      requests:
        cpu: "1"
        memory: "1Gi"
      limits:
        cpu: "1"
        memory: "1Gi"
  priorityClassName: low
```

Quota Manifest:

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-high
  spec:
    hard:
      cpu: "10"
      memory: 200Gi
      pods: 10
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["high"]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-low
  spec:
    hard:
      cpu: "5"
      memory: 100Gi
      pods: 5
    scopeSelector:
      matchExpressions:
      - operator : In
        scopeName: PriorityClass
        values: ["low"]
```


To take advantage of **Priority Scoped Quotas** a pod/container must specify, using a **scopeSelector**, the priority level that it wants in its manifest. The specified priority level must match on that is associated with a Priority Scoped Quota created in the namespace.

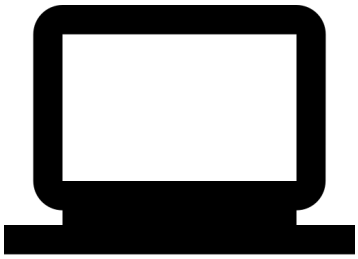
Questions:



Q. What is a Resource Quota in Kubernetes and how is it used?

A. Restriction on the aggregate resources consumed by all pods in a namespace.

Exercises:



8-3: Define Quotas for a Namespace

8-4: Test Quotas for a Namespace

SUSE

Section: 9

Role Based Access Controls in Kubernetes

Section Objectives:

1 Understand Role Based Access Control in Kubernetes

2 Authenticate to a Kubernetes Cluster

3 Configure RBAC in Kubernetes

Understand Role Based Access Control in Kubernetes

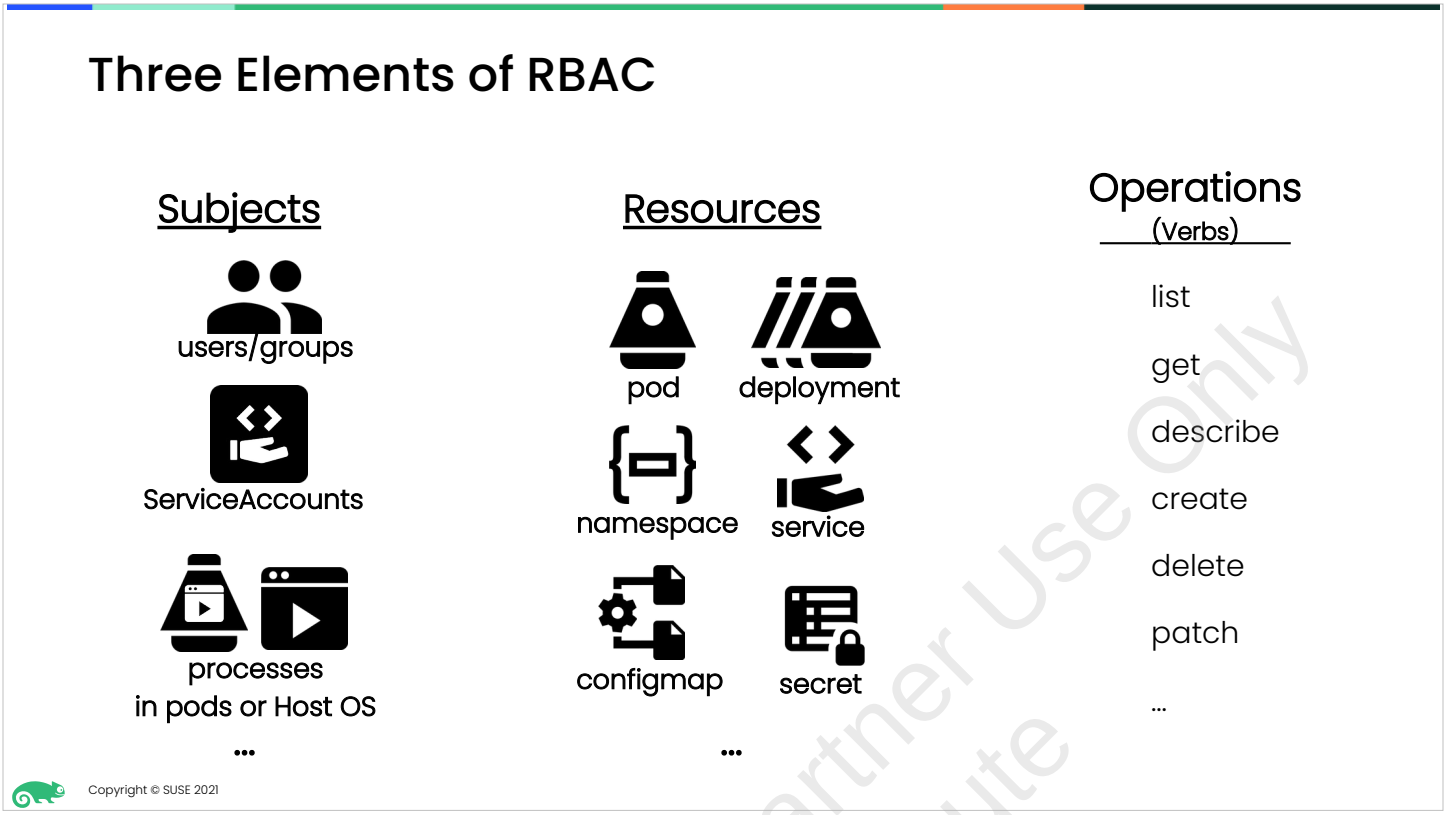
What is Role Based Access Control (RBAC)?

