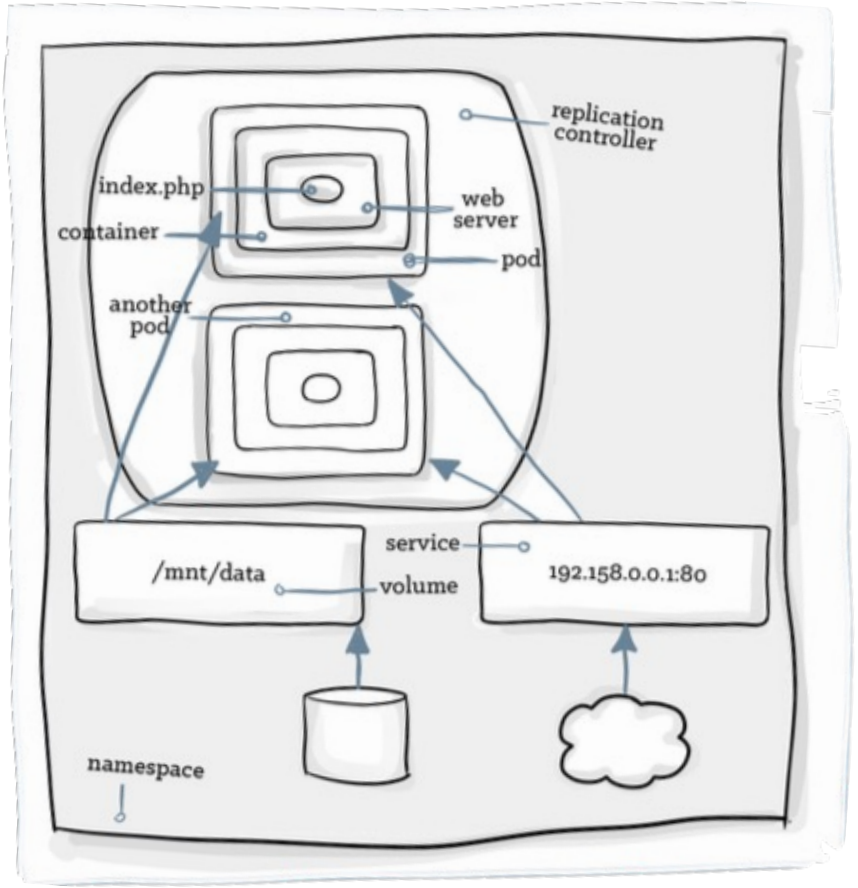
Kubernetes

Lab Guide v8.0-v10

Binny Raphael
Cloud and Infrastructure Services



Class Introductions

- Name
- Base Location
- Project
- Your total experience
- Your experience with Linux/Unix
- Your experience with Docker/Kubernetes/OpenShift if any.
- Why are you attending this session?
- What is your expectation out of this session?

Notes:

- As a courtesy to others, please be on time before the start of the session each-day and after every break. You are free to step outside if you need a break; no need to ask permission.
- Please do not do your project work, email, SMS, WhatsApp, etc during the sessions or talk on phone.
- Please keep you phone on silent mode; in case if you need to take the call, you may step out.
- As informed earlier, you need to be comfortable with the UNIX command line.
- This entire session if aimed towards Administration and Operations team; this is not a Developer Oriented course. (However, there is no harm for a developer in attending this course)
- Feel free to interrupt and ask questions at any time. Sometimes, we may have to take the question offline in the interest of time or wait for a particular topic to be covered first before it can be answered.
- This is a 5-day intense training and you will feel exhausted; we still need you to keep the tempo up all the time.
- The whole session is hands-on; you need to keep up with the average flow of the class. Please DO NOT do the labs during the theory sessions or skip the flow of the lab sessions. We have seen issues earlier.
- We have included as much information as possible in this session. However, *you will need to a lot of reading for many of the topics discussed in class either from internet or vendor sites*. You must read & practice a lot.
- All hands-on sections are identified by a gray rectangle box. Once a while, you may need to follow specific instruction when running those commands in some cases. Please follow the instruction.
- Last not the least, *it will take more than merely attending these sessions to make you an expert.*

Credits: Some of the slides and diagrams are from Docker, Kubernetes and Red Hat OpenShift tutorials on the internet and is reused in this presentation. Such slide have a Docker or Kubernetes or OpenShift logo at the bottom left. Thanks to all of them.

Accessing your Lab

Accessing Docker Lab

Step 1: Use MS Remote Desktop and connect to the Remote Desktop:

MYSIMSTTS01(10.122.198.27) or MYSIMSTTS02(10.122.198.28)

[user-ID: **INFOSYS_USERID** Password: **your AD Password**] -

Step 3: Set the Chrome Proxy Server to :: **172.20.0.xxx** and Port :: **8080**

Step 2: From the remote desktop, Launch Putty and SSH to you Server Instance

Server: **172.20.0.xxx**
User Name: **session7**
Password: **Infy123+**

Step 4: From the Pytty session, again ssh to you Virtual Machine instance

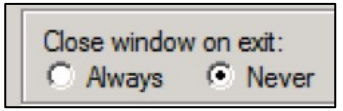
VM/Pod Server: **docker.podNN.example.com**
User Name: **root**
Password: **root123**

Make sure that you use the same POD-NUMBER assigned to you throughout this session.

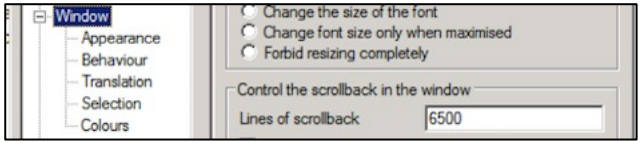
Recommended Putty Configuration

It is recommended to make the following Putty configuration so as to make things easier for you during the labs:

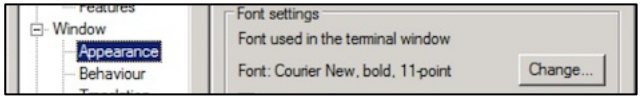
Disable Close window on exit:



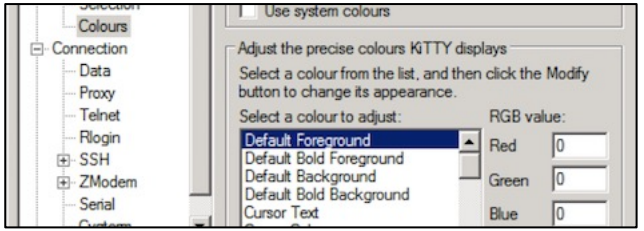
Increase the Scroll back:

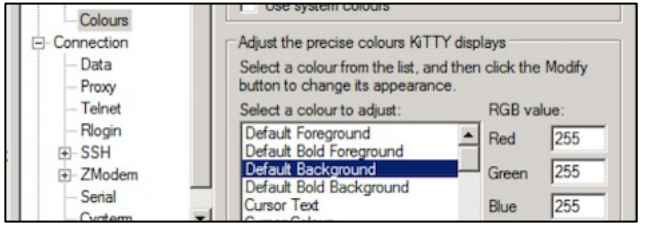


Change the font:



Adjust the Foreground and Background colors:





Note: Make sure that you save these setting; else you will end up doing this changes every time.

Kubernetes

κυβερνήτης*: Greek for “pilot” or “helmsman of a ship”*
the open source cluster manager from Google

What are Containers?

Docker explains Containers as:

“A container image is a lightweight, standalone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.”

What it means is that once an application is packaged as a container, the underlying environment where it is run doesn’t really matter. It can run anywhere, even on a multi-cloud environment.

***Source**: Docker.io and What are containers?*

Why Containers?

Developers develop their application on their work laptop or workstation, which has all the libraries, packages, tools, and framework needed to run that application. It's tested on a platform that resembles the production machine and then it's deployed in production. The problem is that not all three environments are the same; they don't have same tools, frameworks, and libraries. And, the app that works on the dev machine may not work in the production environment.

The solution is Containers.

What it means is that once an application is packaged as a container, the underlying environment where it is run doesn't really matter. It can run anywhere, even on a multi-cloud environment. Now developers or operators don't have to concern themselves with what platforms they are using to run applications. Another big advantage of containers is isolation and security.

***Source**: Docker.io and What are containers?*

What is Kubernetes?

- It is project managed by CNCF (Cloud Native Computing Foundation)
- Kubernetes is a container management and orchestration system
- Designed from the ground-up as a loosely coupled collection of components centered around deploying, maintaining and scaling workloads.
- It runs and manages containerized applications on a cluster
- Provides container grouping, load balancing, auto-healing, scaling features
- Project that was spun out of Google as an open source container orchestration platform
- Built from the lessons learned in the experiences of developing and running Google's Borg and Omega.
- Known as the Linux kernel of distributed systems
- Abstracts away the underlying hardware of the nodes and provides a uniform interface for workloads
- Most importantly, uses the same API across bare metal or VMs or across every cloud provider

Things we can do with Kubernetes

Using Kubernetes we can do cool stuff like:

- Manage containerized applications across a cluster of nodes
- Application auto scaling
- Efficient & Robust deployment of application
- Self healing application
- Blue/green deployment, canary deployment - Easy Service deployments, updates and scalability
- Long running services, but also batch (one-off) jobs
- Overcommit our cluster and evict low-priority jobs
- Manage Stateless and Stateful Applications (databases etc.)
- Fine-grained access control defining what can be done by whom on which resources
- Integrating third party services (service catalog)
- Automating complex tasks (operators)

What is then the best definition for Kubernetes ?

Brian Grant is a Principal Engineer with Kubernetes-Google project and Lead Architect of Kubernetes, and he defines Kubernetes as below and the definition isn't a short one.

"I think of Kubernetes as portable, extensible open-source platform, for managing containerized workloads and services, that facilitates both declarative configuration and automation, with a large, rapidly growing ecosystem."

Kubernetes Timeline

- 2003-2004: Birth of the Borg System
- 2013: From Borg to Omega
- 2014: Google Introduces Kubernetes
  - mid-2014: Google introduced Kubernetes as an open source version of Borg
  - June 7: Initial release - first github commit for Kubernetes
  - July 10: Microsoft, RedHat, IBM, Docker joins the Kubernetes community.
- 2015: The year of Kube v1.0 & CNCF
- 2016: The Year Kubernetes Goes Mainstream!
- 2017: The Year of Enterprise Adoption & Support
  - September 29: Kubernetes 1.8: This release was a milestone for the role-based access control (RBAC) authorizer, a mechanism for controlling access to the Kubernetes API, which would be a fundamental building block for securing Kubernetes clusters.
  - October: Docker Fully Embraces Kubernetes - developers and operators can build apps with Docker and seamlessly test & deploy them using both Docker Swarm and Kubernetes.
- 2018: March 2: Kubernetes 1.10 announced. Kubelet TLS Bootstrapping, API aggregation, and more detailed storage metrics.
- 2019: March Kubernetes 1.14 released

What are some of the Key features that it provides?

- **Scheduling**: Decide where my containers should run
- **Lifecycle and health**: Keep my containers running despite failures
- **Scaling**: Make sets of containers bigger or smaller
- **Naming and discovery**: Find where my containers are now
- **Load balancing**: Distribute traffic across a set of containers
- **Storage volumes**: Provide data to containers
- **Logging and monitoring**: Track what’s happening with my containers
- **Debugging and introspection**: Enter or attach to containers
- **Identity and authorization**: Control who can do things to my containers

Kubernetes High Level Architecture

Kubernetes has just 5 basic components

- 3 Master Components :
  - API Server
  - Controller manager
  - Scheduler

- 2 Node Components :
  - Kubelet
  - Proxy

Kubernetes High Level Architecture: The Master Node(s)

- The Kubernetes logic (its "brains") is a collection of services:
  - the API server (our point of entry to everything!)
  - core services like the scheduler and controller manager
  - etcd (a highly available key/value store; the "database" of Kubernetes)
- Together, these services form what is called the "master"
- These services can run straight on a host, or in containers (that's an implementation detail)
- etcd can be run on separate machines (first schema) or co-located (second schema)
- We need at least one master, but we can have more (for high availability)

Kubernetes High Level Architecture: The (Worker) Node(s)

- The nodes executing our containers run another collection of services:
  - a container Engine (typically Docker)
  - kubelet (the "node agent")
  - kube-proxy (a necessary but not sufficient network component)

- Nodes were formerly called "minions"

- It is recommended to not run apps on the node(s) running master components (Except when using small development clusters)

Kubernetes High Level Architecture: What do these components do?

API Server

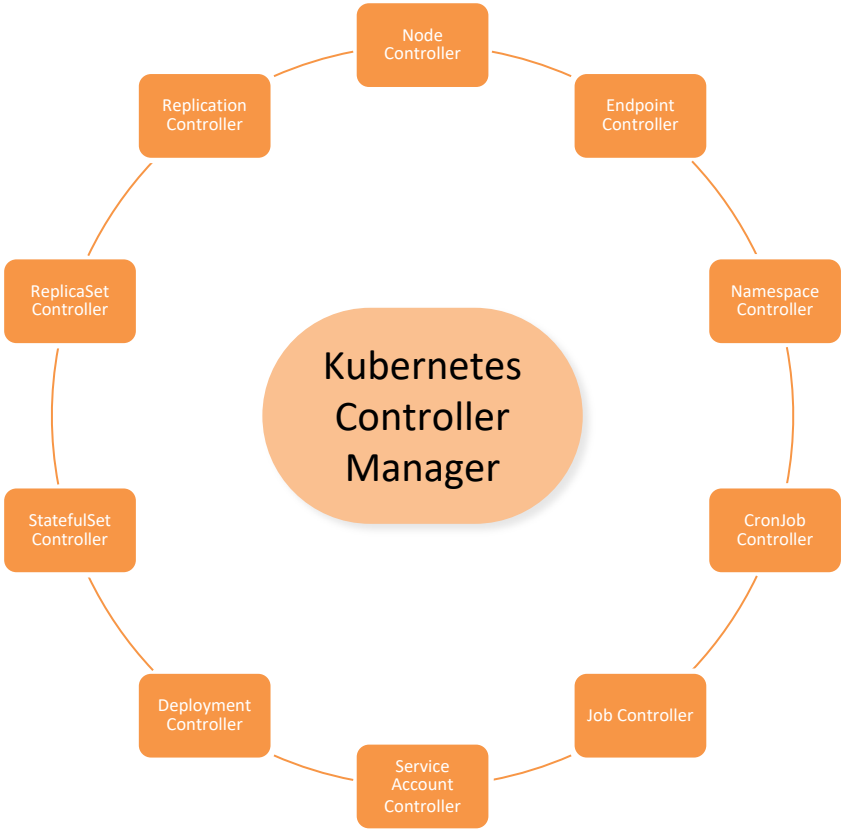
- Provides a forward facing REST interface into the kubernetes control plane and datastore
- All clients and other applications interact with kubernetes strictly through the API Server
- The API server exposes four APIs; Kubernetes API, Extensions API, Autoscaling API, and Batch API.
- These are used for communicating with the Kubernetes cluster and executing container cluster operations.
- **The API Server is the only Kubernetes component that connects to etcd.**
- All the other components must go through the API Server to work with the cluster state (access etc).
- Acts as the gatekeeper to the cluster by handling authentication and authorization, request validation, mutation, and admission control in addition to being the front-end to the backing datastore

Kubernetes High Level Architecture: What do these components do?

Controller Manager

- Serves as the primary daemon that manages all core component control loops.
- Controller manager monitors the current state of the applications deployed on Kubernetes via the API server and makes sure that it meets the desired state.
- Monitors the cluster state via the API server and steers the cluster towards the desired state

Kubernetes Controller Manager



Kubernetes High Level Architecture: What do these components do?

Scheduler

- The Scheduler's responsibility is to monitor the resource usage of each node and scheduling containers according to resource availability.
- Verbose policy-rich engine that evaluates workload requirements and attempts to place it on a matching resource.
- Default scheduler uses bin packing.
- Workload Requirements can include: general hardware requirements, affinity/anti-affinity, labels, and other various custom resource requirements.

Kubernetes High Level Architecture: What do these components do?

etcd

Etcd is a distributed, consistent key-value store used for configuration management, service discovery, and coordinating distributed work and released as an open source by CoreOS.

Kubernetes uses that as the persistence storage of all of its API objects (what nodes exist in the cluster, what pods should be running, which nodes they are running on, etc).

Etcd also implements a watch feature, which provides an event-based interface for asynchronously monitoring changes to keys. These systems allow clients to perform a lightweight subscription for changes to parts of the key namespace. Once a key is changed, its “watchers” get notified immediately. (i.e. one component can write to etcd and other components can immediately react to that change.) This is a crucial feature in the context of Kubernetes, as the API Server component heavily relies on this to get notified and call the appropriate business logic components to move the current state towards the desired state.

Kubernetes High Level Architecture: What do these components do?

Kubelet

- Acts as the node agent responsible for managing the lifecycle of every pod on its host
- It makes use of the pod specification for creating containers and managing them
- Kubelet understands YAML container manifests that it can read from several sources:
  - file path
  - HTTP Endpoint
  - etcd watch acting on any changes
  - HTTP Server mode accepting container manifests over a simple API.

Kubernetes High Level Architecture: What do these components do?

Kube-proxy

- Kube-proxy runs in each node
- Manages the network rules on each node.
- Performs connection forwarding or load balancing for Kubernetes cluster services.
- It uses proxy modes rules for doing simple TCP, UDP stream forwarding or round robin TCP, UDP forwarding and load balancing.
- Available Proxy Modes:
  - Userspace
  - iptables
  - IP Virtual Server (ipvs) - default if supported

Kubernetes High Level Architecture: What do these components do?

Container Runtime Engine

- A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.
  - Containerd (docker)
  - Cri-o
  - Rkt
  - Kata
  - Virtlet (VM CRI compatible runtime)

Kubernetes High Level Architecture: What do these components do?

Cloud Controller Manager (optional)

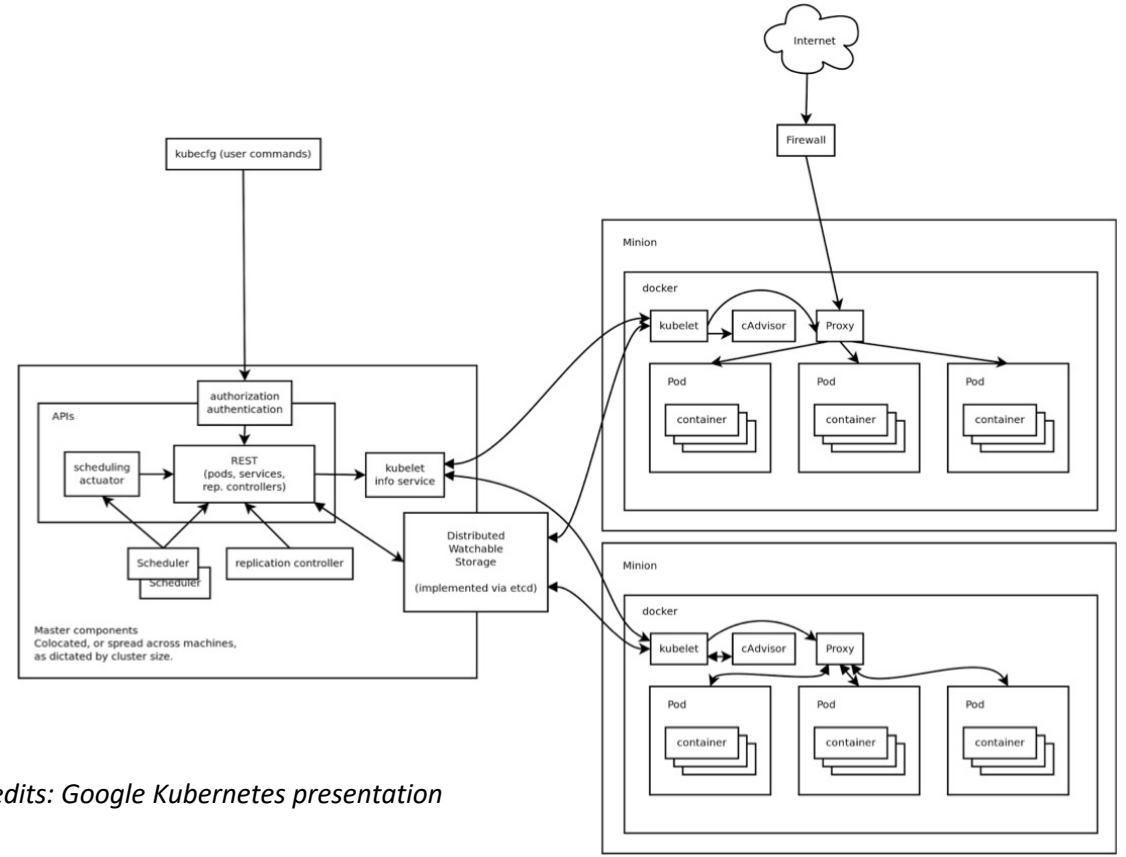
- Provides cloud-provider specific knowledge and integration capability into the core control loop of Kubernetes.
- The controllers include Node, Route, Service, and add an additional controller to handle things such as PersistentVolume Labels.

