**Understanding document Kubernetes**

**By: Mohit Bisht**

**Topics:**

* What is Containerization?
* What is kubernete?
* kubernetes features.
* Other Implementations.
* Kubernetes: The Terminology.
* Basic Building Blocks: Nodes and Pods.
* Labels, Selectors and Namespaces.
* Kubelet and kube proxy.
* Kubernetes installations.
* Scenario: Running first application.
* Scenario: Running deployment services from a single file.
* Scenario: Clean up.
* Scenario: Add, Change and Delete labels.
* Scenario: Select by labels.
* Scenario: Health checks.
* Scenario: Upgrade and roll out.
* Scenario: Basic troubleshooting.
* Scenario: Kubernetes Dashboard
* Scenario: Configuration Data
* Scenario: Dealing with application secrets
* Scenario: Running jobs in Kubernetes
* Scenario: Running stateful set applications
* Production Kubernetes deployment
* Namespaces.
* Monitoring and Logging
* Authentication and Authorization

**What is Containerization?**

* A collection of software processes unified by one namespace with access to an operating system kernel that it shares with other containers and little to no access between containers.
* Docker instance
  + A runtime instance of a Docker image contains three things:
    - A coker image
    - An execution environment
    - A standard set of instructions
  + Docker Engine
    - Composed of the runtime and packaging tool.
    - Must be installed on the hosts that run Docker.
  + Docker store
    - An online cloud service where users can store and share their Docker images.
    - Also known as Docker Hub.
* Difference between VM and Container.
  + VM
    - One or many applications.
    - The necessary binaries and libraries.
    - The entire gusset operating system to interact with the applications.
  + Containers
    - Include the application and all of its dependencies.
    - Share the kernel with other containers.
    - Not tied to infrastructure - only needs Docker Engine installed on the Host.
    - Run as isolated processes in user space on the host OS.
* Container Benefits for Developers
  + Applications are
    - Portable.
    - Packaged in a standard way.
  + Deployment is
    - Easy
    - Repeatable
  + Automated testing, packaging and integrations.
  + Support newer microservice architectures.
  + Alleviate platform compatibility issues.
* Container Benefits for DevOps.
  + Reliable deployment: improve speed and frequency of releases.
  + Consistent application lifecycle: configure once and run multiple times.
  + Consistent environments
    - NO more process difference between dev and production environments.
  + Simple scaling
    - Fast deployments ease the addition of workers and permit workload to grow and shrink for on-demand use cases.
* Create a common languages,
* The DevOps team can isolate and debug issues at the container level.
* Allow the Building of Pipelines
  + Containers bring agility to your code.
  + Help build a continuous integration and deployment pipeline.
  + Push an IT team to develop, test and deploy applications faster.
  + Reason why enterprises have started to adopt containers in such a big way.

**What is kubernete?**

* Orchestrator Features
  + Provision hosts
  + Instantiate containers on a host.
  + Restart failing containers.
  + Expose containers as services outside the cluster.
  + Scale cluster up or down.
* Definition: An open-source platform designed to automate deploying, scaling and operating application containers.
* Goal: to foster an ecosystem of components and tools and relieve the burden of running applications in public and private clouds.
* It was started by google.
* Borg was the predecessor to kubernetes.
* Kubernetes is a platform to schedule and run containers on clusters of virtual machines. It runs on bare metal, virtual machines, private datacenter and public cloud.
* Kubernetes is a container platform.
  + We can use Docker containers to develop and build applications and the use Kubernetes to run these applications on your infrastructure,

**Kubernetes features**

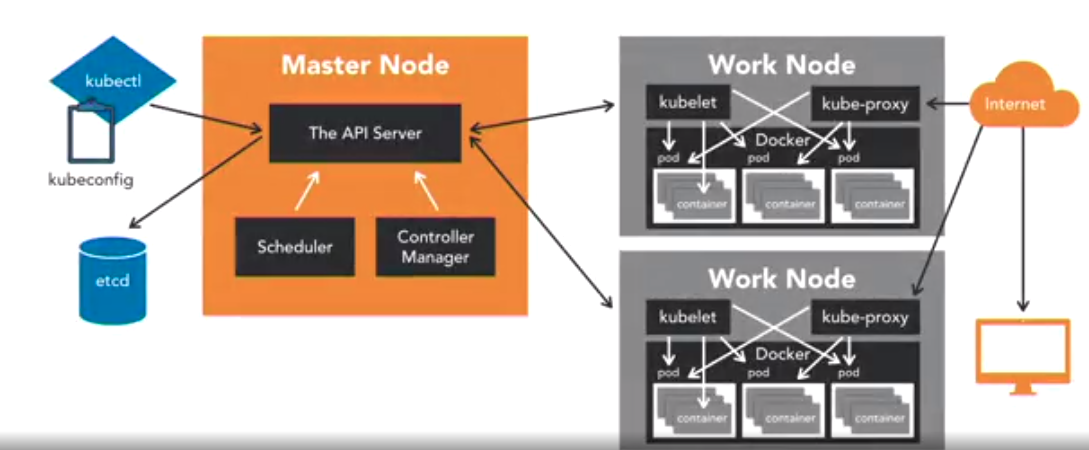
* Multi-Host Container Scheduling
  + Done by the kube-scheduler.
  + Assigns pods to nodes at runtime.
  + Checks resources, quality of service, policies, and user specifications before scheduling.
* Scalability and Availability
  + Kubernetes master can be deployed in a highly available configuration.
  + Multi-region deployments available.
  + Supports 5000 node clusters.
  + 150000 total pods.
  + Maximum of 100 pods per node.
  + Pods can be horizontally scaled via API.
* Flexibility and Modularization.
  + Plug and play architecture.
  + Extend architecture when needed.
  + Add-ons: network drivers, service discovery, container runtime, visualization and command.
* Registration
  + Seamless nodes register themselves with master.
* Service Discovery
  + Automatic detection of services and endpoints via DNS or environment variables.
* Persistent Storage
  + Much requested and an important feature when working with containers.
  + Pods can use persistent volumes to store data.
  + Data retained across pod restarts and crashes.
* Application Upgrades and Downgrades.
  + Upgrades: rolling updates supported.
  + Downgrades: rollbacks are supported.
* Maintenance
  + Features are backward compatible.
  + APIs are versioned.
  + Turn off/on host during maintenance (unschedulable).
* Logging and Monitoring
  + Application monitoring built-in.
    - TCP, HTTP, or container execution health check.
  + Node health check
    - Failures monitored by node controller.
  + Kubernetes status
    - Add-ons: Heapster and cAdvisor.
  + Logging and Monitoring
    - Logging framework
      * In place or extensible.
* Secrets Management
  + Sensitive data is first-class citizens.
  + Mounted as data volumes or environment variables.
  + Specific to namespace.
* Community
  + KubeCon
  + Meetups
  + Slack

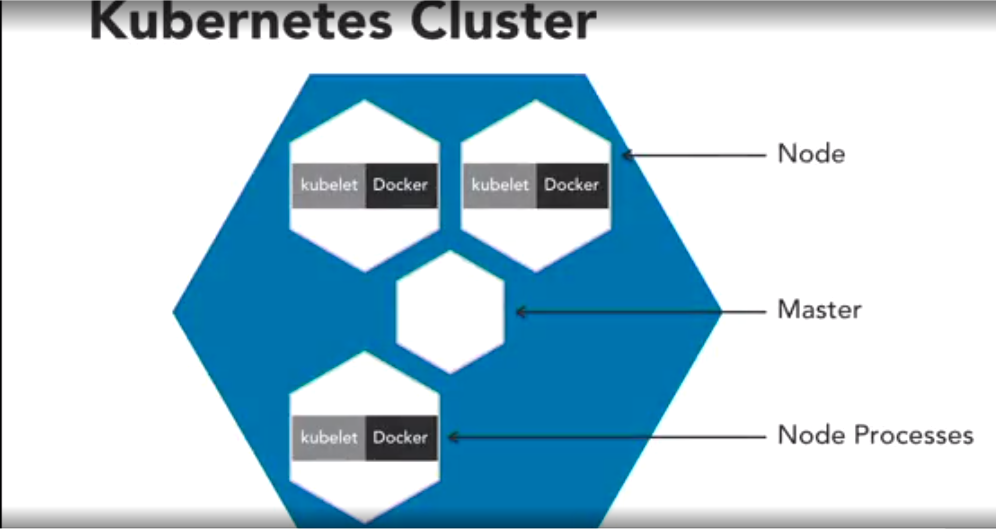
**Other Implementations**

* Major Players in Container Orchestration
  + Kubernetes
  + Docker Swarm
  + Rancher
  + Mesos
* Cloud-Specific Technologies
  + Amazon EC2 Container Service.
  + Google Anthos

**Kubernetes: The Terminology**

* Architecture of a Kubrnetes cluster.