Role Based Access Control (RBAC) is a set of functions in Kubernetes that controls who/what is allowed to perform specific actions in Kubernetes.

Example Use Cases:

- Users who want to deploy new applications
- Applications who need to access specific resources
- Services that need to be accessed by multiple applications
- System-wide application and user permissions

Kubernetes uses Role Based Access Control (RBAC) to determine who can perform what actions on which resource in the cluster. Permissions are additive in nature rather than subtractive. A permission must be explicitly granted for a subject to be able to perform an action. This makes it very important to be specific rather then general when assigning permissions.



Subjects:

Subjects are objects that roles can be assigned to. These objects will then be able to perform actions on resources in the cluster. Typically the type of subject you will be dealing with is a user.

Resources:

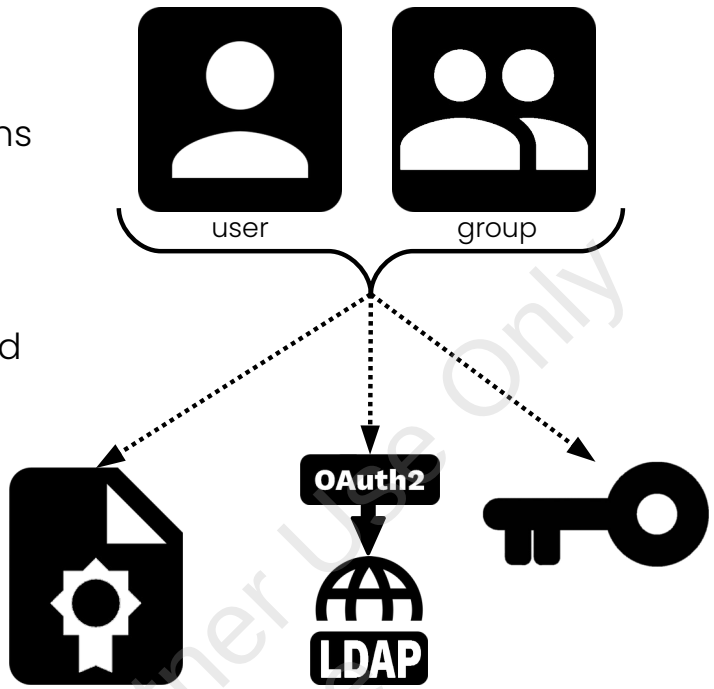
Resources are the objects defined in the Kubernetes API. Some of these resources are scoped globally in the cluster (i.e. nodes) where other resources can be scoped relative to a namespace (i.e. pod, deployment, service, etc).