Kubernetes High Level Architecture: What do these components do?

Cluster DNS

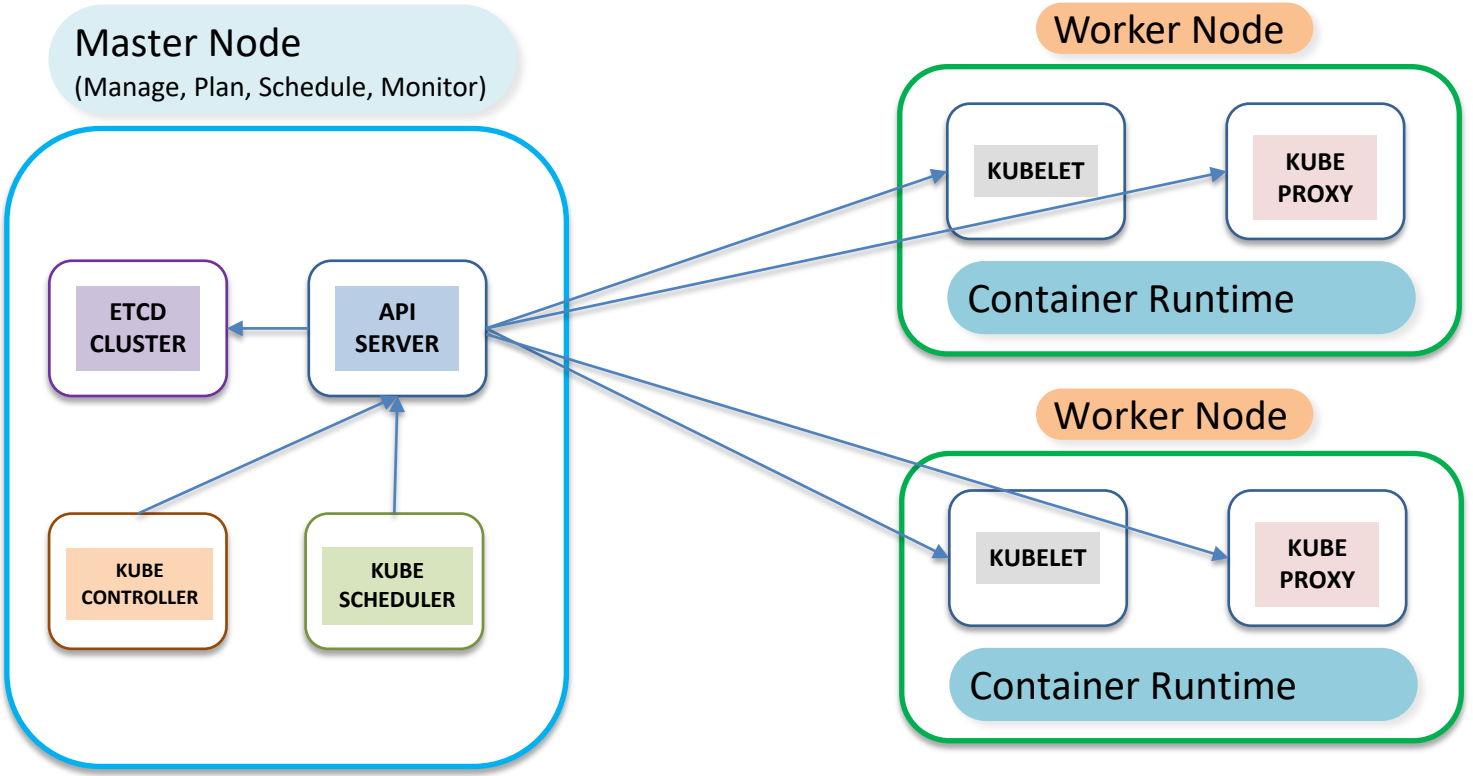
- Provides Cluster Wide DNS for Kubernetes Services.
  - kube-dns (default pre 1.11)
  - CoreDNS (current default)

Kubernetes High Level Architecture: Component Layout

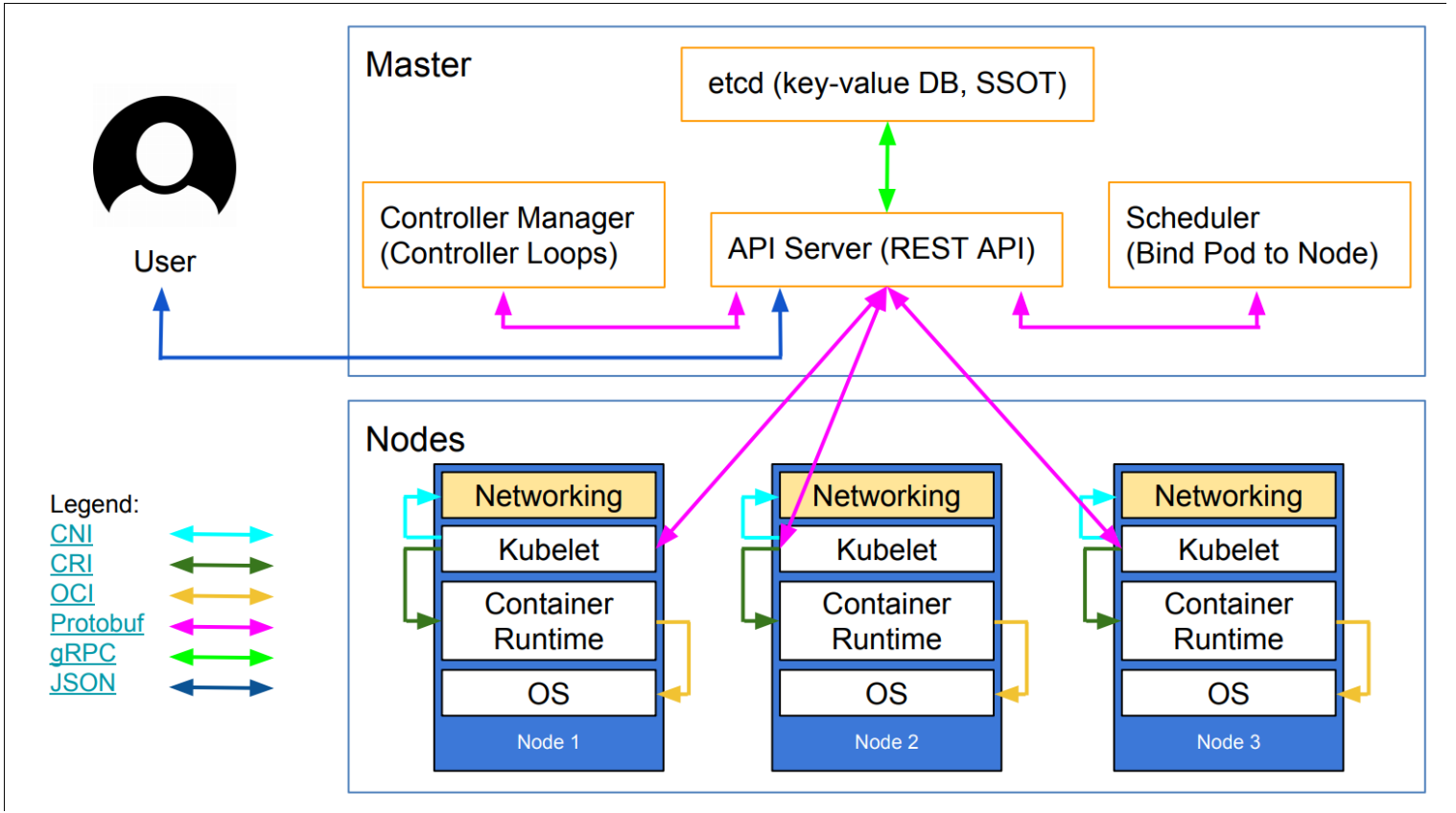


Credits: Google Kubernetes presentation

Kubernetes Architecture: Simplified Component Layout

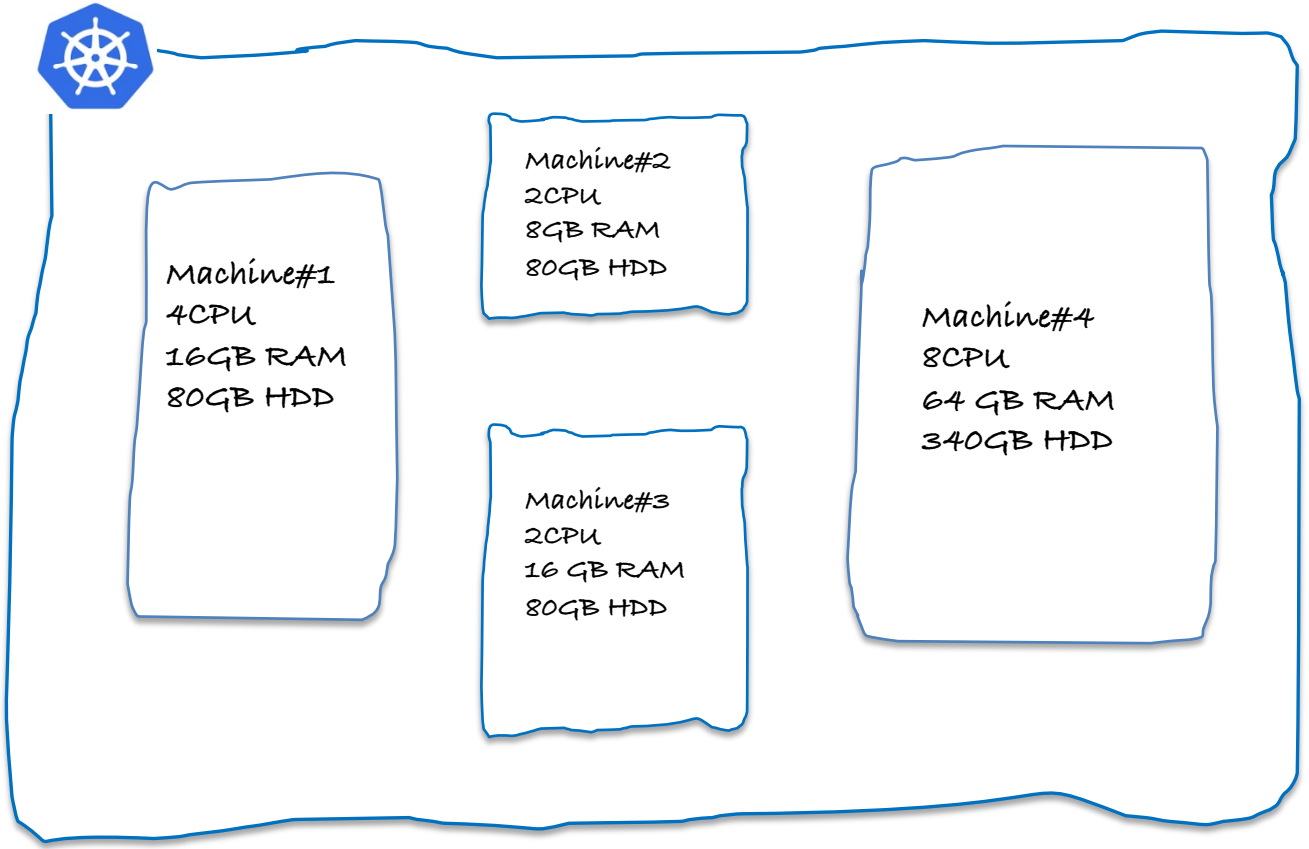


Kubernetes : Communication between Components

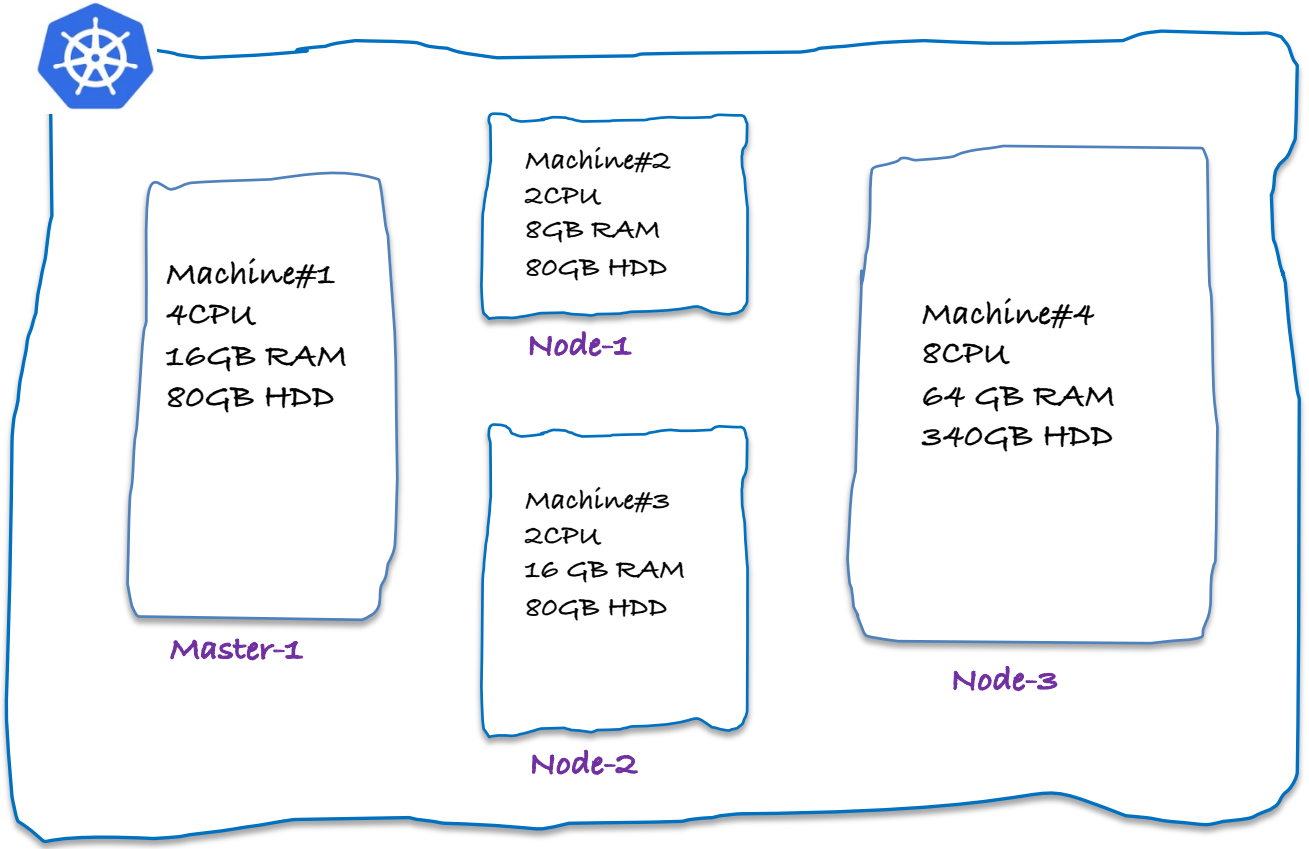


https://docs.google.com/presentation/d/1Gp-2blk5WExI_QR59EUZdwfO2BWLJqa626mK2ej-huo/edit#slide=id.g1e639c415b_0_56

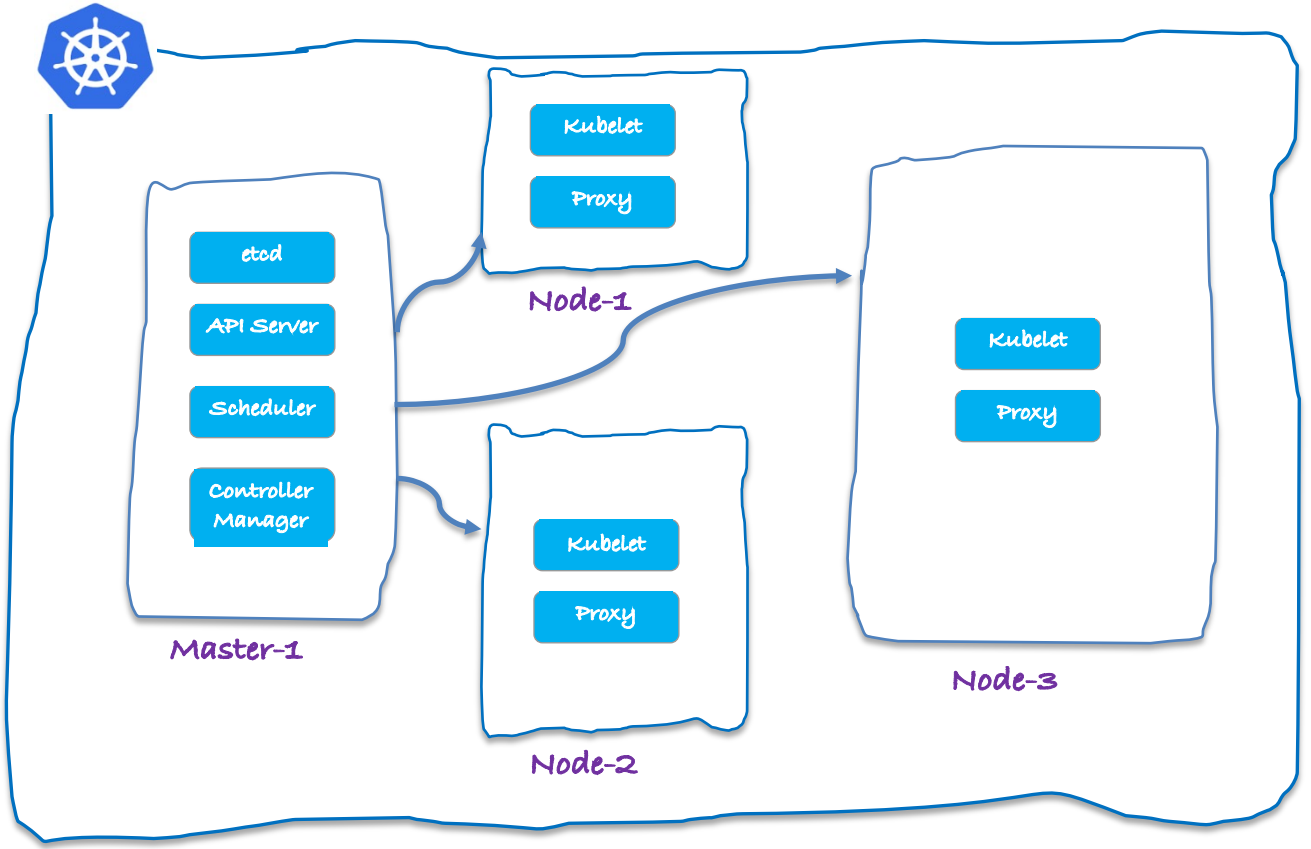
Kubernetes Cluster – Group of machines that act as one