* + Master Node: It is responsible for the overall management of the Kubernetes Cluster. It's got three components that take care of communication, scheduling, and controllers.
    - API Server
    - Scheduler
    - Controller Manager
  + API Server
    - The Kube API Server, as the name states, allows you to interact with the Kubernetes API.
    - It is the front end of the Kubernetes control plane.
  + Scheduler
    - Scheduler watches created Pods, who do not have a Node design yet, and designs the Pod to run on a specific Node.
  + Controller Manager
    - The Controller Manager runs controllers.
    - These are background threads that run tasks in a cluster.
    - The controller actually has a bunch of different roles, but that's all compiled into a single binary.
    - The roles include, the Node Controller, who's responsible for the worker states, the Replication Controller, which is responsible for maintaining the correct number of Pods for the replicated controllers, the End-Point Controller, which joins services and Pods together. Service account and token controllers that handle access management.
    - Finally, there's that CD, which is a simple distributed key value stored.
  + Kubernetes uses etcd as its database and stores all cluster data here. Some of the information that might be stored is job scheduling info, Pod details, stage information, etc. And, that's the Master Node.
  + We can interact with the Master Node using the Kubectl application, which is the command line interface for Kubernetes. Kubectl is also also called Kubectl. Kubectl has a config file called a Kubeconfig. This file has server information, as well as authentication information to access the API Server.
  + Worker Nodes are the Nodes where your applications operate. The Worker Nodes communicate back with the Master Node.
  + Communication to a Worker Node is handled by the Kubelet Process. It's an agent that communicates with the API Server to see if Pods have been designed to the Nodes. It executes Pod containers via the container engine. It mounts and runs Pod volume and secrets. And finally, is aware of Pod of Node states and responds back to the Master.
  + Kubernetes is a container orchestrator, so the expectation is that you have a container native platform running on your Worker Nodes. This is where Docker comes in and works together with Kubelet to run containers on the Node. We could use alternate container platforms, as well, like (mumbles)
  + Kube-proxy process is the Network Proxy and load balancer for the service, on a single Worker Node. It handles the network routing for TCP and UDP Packets, and performs connection forwarding.
  + Having the Docker Daemon allows you to run containers. Containers of an application are tightly coupled together in a Pod.
  + By definition, a Pod is the smallest unit that can be scheduled as a deployment in Kubernetes. This group of containers share storage, Linux namespace, IP addresses, amongst other things. They're also located and share resources that are always scheduled together.
  + Once Pods have been deployed, and are running, the Kubelet process communicates with the Pods to check on state and health, and the Kube-proxy routes any packets to the Pods from other resources that might be wanting to communicate with them.
  + Worker Nodes can be exposed to the internet via load balancer. And, traffic coming into the Nodes is also handled by the Kube-proxy, which is how an End-user ends up talking to a Kubernetes application.
* Basic building Blocks: Nodes and Pods
  + Node: The node serves as a worker machine in a K8s cluster. One important thing to note is that the node can be a physical computer or a virtual machine.
    - Requirements
      * A kubelet running
      * Container tooling like Docker
      * A kube-proxy process running
      * Supervisord
  + Minikube
    - Lightweight kubernetes implementation that creates aVM on your local machine and deploys a simple cluster containing only one node.
      * Pods
        + Pod is the simplest unit that you can interact with. we can create, deploy, and delete Pods, and it represents one running process in your cluster.
        + A Pod contains the following things.
        + Your Docker application container, storage resources
        + A unique network IP, and options that govern how the container should run.
        + In some scenarios, you can have multiple docker containers running in a Pod, but a Pod represents one single unit of deployment, a single instance of an application in Kubernetes that's tightly coupled and shares resources.
        + Pods are designed to be ephemeral, disposable entities. We never create Pods just by themselves in a production application.We only do that when we need to test whether the underlying containers actually work.
        + Pods also don't self-heal. If a Pod dies, for some reason, it will not be rescheduled. Also, if a Pod is exited from a Node because of lack of resources, it will not be restarted on different healthier Nodes.
        + There are higher level constructs to manage and add stability to Pods, called controllers. So pro-tip, don't use a Pod directly. Use a controller instead, like a deployment.
        + Through its life-cycle, a Pod has the following states.
        + Pending, which means that the Pod has been accepted by the Kubernete system, but a container has not been created yet.
        + Running, where a Pod has been scheduled on a Node, and all of its containers are created, and at least one container is in a running state.
        + Succeeded, which means that all the containers in the Pod have exited with an exit stat of zero, which indicates successful execution, and will not be restarted.
        + A failed state, which means all the containers in the Pod have exited and at least one container has failed and returned a non-zero exit status. which is the CrashLoopBackOff. This is where a container fails to start, for some reason, and then Kubernetes tries over and over and over again to restart the Pod.
    - Deployments, ReplicaSets, and Services
    - Before we dive into the details, it's good to understand what problems controllers actually help us solve. These are
    - Application Reliability, where multiple instances of an application running prevent problems if one or more instances fails.
    - Scaling. When your pods experience a high volume of requests, Kubernetes allows you to scale up your pods, allowing for a better user experience.
    - Load balancing, where having multiple versions of a pod running allows traffic to flow to different pods and doesn't overload one single pod or a node.
    - Kinds of controllers
      * ReplicaSets
      * Deployments
      * DaemonSets
      * Jobs
      * Services.
    - ReplicaSets:
      * A ReplicaSet has one job, it ensures that the specified number of replicas for a pod are running at all times.
      * If the number of pods is less than what the ReplicaSet expects, for example, when a pod might have crashed, the ReplicaSet controller will start up a new pod, however, you can't actually declare a ReplicaSet by itself. We need to use it within a deployment.
    - Deployments:
      * Provides declarative updates for pods and ReplicaSets. This means that you can describe the desired state of a deployment in a YAML file and the Deployment Controller will align the actual state to match.
      * Deployments can be defined to create new ReplicaSets or replace existing ones with new ones. Most applications are package deployments, so chances are, we will end up creating Deployments more than anything else. Essentially, a deployment manages a ReplicaSet which, in turn, manages a pod.
      * The benefit of this architecture is that deployments can automatically support a role-back mechanism.
      * A new ReplicaSet is created each time a new Deployment config is deployed, but it also keeps the old ReplicaSet. This allows you to easily roll back to the old state if something isn't quite working correctly.
      * Deployment Controllers and objects are higher-level constructs that were introduced to solve specific issues. Pod management, were running a ReplicaSet allows us to deploy a number of pods and check their status as a single unit.
      * Scaling a ReplicaSet, which scales out the pods and allows for the deployment to handle more traffic.
      * Pod updates and role-backs, where the Deployment Controller allows updates to the PodTemplateSpec.
      * This creates a new ReplicaSet and deploys a newer version of the pod. Also, if we don't like what you see in the newer version of the pod, just roll-back to the old ReplicaSet.
      * Sometimes we have larger changesets or multiple updates that need to happen to a deployment. In these scenarios, we can pause a deployment, make all the necessary updates, and then resume the deployment. Once the deployment is resumed, the new ReplicaSet will be started up and the deployment will update as expected. Please note that while a deployment is paused, it means that only updates are paused, but traffic will still get passed to the existing ReplicaSet as expected.
      * And finally, status. Getting the deployment status is an easy way to check for the health of your pods and identify issues during a roll-out.
    - DaemonSets:
      * DaemonSets ensure that all nodes run a copy of a specific pod. As nodes are added or removed from the cluster, a DaemonSet will add or remove the required pods.
      * Deleting a DaemonSet will also clean up all the pods that it created. The typical use case for a DaemonSet is to run a single log aggregator or monitoring agent on a node.
    - Job:
      * As the name suggests, is basically a supervisor process for pods carrying out batch processes to completion.
      * As the pod completes successfully, the job tracks information about the completion state of the pod.
      * Jobs are used to run individual processes that need to run once and complete successfully.
      * Typically, jobs are run as a cron job to run a specific process at a specific time and repeat at another time. We might use a cron job to run a nightly report or database backups.
    - Service
      * A service provides network connectivity to one or more pods in your cluster. When you create a service, it's designed with a unique IP address that never changes through the lifetime of the service. Pods are then configured to talk to the service and can rely on the service IP on any requests that might be sent to the pod.
      * Services are a really important concept because they allow one set of pods to communicate with another set of pods in an easy way.
      * It's best practice to use a service when we are trying to get two deployments to talk to each other. That way, the pod in the first deployment always has an IP that they can communicate with regardless of whether the pod IPs in the second deployment changes.
      * For example, when your front-end deployment needs to call a back-end deployment, you want to address the back-end with a service IP. Using the Backend Pod IP is a bad choice here because it can change over time and would wreak havoc for your application.
      * A service provides an unchanging address so that the Frontend Pods can effectively talk to them at all times. There's a few kinds of services that you can use.
      * Internal services, where an IP is only reachable from within the cluster, this is the cluster IP in Kubernetes speak.
      * External services where services running web servers, or publicly accessible pods, are exposed through an external endpoint. These endpoints are available on each node through a specific port. This is called a NodePort in Kubernetes speak.
      * Load balancer. This is for use cases when you want to expose your application to the public internet. It's only used when you are using Kubernetes in a cloud environment backed by a cloud provider such as AWS.
* **Labels, Selectors, and Namespaces.**
  + Labels
    - Labels are key value pairs that are attached to objects like pods, services, and deployments.
    - Labels are for us, the users of Kubernetes, to identify attributes for objects. Typically, labels are used to organize clusters in some meaningful way. They can be added at deployment time, or later on and changed at any time.
    - Label keys are unique per object. Here are some examples of labels we might use. We might have a release label that has stable or canary deploys.
    - We could have environment labels that specify the environment, such as dev, qa, production, et cetera, or perhaps tier labels to signify that something is a front end or a back end tier.
    - labels are very specific to your use case, and they are built for users, so think about what your environment looks like, and how you want to organize your applications, and then get your label maker out. By themselves, labels aren't really that powerful.
  + Selectors
    - But when you add selectors, we introduce a very powerful feature. With labels and selectors, you can identify a specific set of objects. There are two kinds of selectors, equality-based and set-based. Equality-based selectors include the equals and not equals.
    - Where the equals represents equality, where two labels or values of labels should be equal.
    - Not equal represents inequality. This means that the values of the labels should not be equal. Next, we have set-based selectors that include IN, NOTIN, and EXISTS operators. The IN operator specifies that the value should be in a set of defined values. The NOTIN operators specified that the value should not be in a set of defined values.
    - And finally, the EXISTS operator is used to determine whether a label exists or not.
    - Labels and label selectors are typically used with a kubectl command to list and filter objects.
  + Namespaces.
    - Unlike labels and selectors, the namespace concept is a feature of Kubernetes that allows you to have multiple virtual clusters backed by the same physical cluster.
    - Namespaces are a great concept to use for large enterprises where there are many users and teams and you want to give access to different teams but at the same time have a rough idea of who owns what in the Kubernetes environment. For example, if you have a big e-commerce company, you might have a namespace for your catalog team, card team and order status team to run their different applications. It's also a great way to divide cluster resources between multiple users and this can be done using resource quotas.
    - Namespaces provide scope for names. Names of resources, like deployments and pods, must be unique within the namespace, but not necessarily across separate namespaces.
    - Example, the catalog team and the card team can have an application name authentication in their own namespaces. When you launch Kubernetes, there is a default namespace where all our objects get placed, but you are allowed to create new namespaces as and when you wish.
    - Also notice that when you install newer application Kubernetes, they'll typically install in a brand new namespace, so that they don't interfere with your existing cluster and cause confusion.
* **Kubelet and Kube proxy**
  + The kubelet is the Kubernetes node agent that runs on each node.
  + It has many roles. It communicates with the API server to see if pods have been assigned to the nodes. It executes the pod containers via the container engine. It mounts and runs pod volumes and secrets. It executes health checks and is aware of pod and node status and reports that back to the API server.
  + The kubelet works in terms of Podspec, which is just a YAML file that describes the pod.
  + The kubelet takes a set of Podspecs that are provided by the kube-apiserver and ensures that the containers described in those Podscpecs are running and healthy.
  + It's important to note that the kubelet only manages containers that were created by the API server, and not any other containers that might be running on the node.
  + We can also manage the kubelet without an API server by using HTTP endpoint or a file
  + The network proxy is called the kube-proxy. This is another process that runs on all the worker nodes.
  + The kube-proxy reflects services that are defined in the API on each node and can do simple network stream or round-robin forwarding across a set of backends.
  + Service cluster IPs and ports are currently found through Docker-link compatible environment variables specifying ports opened by the service proxy.
  + The kube-proxy has three modes: the user space mode, Iptables mode, and ipvs mode, which is an alpha feature in Kubernetes 1.8.
  + The user space is the most common mode,
  + These modes are important when it comes to using services.
  + Services are defined against the API server.
  + The kube-proxy watches the API server for addition and removal of services. For each new service, kube-proxy opens a randomly chosen port on the local node. Any connections made to that port are proxied to one of the corresponding back-end pods.