Operations:

Operations are the verbs that can be used against a resource.

What are Users?

- Subjects that correlate to humans (or applications external to the cluster)
- User/Group accounts are defined externally to the K8s Cluster:
  - Certificate based
  - Token based
  - OAuth2
  - Basic Authentication



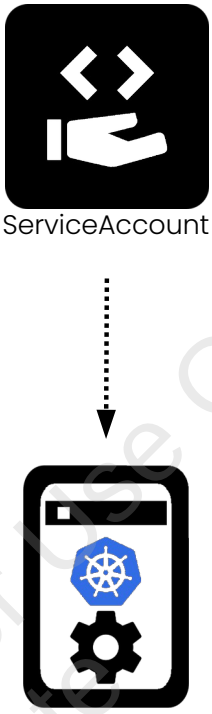
Technically speaking the **User** is the subject that is authenticating. However, roles can be assigned to both users and groups.

Both the user and the groups are defined externally to the cluster. For example you can have users and groups defined as objects in an LDAP directory. You can then use an OAuth2 connector to talk to the LDAP directory to perform user authentication and determine user group membership.

If you are using certificate based authentication the cn= property defined in the certificate is interpreted as the "user". The o= properties defined in the certificate are interpreted as the "groups" the user is a member of.

What are ServiceAccounts?

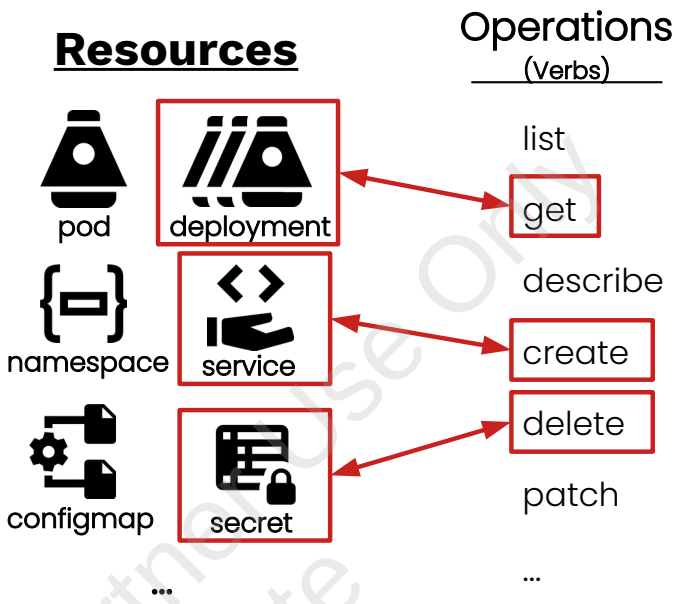
- Subjects that correlate accounts created in the K8s cluster
- ServiceAccounts can be created for and used by applications or by human users



ServiceAccounts are defined internally to the cluster. They can be used for users or "services" to access resources in the cluster. The ServiceAccount will have an access token that it presents to have its access "authorized". This token can be placed in a standard kubeconfg file or passed to the API with the access request.

What are Roles?