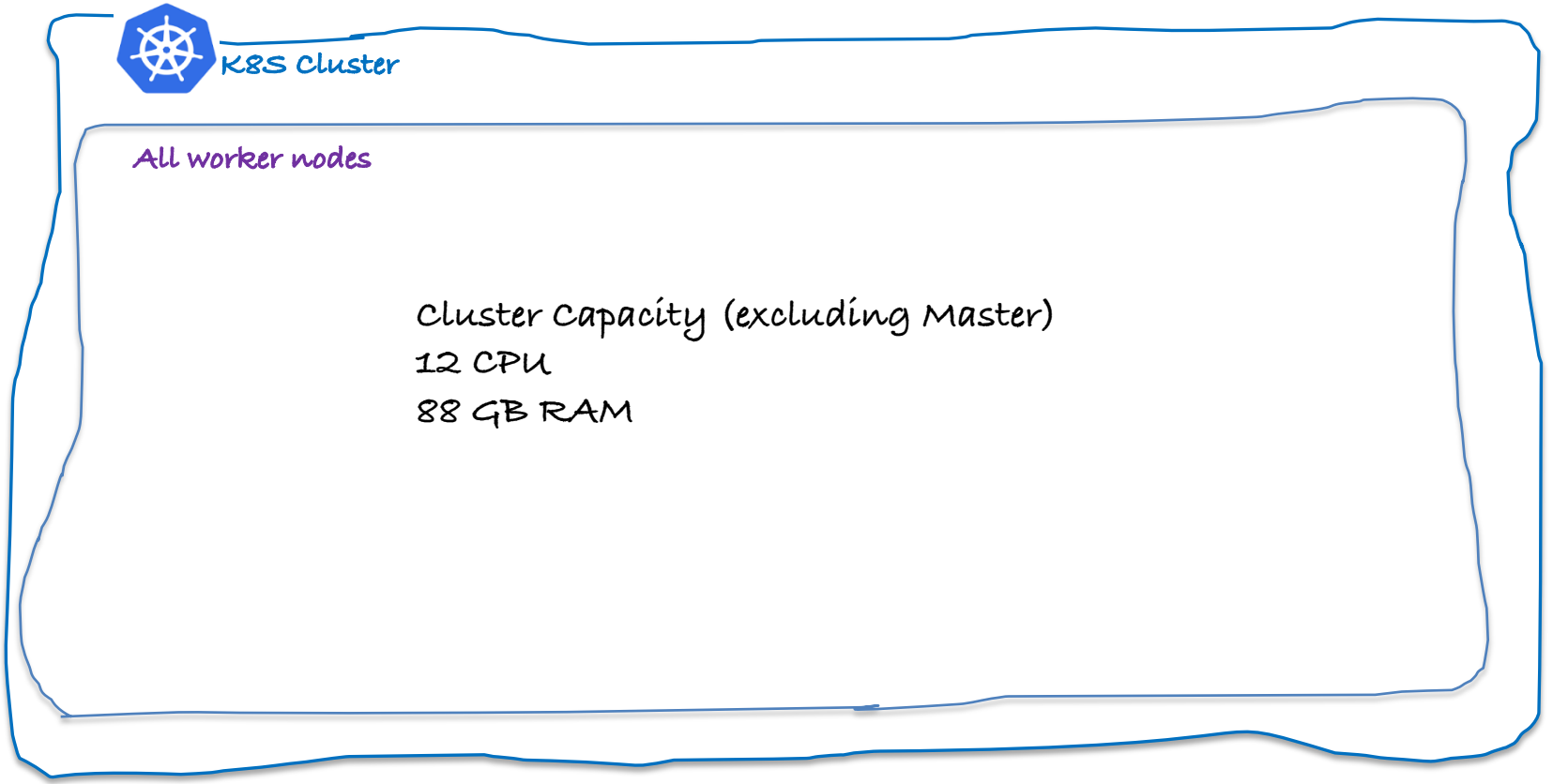
Kubernetes Nodes – Masters and Nodes – can be with different specifications



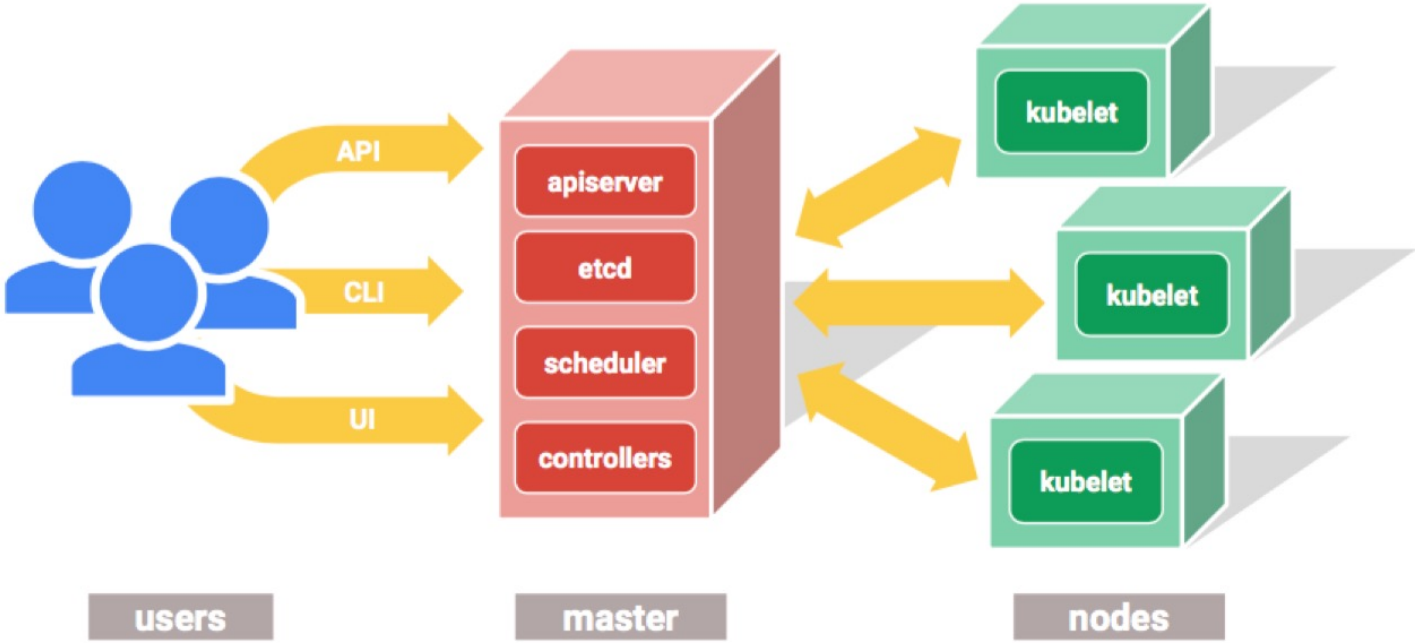
Kubernetes Nodes – Masters and Nodes (after K8S install)



Kubernetes Cluster Capacity – After Install

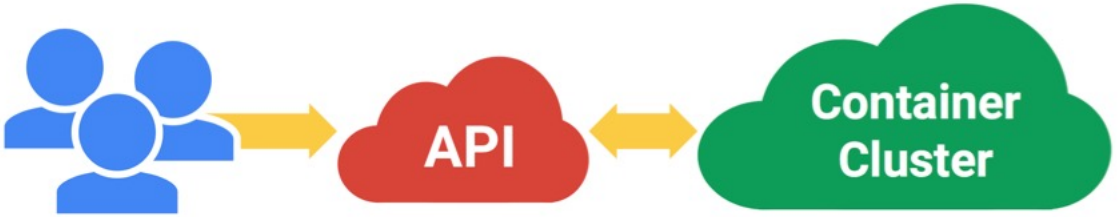


Kubernetes High Level Architecture: 10000 Feet



Credits: Google Kubernetes presentation

Kubernetes High Level Architecture: All you really care about

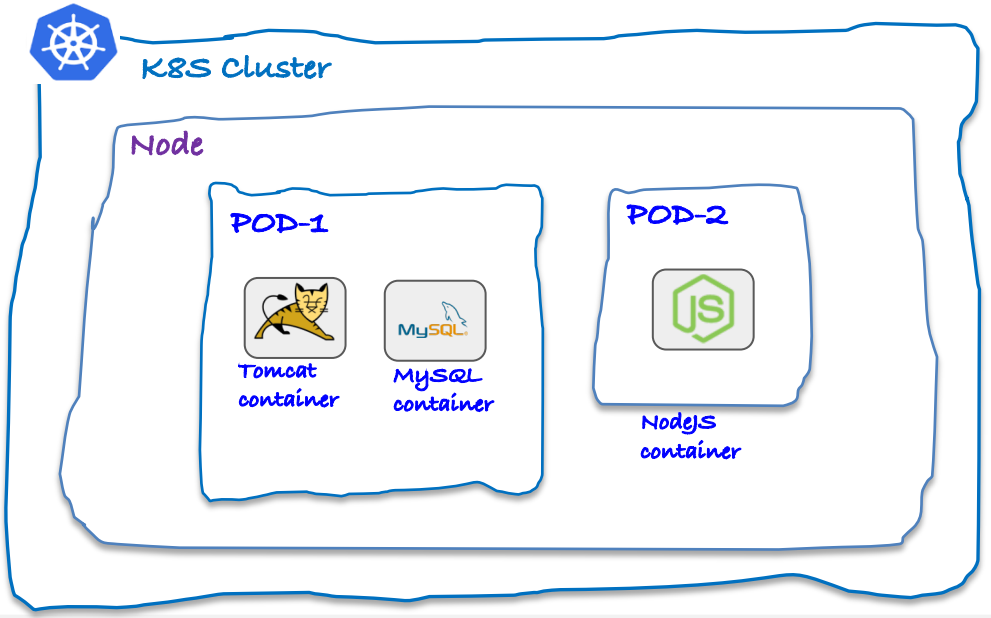


Credits: Google Kubernetes presentation

Key Features of Kubernetes

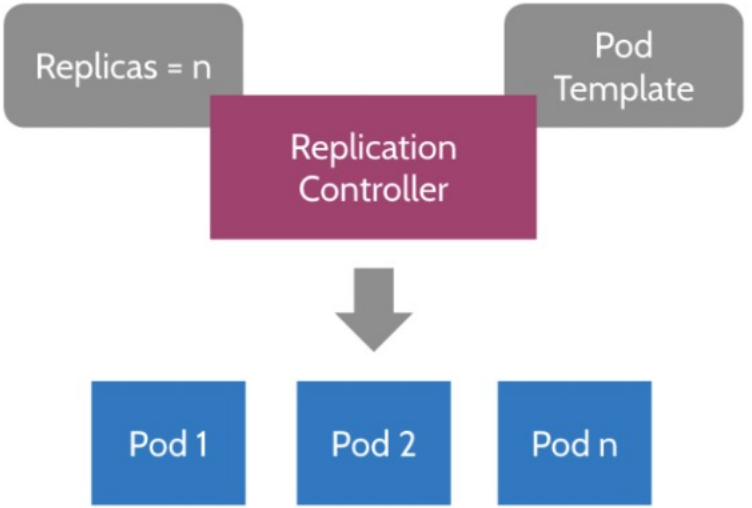
Key Features - Container Grouping - Pods

- Atomic unit or smallest "unit of work" of Kubernetes
- A Pod is a group of containers (one of more) that share the storage, users, network interfaces, etc. using Linux namespaces (ipc, uts, mount, pid, network and user), cgroups, and other kernel features.
- Containers in a pod share an IP address and the port space and storage space
- Pods are transient (ephemeral) – can be deleted anytime, storage can be attached/detached, and talk to each other through abstraction



Key Features - Container Orchestration - Replication Controller

- Replication controller is a logical entity that creates and manages pods.
- Replication controllers uses a pod template for defining the container image identifiers, ports, and labels.
- Replication controllers auto heal pods according to the given health checks.
- These health checks are called liveness probes.
- Replication controllers support manual scaling of pods, and this is handled by the replica count.



Key Features - Health Checking

Software applications fail due to many reasons:

- bugs in the code
- resource limitations
- networking issues
- infrastructure problems, etc.

Monitoring software application deployments is essential.

Kubernetes (via Kubelet agent) provides two main mechanisms for monitoring applications.

- Process Health Checking
- Application Health Checking

Key Features - Health Checking - Process Health Checking

Process Health Checking:

- Kubelet continuously checks the health of the containers via the Docker daemon
- If a container process is not responding, it will get restarted.
- This feature is enabled by default and it’s not customizable

Key Features - Health Checking - Application Health Checking

Kubernetes provides three methods for monitoring the application health; these are known as health checking probes

HTTP GET:

If the application exposes an HTTP endpoint, an HTTP GET request can be used for checking the health status.
HTTP endpoint needs to return a HTTP status code between 200 and 399, for the application to be considered healthy

Container Exec:

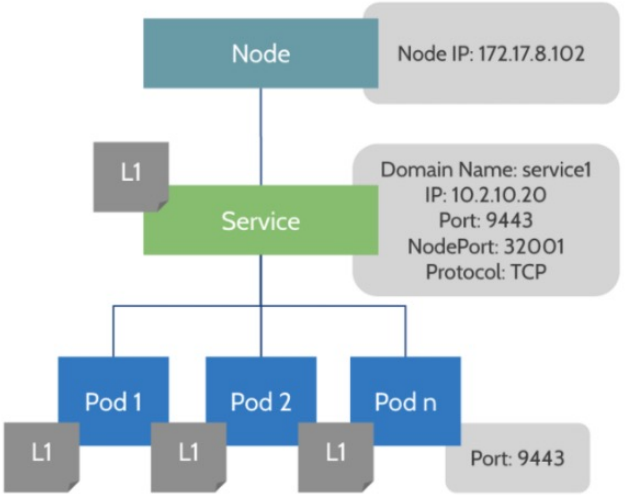
A shell command can be used for this purpose.
This command needs to return a zero to application to be considered healthy.

TCP Socket:

If none of the previous probes works, a simple TCP socket can also be used for checking the health status.
If Kubelet can establish a connection to the given socket, the application is considered healthy.

Key Features - Service Discovery and Load Balancing

- Kubernetes service provides a mechanism for load balancing pods.
- It is implemented using kube-proxy
- Internally uses iptable rules for load balancing at the network layer.
- Each Kubernetes service exposes a DNS entry for accessing the services within the Kubernetes internal network



Key Features - Service Discovery and Load Balancing

A Kubernetes service can be implemented as one of the following types:

ClusterIP: This type will make the service only visible to the internal network for routing internal traffic.

NodeIP: This type will expose the service via node ports to the external network. Each port in a service will be mapped to a node port and those will be accessible via <node-ip>:<node-port>.

Load Balancer: If services need to be exposed via a dynamic load balancer the service type can be set to Load Balancer. This feature is enabled by the underlying cloud provider (example: GCE).

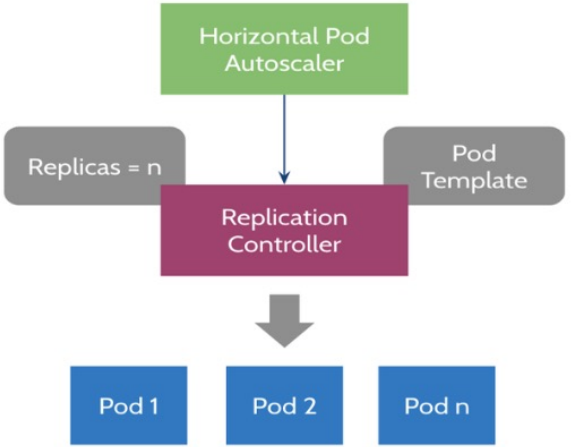
Key Features - Automated Rollouts and Rollbacks

Automated Rollouts and Rollbacks:

- This is one of the distinguishing features of Kubernetes that allows users to do a rollout of a new application version without a service outage.
- A rolling update can be triggered by packaging the new version of the application to a new container image.
- The rolling update process will create a new replication controller and rollout one pod at a time using the new replication controller created.
- Once all the pods are replaced the existing replication controller will be removed.
- Similarly, an application update done via a rolling update can be rolled back if needed.

Key Features - Horizontal Autoscaling

- Horizontal Pod Autoscalers provide autoscaling capabilities for pods.
- It does this by monitoring health statistics sent by the cAdvisor.
- A cAdvisor instance runs in each node and provides information on CPU, memory, and disk usage of containers.
- These statistics get aggregated by Heapster and get accessible via the Kubernetes API server.
- Currently, horizontal autoscaling is only available based on CPU usage.
- Work is in progress to support custom metrics.



Key Features - Secret and Configuration Management

Secret

- Applications that run on pods may need to contain passwords, keys, and other sensitive information.
- Packaging them with the container image may lead to security threats.
- Kubernetes provides a much more secure mechanism to send this sensitive information to the pods at the container startup without packaging them in the container image.
- These entries are called secrets and can be created via the secret API.
- The Secret name can then be given in the replication controller to let the pods access the actual value of the secret at the container startup.

Configuration Management

- Kubernetes uses secrets method for sending the token needed for accessing the Kubernetes API server to the pods.
- Kubernetes supports sending configuration parameters to the pods via ConfigMap API.

Both secrets and config key/value pairs can be accessed inside the container either using a virtual volume mount or using environment variables.

Key Features - Storage Orchestration

- Docker supports mounting storage systems to containers using container host storage or network storage systems.
- We have already see this in the Docker session and all those features are provided by Kubernetes.
- Kubernetes also provides via the Kubernetes API support for NFS, iSCSI, Gluster, Ceph, Cinder, or Flocker.

Key Features - Providing Well Known Ports for Kubernetes Services

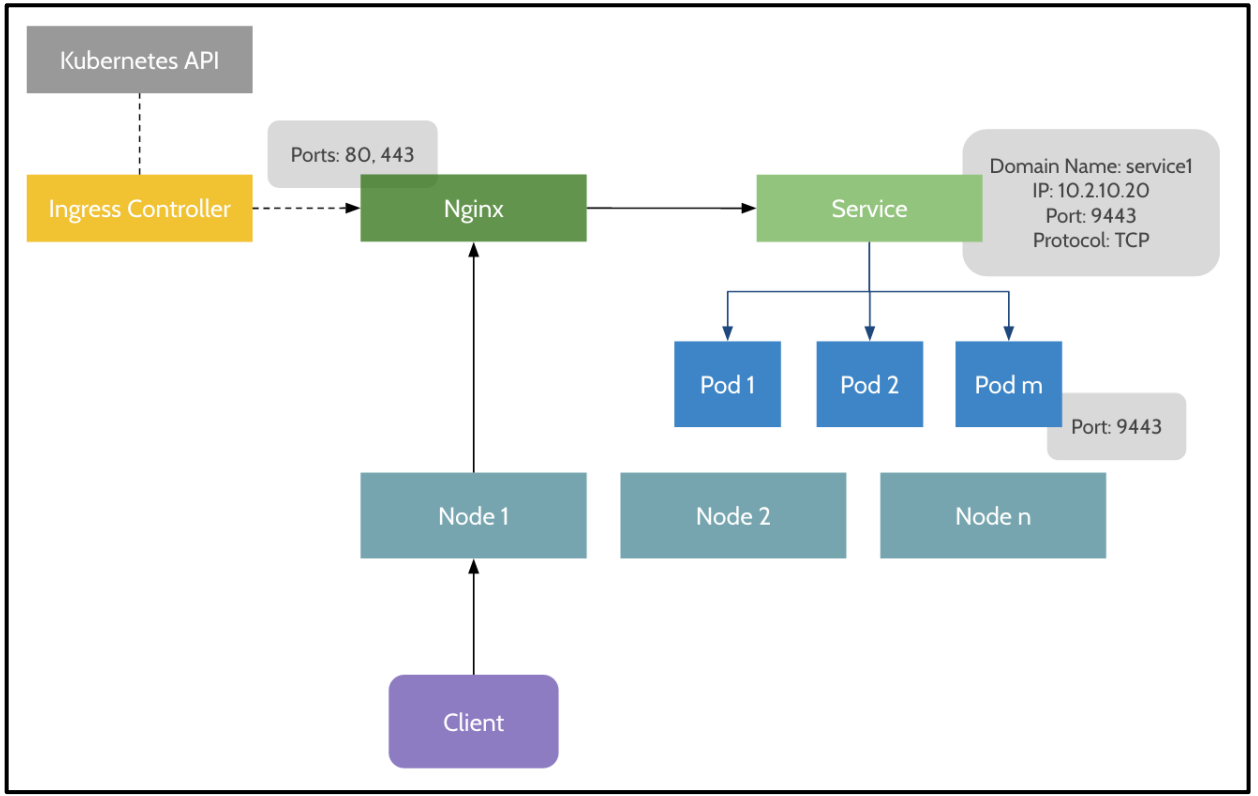
- Kubernetes provides a mechanism for adding a proxy server for Kubernetes services.
- This feature is known as Ingress.
- This is the ability to expose Kubernetes services via well-known ports, such as 80, 443.
- An ingress controller listens to Kubernetes API, generates a proxy configuration in runtime whenever a service is changed, and reloads the Nginx configuration.
- It can expose any given port via a Docker host port.
- Clients can send requests to Kubernetes node IPs, Nginx port and those will get redirected to the relevant service.
- The service will do round robin load balancing in the network layer.
- The service can be identified using an URL context or hostname;

  https://node-ip/foo/,https://foo.bar.com/

Key Features - Service Load Balancers and Sticky Session Management

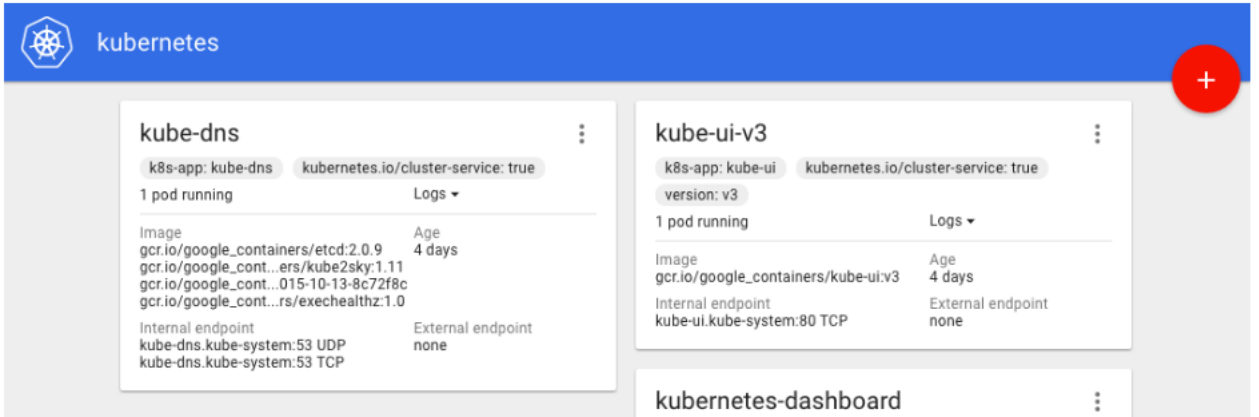
- Similar to ingress controllers, Kubernetes provides another mechanism for load balancing pods using third-party load balancers.
- These are known as service load balancers.
- Unlike ingress, service load balancers don’t route requests to services, rather they are dispatched directly to the pods.
- The main advantage of this feature is the ability to provide sticky session management at the load balancer.