**Installing Kubernetes**

* Prerequisites
  + Docker
  + Hypervisor
    - wget https://download.virtualbox.org/virtualbox/rpm/rhel/virtualbox.repo -P /etc/yum.repos.d/
    - rpm --import https://www.virtualbox.org/download/oracle\_vbox.asc
    - dnf install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
    - dnf update
    - dnf install binutils kernel-devel kernel-headers libgomp make patch gcc glibc-headers glibc-devel dkms -y
    - dnf install VirtualBox-6.1
    - virtualbox
  + Kubernetes
    - curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
    - curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
    - echo "$(<kubectl.sha256) kubectl" | sha256sum --check
    - sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
    - kubectl version --client
  + Minikube
    - curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
    - sudo install minikube-linux-amd64 /usr/local/bin/minikube
    - minikube start
  + Way to Run Kubernetes
    - Minikube
    - Docker Desktop
    - Kubernetes in DOcker
    - Managed Kubernetes services in a cloud - for example Amazon Elastic Kubernetes Service (EK**S)**

**Scenario: Running first application**

* Running Minikube

[mobisht@mobisht kubernetes]$ minikube start

😄 minikube v1.21.0 on Redhat 8.3

✨ Using the virtualbox driver based on existing profile

👍 Starting control plane node minikube in cluster minikube

🔄 Restarting existing virtualbox VM for "minikube" ...

🐳 Preparing Kubernetes v1.20.7 on Docker 20.10.6 ...

🔎 Verifying Kubernetes components...

▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5

🌟 Enabled addons: storage-provisioner, default-storageclass

🏄 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

* Getting Nodes

[mobisht@mobisht kubernetes]$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

minikube Ready control-plane,master 2d15h v1.20.7

* Helloworld yaml file

[mobisht@mobisht 03\_04]$ cat helloworld.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: helloworld

spec:

selector:

matchLabels:

app: helloworld

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

ports:

- containerPort: 80[mobisht@mobisht 03\_04]$

* Get all the running services/kubernetes

[mobisht@mobisht 03\_04]$ kubectl get all

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

* Deploy helloworld application

[mobisht@mobisht 03\_04]$ kubectl create -f helloworld.yaml

deployment.apps/helloworld created

* To get all the services

[mobisht@mobisht 03\_04]$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/helloworld-66f646b9bb-t57lp 0/1 ContainerCreating 0 9s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/helloworld 0/1 1 0 9s

NAME DESIRED CURRENT READY AGE

replicaset.apps/helloworld-66f646b9bb 1 1 0 9s

* Exposing helloworld application

[mobisht@mobisht 03\_04]$ kubectl expose deployment helloworld --type=NodePort

service/helloworld exposed

* Get all the resources

[mobisht@mobisht 03\_04]$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/helloworld-66f646b9bb-t57lp 1/1 Running 0 104s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/helloworld NodePort 10.105.90.2 <none> 80:32743/TCP 8s

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/helloworld 1/1 1 1 104s

NAME DESIRED CURRENT READY AGE

replicaset.apps/helloworld-66f646b9bb 1 1 1 104s

* Running helloworld service

[mobisht@mobisht 03\_04]$ minikube service helloworld

|-----------|------------|-------------|-----------------------------|

| NAMESPACE | NAME | TARGET PORT | URL |

|-----------|------------|-------------|-----------------------------|

| default | helloworld | 80 | http://192.168.99.100:32743 |

|-----------|------------|-------------|-----------------------------|

🎉 Opening service default/helloworld in default browser...

**Scenario: Running deployment services from a single file.**

* To get all resources

[mobisht@mobisht 03\_04]$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/helloworld-66f646b9bb-t57lp 1/1 Running 0 11m

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/helloworld NodePort 10.105.90.2 <none> 80:32743/TCP 10m

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/helloworld 1/1 1 1 11m

NAME DESIRED CURRENT READY AGE

replicaset.apps/helloworld-66f646b9bb 1 1 1 11m

* To get all the deployment

[mobisht@mobisht 03\_04]$ kubectl get deployment

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld 1/1 1 1 12m

* To get deployment details in yaml format.

[mobisht@mobisht 03\_04]$ kubectl get deployment/helloworld -o yaml

apiVersion: apps/v1

kind: Deployment

metadata:

annotations:

deployment.kubernetes.io/revision: "1"

creationTimestamp: "2021-07-05T05:04:21Z"

generation: 1

name: helloworld

namespace: default

resourceVersion: "858"

uid: 7627037a-3823-481b-a9f7-ca987e463888

spec:

progressDeadlineSeconds: 600

replicas: 1

revisionHistoryLimit: 10

selector:

matchLabels:

app: helloworld

strategy:

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

type: RollingUpdate

template:

metadata:

creationTimestamp: null

labels:

app: helloworld

spec:

containers:

- image: karthequian/helloworld:latest

imagePullPolicy: Always

name: helloworld

ports:

- containerPort: 80

protocol: TCP

resources: {}

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

dnsPolicy: ClusterFirst

restartPolicy: Always

schedulerName: default-scheduler

securityContext: {}

terminationGracePeriodSeconds: 30

status:

availableReplicas: 1

conditions:

- lastTransitionTime: "2021-07-05T05:05:01Z"

lastUpdateTime: "2021-07-05T05:05:01Z"

message: Deployment has minimum availability.

reason: MinimumReplicasAvailable

status: "True"

type: Available

- lastTransitionTime: "2021-07-05T05:04:21Z"

lastUpdateTime: "2021-07-05T05:05:01Z"

message: ReplicaSet "helloworld-66f646b9bb" has successfully progressed.

reason: NewReplicaSetAvailable

status: "True"

type: Progressing

observedGeneration: 1

readyReplicas: 1

replicas: 1

updatedReplicas: 1

* To get all the services running

[mobisht@mobisht 03\_04]$ kubectl get service

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

helloworld NodePort 10.105.90.2 <none> 80:32743/TCP 12m

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

[mobisht@mobisht 03\_04]$

[mobisht@mobisht 03\_04]$ kubectl get service/helloworld -o yaml

apiVersion: v1

kind: Service

metadata:

creationTimestamp: "2021-07-05T05:05:57Z"

name: helloworld

namespace: default

resourceVersion: "901"

uid: 5276c79e-ba0c-4917-bfeb-0b49d16cf3c7

spec:

clusterIP: 10.105.90.2

clusterIPs:

- 10.105.90.2

externalTrafficPolicy: Cluster

ports:

- nodePort: 32743

port: 80

protocol: TCP

targetPort: 80

selector:

app: helloworld

sessionAffinity: None

type: NodePort

status:

loadBalancer: {}

* Simple application file

[mobisht@mobisht 03\_05]$ cat helloworld.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: helloworld

spec:

selector:

matchLabels:

app: helloworld

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

ports:

- containerPort: 80

* Deployment file

[mobisht@mobisht 03\_05]$ cat helloworld-deployment.yml

# Helloworld application- just the deployment

apiVersion: apps/v1

kind: Deployment

metadata:

name: helloworld-deployment

spec:

selector:

matchLabels:

app: helloworld

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

ports:

- containerPort: 80

* Service file.

[mobisht@mobisht 03\_05]$ cat helloworld-service.yml

# Helloworld service for nodeports

apiVersion: v1

kind: Service

metadata:

name: helloworld-service

spec:

type: NodePort

ports:

- port: 80

protocol: TCP

targetPort: 80

selector:

app: helloworld

* Running deployment

[mobisht@mobisht 03\_05]$ kubectl create -f helloworld-deployment.yml

deployment.apps/helloworld-deployment created

* Running service

[mobisht@mobisht 03\_05]$ kubectl create -f helloworld-service.yml

service/helloworld-service created

* Get all resources

[mobisht@mobisht 03\_05]$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/helloworld-66f646b9bb-t57lp 1/1 Running 0 17m

pod/helloworld-deployment-66f646b9bb-sbxgs 1/1 Running 0 23s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/helloworld NodePort 10.105.90.2 <none> 80:32743/TCP 16m

service/helloworld-service NodePort 10.106.113.52 <none> 80:32489/TCP 6s

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/helloworld 1/1 1 1 17m

deployment.apps/helloworld-deployment 1/1 1 1 23s

NAME DESIRED CURRENT READY AGE

replicaset.apps/helloworld-66f646b9bb 1 1 1 17m

replicaset.apps/helloworld-deployment-66f646b9bb 1 1 1 24s

* Single file consist of all

[mobisht@mobisht 03\_05]$ cat helloworld-all.yml

apiVersion: apps/v1

kind: Deployment

metadata:

name: helloworld-all-deployment

spec:

selector:

matchLabels:

app: helloworld

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: helloworld-all-service

spec:

type: NodePort

ports:

- port: 80

protocol: TCP

targetPort: 80

selector:

app: helloworld

* Running file consist of all.

[mobisht@mobisht 03\_05]$ kubectl create -f helloworld-all.yml

deployment.apps/helloworld-all-deployment created

service/helloworld-all-service created

* Getting all resources.

[mobisht@mobisht 03\_05]$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/helloworld-66f646b9bb-t57lp 1/1 Running 0 18m

pod/helloworld-all-deployment-66f646b9bb-h9zsv 0/1 ContainerCreating 0 7s

pod/helloworld-deployment-66f646b9bb-sbxgs 1/1 Running 0 102s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/helloworld NodePort 10.105.90.2 <none> 80:32743/TCP 17m

service/helloworld-all-service NodePort 10.100.92.142 <none> 80:31131/TCP 7s

service/helloworld-service NodePort 10.106.113.52 <none> 80:32489/TCP 85s

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/helloworld 1/1 1 1 18m

deployment.apps/helloworld-all-deployment 0/1 1 0 7s

deployment.apps/helloworld-deployment 1/1 1 1 102s

NAME DESIRED CURRENT READY AGE

replicaset.apps/helloworld-66f646b9bb 1 1 1 18m

replicaset.apps/helloworld-all-deployment-66f646b9bb 1 1 0 7s

replicaset.apps/helloworld-deployment-66f646b9bb 1 1 1 102s

* Getting all deployment

[mobisht@mobisht 03\_05]$ kubectl get deployment

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld 1/1 1 1 19m

helloworld-all-deployment 1/1 1 1 30s

helloworld-deployment 1/1 1 1 2m5s

* Getting all services

[mobisht@mobisht 03\_05]$ kubectl get services

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

helloworld NodePort 10.105.90.2 <none> 80:32743/TCP 17m

helloworld-all-service NodePort 10.100.92.142 <none> 80:31131/TCP 35s

helloworld-service NodePort 10.106.113.52 <none> 80:32489/TCP 113s

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d15h

**Scenarios: Clean up**

kubectl get pods,services,deployments,jobs,daemonset

kubectl delete deployments <deployment>

kubectl delete services <services>

kubectl delete pods <pods>

kubectl delete daemonset <daemonset>

**Scenario: Add, Change and Delete labels**

* File with labels:

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ cat helloworld-pod-with-labels.yml

apiVersion: v1

kind: Pod

metadata:

name: helloworld

labels:

env: production

author: karthequian

application\_type: ui

release-version: "1.0"