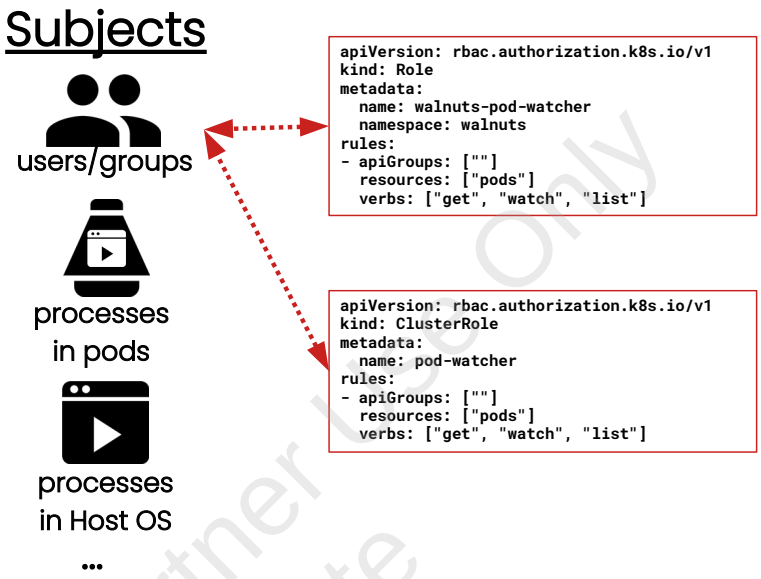
- Link **Operations** with **Resources**
- Act as sets of permissions
- Permissions add abilities but can not remove abilities
- Can be namespace scoped (Roles) or cluster scoped (ClusterRoles)



Roles are the mechanism that is used to link actions with resources. They are effectively the "permissions" that are assigned to subjects.

What are RoleBindings/ClusterRoleBindings?

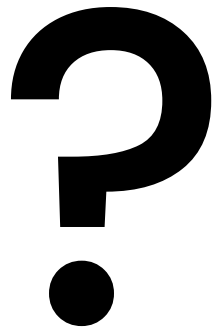
- Constructs that associate (binds) a **Subject** to a **Role** or **ClusterRole**



RoleBindings are the mechanism that is used to link, or bind, a **Role** to a **Subject** thereby granting the Subject the permission defined by the **Role**.

Role bindings come in two different types that an be assigned at the namespace level or at the cluster level. Cluster scoped role bindings are defined as an object of kind **ClusterRoleBinding** and can only reference **ClusterRoles** where namespace scoped role bindings are of kind **RoleBinding** and can only reference **Roles**.

Questions:



Q. What are the three elements of Role Based Access Control in Kubernetes?

A. Subjects, Resources, Operations.

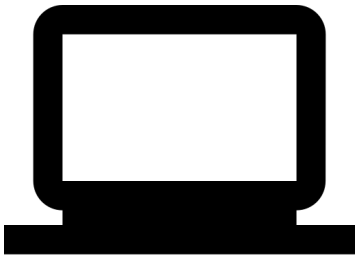
Q. What are used to link users to an operations?

A. Roles.

Q. What is a ServiceAccount?

A. An account created internally to the Kubernets cluster that can be used by applications or users to authenticate to the cluster.

Exercises:



9-1: Create Service Accounts

Authenticate to a Kubernetes Cluster

Kubeconfig file

- Used to authenticate to the cluster
  - Default location: ~/.kube/config
  - Can be supplied on the CLI or environment variable
- Contains 3 main sections:
  - Clusters
  - Contexts
  - Users