Key Features - Service Load Balancers Architecture



Key Features - Kubernetes Dashboard

- Kubernetes dashboard provides features for deploying and monitoring applications.
- Any server cluster can be deployed by specifying a Docker image ID and required service ports.
- Once deployed, server logs can be viewed via the same UI.



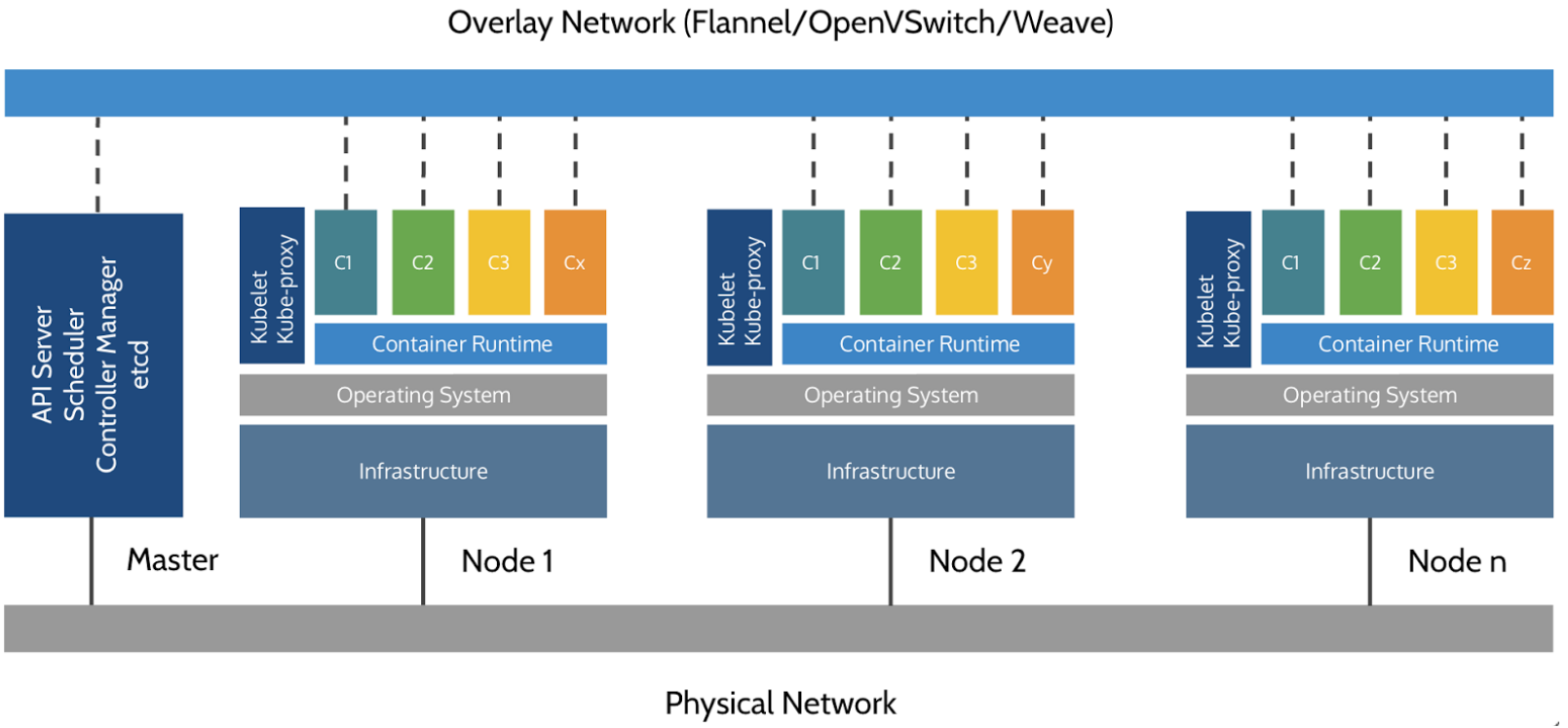
Key Features - Resource Usage Monitoring

Resource Usage Monitoring

- Kubernetes uses cAdvisor for monitoring containers in each node.
- It provides information on CPU usage, memory consumption, disk usage, network statistics, etc.
- A component called Metrics Server aggregates above data and makes them available via Kubernetes API.
- Data can be written to a data store and visualized via a UI.
- Prometheus and Grafana can be used for this purpose.

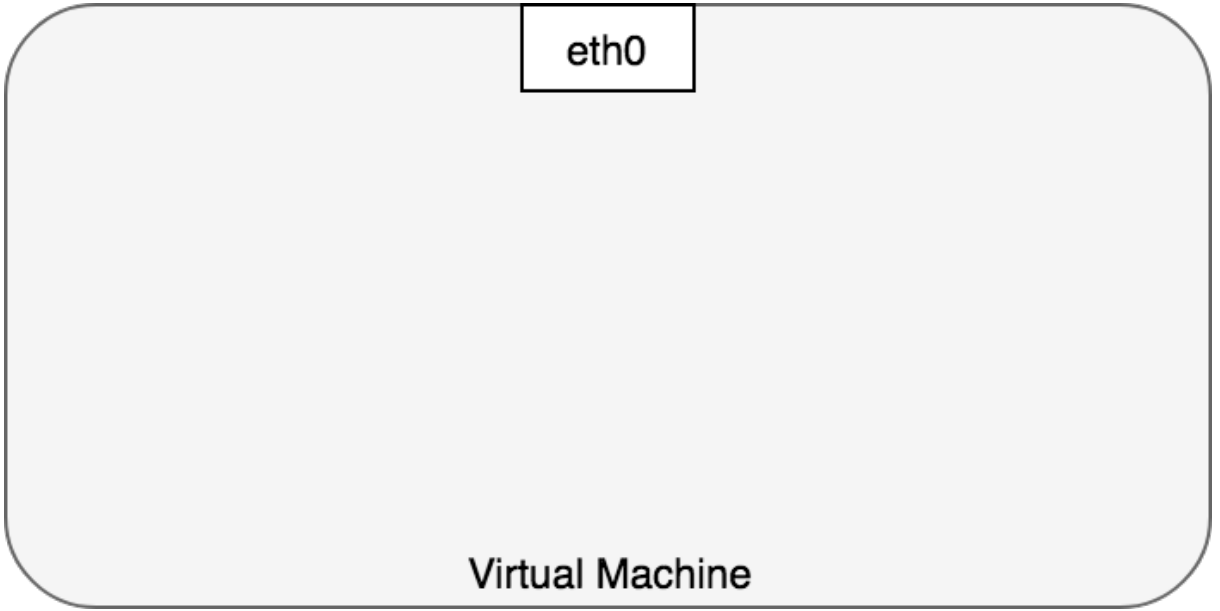
Kubernetes Network Architecture

Kubernetes High Level Architecture

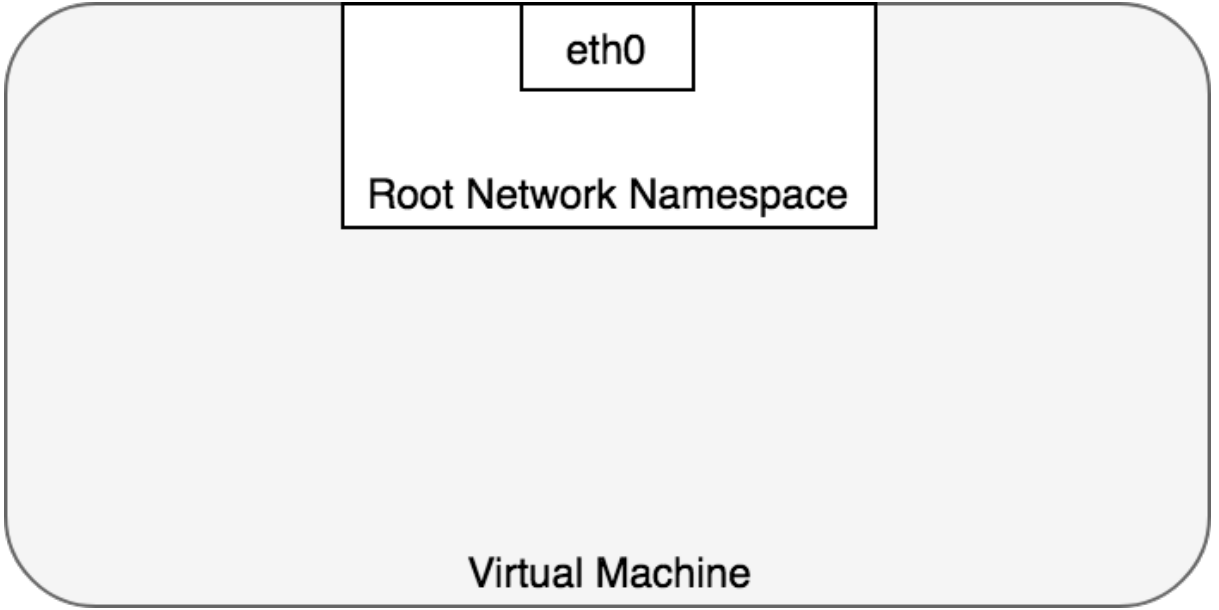


Credits: Image source : https://goo.gl/bZOet6

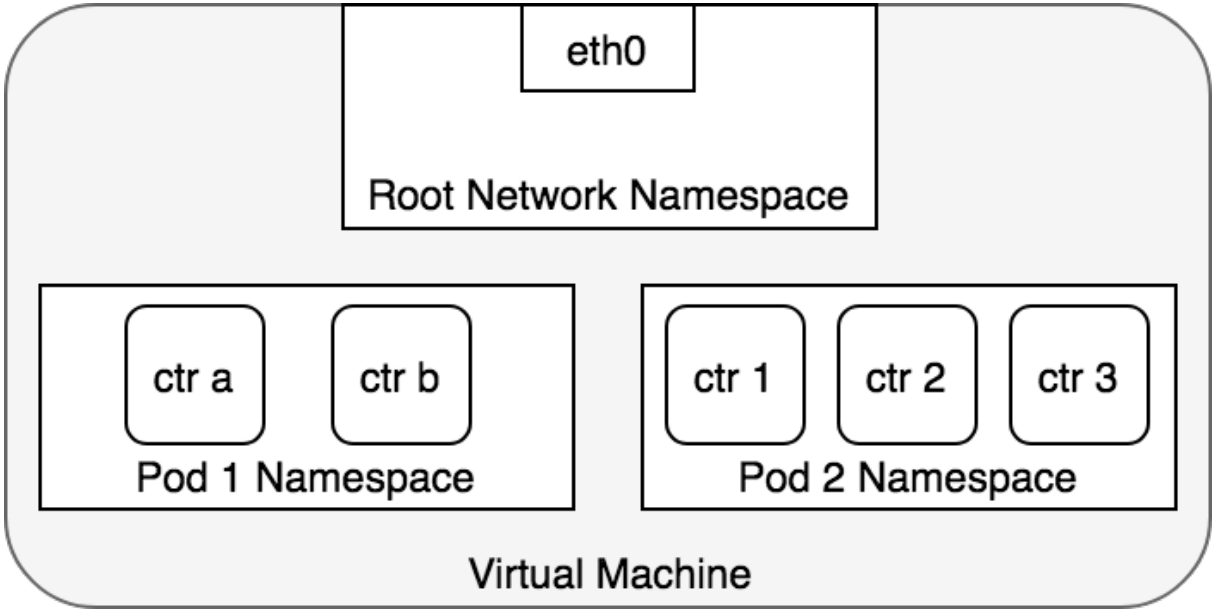
Container-to-Container Networking



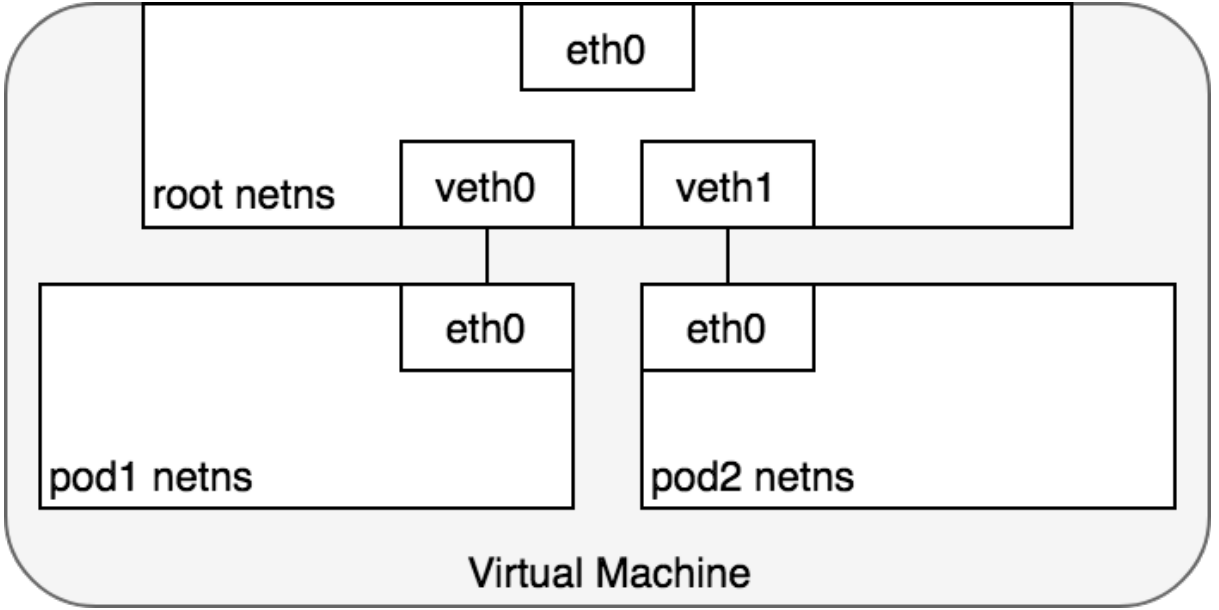
How Overlay Network Works



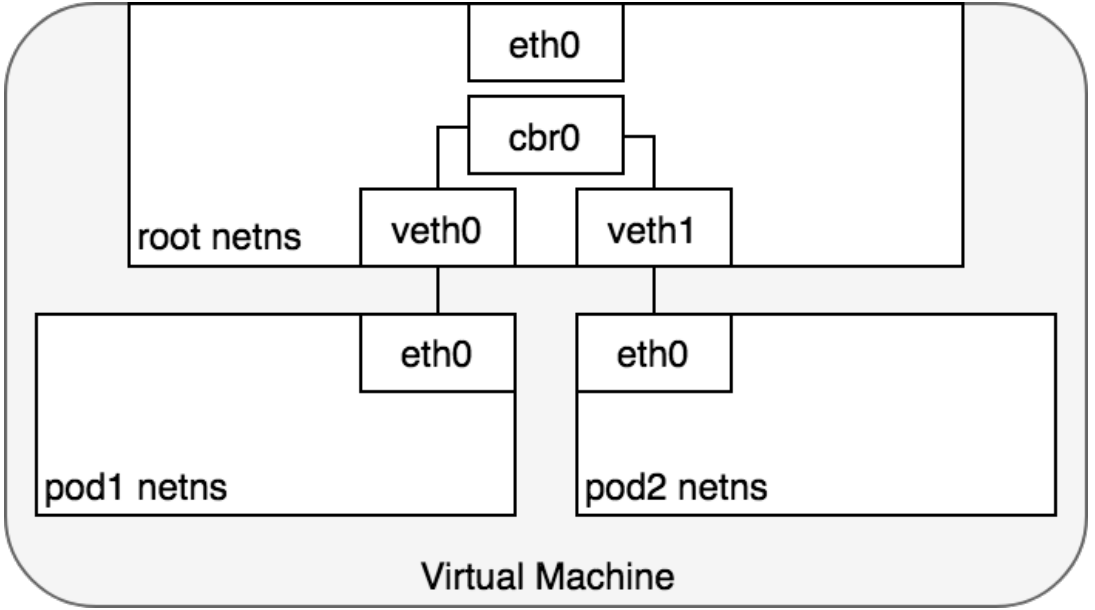
How Overlay Network Works



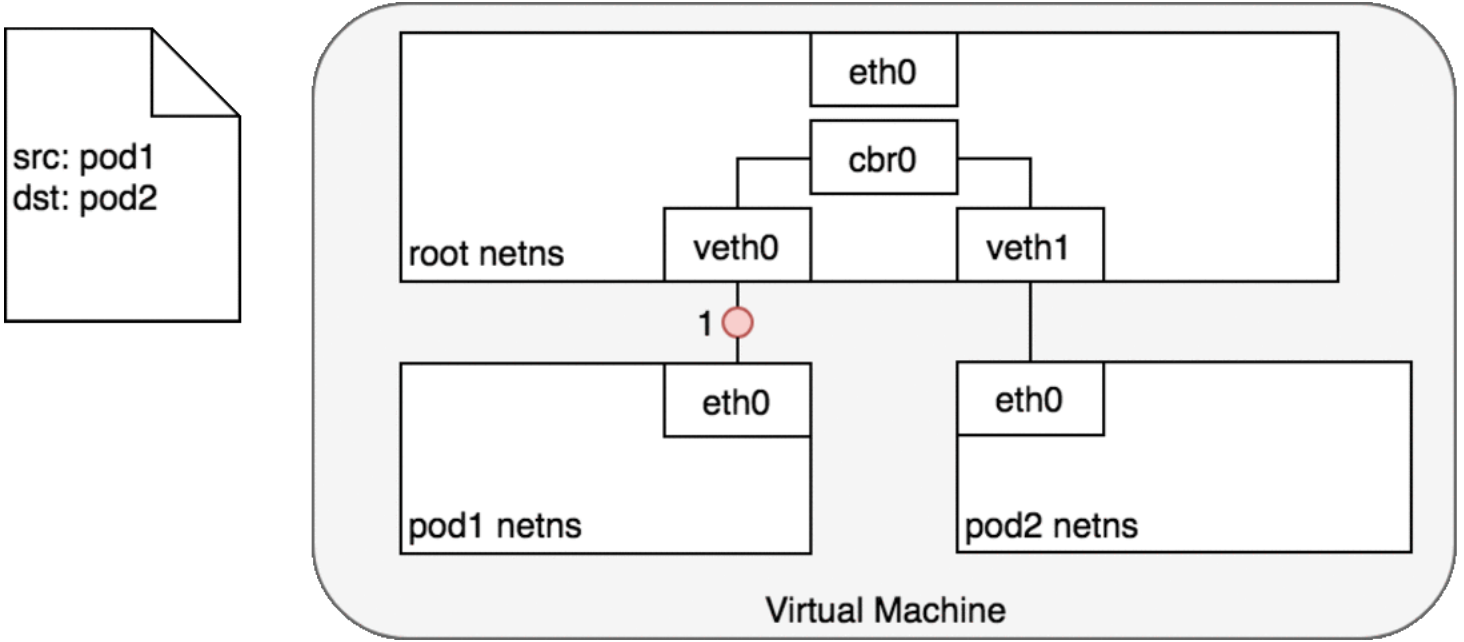
Pod-to-Pod Networking - veth pairs per Pod