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

* Deploying app

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl create -f helloworld-pod-with-labels.yml

pod/helloworld created

* Pods details

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld 1/1 Running 0 12s

* Show labels

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

helloworld 1/1 Running 0 29s application\_type=ui,author=karthequian,env=production,release-version=1.0

* Tagging labels

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl label po/helloworld app=helloworldapp --overwrite

pod/helloworld labeled

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

helloworld 1/1 Running 0 2m23s app=helloworldapp,application\_type=ui,author=karthequian,env=production,release-version=1.0

* Delete labels

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl label pod/helloworld app-

pod/helloworld labeled

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

helloworld 1/1 Running 0 3m9s application\_type=ui,author=karthequian,env=production,release-version=1.0

**Scenario: Select by labels**

* Creating pods

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl create -f sample-infrastructure-with-labels.yml

pod/homepage-dev created

pod/homepage-staging created

pod/homepage-prod created

pod/login-dev created

pod/login-staging created

pod/login-prod created

pod/cart-dev created

pod/cart-staging created

pod/cart-prod created

pod/social-dev created

pod/social-staging created

pod/social-prod created

pod/catalog-dev created

pod/catalog-staging created

pod/catalog-prod created

pod/quote-dev created

pod/quote-staging created

pod/quote-prod created

pod/ordering-dev created

pod/ordering-staging created

pod/ordering-prod created

* Check pods

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

cart-dev 1/1 Running 0 2m18s

cart-prod 1/1 Running 0 2m18s

cart-staging 1/1 Running 0 2m18s

catalog-dev 1/1 Running 0 2m18s

catalog-prod 1/1 Running 0 2m18s

catalog-staging 1/1 Running 0 2m18s

helloworld 1/1 Running 0 10m

homepage-dev 1/1 Running 0 2m18s

homepage-prod 1/1 Running 0 2m18s

homepage-staging 1/1 Running 0 2m18s

login-dev 1/1 Running 0 2m18s

login-prod 1/1 Running 0 2m18s

login-staging 1/1 Running 0 2m18s

ordering-dev 1/1 Running 0 2m18s

ordering-prod 1/1 Running 0 2m18s

ordering-staging 1/1 Running 0 2m18s

quote-dev 1/1 Running 0 2m18s

quote-prod 1/1 Running 0 2m18s

quote-staging 1/1 Running 0 2m18s

social-dev 1/1 Running 0 2m18s

social-prod 1/1 Running 0 2m18s

social-staging 1/1 Running 0 2m18s

* Show labels of all the pods

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

cart-dev 1/1 Running 0 2m38s application\_type=api,dev-lead=carisa,env=development,release-version=1.0,team=ecommerce

cart-prod 1/1 Running 0 2m38s application\_type=api,dev-lead=carisa,env=production,release-version=1.0,team=ecommerce

cart-staging 1/1 Running 0 2m38s application\_type=api,dev-lead=carisa,env=staging,release-version=1.0,team=ecommerce

catalog-dev 1/1 Running 0 2m38s application\_type=api,dev-lead=daniel,env=development,release-version=4.0,team=ecommerce

catalog-prod 1/1 Running 0 2m38s application\_type=api,dev-lead=daniel,env=production,release-version=4.0,team=ecommerce

catalog-staging 1/1 Running 0 2m38s application\_type=api,dev-lead=daniel,env=staging,release-version=4.0,team=ecommerce

helloworld 1/1 Running 0 10m application\_type=ui,author=karthequian,env=production,release-version=1.0

homepage-dev 1/1 Running 0 2m38s application\_type=ui,dev-lead=karthik,env=development,release-version=12.0,team=web

homepage-prod 1/1 Running 0 2m38s application\_type=ui,dev-lead=karthik,env=production,release-version=12.0,team=web

homepage-staging 1/1 Running 0 2m38s application\_type=ui,dev-lead=karthik,env=staging,release-version=12.0,team=web

login-dev 1/1 Running 0 2m38s application\_type=api,dev-lead=jim,env=development,release-version=1.0,team=auth

login-prod 1/1 Running 0 2m38s application\_type=api,dev-lead=jim,env=production,release-version=1.0,team=auth

login-staging 1/1 Running 0 2m38s application\_type=api,dev-lead=jim,env=staging,release-version=1.0,team=auth

ordering-dev 1/1 Running 0 2m38s application\_type=backend,dev-lead=chen,env=development,release-version=2.0,team=purchasing

ordering-prod 1/1 Running 0 2m38s application\_type=backend,dev-lead=chen,env=production,release-version=2.0,team=purchasing

ordering-staging 1/1 Running 0 2m38s application\_type=backend,dev-lead=chen,env=staging,release-version=2.0,team=purchasing

quote-dev 1/1 Running 0 2m38s application\_type=api,dev-lead=amy,env=development,release-version=2.0,team=ecommerce

quote-prod 1/1 Running 0 2m38s application\_type=api,dev-lead=amy,env=production,release-version=1.0,team=ecommerce

quote-staging 1/1 Running 0 2m38s application\_type=api,dev-lead=amy,env=staging,release-version=2.0,team=ecommerce

social-dev 1/1 Running 0 2m38s application\_type=api,dev-lead=carisa,env=development,release-version=2.0,team=marketing

social-prod 1/1 Running 0 2m38s application\_type=api,dev-lead=marketing,env=production,release-version=1.0,team=marketing

social-staging 1/1 Running 0 2m38s application\_type=api,dev-lead=marketing,env=staging,release-version=1.0,team=marketing

* Search by labels

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods -selector env=production

Unable to connect to the server: dial tcp: lookup elector on 10.75.5.25:53: no such host

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --selector env=production

NAME READY STATUS RESTARTS AGE

cart-prod 1/1 Running 0 4m13s

catalog-prod 1/1 Running 0 4m13s

helloworld 1/1 Running 0 12m

homepage-prod 1/1 Running 0 4m13s

login-prod 1/1 Running 0 4m13s

ordering-prod 1/1 Running 0 4m13s

quote-prod 1/1 Running 0 4m13s

social-prod 1/1 Running 0 4m13s

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --selector env=production --show-labels

NAME READY STATUS RESTARTS AGE LABELS

cart-prod 1/1 Running 0 4m43s application\_type=api,dev-lead=carisa,env=production,release-version=1.0,team=ecommerce

catalog-prod 1/1 Running 0 4m43s application\_type=api,dev-lead=daniel,env=production,release-version=4.0,team=ecommerce

helloworld 1/1 Running 0 12m application\_type=ui,author=karthequian,env=production,release-version=1.0

homepage-prod 1/1 Running 0 4m43s application\_type=ui,dev-lead=karthik,env=production,release-version=12.0,team=web

login-prod 1/1 Running 0 4m43s application\_type=api,dev-lead=jim,env=production,release-version=1.0,team=auth

ordering-prod 1/1 Running 0 4m43s application\_type=backend,dev-lead=chen,env=production,release-version=2.0,team=purchasing

quote-prod 1/1 Running 0 4m43s application\_type=api,dev-lead=amy,env=production,release-version=1.0,team=ecommerce

social-prod 1/1 Running 0 4m43s application\_type=api,dev-lead=marketing,env=production,release-version=1.0,team=marketing

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --selector dev-lead=chen --show-labels

NAME READY STATUS RESTARTS AGE LABELS

ordering-dev 1/1 Running 0 5m38s application\_type=backend,dev-lead=chen,env=development,release-version=2.0,team=purchasing

ordering-prod 1/1 Running 0 5m38s application\_type=backend,dev-lead=chen,env=production,release-version=2.0,team=purchasing

ordering-staging 1/1 Running 0 5m38s application\_type=backend,dev-lead=chen,env=staging,release-version=2.0,team=purchasing

* Search multiple labels

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods --selector dev-lead=chen,env=production --show-labels

NAME READY STATUS RESTARTS AGE LABELS

ordering-prod 1/1 Running 0 6m19s application\_type=backend,dev-lead=chen,env=production,release-version=2.0,team=purchasing

* Using IN

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods -l 'release-version in (1.0,2.0)'

NAME READY STATUS RESTARTS AGE

cart-dev 1/1 Running 0 7m59s

cart-prod 1/1 Running 0 7m59s

cart-staging 1/1 Running 0 7m59s

helloworld 1/1 Running 0 16m

login-dev 1/1 Running 0 7m59s

login-prod 1/1 Running 0 7m59s

login-staging 1/1 Running 0 7m59s

ordering-dev 1/1 Running 0 7m59s

ordering-prod 1/1 Running 0 7m59s

ordering-staging 1/1 Running 0 7m59s

quote-dev 1/1 Running 0 7m59s

quote-prod 1/1 Running 0 7m59s

quote-staging 1/1 Running 0 7m59s

social-dev 1/1 Running 0 7m59s

social-prod 1/1 Running 0 7m59s

social-staging 1/1 Running 0 7m59s

* NotIN

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl get pods -l 'release-version notin (1.0,2.0)' --show-labels

NAME READY STATUS RESTARTS AGE LABELS

catalog-dev 1/1 Running 0 8m36s application\_type=api,dev-lead=daniel,env=development,release-version=4.0,team=ecommerce

catalog-prod 1/1 Running 0 8m36s application\_type=api,dev-lead=daniel,env=production,release-version=4.0,team=ecommerce

catalog-staging 1/1 Running 0 8m36s application\_type=api,dev-lead=daniel,env=staging,release-version=4.0,team=ecommerce

homepage-dev 1/1 Running 0 8m36s application\_type=ui,dev-lead=karthik,env=development,release-version=12.0,team=web

homepage-prod 1/1 Running 0 8m36s application\_type=ui,dev-lead=karthik,env=production,release-version=12.0,team=web

homepage-staging 1/1 Running 0 8m36s application\_type=ui,dev-lead=karthik,env=staging,release-version=12.0,team=web

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl

helloworld-pod-with-labels.yml Readme.md sample-infrastructure-with-labels.yml

* Deleting pod using labels

[mobisht@mobisht 04\_01\_Adding\_labels\_to\_the\_app]$ kubectl delete pods -l dev-lead=karthik

pod "homepage-dev" deleted

pod "homepage-prod" deleted

pod "homepage-staging" deleted

**Scenarios: Health checks**

In this scenario we will check how to check the health of pods and deployment.