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <ca certificate>
    server: <server URL and port>
  name: <cluster name>
contexts:
- context:
    cluster: <cluster name>
    user: <user name>
  name: <context name>
current-context: <context name>
kind: Config
preferences: {}
users:
- name: <user/serviceaccount name>
  user:
    token: <authentication token>
```

The kubeconfig files are what the **kubectl** command uses to determine how to access a Kubernetes cluster as well as the context (subject/user and cluster name) in the cluster. The command also uses the file to determine the user and the user's authentication token used to access the cluster.

The default location that **kubectl** looks for a kubeconfig file is **~/.kube/config** but the path to a kubeconfig file can also either be passed to the **kubectl** command using the **--kubeconfig** CLI option or set in the KUBECONFIG environment variable.

The tree main sectionsof a kubeconfig file are **clusters:**, **contexts**: and **users:**.

The **clusters:** section contains connection information for one or more clusters. Connection information includes the address to connect to and the CA certificate for the cluster.

The **users:** section contains one ofr more user that will be used when connectiong to the cluster(s). The user info includes the user name and authentication infor such as a token or connection info for a authetication provider. The user can be ueiother a "user" or a serviceaccount.

The **contexts:** section contains the mappings of the user(s) to the cluster(s).

Kubeconfig File Example: User

- User: section contains authentication provider information

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: ...
    server: https://control01.example.com:6443
  name: local
contexts:
- context:
    cluster: local
    user: charlie
  name: local
current-context: local
kind: Config
preferences: {}
users:
- name: charlie
  user:
    auth-provider:
      config:
        client-id: oidc
        client-secret: ...
        id-token: ...
        idp-issuer-url: https://master01.example.com:32000
        refresh-token: ...
      name: oidc
```


In a kubeconfig for a user the **`user:`** entry in the **`users:`** section will contain the connection information for the authentication provider that will be used to autheinticat the users.

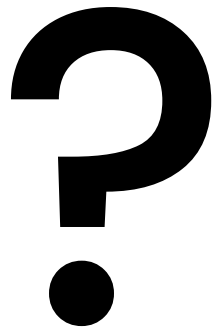
Kubeconfig File Example: Service Account

- User: section contains authentication token

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <ca certificate>
    server: https://control01.example.com:6443
  name: local
contexts:
- context:
    cluster: local
    user: charlie
  name: local
current-context: local
kind: Config
preferences: {}
users:
- name: charlie
  user:
    token: <authentication token>
```


In a kubeconfig for a serviceaccount the **`user:`** entry in the **`users:`** section will contain the autheintication token for the services account. This token can be retrieved from the secret that was created for and associated with the serviceaccount..

Questions:



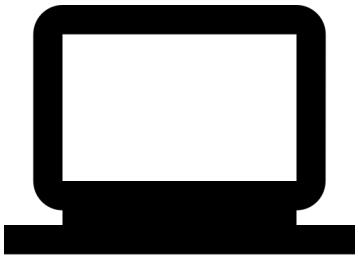
Q. What is a kubeconfig file and how is it used?

A. File containing authentication information for interacting with a Kubernetes cluster.

Q. What are the three main sections of a kubeconfig file?

A. Clusters, Contexts, Users.

Exercises:



9-2: Create kubeconfig Files for Service Accounts

Configure RBAC in Kubernetes

Role Manifest

- Roles are given a name
- Roles are scoped to a specific namespace
- Rules are comprised of:
  - apiGroups
  - resources
  - verbs

Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: walnuts-pod-watcher
  namespace: walnuts
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Command: **`kubectl apply -f walnuts-pod-watcher.yaml`**

When creating a role, you must first give the role a name. In this example, the role is called **walnuts-pod-watcher**. It uses the default apiGroup (which is normal for pods). It is allowed by the verb list to "**get**", "**watch**", and "**list**".
The resources list only has "**pods**".
This role also only works in the **walnuts** namespace.
Therefore any subject that is bound to this role will have the ability to get, watch, and list pods in the walnuts namespace.

Information About Roles

Display roles defined in a namespace

```
> kubectl get roles -n walnuts
NAME                          CREATED AT
walnuts-admin                 2020-08-19T16:47:05Z
walnuts-deployment-manager    2020-08-19T16:47:15Z
walnuts-pod-watcher           2020-08-19T16:47:22Z
```

Display info about a role defined in a namespace