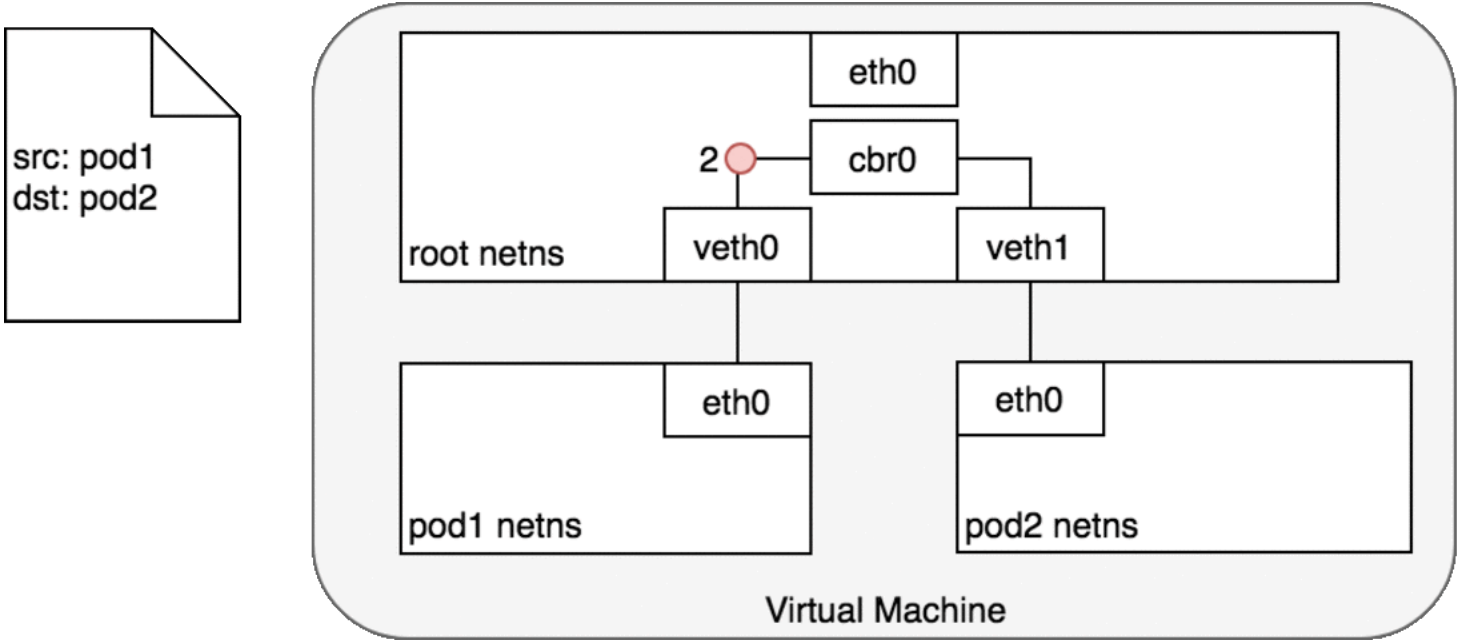
Pod-to-Pod Networking - Connecting namespaces using a bridge



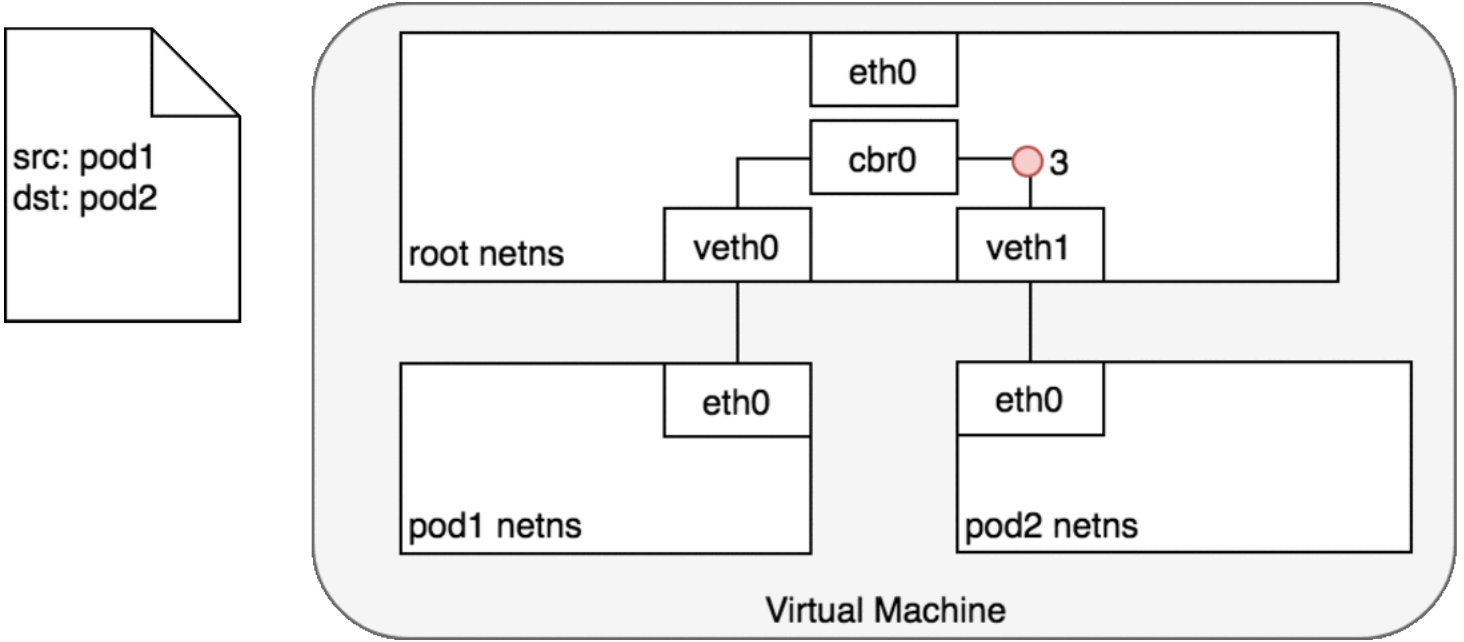
Pod-to-Pod Networking - Life of a packet: Pod-to-Pod, same Node



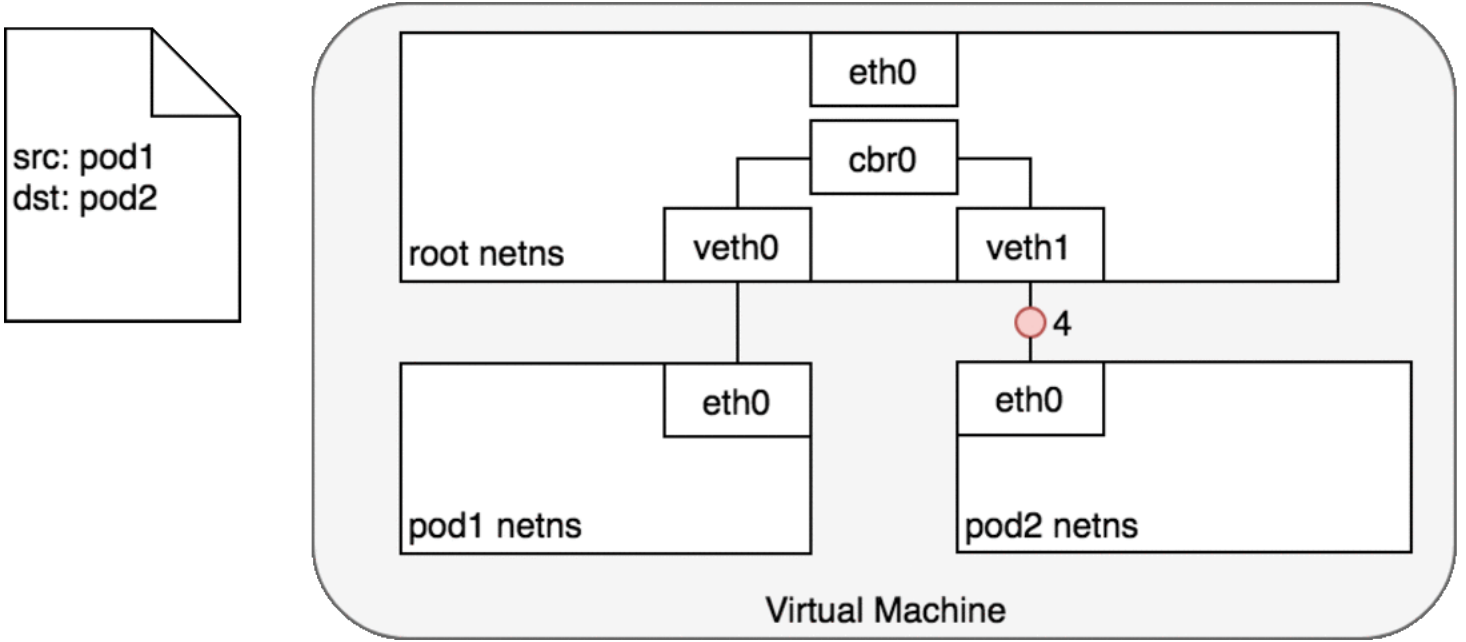
Pod-to-Pod Networking - Life of a packet: Pod-to-Pod, same Node



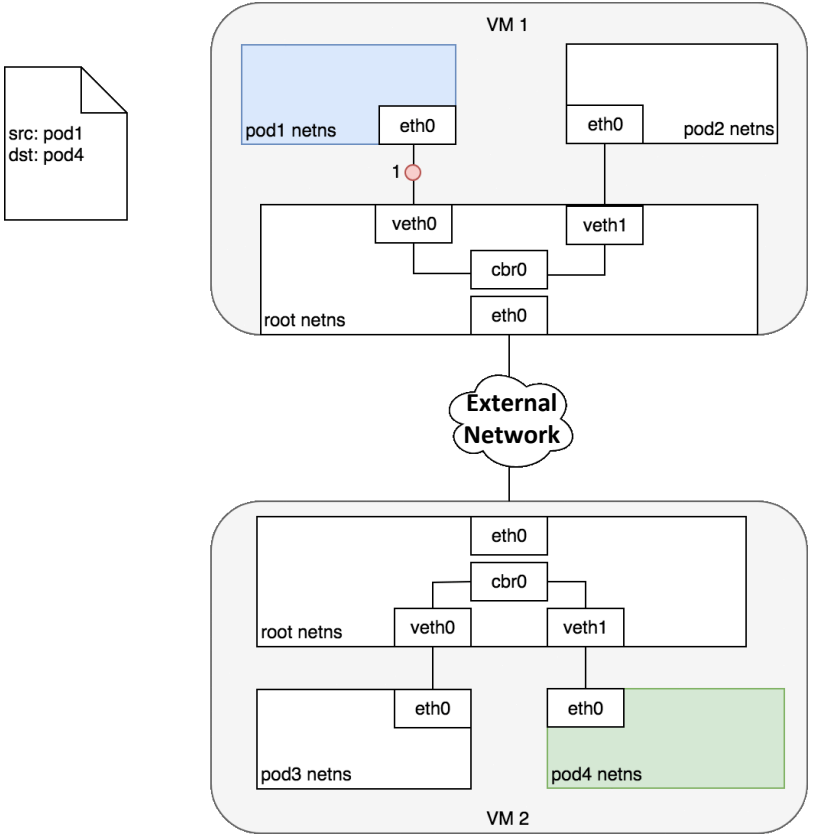
Pod-to-Pod Networking - Life of a packet: Pod-to-Pod, same Node



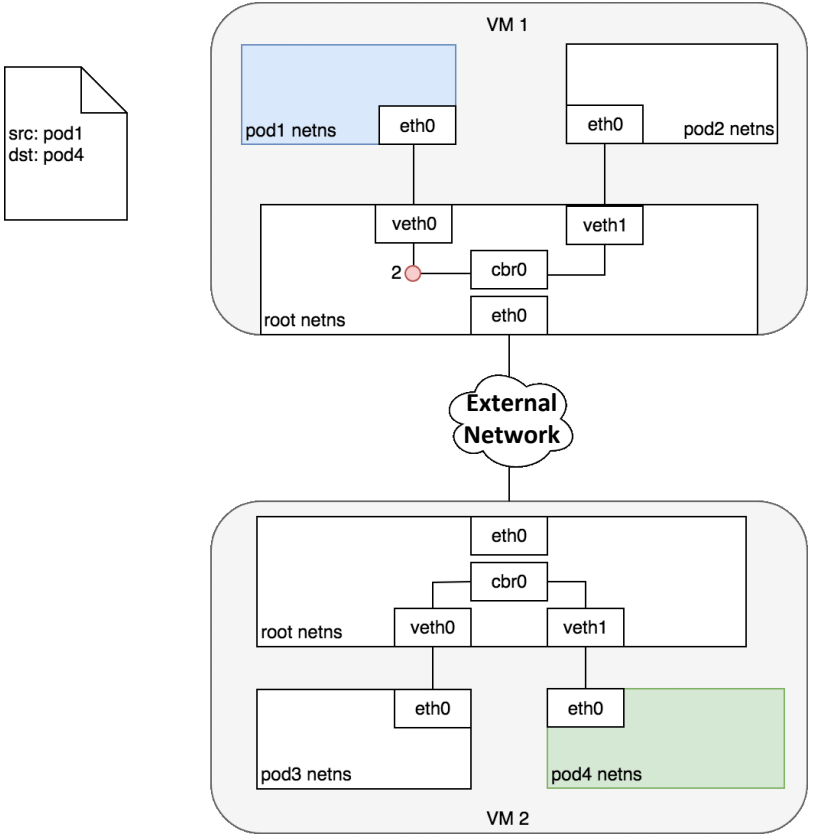
Pod-to-Pod Networking - Life of a packet: Pod-to-Pod, same Node



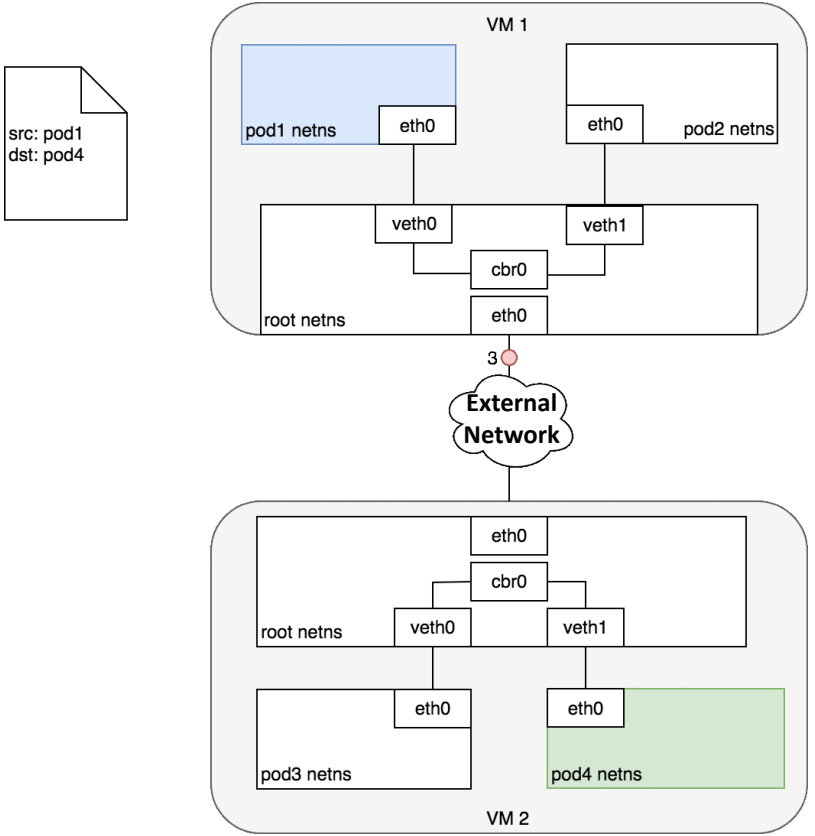
Life of a packet: Pod-to-Pod, across Nodes



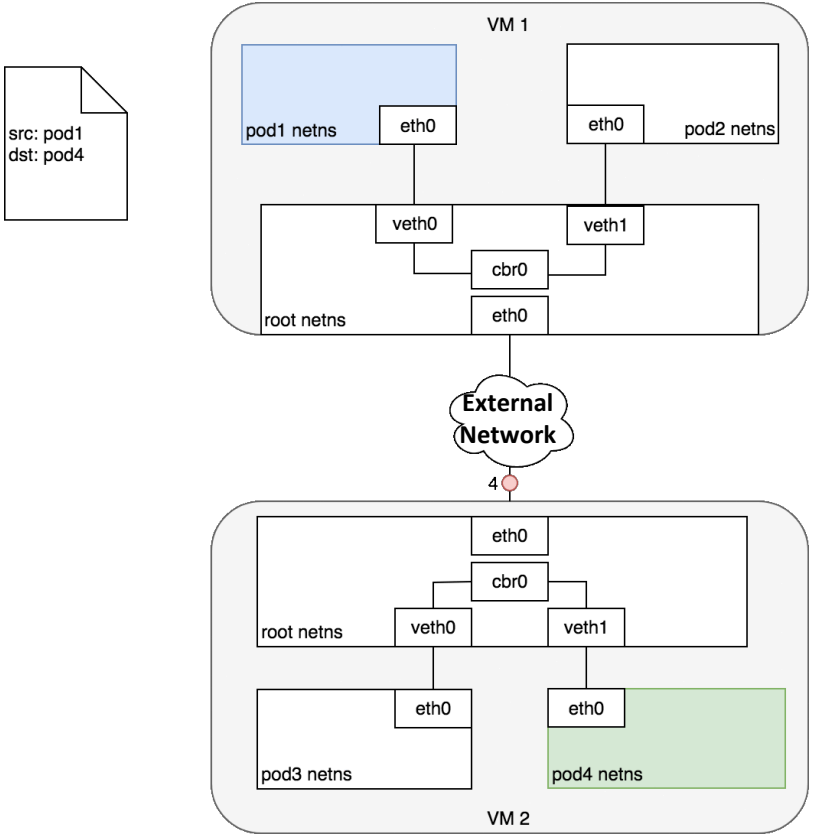
Life of a packet: Pod-to-Pod, across Nodes



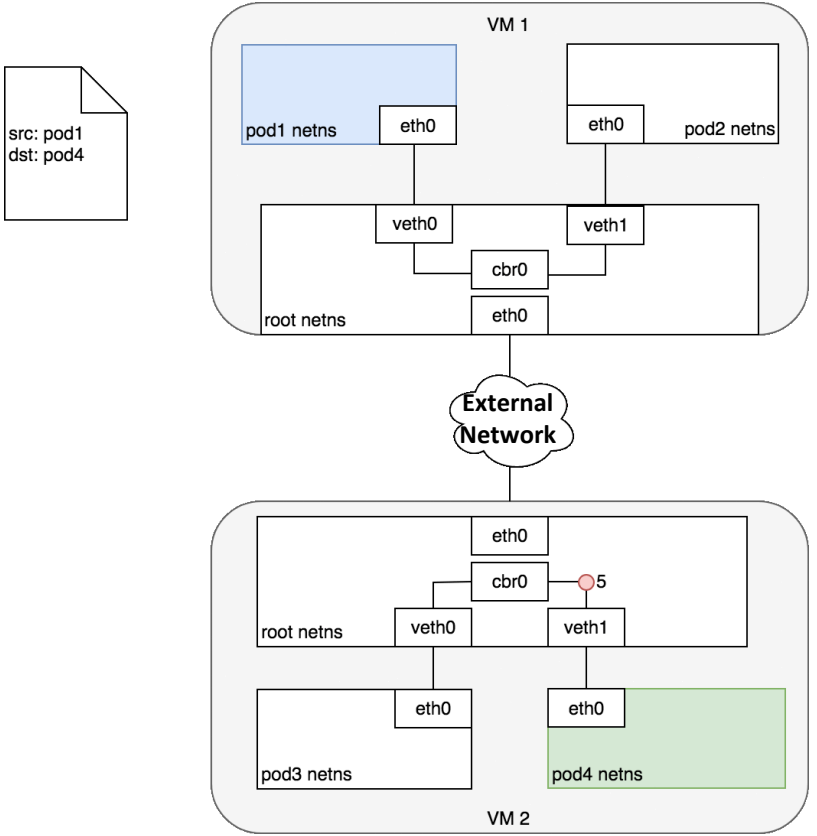
Life of a packet: Pod-to-Pod, across Nodes



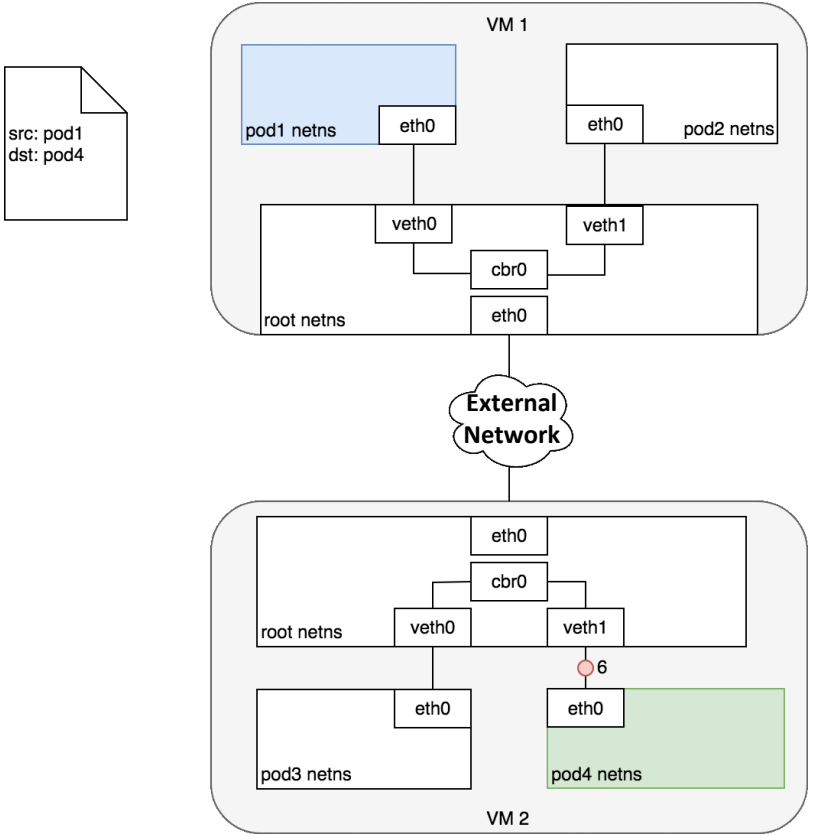
Life of a packet: Pod-to-Pod, across Nodes



Life of a packet: Pod-to-Pod, across Nodes



Life of a packet: Pod-to-Pod, across Nodes



Key Features - Kubernetes Network Architecture

Pod Network

- Cluster-wide network used for pod-to-pod communication and Pod networking managed by a CNI (Container Network Interface) plugin.
- CNI functions as an interface between the container runtime and a network implementation plugin

Service Network

- Cluster-wide range of Virtual IPs managed by kube-proxy for service discovery

Key Features - Kubernetes Network Architecture

CNI Plugins

- Amazon ECS
- Calico
- Cillium
- Contiv
- Contrail
- Flannel
- GCE
- kube-router
- Multus
- OpenVSwitch
- Romana
- Weave

Key Features - Kubernetes Network Architecture

Fundamental Kubernetes Network Rules

- Everything in Kubernetes cluster (nodes and pods) is one big flat IP network.
- All nodes must be able to reach each other, without NAT
- All pods must be able to reach each other, without NAT
- Pods and nodes must be able to reach each other, without NAT
- Each pod is aware of its IP address (no NAT)
- Kubernetes doesn't mandate any particular implementation

What this means:

- Everything can reach everything.
- No address translation, No port translation, No new protocol
- Pods **cannot** move from a node to another and keep their IP address
- If you want security, you need to add network policies
- There are many implementations out there for networking

Key Features - Kubernetes Network Architecture

Networking: Container-to-Container

- Containers within a pod exist within the same network namespace and share an IP.
- Enables intrapod communication over localhost.