* With no Error:

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ cat helloworld-with-probes.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: helloworld-deployment-with-probe

spec:

selector:

matchLabels:

app: helloworld

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

ports:

- containerPort: 80

readinessProbe:

# length of time to wait for a pod to initialize

# after pod startup, before applying health checking

initialDelaySeconds: 5

# Amount of time to wait before timing out

timeoutSeconds: 1

# Probe for http

httpGet:

# Path to probe

path: /

# Port to probe

port: 80

livenessProbe:

# length of time to wait for a pod to initialize

# after pod startup, before applying health checking

initialDelaySeconds: 5

# Amount of time to wait before timing out

timeoutSeconds: 1

# Probe for http

httpGet:

# Path to probe

path: /

# Port to probe

port: 80

* Deploying the application

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl create -f helloworld-with-probes.yaml

deployment.apps/helloworld-deployment-with-probe created

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld-deployment-with-probe  **0/1**  1 0 12s

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld-deployment-with-probe **1/1**  1 1 37s

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get replicaset

NAME DESIRED CURRENT READY AGE

helloworld-deployment-with-probe-5c5b86d4c4 1 1 1 52s

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

helloworld-deployment-with-probe-5c5b86d4c4-gxkcp  **1/1**  Running 0 5m44s app=helloworld,pod-template-hash=5c5b86d4c4

* Deploying with error (Port is different)

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ cat helloworld-with-bad-readiness-probe.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: helloworld-deployment-with-bad-readiness-probe

spec:

selector:

matchLabels:

app: helloworld

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

ports:

**- containerPort: 80**

readinessProbe:

# length of time to wait for a pod to initialize

# after pod startup, before applying health checking

initialDelaySeconds: 5

# Amount of time to wait before timing out

timeoutSeconds: 1

# Probe for http

httpGet:

# Path to probe

path: /

# Port to probe

**port: 90**

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl create -f helloworld-with-bad-readiness-probe.yaml

deployment.apps/helloworld-deployment-with-bad-readiness-probe created

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get deployment

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld-deployment-with-bad-readiness-probe **0/1**  1 0 9s

helloworld-deployment-with-probe 1/1 1 1 7m4s

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v **0/1** Running 0 29s

helloworld-deployment-with-probe-5c5b86d4c4-gxkcp 1/1 Running 0 7m24s

[[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl describe pod/helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v

Name: helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v

Namespace: default

Priority: 0

Node: minikube/192.168.99.100

Start Time: Mon, 05 Jul 2021 12:27:38 +0530

Labels: app=helloworld

pod-template-hash=6fb9c97fdd

Annotations: <none>

Status: Running

IP: 172.17.0.2

IPs:

IP: 172.17.0.2

Controlled By: ReplicaSet/helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd

Containers:

helloworld:

Container ID: docker://ddfaf040bb979112ad790a5abc971eee7aa3c55f6c05017921168c7329cf3b21

Image: karthequian/helloworld:latest

Image ID: docker-pullable://karthequian/helloworld@sha256:48413fdddeae11e4732896e49b6d82979847955666ed95e4d6e57b433920c9e1

Port: 80/TCP

Host Port: 0/TCP

State: Running

Started: Mon, 05 Jul 2021 12:27:43 +0530

Ready: False

Restart Count: 0

Readiness: http-get http://:90/ delay=5s timeout=1s period=10s #success=1 #failure=3

Environment: <none>

Mounts:

/var/run/secrets/kubernetes.io/serviceaccount from default-token-khvqr (ro)

Conditions:

Type Status

Initialized True

Ready False

ContainersReady False

PodScheduled True

Volumes:

default-token-khvqr:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-khvqr

Optional: false

QoS Class: BestEffort

Node-Selectors: <none>

Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s

node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Scheduled 70s default-scheduler Successfully assigned default/helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v to minikube

Normal Pulling 69s kubelet Pulling image "karthequian/helloworld:latest"

Normal Pulled 65s kubelet Successfully pulled image "karthequian/helloworld:latest" in 4.005636029s

Normal Created 65s kubelet Created container helloworld

**Normal Started 65s kubelet Started container helloworld**

**Warning Unhealthy 10s (x6 over 60s) kubelet Readiness probe failed: Get "http://172.17.0.2:90/": dial tcp 172.17.0.2:90: connect: connection refused**

* With error but putting “failureThreshold: 2”

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ cat helloworld-with-bad-liveness-probe.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: helloworld-deployment-with-bad-liveness-probe

spec:

selector:

matchLabels:

app: helloworld

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:latest

ports:

- **containerPort: 80**

livenessProbe:

# length of time to wait for a pod to initialize

# after pod startup, before applying health checking

initialDelaySeconds: 5

# How often (in seconds) to perform the probe.

periodSeconds: 5

# Amount of time to wait before timing out

timeoutSeconds: 1

# Kubernetes will try failureThreshold times before giving up and restarting the Pod

**failureThreshold: 2**

# Probe for http

httpGet:

# Path to probe

path: /

# Port to probe

**port: 90**

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl create -f helloworld-with-bad-liveness-probe.yaml

deployment.apps/helloworld-deployment-with-bad-liveness-probe created

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get deployment

NAME READY UP-TO-DATE AVAILABLE AGE

helloworld-deployment-with-bad-liveness-probe  **0/1**  1 0 6s

helloworld-deployment-with-bad-readiness-probe 0/1 1 0 2m51s

helloworld-deployment-with-probe 1/1 1 1 9m46s

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg  **1/1** Running 0 25s

helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v 0/1 Running 0 3m10s

helloworld-deployment-with-probe-5c5b86d4c4-gxkcp 1/1 Running 0 10m

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg 1/1 Running 1 42s

helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v 0/1 Running 0 3m27s

helloworld-deployment-with-probe-5c5b86d4c4-gxkcp 1/1 Running 0 10m

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg 1/1 Running **2** 48s

helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v 0/1 Running 0 3m33s

helloworld-deployment-with-probe-5c5b86d4c4-gxkcp 1/1 Running 0 10m

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg 1/1 Running **3**  63s

helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v 0/1 Running 0 3m48s

helloworld-deployment-with-probe-5c5b86d4c4-gxkcp 1/1 Running 0 10m

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg 0/1 **CrashLoopBackOff** 3 73s

helloworld-deployment-with-bad-readiness-probe-6fb9c97fdd-mwt5v 0/1 Running 0 3m58s

helloworld-deployment-with-probe-5c5b86d4c4-gxkcp 1/1 Running 0 10m

[mobisht@mobisht 04\_03\_Application\_health\_checks]$ kubectl describe pod/helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg

Name: helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg

Namespace: default

Priority: 0

Node: minikube/192.168.99.100

Start Time: Mon, 05 Jul 2021 12:30:23 +0530

Labels: app=helloworld

pod-template-hash=7dcb4c6699

Annotations: <none>

Status: Running

IP: 172.17.0.4

IPs:

IP: 172.17.0.4

Controlled By: ReplicaSet/helloworld-deployment-with-bad-liveness-probe-7dcb4c6699

Containers:

helloworld:

Container ID: docker://5622eef47559a38abbe7be98f112e4b371310d3ff7b548a75d0c0e42f3a61894

Image: karthequian/helloworld:latest

Image ID: docker-pullable://karthequian/helloworld@sha256:48413fdddeae11e4732896e49b6d82979847955666ed95e4d6e57b433920c9e1

Port: 80/TCP

Host Port: 0/TCP

State: Waiting

Reason: CrashLoopBackOff

Last State: Terminated

Reason: Completed

Exit Code: 0

Started: Mon, 05 Jul 2021 12:31:21 +0530

Finished: Mon, 05 Jul 2021 12:31:32 +0530

Ready: False

Restart Count: 3

Liveness: http-get http://:90/ delay=5s timeout=1s period=5s #success=1 #failure=2

Environment: <none>

Mounts:

/var/run/secrets/kubernetes.io/serviceaccount from default-token-khvqr (ro)

Conditions:

Type Status

Initialized True

Ready False

ContainersReady False

PodScheduled True

Volumes:

default-token-khvqr:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-khvqr

Optional: false

QoS Class: BestEffort

Node-Selectors: <none>

Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s

node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Scheduled 92s default-scheduler Successfully assigned default/helloworld-deployment-with-bad-liveness-probe-7dcb4c6699-cscjg to minikube

Normal Pulled 81s kubelet Successfully pulled image "karthequian/helloworld:latest" in 10.203574532s

Normal Pulled 64s kubelet Successfully pulled image "karthequian/helloworld:latest" in 3.616070048s

Normal Pulled 49s kubelet Successfully pulled image "karthequian/helloworld:latest" in 3.779540318s

Warning Unhealthy 38s (x6 over 73s) kubelet Liveness probe failed: Get **"http://172.17.0.4:90/": dial tcp 172.17.0.4:90: connect: connection refused**

**Normal Killing 38s (x3 over 68s) kubelet Container helloworld failed liveness probe,** will be restarted

Normal Pulling 38s (x4 over 91s) kubelet Pulling image "karthequian/helloworld:latest"

Normal Created 34s (x4 over 81s) kubelet Created container helloworld

Normal Started 34s (x4 over 81s) kubelet Started container helloworld

Normal Pulled 34s kubelet Successfully pulled image

**Scenario: Upgrade and rollout**

* File:

[mobisht@mobisht 04\_04\_Rolling\_updates]$ cat helloworld-black.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: navbar-deployment

spec:

selector:

matchLabels:

app: helloworld

replicas: 3 # tells deployment to run 3 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: helloworld

spec:

containers:

- name: helloworld

image: karthequian/helloworld:black

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: navbar-service

spec:

# if your cluster supports it, uncomment the following to automatically create

# an external load-balanced IP for the frontend service.

type: NodePort

ports:

- port: 80

protocol: TCP

targetPort: 80

selector:

app: helloworld

* Deploy application:

[mobisht@mobisht 04\_04\_Rolling\_updates]$ kubectl create -f helloworld-black.yaml --record

deployment.apps/navbar-deployment created

service/navbar-service created

[mobisht@mobisht 04\_04\_Rolling\_updates]$ minikube service navbar-service

|-----------|----------------|-------------|-----------------------------|

| NAMESPACE | NAME | TARGET PORT | URL |

|-----------|----------------|-------------|-----------------------------|

| default | navbar-service | 80 | http://192.168.99.100:31001 |

|-----------|----------------|-------------|-----------------------------|

🎉 Opening service default/navbar-service in default browser…

* Update

[mobisht@mobisht 04\_04\_Rolling\_updates]$ kubectl set image deployment/navbar-deployment helloworld=karthequian/helloworld:blue

deployment.apps/navbar-deployment image updated

[mobisht@mobisht 04\_04\_Rolling\_updates]$ minikube service navbar-service

|-----------|----------------|-------------|-----------------------------|

| NAMESPACE | NAME | TARGET PORT | URL |

|-----------|----------------|-------------|-----------------------------|

| default | navbar-service | 80 | http://192.168.99.100:31001 |

|-----------|----------------|-------------|-----------------------------|

🎉 Opening service default/navbar-service in default browser...

* Roll back

[mobisht@mobisht 04\_04\_Rolling\_updates]$ kubectl rollout undo deployment/navbar-deployment

deployment.apps/navbar-deployment rolled back

**Scenario: Basic troubleshooting**

* Deploy application with wrong image.