```
> kubectl describe role walnuts-pod-watcher -n walnuts
Name:         walnuts-pod-watcher
Labels:       <none>
Annotations:  PolicyRule:
  Resources   Non-Resource URLs  Resource Names  Verbs
  ---------   -----------------  --------------  -----
  pods        []                 []              [get watch list]
```

ClusterRole Manifest

- ClusterRoles are given a name
- ClusterRoles are not scoped to a specific namespace but are cluster scoped
- Rules are comprised of:
  - apiGroups
  - resources
  - verbs

Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-pod-watcher
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Command: **kubectl apply -f cluster-pod-watcher.yaml**

When creating a ClusterRole, you must first give the role a name. In this example, the ClusterRole is called **pod-watcher**. It uses the default apiGroup (which is normal). It is allowed by the verb list to "**get**", "**watch**", and "**list**".
The resources list only has "**pods**".
This role works in all namespaces.
Therefore any subject that is bound to this role will have the ability to get, watch, and list pods in the all namespaces.

Information About ClusterRoles

Display ClusterRoles defined in a cluster

```
> kubectl get clusterroles
NAME                                   CREATED AT
admin                                  2020-08-17T17:59:05Z
cilium                                 2020-08-17T17:59:10Z
cilium-operator                        2020-08-17T17:59:10Z
cluster-admin                          2020-08-17T17:59:05Z
cluster-pod-watcher                    2020-08-20T17:01:32Z

...

view                                   2020-08-17T17:59:05Z
```

Information About ClusterRoles

Display info about a cluster role

```
> kubectl describe clusterRole cluster-pod-watcher
Name:         cluster-pod-watcher
Labels:       <none>
Annotations:  PolicyRule:
  Resources   Non-Resource URLs  Resource Names  Verbs
  ---------   -----------------  --------------  -----
  pods        []                 []              [get watch list]
```

Role vs ClusterRole Manifest Differences

Role Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: walnuts-pod-watcher
  namespace: walnuts
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRole Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-watcher

rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```


The only real differences in the manifests for Roles and ClusterRoles are that the kind: is different (Role vs ClusterRole) and a ClusterRole does not have a **namespace**: property in the metadata section.

RoleBinding Manifest - Users

- RolesBindings are given a name
- RoleBindings are scoped to a specific **namespace**
- RoleBindings associate a **Subject** with a **Role**
- When the **Subject** is a **User** you must specify the **rbac apiGroup**

Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: charlie-read-walnuts-pods
  namespace: walnuts
subjects:
- kind: User
  name: charlie
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: walnuts-pod-watcher
  apiGroup: rbac.authorization.k8s.io
```

Command: **`kubectl apply -f charlie-read-walnuts-pods.yaml`**

When creating a RoleBinding, you must first give the RoleBinding a name. In this example, the RoleBinding is called **charlie-read-walnuts-pods**. It uses, in the **`subjects:`** section **User** for the **`kind:`** and **rbac.authorization.k8s.io** for the **`apiGroup:`**.

This example links the User named **charlie** to the **walnuts-pod-watcher** role in the **walnuts** namespace.

RoleBinding Manifest - ServiceAccounts

- RolesBindings are given a name
- RoleBindings are scoped to a specific **namespace**
- RoleBindings associate a **Subject** with a **Role**
- When the **Subject** is a **ServiceAccount** you must specify the **Namespace** in which it resides

Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: charlie-read-walnuts-pods
  namespace: walnuts
subjects:
- kind: ServiceAccount
  name: charlie
  namespace: walnuts
roleRef:
  kind: Role
  name: walnuts-pod-watcher
  apiGroup: rbac.authorization.k8s.io
```

Command: **`kubectl apply -f charlie-read-walnuts-pods.yaml`**

RoleBindings for ServiceAccounts differ from those for Users/Groups in that in the **`subjects:`** section you use **ServiceAccount** for the **`kind:`** and you specify the **`namespace:`** in whigh the ServiceAccount resindes.

This example links the ServiceAccount named **charlie** to the **walnuts-pod-watche**r role in the **walnuts** namespace.

ClusterRoleBinding Manifest - User

- ClusterRolesBindings are given a name
- Cluster RoleBindings are not scoped to a specific namespace
- ClusterRoleBindings associate a **Subject** with a **ClusterRole**
- When the **Subject** is a **User** you must specify the **rbac apiGroup**

Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: charlie-read-pods
subjects:
- kind: User
  name: charlie
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-pod-watcher
  apiGroup: rbac.authorization.k8s.io
```