Networking: Pod-to-Pod

- Allocated cluster unique IP for the duration of its life cycle.
- Pods themselves are fundamentally ephemeral.

Networking: Pod-to-Service

- managed by kube-proxy and given a persistent cluster unique IP
- exists beyond a Pod’s lifecycle.

Networking: External-to-Service

- Handled by kube-proxy
- Works in cooperation with a cloud provider or other external entity (load balancer or network).

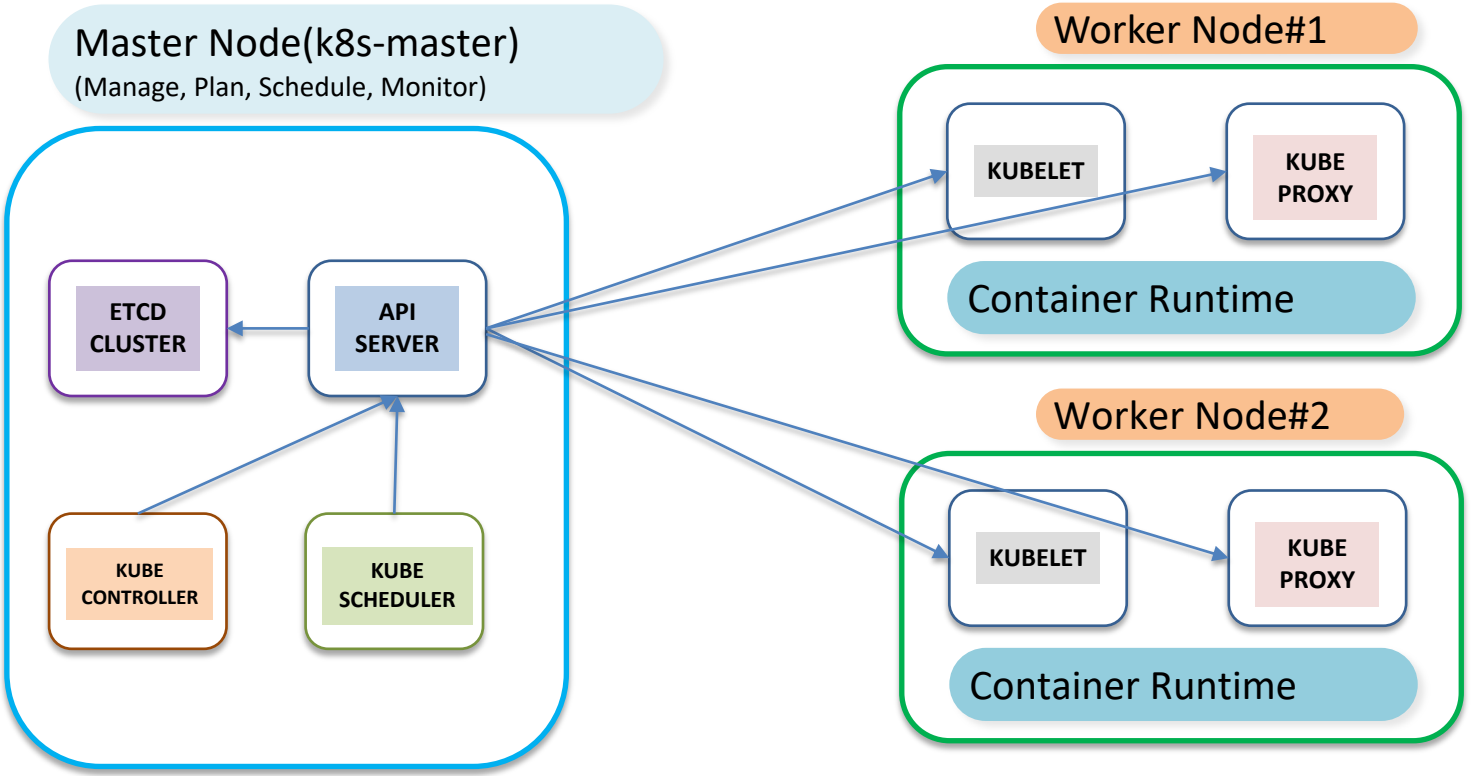
Installing Kubernetes

Install all prerequisites

- Check and confirm all hostname and make sure that the IP address is matching with eth0
- Install Network Tools, Bind tools, wget
- Disable **firewalld, NetworkManager and iptables**
- Enable legacy network services
- Disable SELINUX
- Load and enable br_netfilter module (bridge-netfilter)

  [See http://ebtables.netfilter.org/documentation/bridge-nf.html for features of bridge-netfilter]

Kubernetes Lab Layout



Confirm the Hostname and POD Number

On all hosts confirm the following:

```
[root@ALL_NODES|/root]# hostname

[root@ALL_NODES|/root]# cat pod*.TXT

[root@ALL_NODES|/root]# echo ${PODURL}

[root@ALL_NODES|/root]# cat /etc/hosts
```

Installing Kubernetes – check /etc/resolv.conf

```
[root@ALL_NODES|/root]# cat /etc/resolv.conf

search podx.example.com
nameserver 172.20.XX.YY
```

[[Note: the “search” should match as above on ALL_NODES with you POD-NAME.
Else correct it using the Unix command]]

Install Docker as the container platform

Install Docker (on Master and on both Worker Nodes), enable and start the service

```
[root@ALL_NODES|/root]# systemctl enable docker

[root@ALL_NODES|/root]# systemctl start docker
```

Check and make sure that the service is up and running

```
[root@ALL_NODES |/root]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Thu 2017-12-14 19:29:47 IST; 4s ago
     Docs: http://docs.docker.com
 Main PID: 8920 (dockerd-current)
   CGroup: /system.slice/docker.service


[root@ALL_NODES |/root]# docker version
```

Preload all Docker images that we need for our lab [For reference Only]

We will need to load all the Docker images required for Kubernetes since we are working in a restricted lab environment. All the images are available in your 'root' user home directory. This needs to be done an all nodes.

```
[root@ALL_NODES|/root]# cd /root/Kubernetes-installables/images/


[root@ALL_NODES|/root/Kubernetes-installables/images/]# sh -x import_main_k8s_images.sh

[this will take few minutes]
```

Verify that all images are imported

```
[root@ALL_NODES|/root]# docker images
REPOSITORY                                                    TAG                 IMAGE ID            CREATED             SIZE
docker.io/nginx                                               alpine              b6753551581f        9 days ago          21.4 MB
k8s.gcr.io/kube-proxy                                         v1.16.2             8454cbe08dc9        2 weeks ago         86.1 MB
k8s.gcr.io/kube-controller-manager                            v1.16.2             6e4bffa46d70        2 weeks ago         163 MB
k8s.gcr.io/kube-scheduler                                     v1.16.2             ebac1ae204a2        2 weeks ago         87.3 MB
k8s.gcr.io/kube-apiserver                                     v1.16.2             c2c9a0406787        2 weeks ago         217 MB
docker.io/kubernetesui/dashboard                              v2.0.0-beta5        b46f92585200        2 weeks ago         91.5 MB
k8s.gcr.io/metrics-server-amd64                               v0.3.6              9dd718864ce6        2 weeks ago         39.9 MB
k8s.gcr.io/etcd                                               3.3.15-0            b2756210eeab        8 weeks ago         247 MB
k8s.gcr.io/coredns                                            1.6.2               bf261d157914        2 months ago        44.1 MB
docker.io/kubernetesui/metrics-scraper                        v1.0.1              709901356c11        3 months ago        40.1 MB
quay.io/coreos/flannel                                        v0.11.0-amd64       ff281650a721        9 months ago        52.6 MB
k8s.gcr.io/pause                                              3.1                 da86e6ba6ca1        22 months ago       742 kB
docker.io/nginx                                               run                 f895b3fb9e30        22 months ago       108 MB
quay.io/kubernetes-ingress-controller/nginx-ingress-controller  0.9.0             e56d6a14283a        23 months ago       190 MB
docker.io/centos                                              7                   3fa822599e10        23 months ago       204 MB
docker.io/busybox                                             latest              6ad733544a63        24 months ago       1.13 MB
docker.io/busybox                                             run                 6ad733544a63        24 months ago       1.13 MB
docker.io/mhausenblas/simpleservice                           0.5.0               601917f29430        2 years ago         683 MB
gcr.io/google_containers/defaultbackend                       1.0                 137a07dfd084        4 years ago         7.51 MB
gcr.io/google_containers/hpa-example                          run                 4ca4c13a6d7c        4 years ago         481
MB
```

Install Kubernetes binaries

Again, we will install the pre-downloaded Kubernetes binaries since we are working in a restricted lab environment. This needs to be installed an all nodes.

```
[root@ALL_NODES|]# cd /root/Kubernetes-installables/binary/

[root@ALL_NODES|]# ls -l

[root@ALL_NODES|]# yum install -y *rpm

[this will take few minutes and also install any dependent rpms needed]

[root@ALL_NODES|]# cd /root

[root@ALL_NODES|]# rpm -qa | grep kube
kubeadm-1.16.2-0.x86_64
kubectl-1.16.2-0.x86_64
kubelet-1.16.2-0.x86_64
kubernetes-cni-0.7.5-0.x86_64
```

Enable Kubernetes ‘kubelet’ service [on k8s-master]

Enable and Start the ‘kubelet’ service and verify it. [this needs to be done on **k8s-master**]

```
[root@k8s-master|/root]# systemctl  enable kubelet

[root@k8s-master|/root]# systemctl  start kubelet

[root@k8s-master|/root]# systemctl  status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: activating (auto-restart) (Result: exit-code) since Fri 2019-11-01 21:42:55 UTC; 4s ago
     Docs: https://kubernetes.io/docs/
  Process: 9059 ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS (code=exited, status=255)
 Main PID: 9059 (code=exited, status=255)


[Service is in activating state]
```

Initialize the Kubernetes **Master** node on the new cluster [on k8s-master]

Use the “kubeadm init” command to initialize the cluster [[**PLEASE NOTE DOWN THE JOIN TOKEN]]**.

```
[root@k8s]# kubeadm init --pod-network-cidr=10.244.0.0/16 --kubernetes-version=1.22.2 | tee k8s-install.log
[init] Using Kubernetes version: v1.22.2
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
.
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Activating the kubelet service
..
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
.
.
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.122.162:6443 --token m1qx7i.xxxxxxx \
    --discovery-token-ca-cert-hash yyyyyyyyyyyyyyyyy
```

← **PLEASE NOTE THIS DOWN**

Now check the ‘kubelet’ service status again [on k8s-master]

After executing the ‘kubeadm init’ command, check the status of kubelet service:

```
[root@k8s-master|/root]# systemctl  status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Fri 2019-11-01 21:44:49 UTC; 5min ago
     Docs: https://kubernetes.io/docs/
 Main PID: 9291 (kubelet)
   CGroup: /system.slice/kubelet.service
           └─9291 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --
kubeconfig=/etc/kubernetes/kubelet.conf --c...


[Service is now in active state]
```

Start interacting with you cluster

Currently we have initialized the k8s cluster; we can start interacting with the cluster using the ‘kubectl’ command. For this we need the environment to be setup with the details of the k8s Custer.

The ‘kubeadm init’ command outputs the details of how to do it [shown below also]

```
To start using your cluster, you need to run (regular user - for our lab, we will run as ‘root’ user):

[root@k8s-master|/root]# mkdir -p $HOME/.kube
[root@k8s-master|/root]# sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[root@k8s-master|/root]# sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Check the initial status of the cluster

```
[root@k8s-master|/root]# kubectl get nodes
NAME                          STATUS     ROLES    AGE     VERSION
k8s-master.pod1.example.com   NotReady   master   5m44s   v1.16.2

[root@k8s-master|/root]# kubectl get pods --all-namespaces  -o wide -w
NAMESPACE     NAME                                                READY   STATUS    RESTARTS   AGE
kube-system   coredns-5644d7b6d9-6qhf4                            0/1     Pending   0          5m41s
kube-system   coredns-5644d7b6d9-j2f7d                            0/1     Pending   0          5m41s
kube-system   etcd-k8s-master.pod1.example.com                    1/1     Running   0          4m35s
kube-system   kube-apiserver-k8s-master.pod1.example.com          1/1     Running   0          4m51s
kube-system   kube-controller-manager-k8s-master.pod1.example.com 1/1     Running   0          5m1s
kube-system   kube-proxy-g8kbx                                    1/1     Running   0          5m41s
kube-system   kube-scheduler-k8s-master.pod1.example.com          1/1     Running   0          4m49s
```

We see that the Master is in **“NotReady”** state and that the k8s cluster DNS is in **“Pending”** status.

Note: We will cover the details of the above command in detail in the later slides. Don’t worry about the options for now.

Let’s quickly check the /var/log/message

Take a look at the /var/log/message file on the master node. What do you see? You will see tha there are messages like “network plugin is not ready: cni config uninitialized”

```
[root@k8s-master|/root]# tail -3 /var/log/messages
Nov  1 21:52:39 k8s-master kubelet: W1101 21:52:39.175741    9291 cni.go:237] Unable to update cni config: no networks
found in /etc/cni/net.d
Nov  1 21:52:39 k8s-master kubelet: E1101 21:52:39.743364    9291 kubelet.go:2187] Container runtime network not ready:
NetworkReady=false reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni config uninitialized
Nov  1 21:53:04 k8s-master kubelet: W1101 21:53:04.177061    9291 cni.go:237] Unable to update cni config: no networks
found in /etc/cni/net.d
Nov  1 21:53:04 k8s-master kubelet: E1101 21:53:04.753158    9291 kubelet.go:2187] Container runtime network not ready:
NetworkReady=false reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni config uninitialized
Nov  1 21:53:09 k8s-master kubelet: W1101 21:53:09.177339    9291 cni.go:237] Unable to update cni config: no networks
found in /etc/cni/net.d
```

Let's install the pod network

You **must** install a pod network add-on so that your pods can communicate with each other.

The network must be deployed before any applications. **Also, kube-dns, which we saw earlier, a helper service, will not start up before a network is installed.**

Check: https://github.com/flannel-io/flannel#flannel

kubeadm only supports Container Network Interface (CNI) based networks

```
[root@k8s-master|/root]# cd /root/Kubernetes-installables/yamls-for-cni

[root@k8s-master yamls]# kubectl apply -f kube-flannel.yml
Warning: policy/v1beta1 PodSecurityPolicy is deprecated in v1.21+, unavailable in v1.25+
podsecuritypolicy.policy/psp.flannel.unprivileged created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds created
```

Now check the status of the cluster

```
[root@k8s-master|/root]# kubectl get nodes
NAME                          STATUS   ROLES                  AGE     VERSION
k8s-master.pod1.example.com   Ready    control-plane,master   4m10s   v1.22.2

[root@k8s-master|/root]# kubectl get pods --all-namespaces  -o wide -w
NAMESPACE     NAME                                                 READY   STATUS    RESTARTS   AGE
kube-system   coredns-5644d7b6d9-6qhf4                             1/1     Running   0          11m
kube-system   coredns-5644d7b6d9-j2f7d                             1/1     Running   0          11m
kube-system   etcd-k8s-master.pod1.example.com                     1/1     Running   0          10m
kube-system   kube-apiserver-k8s-master.pod1.example.com           1/1     Running   0          10m
kube-system   kube-controller-manager-k8s-master.pod1.example.com  1/1     Running   0          10m
kube-system   kube-flannel-ds-amd64-zzbxt                          1/1     Running   0          55s
kube-system   kube-proxy-g8kbx                                     1/1     Running   0          11m
kube-system   kube-scheduler-k8s-master.pod1.example.com           1/1     Running   0          10m
```

We see that the master node is in ‘Ready’ and the service are in ‘Running’ state. You can check the /var/log/message to verify

So what is the story with the two nodes?

What about the other two nodes? The nodes are where the workloads (containers and pods, etc) run. We need to join the two nodes to the K8S cluster.

TIP: Please keep “`kubectl get pods --all-namespaces  -o wide -w`” running in one Putty window

Enable Kubernetes ‘kubelet’ service [on worker-node1 and worker-node2]

Enable and Start the ‘kubelet’ service and verify it.