[mobisht@mobisht 04\_05\_Basic\_troubleshooting]$ kubectl create -f helloworld-with-bad-pod.yaml

deployment.apps/bad-helloworld-deployment created

[mobisht@mobisht 04\_05\_Basic\_troubleshooting]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

bad-helloworld-deployment-b564cfb94-ctrj8 0/1 **ErrImagePull** 0 11s

* Describe deployment

[mobisht@mobisht 04\_05\_Basic\_troubleshooting]$ kubectl describe deployment/bad-helloworld-deployment

Name: bad-helloworld-deployment

Namespace: default

CreationTimestamp: Mon, 05 Jul 2021 12:59:51 +0530

Labels: <none>

Annotations: deployment.kubernetes.io/revision: 1

Selector: app=bad-helloworld

Replicas: 1 desired | 1 updated | 1 total | 0 available | 1 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 25% max unavailable, 25% max surge

Pod Template:

Labels: app=bad-helloworld

Containers:

helloworld:

Image: karthequian/unkown-pod:latest

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Conditions:

Type Status Reason

---- ------ ------

Available False MinimumReplicasUnavailable

Progressing True ReplicaSetUpdated

OldReplicaSets: <none>

NewReplicaSet: bad-helloworld-deployment-b564cfb94 (1/1 replicas created)

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ScalingReplicaSet 62s deployment-controller Scaled up replica set bad-helloworld-deployment-b564cfb94 to 1

* Describe pods

[mobisht@mobisht 04\_05\_Basic\_troubleshooting]$ kubectl describe pod/bad-helloworld-deployment-b564cfb94-ctrj8

Name: bad-helloworld-deployment-b564cfb94-ctrj8

Namespace: default

Priority: 0

Node: minikube/192.168.99.100

Start Time: Mon, 05 Jul 2021 12:59:51 +0530

Labels: app=bad-helloworld

pod-template-hash=b564cfb94

Annotations: <none>

Status: Pending

IP: 172.17.0.2

IPs:

IP: 172.17.0.2

Controlled By: ReplicaSet/bad-helloworld-deployment-b564cfb94

Containers:

helloworld:

Container ID:

Image: karthequian/unkown-pod:latest

Image ID:

Port: 80/TCP

Host Port: 0/TCP

State: Waiting

Reason: ImagePullBackOff

Ready: False

Restart Count: 0

Environment: <none>

Mounts:

/var/run/secrets/kubernetes.io/serviceaccount from default-token-khvqr (ro)

Conditions:

Type Status

Initialized True

Ready False

ContainersReady False

PodScheduled True

Volumes:

default-token-khvqr:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-khvqr

Optional: false

QoS Class: BestEffort

Node-Selectors: <none>

Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s

node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Scheduled 106s default-scheduler Successfully assigned default/bad-helloworld-deployment-b564cfb94-ctrj8 to minikube

Normal BackOff 19s (x4 over 100s) kubelet Back-off pulling image "karthequian/unkown-pod:latest"

Warning Failed 19s (x4 over 100s) kubelet Error: ImagePullBackOff

Normal Pulling 6s (x4 over 105s) kubelet Pulling image "karthequian/unkown-pod:latest"

Warning Failed 0s (x4 over 100s) kubelet Failed to pull image "karthequian/unkown-pod:latest": rpc error: code = Unknown desc = Error response from daemon: pull access denied for karthequian/unkown-pod, repository does not exist or may require 'docker login': denied: requested access to the resource is denied

Warning Failed 0s (x4 over 100s) kubelet Error: ErrImagePull

[mobisht@mobisht 04\_05\_Basic\_troubleshooting]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

bad-helloworld-deployment-b564cfb94-ctrj8 0/1 ImagePullBackOff 0 3m5s

* Get Logs

[mobisht@mobisht 04\_05\_Basic\_troubleshooting]$ kubectl logs bad-helloworld-deployment-b564cfb94-ctrj8

Error from server (BadRequest): container "helloworld" in pod "bad-helloworld-deployment-b564cfb94-ctrj8" is waiting to start: image can't be pulled

[mobisht@mobisht 04\_05\_Basic\_troubleshooting]$

**Scenario: Deploying PHP Guestbook application with MangoDB**

Link: <https://kubernetes.io/docs/tutorials/stateless-application/guestbook/>

**Scenario: Kubernetes Dashboard**

Link: <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>

mobisht@mobisht 05\_01\_RealWorld]$ minikube addons list

|-----------------------------|----------|--------------|

| ADDON NAME | PROFILE | STATUS |

|-----------------------------|----------|--------------|

| ambassador | minikube | disabled |

| auto-pause | minikube | disabled |

| csi-hostpath-driver | minikube | disabled |

| dashboard | minikube | disabled |

| default-storageclass | minikube | enabled ✅ |

| efk | minikube | disabled |

| freshpod | minikube | disabled |

| gcp-auth | minikube | disabled |

| gvisor | minikube | disabled |

| helm-tiller | minikube | disabled |

| ingress | minikube | disabled |

| ingress-dns | minikube | disabled |

| istio | minikube | disabled |

| istio-provisioner | minikube | disabled |

| kubevirt | minikube | disabled |

| logviewer | minikube | disabled |

| metallb | minikube | disabled |

| metrics-server | minikube | disabled |

| nvidia-driver-installer | minikube | disabled |

| nvidia-gpu-device-plugin | minikube | disabled |

| olm | minikube | disabled |

| pod-security-policy | minikube | disabled |

| registry | minikube | disabled |

| registry-aliases | minikube | disabled |

| registry-creds | minikube | disabled |

| storage-provisioner | minikube | enabled ✅ |

| storage-provisioner-gluster | minikube | disabled |

| volumesnapshots | minikube | disabled |

|-----------------------------|----------|--------------|

[mobisht@mobisht 05\_01\_RealWorld]$ minikube addons enable dashboard

▪ Using image kubernetesui/dashboard:v2.1.0

▪ Using image kubernetesui/metrics-scraper:v1.0.4

💡 Some dashboard features require the metrics-server addon. To enable all features please run:

minikube addons enable metrics-server

🌟 The 'dashboard' addon is enabled

[mobisht@mobisht 05\_01\_RealWorld]$ minikube addons enable metrics-server

▪ Using image k8s.gcr.io/metrics-server/metrics-server:v0.4.2

🌟 The 'metrics-server' addon is enabled

[mobisht@mobisht 05\_01\_RealWorld]$ minikube addons list

|-----------------------------|----------|--------------|

| ADDON NAME | PROFILE | STATUS |

|-----------------------------|----------|--------------|

| ambassador | minikube | disabled |

| auto-pause | minikube | disabled |

| csi-hostpath-driver | minikube | disabled |

**| dashboard | minikube | enabled ✅ |**

**| default-storageclass | minikube | enabled ✅ |**

| efk | minikube | disabled |

| freshpod | minikube | disabled |

| gcp-auth | minikube | disabled |

| gvisor | minikube | disabled |

| helm-tiller | minikube | disabled |

| ingress | minikube | disabled |

| ingress-dns | minikube | disabled |

| istio | minikube | disabled |

| istio-provisioner | minikube | disabled |

| kubevirt | minikube | disabled |

| logviewer | minikube | disabled |

| metallb | minikube | disabled |

| metrics-server | minikube | enabled ✅ |

| nvidia-driver-installer | minikube | disabled |

| nvidia-gpu-device-plugin | minikube | disabled |

| olm | minikube | disabled |

| pod-security-policy | minikube | disabled |

| registry | minikube | disabled |

| registry-aliases | minikube | disabled |

| registry-creds | minikube | disabled |

| storage-provisioner | minikube | enabled ✅ |

| storage-provisioner-gluster | minikube | disabled |

| volumesnapshots | minikube | disabled |

|-----------------------------|----------|--------------|

[mobisht@mobisht 05\_01\_RealWorld]$ minikube dashboard

🤔 Verifying dashboard health ...

🚀 Launching proxy ...

🤔 Verifying proxy health ...

🎉 Opening http://127.0.0.1:42345/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/ in your default browser...

**Scenario: Configuration Data**

mobisht@mobisht 05\_03\_Configmaps]$ cat reader-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: logreader

labels:

app: logreader

spec:

replicas: 1

selector:

matchLabels:

app: logreader

template:

metadata:

labels:

app: logreader

spec:

containers:

- name: logreader

image: karthequian/reader:latest

env:

- name: log\_level

value: "error"

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl create -f reader-deployment.yaml

deployment.apps/logreader created

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

logreader-6cf6586757-pnzzs 1/1 Running 0 76s

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl logs logreader-6cf6586757-pnzzs

**Log level passed via env variables was: 'error'**

**Log level passed via env variables was: 'error'**

**Log level passed via env variables was: 'error'**

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl create configmap logger --from-literal=log\_level=debug

configmap/logger created

[mobisht@mobisht 05\_03\_Configmaps]$ cat reader-configmap-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: logreader-dynamic

labels:

app: logreader-dynamic

spec:

replicas: 1

selector:

matchLabels:

app: logreader-dynamic

template:

metadata:

labels:

app: logreader-dynamic

spec:

containers:

- name: logreader

image: karthequian/reader:latest

env:

- name: log\_level

**valueFrom:**

**configMapKeyRef:**

**name: logger #Read from a configmap called log-level**

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl create -f reader-configmap-deployment.yaml

deployment.apps/logreader-dynamic created

[mobisht@mobisht 05\_03\_Configmaps]$

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl get configmaps

NAME DATA AGE

kube-root-ca.crt 1 2d19h

logger 1 88s

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl get configmap/logger -o yaml

apiVersion: v1

data:

log\_level: debug

kind: ConfigMap

metadata:

creationTimestamp: "2021-07-05T09:02:46Z"

name: logger

namespace: default

resourceVersion: "12594"

uid: 8f5fc621-11c7-49c3-ad74-ac6ef6a0d0b8

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl get pod

NAME READY STATUS RESTARTS AGE

logreader-6cf6586757-pnzzs 1/1 Running 0 5m

logreader-dynamic-8696bbff75-zmg4z 1/1 Running 0 66s

[mobisht@mobisht 05\_03\_Configmaps]$ kubectl logs logreader-dynamic-8696bbff75-zmg4z

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

Log level passed via env variables was: 'debug'

**Scenario: Dealing with application secrets**

[mobisht@mobisht 05\_04\_secrets]$ kubectl create secret generic apikey --from-literal=api\_key=123456767

secret/apikey created

[mobisht@mobisht 05\_04\_secrets]$ kubectl get secrets

NAME TYPE DATA AGE

apikey Opaque 1 11s

default-token-khvqr kubernetes.io/service-account-token 3 2d19h

[mobisht@mobisht 05\_04\_secrets]$ kubectl get secret apikey

NAME TYPE DATA AGE

apikey Opaque 1 27s

[mobisht@mobisht 05\_04\_secrets]$ kubectl get secret apikey -o yaml

apiVersion: v1

data:

**api\_key: MTIzNDU2NzY3**

kind: Secret

metadata:

creationTimestamp: "2021-07-05T09:11:23Z"

name: apikey

namespace: default

resourceVersion: "12978"

uid: 678f798c-1b0b-44f6-9843-95c306ea258f

type: Opaque

[mobisht@mobisht 05\_04\_secrets]$ cat secretreader-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: secretreader

labels:

name: secretreader

spec:

replicas: 1

selector:

matchLabels:

name: secretreader

template:

metadata:

labels:

name: secretreader

spec:

containers:

- name: secretreader

image: karthequian/secretreader:latest

env:

- name: api\_key

valueFrom:

secretKeyRef:

**name: apikey**

key: api\_key[mobisht@mobisht 05\_04\_secrets]$

[mobisht@mobisht 05\_04\_secrets]$ kubectl create -f secretreader-deployment.yaml

deployment.apps/secretreader created

[mobisht@mobisht 05\_04\_secrets]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

logreader-6cf6586757-pnzzs 1/1 Running 0 12m

logreader-dynamic-8696bbff75-zmg4z 1/1 Running 0 9m1s

secretreader-74957f7d88-6fqrt 1/1 Running 0 22s

[mobisht@mobisht 05\_04\_secrets]$ kubectl logs secretreader-74957f7d88-6fqrt

api\_key passed via env variable was: '123456767'

api\_key passed via env variable was: '123456767'

api\_key passed via env variable was: '123456767'

api\_key passed via env variable was: '123456767'

**Scenario: Running jobs in Kubernetes**

[mobisht@mobisht 05\_05\_jobs]$ cat simplejob.yaml

apiVersion: batch/v1

**kind: Job**

metadata:

name: finalcountdown

spec:

template:

metadata:

name: finalcountdown

spec:

containers:

- name: counter

image: busybox

command:

- bin/sh

- -c

- "for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done"

restartPolicy: Never #could also be Always or OnFailure[mobisht@mobisht 05\_05\_jobs]$

[mobisht@mobisht 05\_05\_jobs]$ kubectl create -f simplejob.yaml

[mobisht@mobisht 05\_05\_jobs]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

finalcountdown-5x885 0/1 Completed 0 48s

logreader-6cf6586757-pnzzs 1/1 Running 0 16m

logreader-dynamic-8696bbff75-zmg4z 1/1 Running 0 12m

secretreader-74957f7d88-6fqrt 1/1 Running 0 4m17s

[mobisht@mobisht 05\_05\_jobs]$ kubectl get jobs

NAME COMPLETIONS DURATION AGE

[mobisht@mobisht 05\_05\_jobs]$ kubectl get jobs

NAME COMPLETIONS DURATION AGE

finalcountdown 1/1 47s 52s

[mobisht@mobisht 05\_05\_jobs]$ kubectl logs finalcountdown-5x885

9

8

7

6

5

4

3

2

1

**cron are job but they can run periodically**

[mobisht@mobisht 05\_05\_jobs]$ cat cronjob.yaml

apiVersion: batch/v1beta1

**kind: CronJob**

metadata:

name: hellocron

spec:

schedule: "\*/1 \* \* \* \*" #Runs every minute (cron syntax) or @hourly.

jobTemplate:

spec:

template:

spec:

containers:

- name: hellocron

image: busybox

args:

- /bin/sh

- -c

- date; echo Hello from your Kubernetes cluster

restartPolicy: OnFailure #could also be Always or Never

suspend: false #Set to true if you want to suspend in the future

[mobisht@mobisht 05\_05\_jobs]$ kubectl create -f cronjob.yaml

cronjob.batch/hellocron created

[mobisht@mobisht 05\_05\_jobs]$ kubectl get cronjob

NAME SCHEDULE SUSPEND ACTIVE LAST SCHEDULE AGE

hellocron \*/1 \* \* \* \* False 0 33s 2m10s

[mobisht@mobisht 05\_05\_jobs]$ kubectl get cronjob

**Edit SUSPEND as true**

NAME SCHEDULE SUSPEND ACTIVE LAST SCHEDULE AGE

hellocron \*/1 \* \* \* \* True 0 39s 3m16s

**Scenario: Running stateful set applications**

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ cat daemonset.yaml

apiVersion: apps/v1

**kind: DaemonSet**

metadata:

name: example-daemonset

namespace: default

labels:

k8s-app: example-daemonset

spec:

selector:

matchLabels:

name: example-daemonset

template:

metadata:

labels:

name: example-daemonset

spec:

#nodeSelector: minikube # Specify if you want to run on specific nodes

containers:

- name: example-daemonset

image: busybox

args:

- /bin/sh

- -c

- date; sleep 1000

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

terminationGracePeriodSeconds: 30

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl create -f daemonset.yaml

daemonset.apps/example-daemonset created

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl get daemonset

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

example-daemonset-6g9dw 1/1 Running 0 33s

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ cat daemonset-infra-development.yaml

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: example-daemonset2

namespace: default

labels:

k8s-app: example-daemonset2

spec:

selector:

matchLabels:

name: example-daemonset2

template:

metadata:

labels:

name: example-daemonset2

spec:

containers:

- name: example-daemonset2

image: busybox

args:

- /bin/sh

- -c

- date; sleep 1000

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

terminationGracePeriodSeconds: 30

nodeSelector:

**infra: "development"**

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl get nodes --show-labels

NAME STATUS ROLES AGE VERSION LABELS

minikube Ready control-plane,master 2d19h v1.20.7 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=minikube,kubernetes.io/os=linux,minikube.k8s.io/commit=76d74191d82c47883dc7e1319ef7cebd3e00ee11,minikube.k8s.io/name=minikube,minikube.k8s.io/updated\_at=2021\_07\_02T19\_15\_51\_0700,minikube.k8s.io/version=v1.21.0,node-role.kubernetes.io/control-plane=,node-role.kubernetes.io/master=

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl create -f daemonset-infra-development.yaml

daemonset.apps/example-daemonset2 created

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl get daemonset

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

example-daemonset 1 1 1 1 1 <none> 2m35s

**example-daemonset2 0 0 0 0 0 infra=development 9s**

daemonset-infra-development.yaml daemonset-infra-prod.yaml daemonset.yaml Readme.md statefulset.yaml

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ cat daemonset-infra-prod.yaml

apiVersion: apps/v1

**kind: DaemonSet**

metadata:

name: prod-daemonset

namespace: default

labels:

k8s-app: prod-daemonset

spec:

selector:

matchLabels:

name: prod-daemonset

template:

metadata:

labels:

name: prod-daemonset

spec:

containers:

- name: prod-daemonset

image: busybox

args:

- /bin/sh

- -c

- date; sleep 1000

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

terminationGracePeriodSeconds: 30

nodeSelector:

**infra: "production"**

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl create -f daemonset-infra-prod.yaml

daemonset.apps/prod-daemonset created

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl get daemonset

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

example-daemonset 1 1 1 1 1 <none> 3m31s

example-daemonset2 0 0 0 0 0 infra=development 65s

**prod-daemonset 0 0 0 0 0 infra=production 3s**

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ cat statefulset.yaml

apiVersion: v1

**kind: Service**

metadata:

name: zk-hs

labels:

app: zk

spec:

ports:

- port: 2888

name: server

- port: 3888

name: leader-election

clusterIP: None

selector:

app: zk

---

apiVersion: v1

kind: Service

metadata:

name: zk-cs

labels:

app: zk

spec:

ports:

- port: 2181

name: client

selector:

app: zk

---

apiVersion: policy/v1beta1

kind: PodDisruptionBudget

metadata:

name: zk-pdb

spec:

selector:

matchLabels:

app: zk

maxUnavailable: 1

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: zk

spec:

selector:

matchLabels:

app: zk

serviceName: zk-hs

replicas: 3

updateStrategy:

type: RollingUpdate

podManagementPolicy: OrderedReady

template:

metadata:

labels:

app: zk

spec:

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: "app"

operator: In

values:

- zk

topologyKey: "kubernetes.io/hostname"

containers:

- name: kubernetes-zookeeper

imagePullPolicy: Always

image: "k8s.gcr.io/kubernetes-zookeeper:1.0-3.4.10"

resources:

requests:

memory: "1Gi"

cpu: "0.5"

ports:

- containerPort: 2181

name: client

- containerPort: 2888

name: server

- containerPort: 3888

name: leader-election

command:

- sh

- -c

- "start-zookeeper \

--servers=3 \

--data\_dir=/var/lib/zookeeper/data \

--data\_log\_dir=/var/lib/zookeeper/data/log \

--conf\_dir=/opt/zookeeper/conf \

--client\_port=2181 \

--election\_port=3888 \

--server\_port=2888 \

--tick\_time=2000 \

--init\_limit=10 \

--sync\_limit=5 \

--heap=512M \

--max\_client\_cnxns=60 \

--snap\_retain\_count=3 \

--purge\_interval=12 \

--max\_session\_timeout=40000 \

--min\_session\_timeout=4000 \

--log\_level=INFO"

readinessProbe:

exec:

command:

- sh

- -c

- "zookeeper-ready 2181"

initialDelaySeconds: 10

timeoutSeconds: 5

livenessProbe:

exec:

command:

- sh

- -c

- "zookeeper-ready 2181"

initialDelaySeconds: 10

timeoutSeconds: 5

volumeMounts:

- name: datadir

mountPath: /var/lib/zookeeper

securityContext:

runAsUser: 1000

fsGroup: 1000

volumeClaimTemplates:

- metadata:

name: datadir

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 10Gi[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$

**Stateful set manage deployments for pods**

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl create -f statefulset.yaml

service/zk-hs created

service/zk-cs created

poddisruptionbudget.policy/zk-pdb created

statefulset.apps/zk created

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl get statefulsets

NAME READY AGE

**zk 0/3 18s**

[mobisht@mobisht 05\_06\_daemonset\_statefulsets]$ kubectl get pods

NAME READY STATUS RESTARTS AGE

example-daemonset-6g9dw 1/1 Running 0 6m25s

**zk-0 1/1 Running 0 31s**

**zk-1 0/1 Pending 0 0s**

**Production Kubernetes deployment**

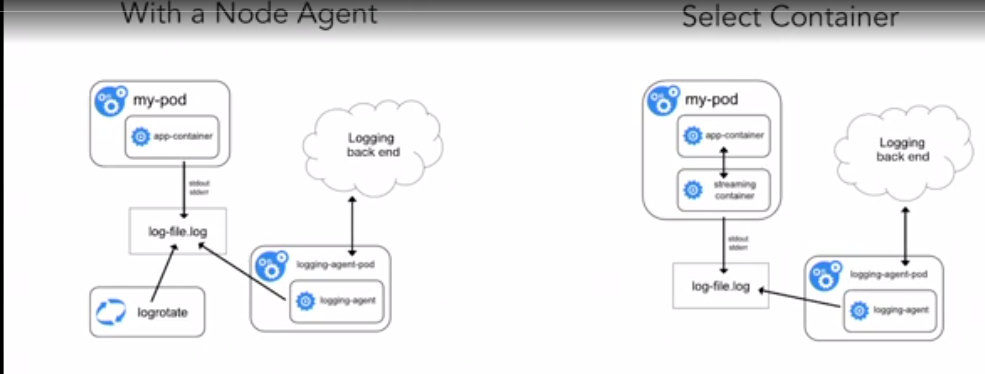
* Common ways
  + Kelsey hightower tutorial.
  + Kubeadm
  + Install steps:
    - Initially provision the master host with docker and kubernetes distribution.
    - Run kubeadm init, which starts kubeadm, provisions the kubernetes control plane, and provides a join token.
    - Run kubeadm join with join token on each worker node. The worker will join the cluster.
  + Install a Pod Network
    - Evaluate your network strategies.
    - Consider Flannel and Weave Net as starting possibilities.
  + Kobs
    - Automate the K8s cluster provisioning in AWS.
    - Deploy high-availability (HA) master.
    - Permits upgrading with kube-up.
    - Use a state-sync model for dry runs and automatic idempotency.
    - Generates configuration files for AWS CloudFormation and Terraform configuration, supports custom add-ons, and finally uses a manifest-based API configuration.
    - Kops is a very popular way to deploy Kubernetes on AWS and looks very similar to the way kubectl operates.
  + If you're going to run Kubernetes on a cloud provider like AWS, Azure, Google or Oracle Cloud, you might want to consider one of their native container services like Amazon Elastic Kubernetes Service on AWS, Azure Container Service for Azure, Google Container Engine for Google Cloud or Oracle Container Engine for Kubernetes in Oracle Cloud.