Command: **`kubectl apply -f charlie-read-pods.yaml`**

When creating a ClusterRoleBinding, you must first give the ClusterRoleBinding a name. In this example, the ClusterRoleBinding is called **charlie-read-pods**. Like with RoleBindings with users it uses **User** for the **`kind:`** and the **rbac.authorization.k8s.io** **`apiGroup;`** for the apiGroup in the **`subjects:`** section.

This example links the User named **charlie** to the **pod-watcher** role in all namespaces.

ClusterRoleBinding Manifest - ServiceAccount

- ClusterRolesBindings are given a name
- Cluster RoleBindings are not scoped to a specific namespace
- ClusterRoleBindings associate a **Subject** with a **ClusterRole**
- When the **Subject** is a **ServiceAccount** you must specify the **Namespace** in which it resides

Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: charlie-read-pods
subjects:
- kind: ServiceAccount
  name: charlie
  namespace: walnuts
roleRef:
  kind: ClusterRole
  name: cluster-pod-watcher
  apiGroup: rbac.authorization.k8s.io
```

Command: **`kubectl apply -f charlie-read-pods.yaml`**

When creating a ClusterRoleBinding, you must first give the ClusterRoleBinding a name. In this example, the ClusterRoleBinding is called **charlie-read-pods**. ILike with RoleBindings with users it uses **ServiceAccount** for the **`kind:`** and the **`namspace:`** in which the ServiceAccount resides in the **`subjects:`** section.

This example links the ServiceAccount named **charlie** to the **pod-watcher** role in all namespaces.

RoleBinding vs ClusterRoleBinding Manifest Differences

RoleBinding Manifest:

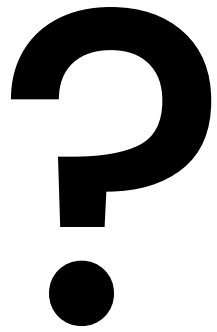
```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: charlie-read-walnuts-pods
  namespace: walnuts
subjects:
- kind: User
  name: charlie@example.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: walnut-pod-watcher
  apiGroup: rbac.authorization.k8s.io
```

ClusterRoleBinding Manifest:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-pods

subjects:
- kind: User
  name: charlie@example.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-watcher
  apiGroup: rbac.authorization.k8s.io
```


The only real differences in the manifests for Roles and ClusterRoles are that the kind: is different (Role vs ClusterRole) and a ClusterRole does not have a namespace: property in the metadata section.

Questions:



Q. In what two scopes can rules be defined?

A. Cluster, namespace.

Q. What kind of role is used for cluster scoped resources?

A. ClusterRole

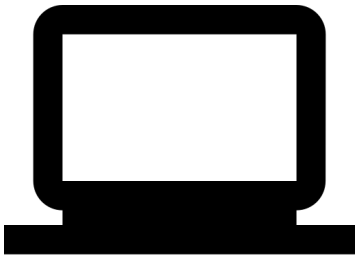
Q. What is used to link a role with rules relating to namespace scoped resources to a user?

A. RoleBinding.

Q. What is used to link a role with rules relating to cluster scoped resources to a user?

A. ClusterRoleBinding.

Exercises:



9-3: Create Roles and CLusterRoles

9-4: Create RoleBindings and ClusterRoleBindings

9-5: Test RBAC in Kubernetes

SUSE

Thank you

For more information, contact SUSE at:

+1 800 796 3700 (U.S./Canada)

+49 (0)911-740 53-0 (Worldwide)

Maxfeldstrasse 5

90409 Nuremberg

www.suse.com