```
[root@BOTH_WORKER_NODES|/root]# systemctl  enable kubelet

[root@BOTH_WORKER_NODES |/root]# systemctl  start kubelet

[root@BOTH_WORKER_NODES |/root]# systemctl  status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/etc/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: activating (auto-restart) (Result: exit-code) since Thu 2017-12-14 20:57:20 IST; 55ms ago

[Service is in activating state]
```

Joining your nodes

To add new nodes to your cluster do the following for each node [on worker-node1 and worker-node2]

```
[This command is from the output of the 'kubeadm init' command
[root@BOTH_WORKER_NODES|/root]# kubeadm join --token a355e6.xxxxx 10.0.2.180:6443 --discovery-token-ca-cert-
hash yyyyyyyyy

[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[kubelet-start] Downloading configuration for the kubelet from the "kubelet-config-1.16" ConfigMap in the kube-system namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Activating the kubelet service
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

Now check the status of the cluster once more

```
[root@k8s-master|/root]# kubectl get nodes
NAME                              STATUS   ROLES                  AGE    VERSION
k8s-master.pod1.example.com       Ready    control-plane,master   20m    v1.22.2
worker-node1.pod1.example.com     Ready    <none>                 106s   v1.22.2
worker-node1.pod1.example.com     Ready    <none>                 106s   v1.22.2

[root@k8s-master|/root]# kubectl get pods --all-namespaces  -o wide -w
NAMESPACE     NAME                                                READY   STATUS    RESTARTS   AGE
kube-system   coredns-5644d7b6d9-6qhf4                            1/1     Running   0          17m
kube-system   coredns-5644d7b6d9-j2f7d                            1/1     Running   0          17m
kube-system   etcd-k8s-master.pod1.example.com                    1/1     Running   0          16m
kube-system   kube-apiserver-k8s-master.pod1.example.com          1/1     Running   0          16m
kube-system   kube-controller-manager-k8s-master.pod1.example.com 1/1     Running   0          16m
kube-system   kube-flannel-ds-amd64-86bg9                         1/1     Running   0          3m43s
kube-system   kube-flannel-ds-amd64-tq4l4                         1/1     Running   0          119s
kube-system   kube-flannel-ds-amd64-zzbxt                         1/1     Running   0          7m12s
kube-system   kube-proxy-g8kbx                                    1/1     Running   0          17m
kube-system   kube-proxy-mkqhb                                    1/1     Running   0          119s
kube-system   kube-proxy-w45db                                    1/1     Running   0          3m43s
kube-system   kube-scheduler-k8s-master.pod1.example.com          1/1     Running   0          16m
```

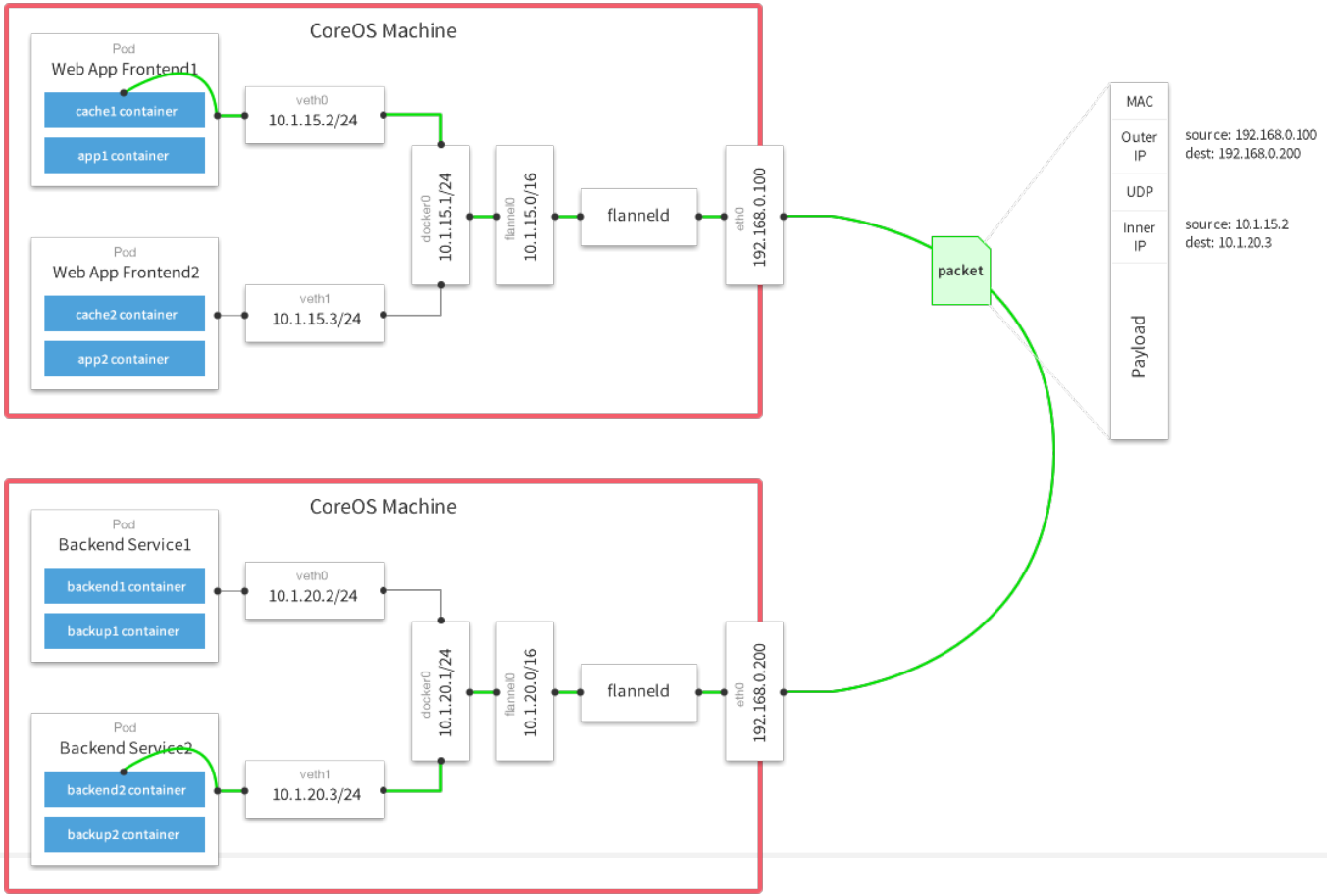
What are all these pods

- **etcd** is our etcd server
- **kube-apiserver** is the API server
- **kube-controller-manager** and **kube-scheduler** are other master components
- **coredns** is an DNS component
- **kube-proxy** is the (per-node) component managing port mappings and such
- **kube-flannel** is the (per-node) component managing the network overlay
- the **READY** column indicates the number of containers in each pod
- the pods with a name ending with "-**k8s-master.xxxx**" are the master components (they have been specifically "pinned" to the master node)

Check reachability of node over the flannel overlay network (from all nodes)

```
[root@k8s-master|/root]# ifconfig flannel.1
flannel.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.244.0.0  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::8ee:bbff:fe6b:891  prefixlen 64  scopeid 0x20<link>

[root@k8s-master|/root]# ping -c 2 10.244.0.0
[root@k8s-master|/root]# ping -c 2 10.244.1.0
[root@k8s-master|/root]# ping -c 2 10.244.2.0

[root@worker-node1|/root]# ping -c 2 10.244.0.0
[root@worker-node1|/root]# ping -c 2 10.244.1.0
[root@worker-node1|/root]# ping -c 2 10.244.2.0

[root@worker-node2|/root]# ping -c 2 10.244.0.0
[root@worker-node2|/root]# ping -c 2 10.244.1.0
[root@worker-node2|/root]# ping -c 2 10.244.2.0
```

Flannel overlay network (quick 101 -- see documents for more details)



Quick review of how we set up the Kubernetes cluster

- Installed prerequisites
- Installed Docker
- Installed Kubernetes packages
- Ran kubeadm init on the master node
- Set up Flannel (the overlay network)
- Ran kubeadm join on the other nodes (with the token produced by kubeadm init)
- Copied the configuration file generated by kubeadm init to setup the environment for kubectl
- Checked the Flannel network across all three nodes

Check the official site….

- https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
- https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/
- https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/
- https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/
- https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/setup-ha-etcd-with-kubeadm/

Installing the Kubernetes Web UI (Dashboard)

Kubernetes Web UI (Dashboard)

Kubernetes resources can also be viewed with a web dashboard.

We are going to deploy that dashboard with three commands:

- one to actually run the dashboard (using kubectl create)
- one to bypass authentication for the dashboard (using kubectl create)
- one to make the dashboard available from outside (using NodePort)

Installing and Running the Kubernetes Web UI (Dashboard)

We need to create a deployment and a service for the dashboard
We will also need a secret, a service account, a role and a role binding
All these things can be defined in a YAML file and created with kubectl apply -f

Create all the dashboard resources, with the following command:

```
[root@k8s-master|/root]# cd /root/labs/dashboard

[root@k8s-master|xx]# ls -l  dashboard*
-rw-r--r--. 1 1001 games  375 Oct 27 06:36 dashboard-v2.3.1-admin-service-account.yaml
-rw-r--r--. 1 1001 games 7639 Oct 27 06:36 dashboard-v2.3.1-recommended.yaml

[root@k8s-master|xx]# kubectl create -f dashboard-v2.3.1-recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
.
.
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

Installing and Running the Kubernetes Web UI (Dashboard)

Granting more rights to the dashboard and bypass authentication for the dashboard:

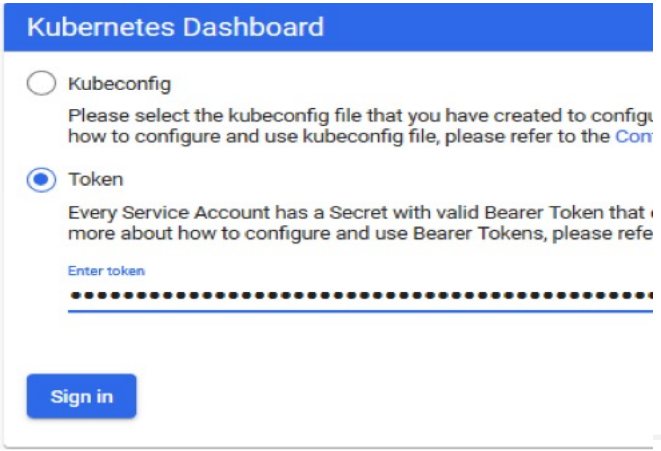
```
[root@k8s-master|xx]# kubectl create -f dashboard-v2.3.1-admin-service-account.yaml
serviceaccount/dashboard-admin created
clusterrolebinding.rbac.authorization.k8s.io/dashboard-admin created

[root@k8s-master|xx]# kubectl -n kubernetes-dashboard get svc
kubernetes-dashboard        NodePort    10.111.40.244   <none>        443:31234/TCP   5m31s

[root@k8s-master|xx]#  kubectl -n kubernetes-dashboard get secret | grep admin-user
admin-user-token-8xpgb             kubernetes.io/service-account-token   3      6m32s


[root@k8s-master|xx]# kubectl -n kubernetes-dashboard describe secret admin-user-token-8xpgb
token:      eyJhbGciOiJS. . . . . . . nZpY2VhY

[root@k8s-master|xx]# echo  $PODSURL:31234/
https://k8s-master.podxx.example.com:31234/
```

Accessing the dashboard:

Use Firefox and connect to the URL above to access Dashboard from your local workstation. On the screen below, copy-paste the value of "**Token:**" and click "**Sign In**" (Set the browser Proxy Server that is give to you)

Note:

```
"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe"  --ignore-certificate-errors
```



Default Kubernetes Web UI is as below:

kubernetes

Search

Cluster

Cluster
- Namespaces
- Nodes
- Persistent Volumes
- Roles
- Storage Classes

Namespace

All namespaces

Overview

Workloads
- Cron Jobs
- Daemon Sets
- Deployments

Namespaces

Name	Labels
kube-public	-
kube-system	-
default	-

Nodes

Name	Labels	Ready
worker-node2	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux kubernetes.io/hostname: worker-node2	True
worker-node1	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux	True

Kubectl - Our CLI to the Kubernetes Cluster

- kubectl - pronounced as "Cube C T L” or "Cube cuttle” or "Cube cuddle"...
- It has is a rich CLI tool around the Kubernetes API
- By default it uses the ~/.kube/config file

  - kubectl get
  - kubectl get node
  - kubectl get nodes -o wide
  - kubectl get nodes -o yaml
  - kubectl get services
  - kubectl get svc
  - kubectl get pods
  - kubectl get namespaces
  - kubectl get ns

Deploying Pod(s)

Credits: *Some of the example files and contents are from Kubernetes Tutorial and is reused in this presentation*

Pods - Launching the first pod

A pod is a collection of containers sharing a network and mount namespace and is the basic unit of deployment in Kubernetes. All containers in a pod are scheduled on the same node.

Let us launch a pod with a docker.io/`mhausenblas/simpleservice:0.5.0` image. (If you are in Lab environment, make sure you have this on all nodes using the 'docker images' command)

```
[root@k8s-master|/root]# kubectl create deployment simplesrv --image=docker.io/mhausenblas/simpleservice:0.5.0
deployment "simplesrv" created
```

Check if the pod is running and also check its details

```
[root@k8s-master|/root]# kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
simplesrv-7867dfcfb7-gr8dz    1/1       Running   0          8s
```

Pods - Show details of pod and its resource

Get the details of the pod

```
[root@k8s-master|/root]# kubectl describe pod simplesrv-7867dfcfb7-gr8dz | more
Name:           simplesrv-7867dfcfb7-gr8dz
Namespace:      default
Node:           worker-node2/10.0.2.182
Start Time:     Sun, 17 Dec 2017 16:55:32 +0530
Labels:         pod-template-hash=3114327547
.
.
<<snip snip snip snip>>
```

Wow!!! Way to much information.

Let us see if we can see which node it is running on:

```
[root@k8s-master|/root]# kubectl get pods -o wide
simplesrv-7867dfcfb7-gr8dz   1/1       Running   0          1m        10.244.1.3   worker-node1
```

Pods - Lets check some logs

Get the logs of the pod

```
[root@k8s-master|/root]# kubectl logs simplesrv-7867dfcfb7-gr8dz
2017-12-17T11:40:51 INFO This is simple service in version v0.5.0 listening on port 9876 [at line 142]
```

Pods - Accessing our pod

Lets get the IP-Address of the pod

```
[root@k8s-master|/root]# kubectl describe pod simplesrv-7867dfcfb7-gr8dz | grep IP:
IP:               10.244.1.3
```

From the master node we can access the pod using the IP we got earlier:

```
[root@k8s-master|/root]# curl 10.244.1.3:9876/info ; echo
{"host": "10.244.1.3:9876", "version": "0.5.0", "from": "10.244.0.0"}
```

Can we ping out pod IP?

```
[root@k8s-master|/root]# ping -c 2 10.244.1.3
PING 10.244.1.3 (10.244.1.3) 56(84) bytes of data.
64 bytes from 10.244.1.3: icmp_seq=1 ttl=63 time=1.61 ms
64 bytes from 10.244.1.3: icmp_seq=2 ttl=63 time=3.42 ms
```

Pods - Accessing our pod - from another node

From worker-node2, we can access the pod using the IP we got earlier:

```
[root@worker-node2|/root]# curl 10.244.1.3:9876/info ; echo
{"host": "10.244.1.3:9876", "version": "0.5.0", "from": "10.244.2.0"}
```

Did you notice any difference in the 'curl' output above?

What if? What if we try from the node worker-node1 where it is running?

```
[root@worker-node1|/root]# curl 10.244.1.3:9876/info ; echo
{"host": "10.244.1.3:9876", "version": "0.5.0", "from": "10.244.1.1"}
```

What is the difference? What did you notice? Why is it so?

Pods - deleting our pod

Before deleting the pod, open another putty session and execute the below command. The '-w' is the watch option for and it will keep watching for any changes.

```
[root@k8s-master|/root]# kubectl get pods -o wide -w
simplesrv-7867dfcfb7-4bq4s   1/1       Running   0          1m        10.244.1.3   worker-node1
```

Delete the pod now and watch what happens (please watch the output for about 30-seconds)

```
[root@k8s-master|/root]# kubectl delete pod  simplesrv-7867dfcfb7-4bq4s
Pod "simplesrv-7867dfcfb7-4bq4s" deleted
```

Can you try to explain what just happened?

Pods - deleting our pod

'kubectl run' creates a deployment, and this deployment will make sure that as per its instruction, to keep one pod running. You can keep on deleting the pod and it will notice it and create new pod it. It will have a new IP address and it may be launched on a new node. (more on this later).

You can see this is the below output (check out your watch output)

```
[root@k8s-master|/root]# kubectl get pods -o wide -w
NAME                          READY     STATUS    RESTARTS   AGE       IP           NODE
simplesrv-7867dfcfb7-4bq4s    1/1       Running   0          23m       10.244.1.3   worker-node1
simplesrv-7867dfcfb7-4bq4s    1/1       Terminating   0         25m       10.244.1.3   worker-node1
simplesrv-7867dfcfb7-gpxzf    0/1       Pending   0         0s        <none>    <none>
simplesrv-7867dfcfb7-gpxzf    0/1       Pending   0         0s        <none>    worker-node2
simplesrv-7867dfcfb7-gpxzf    0/1       ContainerCreating   0         0s        <none>    worker-node2
simplesrv-7867dfcfb7-gpxzf    1/1       Running   0         2s        10.244.2.4   worker-node2
simplesrv-7867dfcfb7-4bq4s    0/1       Terminating   0         25m       <none>    worker-node1
simplesrv-7867dfcfb7-4bq4s    0/1       Terminating   0         25m       <none>    worker-node1
simplesrv-7867dfcfb7-4bq4s    0/1       Terminating   0         25m       <none>    worker-node1
```

Pods - so how do I delete this

We use ‘kubectl run’; this created a ‘deployment’. So you will have to delete the ‘deployment’ to get rid of the pod. (while you are delteing keep the ‘watch’ session active.

```
[root@k8s-master|/root]# kubectl get deployment
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
simplesrv   1         1         1            1           37m

[root@k8s-master|/root]# kubectl delete deployment simplesrv
deployment "simplesrv" deleted
```

What is the state? Did it work?

Pod Yaml Manifest

```yaml
apiVersion:
kind:
metadata:




spec:
```

Minimum 4 sections

Pod Yaml Manifest

```
apiVersion:  (the minimum version)
kind:   (type of object we are creating)
metadata:

 (data about the object - like the name and
labels)



spec:
(defines what is inside the object you are
creating. In this case the container image
itself)
```

Minimum 4 sections

Pod Yaml Manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: my-nginx
    tier: front-end
    env: development
spec:
  containers:
  - name: nginx-container
    image: bitnami/nginx
```

```
>> kubectl create -f pod-manifest.yaml
>> kubectl get pods
>> kubectl describe pod my-nginx-pod
```

Pods - Launching the pod using a yaml configuration file

Launch the pod:

```
[root@k8s-master|/root/labs/yamls]# kubectl create -f pod.yaml
pod "twocontainers" created

[root@k8s-master|/root/labs/yamls]# kubectl get pods -o wide
NAME            READY     STATUS    RESTARTS   AGE       IP           NODE
twocontainers   2/2       Running   0          44s       10.244.2.5   worker-node2.podx.example.com
```

Pods - Connecting to the pod?

Can we connect to this pod? Yes we can. Using the ‘kubectl exec’

```
[root@k8s-master|/root]# kubectl exec twocontainers -c shell -i -t -- bash
[root@twocontainers /]#
[root@twocontainers /]#
[root@twocontainers /]#
[root@twocontainers /]# curl -s localhost:9876/info
{"host": "localhost:9876", "version": "0.5.0", "from": "127.0.0.1"}
[root@twocontainers /]#
[root@twocontainers /]#
```

Usage : `kubectl exec POD [-c CONTAINER] -- COMMAND [args...] [options]`

Pods - Checking the CPU and Memory usage of the pods

Find the container ID and connect to the node where the container is running and use ‘docker stats’

```
[root@k8s-master|/root]# kubectl describe pod twocontainers | grep "Container ID:"
    Container ID:   docker://f5254581b881e82a3c68dc1b7dfc707d375db16e6671f403801ed04a295f2137
    Container ID:   docker://bbf68816b762b466e64cde983ac2860d72aab9df57d37674a4297f399c915208

[[
you can select first few characters from the docker ID# as below

>>> This needs to be executed on the node where the pod ‘twocontainers’ is running <<<
]]

[root@worker-node1|/root]# docker stats | egrep '(CONT|f5254581b881|bbf68816b762)'
bbf68816b762        0.00%               92 KiB / 1.953 GiB      0.00%               0 B / 0 B           3.921 MB / 0 B      1
f5254581b881        0.00%               10.04 MiB / 1.953 GiB   0.50%               0 B / 0 B           21.65 MB / 0 B      1
bbf68816b762        0.00%               92 KiB / 1.953 GiB      0.00%               0 B / 0 B           3.921 MB / 0 B      1
f5254581b881        0.00%               10.04 MiB / 1.953 GiB   0.50%               0 B / 0 B           21.65 MB / 0 B      1
bbf68816b762        0.00%               92 KiB / 1.953 GiB      0.00%               0 B / 0 B           3.921 MB / 0 B      1
```

Keep this “docker stats” putty session running

Pods – Load test the Pods and check the results….

Run some quick load test on the ‘twocontainers’ pod and see the load

```
[root@k8s-master|/root]# while true; do curl -s 10.244.1.5:9876/info -O /dev/null ; done
<<use the IP-Address of twocontainers>>
```

Check the CPU and Memory usage of the containers on the respective worker-nodes.

Once you have check the load please use “Ctrl-C” on both `"docker stats"` and current `"while"` putty sessions.

Check the CPU and Memory usage of the containers on the respective worker-nodes.

What happened? What is your observation? What is the load?

Pods - Launching the pod with resource constraint

You can use resource constraint for all the pods you launch. You can limit the amount of CPU and/or Memory allocated to a container in a pod.

```
[root@k8s-master|/root]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# cat constraint-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraintpod
spec:
  containers:
  - name: sise
    image: mhausenblas/simpleservice:0.5.0
    ports:
    - containerPort: 9876
    resources:
      limits:
        memory: "64Mi"
        cpu: "50m"
```

Pods - Launching the pod with resource constraint

Launch the pod:

```
[root@k8s-master|/root/labs/yamls]# kubectl create -f constraint-pod.yaml
pod "constraintpod" created

[root@k8s-master|/root]# kubectl get pods -o wide
NAME            READY     STATUS    RESTARTS   AGE       IP           NODE
constraintpod   1/1       Running   0          6m        10.244.1.5   worker-node1


[root@k8s-master|/root]# kubectl describe pod constraintpod | egrep '(Limits|cpu|memory|Requests)'
 Limits:
      cpu:     50m
      memory:  64Mi
    Requests:
      cpu:        50m
      memory:     64Mi
```

Pods - Launching the pod with resource constraint - Let's validate

Let us check and see if the resource constraint is applied:

Find the container ID and connect to the node where the container is running and use 'docker stats'

```
[root@k8s-master|/root]# kubectl describe pod constraintpod | grep "Container ID:"
 Container ID:   docker://699418033cad34515ffeee586de0ad3680a2d5d68043f032a185f65b45f54039

[root@worker-node1|/root]# docker stats | egrep '(CONT|699418033c)'
CONTAINER                                                           CPU %               MEM USAGE / LIMIT
MEM %               NET I/O             BLOCK I/O           PIDS
699418033cad34515ffeee586de0ad3680a2d5d68043f032a185f65b45f54039   0.00%               10.04 MiB / 64 MiB
15.69%              0 B / 0 B           21.65 MB / 0 B      1
```

Check the official documents to learn more about resource constraints in Kubernetes.