**Namespaces**

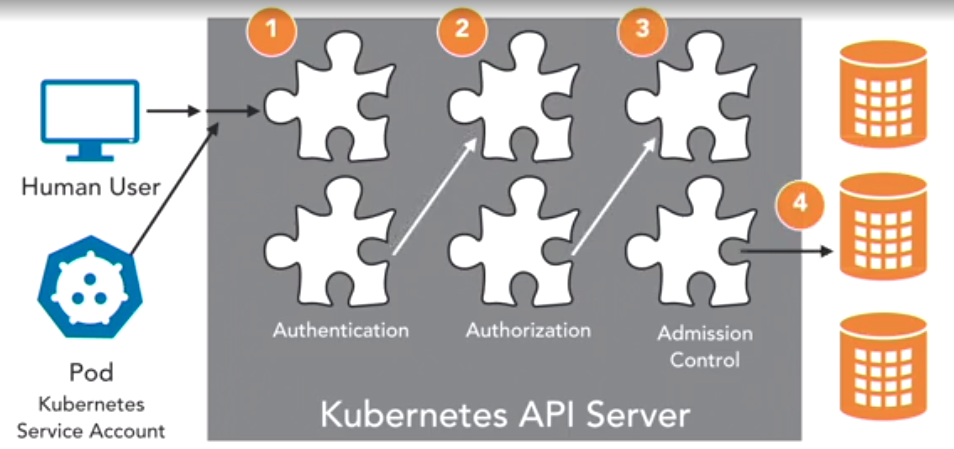
* Namespaces are a fundamental concept to add multi-tenancy to your Kubernetes instance.
* Kubernetes provides multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.
* There are four primary use cases.
  + One, roles and responsibilities in an enterprise.
  + Two, partitioning landscapes, for example, dev vs test vs production.
  + Three, customer partitioning for non-multi-tenant scenarios. For example, if you're a consulting company and you have multiple customers, you can use this feature. And finally, for application partitioning as well.
* Enterprises have many teams that typically operate independently of each other and might have some shared interfaces and APIs to communicate with each other. The use of namespaces is perfect in this scenario, because teams are able to operate independently without stepping on each other.
* Namespaces also prevent them from confusing services and deployments that might not belong to them.
* Another common pattern is using namespaces for Dev, Test, and Production environments. With most teams building a continuous integration, or CI pipelines, you can easily define CI pipelines to deploy an entire environment in a quick and easy fashion.
* Some anti-patterns to watch out for are, sometimes you will end up with a really large namespace because you have many applications that are running.
* In these scenarios, it might be good to create additional namespaces for groups of applications. For example, E-Commerce Dev namespace for all E-commerce related applications.
* Another scenario is, you might end up with too many environments. Just because you can, does not mean that you should.
* Don't abuse the feature because it will result in too many unnecessary environments. If you don't use a staging deployment, or you don't use a staging namespace, then don't create one.
* Consulting companies and small software vendors might use this method a lot. You could create a namespace for each customer or project to keep them distinct, while not needing to worry about reusing the same names or resources across different projects.
* Let's discuss the basic commands related to them. To just get all your existing namespaces, you can just do a **kubectl get namespaces**. To create one, all you have to do is do a **kubectl create namespace** followed by the name you want to use. Similarly, to delete the namespace, it's a **kubectl delete namespace** followed by the namespace you'd like to delete. When deploying a specific resource, like a deployment to a specific namespace, just add the -n namespace-name, and the resource will exist in that specific namespace.

**Monitoring and Logging**

* Once your infrastructure and your application have been deployed successfully, you run into your next challenge: Are they operating correctly, have there been any errors, are they even online?
* Logging
  + If your application isn't outputting logs, it probably should.
  + From a kubernetes perspective, starting out, it's better for your applications, running in a pod, to write your standard out. This is because kubernetes will automatically pick up the logs and show you anything in the log info when you use the command **kubectl logs** to fetch the logs.
  + This is really useful when you're doing development. Once your application is running in staging or production, you might consider using a logging platform like **Elasticsearch in Kibana, with logs being shipped to it from pods, using Fluentd, Filebeat or Logstash.**

****

* + The architecture of this paradigm is shown here. You would typically run a deployment for your Elasticsearch in Kibana instance outside of your application.
  + The Elasticsearch endpoint needs to be accessible to pods in the cluster, and the Kibana instance should be exposed as a service so that you can see log information in the Kibana user interface. Then, in your application deployment, you'll want the log shipper, Fluentd, Logstash or Filebeat, to gather log data and periodically send it to the Elasticsearch instance.
* Monitoring
  + From a monitoring perspective, we have a few things we'd like to monitor. The node health, the health of the kubernetes cluster, and application health and metrics.
  + For node and kubernetes health, the combination of two open source projects, **cAadvisor and Prometheus,** give an elegant way to monitor your cluster and applications.
  + cAdvisor is an open-source resource usage collector that was built for containers. It auto discovers all containers in a given node and collects CPU, memory, file system and network usage statistics.
  + cAdvisor also provides the overall machine usage by analyzing the root container on the machine.
  + The other project you really want to look at is called Prometheus. Prometheus is an open-source system monitoring and alerting toolkit with a query language that works really well for application-specific metrics and can also be used from an alerting standpoint.
  + Prometheus was initially used to collect application metrics like number of successful logins per hour, but can also be used natively to monitor kubernetes itself, with projects like kube-prometheus.
  + We can instrument applications to save application monitoring data at a /metrics endpoint, that Prometheus queries in a timely manner.
  + Prometheus and cAdvisor are typically linked to Grafana, which is an open-source tool to visualize monitoring data.

**Authentication and Authorization  
**

* Authentication and authorization are two of the most common topics that arise when we are trying to integrate Kubernetes into their infrastructure. In this section,
* From an architecture perspective, users who are trying to access the Kubernetes API, via the kubectl command, first have to be authenticated to access the API.
* And then authorized to run specific actions.
* Occasionally, users will also need to have admission control to approve or reject a request.
* There are actually two kinds of users.
  + Normal users, like you and me.
  + service accounts which are internal users to the system.
* We typically think of normal users as users who are in an LDAP or an SSO system in our enterprises.
* From a Kubernetes perspective, these users are managed outside of Kubernetes. Service accounts are managed by the Kube API server, bound to a specific namespace, and tied to credentials managed in secrets.
* There are four pieces of information that define a user.
  + The username, which is basically a string. For example, Karthik or Karthik@example.com.
  + A UID, or a unique ID, which is an identifier that is more consistent and unique than the username.
  + A group, which is a string that associates users with a set of commonly grouped users.
* These are used later on by the authorization model. And extra fields which is basically a map of strings that holds additional information used by the authorization system.
* Authentication in Kubernetes defines whether a user has access to communicate with the Kube API server.
* And authorization defines whether a user has actual permissions to perform specific actions. There are a number of ways you can authenticate.
* The entire list can be found in the Kubernetes documentation but we look at the four most popular modules that enterprises use today.
  + Client certs
  + static tokens
  + OpenID connect
  + webhook mode.
  + Client cert
    - Authentication is enabled by passing the client ca file option to the API server.
    - The reference file must contain certificate authorities to validate client certificates passed to the API server.
    - If a client cert is presented and verified, the common name of the client certificate is used as the username for the request.
    - This is a straightforward way to manage Kubernetes if you're comfortable with handling certificate managements in your org.
    - The API server reads the bare tokens from a file when given the dash dash token auth file option on the command line. The token auth file is a CSV file with four columns: token, username, user UID, followed by optional group names as shown in the example.
    - The caveat to this method is that these tokens last indefinitely and you're required to restart the API server for any changes in tokens to take place. The static password file is similar to this method but the values are passed as basic auth parameters.
    - The benefit of this method is that it's very simple and easy to manage a small batch of users this way.
    - For larger organizations that already have oAuth, such as Google, or Azure Active Directory, OpenID connects tokens to authenticate against Kubernetes. The details of this are complex and I'd recommend reading the docs for this method.
    - And finally, there's authentication using webhook tokens. This is one of the most common methods that's used by a number of organizations who want to have the users interacting with Kubernetes but also manage tokens in an easier way. Using this method, the Kube API server makes a request to an API that's defined by you to authenticate.
    - The remote API decides whether a token passed to it is a valid token or not, and then responds back to the server.
    - On the authorization front, there are three common ways to perform authorizations. ABAC, RBAC, or Webhook.
    - \ABAC, or attribute based access control, defines an access control paradigm whereby access rights are granted to users through the use of policies that combine attributes together.
    - The ABAC file defines what access a specific user might have to all resources. In the example config shown, Karthik has master access to all attributes, whereas the user Carisa only has read access to all the repos and name spaces.
    - RBAC is role based access control. This is the most common authorization mechanism used in Kubernetes and a lot of applications end up using RBAC to authorize their service accounts.
    - Recommend that you turn on this functionality on your KubeAPI server even if you're using another authorization mechanism, because you'll end up using something that will require RBAC being installed.
    - RBAC depends on roles and cluster roles. These are rules that represent a set of permissions. A role can be defined within the namespace with a role or a clusterwide with a cluster role. In this example, we see that a role is in the default namespace that allows read access to all pods. Role bindings grant permissions defined in a role to users.
    - They hold the users or group as a reference to the role being granted. Permissions can then be granted within a namespace with a role binding or cluster wide with a cluster role binding. In the example shown, the user Karthik can read pods in the default namespace.
    - The webhook mode for authorization, like the authentication webhook mode, allows you to define what permissions are allowed for a specified user. The Kube API server will send a request with the user and resource attribute data to a remote server that you define, that interprets the request and defines whether a request is allowed or not. This method works really well if you're trying to integrate with a third party authorization system, or if you want a complex set of rules.