Keep this "docker stats" putty session also running

Pods - Don’t trust it till you test it and see the results for yourself !!

Run some quick load test on the ‘constraintpod’ pod and see the difference.

```
[root@k8s-master|/root]# while true; do curl -s 10.244.2.6:9876/info -O /dev/null ; done
<<use the IP-Address of constraintpod>>
```

Check the CPU and Memory usage of the containers on the respective worker-nodes.

Once you have check the load please use “Ctrl-C” on both `”docker stats”` and current `”while”` putty sessions.

Check the CPU and Memory usage of the containers on the respective worker-nodes.

What happened? What is your observation? Is there a difference in CPU usage of the containers? If so why?

Pods - Can we try deleting `twocontainers` pod?

Try deleting the pod: (keep the watch window active) -- wait again for about 30-seconds.

```
[root@k8s-master|/root]# kubectl delete pod twocontainers
pod "twocontainers" deleted
```

What happened? Did it behave like the earlier delete scenario? Why the difference?

Launching one or more containers (together) in Kubernetes is simple, however doing it directly as shown above comes with a serious limitation: you have to manually take care of keeping them running in case of a failure.

A better way to supervise pods is to use replication controllers, or even better deployments (we saw this sometime earlier), giving you much more control. We will see more of this later.

Pods – Clean up the `constraintpod`

Delete the `constraintpod` using the manifest file.

```
[root@k8s-master|/root/labs/yamls]# kubectl delete -f constraint-pod.yaml

pod "constraintpod" deleted
```

Kubernetes Metrics server

Kubernetes Metrics server

The Metrics server is a cluster-wide aggregator of resource usage data. It currently supports Kubernetes natively and works on all Kubernetes setups. Heapster was the earlier aggregator and is deprecated with Kubernetes v1.11.

It collects metrics like CPU or memory consumption for containers or nodes, from the Summary API, exposed by Kubelet on each node

Long term storage of resource usage data will require Prometheus and to visualize this as charts you will need something like Grafana.

If you want to use k8s' features like **horizontal pod autoscaler** or **"kubectl top"** command you need to install metrics-server.

All these can be configure very easily and requires minimum efforts.

In the lab, we will only install the Metrics server.

Installing the Metrics server components…

Install the Metrics server

```
[root@k8s-master|/root]# cd /root/labs/

[root@k8s-master|/xx]# ls -l metrics-server
-rw-r--r-- 1 root root 4148 Sep 22 12:39 components.yaml
```

https://github.com/kubernetes-incubator/metrics-server

https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/

Installing the Metrics server components…

Install the Metrics server

```
[root@k8s-master| #]# kubectl apply -f metrics-server/

clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
serviceaccount/metrics-server created
deployment.apps/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
```

Installing the Metrics server components…

Check the pod deployment status:

```
[root@k8s-master|/root]# kubectl get pods -n kube-system

NAME                                                READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-6qhf4                            1/1     Running   0          2d10h
coredns-5644d7b6d9-j2f7d                            1/1     Running   0          2d10h
etcd-k8s-master.pod1.example.com                    1/1     Running   0          2d10h
kube-apiserver-k8s-master.pod1.example.com          1/1     Running   0          2d10h
kube-controller-manager-k8s-master.pod1.example.com 1/1     Running   0          2d10h
kube-flannel-ds-amd64-86bg9                         1/1     Running   0          2d10h
kube-flannel-ds-amd64-tq4l4                         1/1     Running   0          2d10h
kube-flannel-ds-amd64-zzbxt                         1/1     Running   0          2d10h
kube-proxy-g8kbx                                    1/1     Running   0          2d10h
kube-proxy-mkqhb                                    1/1     Running   0          2d10h
kube-proxy-w45db                                    1/1     Running   0          2d10h
kube-scheduler-k8s-master.pod1.example.com          1/1     Running   0          2d10h
metrics-server-774f866674-kk4xl                     1/1     Running   0          61s
```

Accessing the Metrics server data

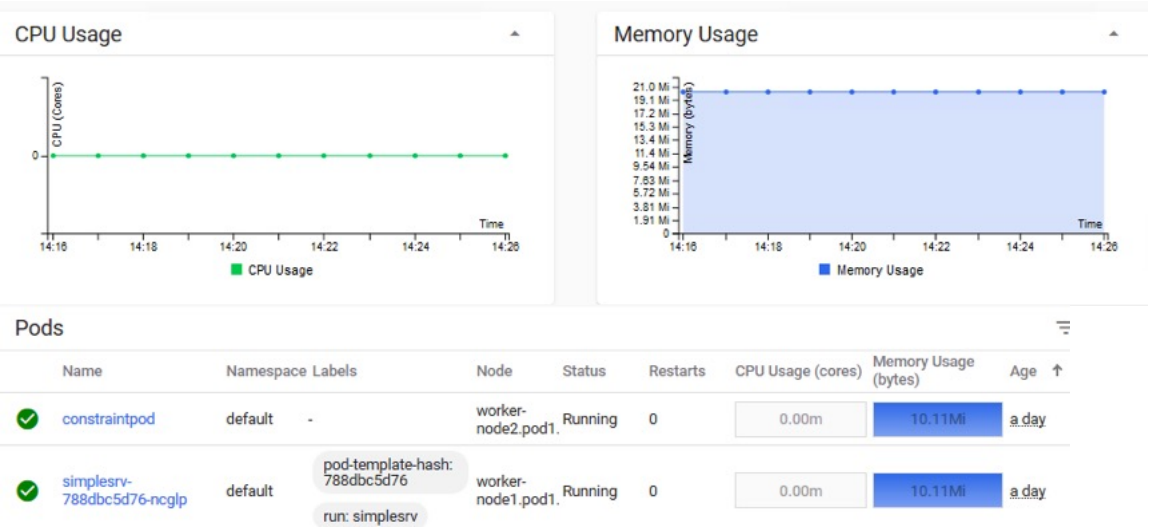
```
[root@k8s-master|/root]# kubectl top nodes
NAME                              CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
k8s-master.pod1.example.com       113m         5%     1226Mi          70%
worker-node1.pod1.example.com     35m          3%     568Mi           32%
worker-node2.pod1.example.com     35m          3%     599Mi           34%

[root@k8s-master|/root]# kubectl top pods -n kube-system
NAME                                                CPU(cores)   MEMORY(bytes)
coredns-5644d7b6d9-6qhf4                            3m           23Mi
coredns-5644d7b6d9-j2f7d                            3m           11Mi
etcd-k8s-master.pod1.example.com                    15m          175Mi
kube-apiserver-k8s-master.pod1.example.com          34m          431Mi
kube-controller-manager-k8s-master.pod1.example.com 14m          70Mi
kube-flannel-ds-amd64-86bg9                         3m           10Mi
kube-flannel-ds-amd64-tq4l4                         3m           9Mi
```

You will have to wait for few minutes for the Metrics data to be collected to see the above output

Checking the Kubernetes Web UI once again…

Go back to the Kubernetes Web UI and explore the Node and Pods; what do you see? Do you see any difference?

Since we installed Metrics server, the Kubernetes Web UI has started pulling resource utilizations from them automatically.



Checking the Kubernetes ‘raw’ Metrics data…

You can call the apiservice via kubectl and access the raw metrics data.

For example to get the basic node metrics, run the below command

```
[[There is a ’.’ at the end of the below command:]]

[root@k8s-master|/root]# kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | jq .

"kind": "NodeMetricsList",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes"
  },
  "items": [
    {
      "metadata": {
        "name": "k8s-master.pod1.example.com",
        "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/k8s-master.pod1.example.com",
        "creationTimestamp": "2019-11-04T09:19:07Z”
.
.
.
.
```

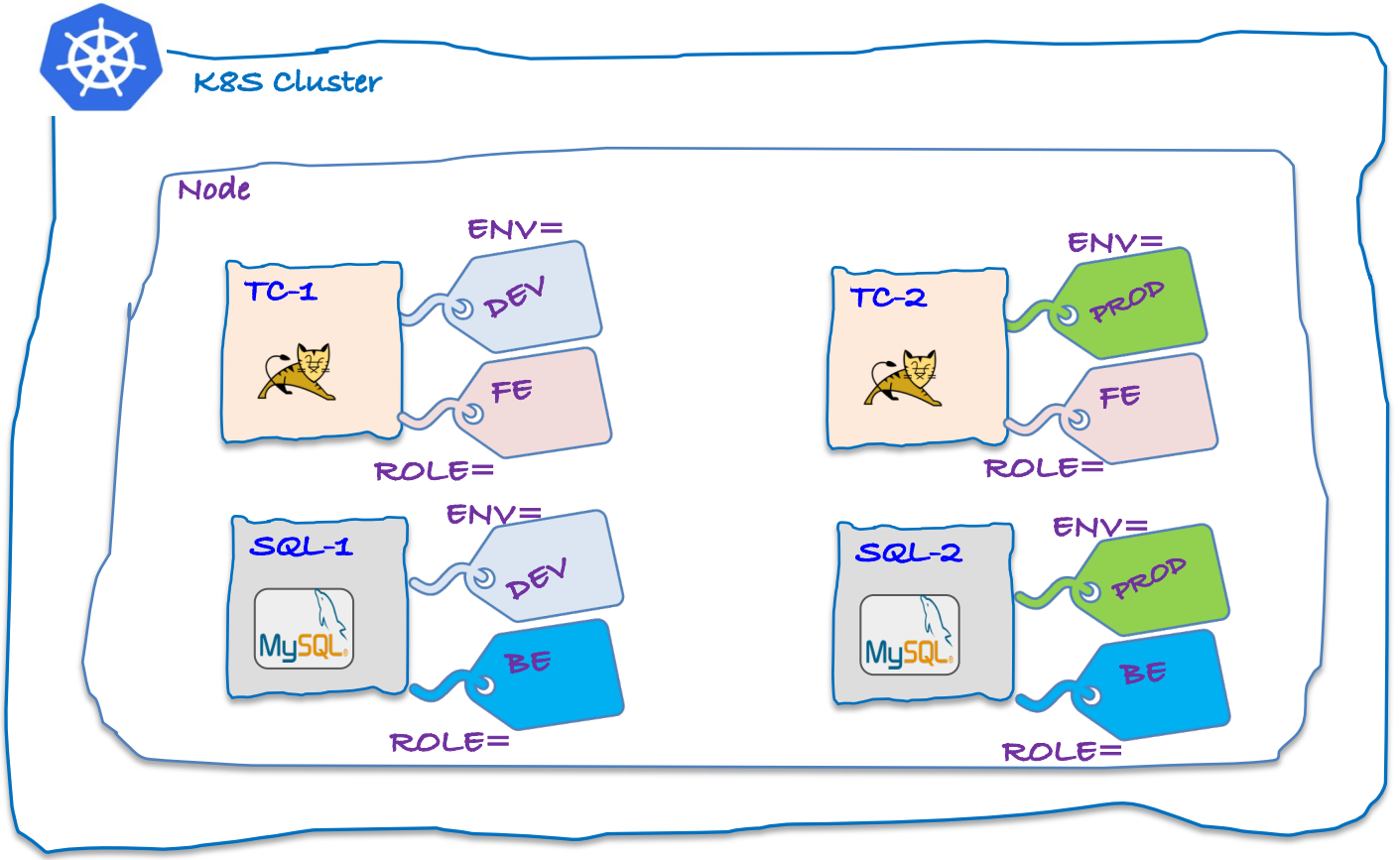
Labels

Labels - Identifying metadata attached to objects

- Metadata
- Attached to any API object
- Generally represent identity
- Can be queried by selectors
- The only grouping mechanism
- Used to determine which objects to apply an operation to
  - pods under a Replication Controller
  - pods in a Service
  - capabilities of a node (scheduling constraints) e.g. Production node or QA node

- **We can attach/apply a label to any object in the cluster including the nodes itself.**

Labels - Identifying metadata attached to objects



Labels

Labels are the mechanism we use to organize Kubernetes objects. A label is a key-value pair

Let's create a pod that initially has one label (env=development):

```
[root@k8s-master|/root]# cd /root/labs/yamls

[root@k8s-master|/root/labs/yamls]# kubectl create -f labels-pod.yaml
pod "labelex" created
```

Check if the pod is running and also display the labels using '--show-labels'

```
[root@k8s-master|/root/labs/yamls]# kubectl get pods --show-labels
NAME      READY     STATUS    RESTARTS   AGE       LABELS
labelex   1/1       Running   0          1m        env=development
```

Note that labels are not restricted to pods. In fact you can apply them to all sorts of objects, such as nodes or services.

Labels - add a new label

You can add a label to the pod as:

```
[root@k8s-master|/root/labs/yamls]# kubectl label pods labelex owner=michael
pod "labelex" labeled

[root@k8s-master|/root/labs/yamls]# kubectl get pods --show-labels
NAME      READY     STATUS    RESTARTS   AGE       LABELS
labelex   1/1       Running   0          2m        env=development,owner=michael
```

To use a label for filtering, for example to list only pods that have an owner that equals michael, use the --selector option:

```
[root@k8s-master|/root/labs/yamls]# kubectl get pods --selector owner=michael
NAME      READY     STATUS    RESTARTS   AGE
labelex   1/1       Running   0          2m
```

Labels - add two labels

You can add multiple labels to the pod as:

```
[root@k8s-master|/root/labs/yamls]# kubectl create -f labels-anotherpod.yaml
pod "labelexother" created

[root@k8s-master|/root/labs/yamls]# kubectl get pods --show-labels
NAME            READY     STATUS    RESTARTS   AGE       LABELS
labelex         1/1       Running   0          7m        env=development,owner=michael
labelexother    1/1       Running   0          42s       env=production,owner=michael
```

Now, let's list all pods that are either labelled with env=development or with env=production:

```
[root@k8s-master|/root/labs/yamls]# kubectl get pods -l 'env in (production, development)'
NAME            READY     STATUS    RESTARTS   AGE
labelex         1/1       Running   0          9m
labelexother    1/1       Running   0          2m
```

Clean up the lab

You can now clean up the lab:

```
[root@k8s-master|/root/labs/yamls]# kubectl delete -f labels-pod.yaml

pod "labelex" deleted

[root@k8s-master|/root/labs/yamls]# kubectl delete -f labels-anotherpod.yaml

pod "labelexother" deleted
```

Replication Controllers

Replication Controllers

A replication controller (RC) is a supervisor for long-running pods. An RC will launch a specified number of pods called replicas and makes sure that they keep running, for example when a node fails or something inside of a pod, like something inside one of its containers goes wrong.

Note that, going forward, the RCs are called replica sets (RS), supporting set-based selectors. The RS are already in use in the context of deployments.

```
[root@k8s-master|/root]# cd /root/labs/yamls


[root@k8s-master|/root/labs/yamls]# cat rc-rc.yaml
.
.
.
```

Replication Controller Yaml Manifest

```yaml
apiVersion:
kind:
metadata:





spec:
```

Replication Controller Yaml Manifest

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
(defines what is inside the object you
are creating - In this case multiple
instances of a Pod)
```

Replication Controller Yaml Manifest

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
```

Replication Controller Yaml Manifest

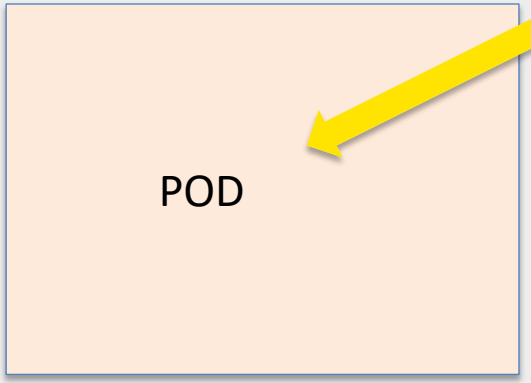
```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
```

POD

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: my-nginx
    tier: front-end
    env: development
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Replication Controller Yaml Manifest

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
```




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: my-nginx
    tier: front-end
    env: development
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Replication Controller Yaml Manifest

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
    metadata:
      name: my-nginx-pod
      labels:
        app: my-nginx
        tier: front-end
        env: development
    spec:
      containers:
      - name: nginx-container
        image: bitnami/nginx
```

Replication Controller Yaml Manifest

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
      metadata:
        name: my-nginx-pod
        labels:
          app: my-nginx
          tier: front-end
          env: development
      spec:
        containers:
        - name: nginx-container
          image: bitnami/nginx
 replicas: 3
```

Now we have two “metadata:” section and two “spec:” section

“replicas: ” tells how many instances/pods to create

Replication Controller Yaml Manifest

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replication-controller
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
     metadata:
       name: my-nginx-pod
       labels:
         app: my-nginx
         tier: front-end
         env: development
     spec:
       containers:
       - name: nginx-container
         image: bitnami/nginx
 replicas: 3
```

```
>> kubectl create -f replication-controller-manifest.yaml
>> kubectl get pods
>> kubectl get replicationcontroller
>> kubectl describe replicationcontroller my-replication-controller
```

Replication Controllers

A replication controller (RC) is a supervisor for long-running pods. An RC will launch a specified number of pods called replicas and makes sure that they keep running, for example when a node fails or something inside of a pod, like something inside one of its containers goes wrong.

Note that, going forward, the RCs are called replica sets (RS), supporting set-based selectors. The RS are already in use in the context of deployments.

```
[root@k8s-master|/root]# cd /root/labs/yamls


[root@k8s-master|/root/labs/yamls]# kubectl create -f rc-rc.yaml
replicationcontroller "rcex" created
```

Replication Controllers - let's check the details

You can check the RC and the pod as below:

```
[root@k8s-master|/root/labs/yamls]# kubectl get rc
NAME      DESIRED   CURRENT   READY     AGE
rcex      1         1         1         2m


[root@k8s-master|/root/labs/yamls]# kubectl get pods --show-labels -o wide -w
NAME         READY     STATUS    RESTARTS   AGE       IP            NODE           LABELS
rcex-vf7bd   1/1       Running   0          6m        10.244.2.10   worker-node2   app=sise
```

Note two things here:
the supervised pod got a random name assigned (rcex-vf7bd)
the way the RC keeps track of its pods is via the label, here app=sise

Replication Controllers - Scaling Up and Down

To scale up, that is, to increase the number of replicas, do:

```
[root@k8s-master|/root/labs/yamls]#    kubectl scale --replicas=3 rc/rcex
replicationcontroller "rcex" scaled


[root@k8s-master|/root/labs/yamls]# kubectl get pods -l app=sise -o wide -w
NAME          READY     STATUS    RESTARTS   AGE       IP            NODE
rcex-dqv4w    1/1       Running   0          59s       10.244.1.10   worker-node1
rcex-r4vx5    1/1       Running   0          59s       10.244.2.11   worker-node2
rcex-vf7bd    1/1       Running   0          9m        10.244.2.10   worker-node2
```

You can try scaling to 5-nodes and then scale down to 2-nodes and check the different outputs. If you are curious, you can try scaling it to 30-nodes and check the CPU and Memory usage across all three nodes and you will be surprised to see that there is hardly any resource bottleneck.

Note: DO NOT use any higher value than 30; else you may crash the k8s cluster and you will not be able to complete the other labs.

Clean up the lab - Replication Controllers - Deleting the RC

To get rid of the RC and the pods it is supervising, use:

```
[root@k8s-master|/root]# kubectl delete rc rcex
replicationcontroller "rcex" deleted
```

Services

Services

- Service is an abstraction for pods, providing a stable, virtual IP (VIP) address.
- Unified method of accessing the exposed workloads of Pods
- Pods may come and go and the IPs will keep changing
- A service (VIP) is a stable address for a pod (or a bunch of pods)
- Services allow clients to reliably connect to the containers running in the pods, using the VIP
- Target Pods using equality based selectors.
- kube-proxy acts as a daemon that creates local entries in the host's iptables for every service
- kube-proxy is responsible to keep the mapping between the VIP and the pods up-to-date
- Uses kube-proxy to provide simple load-balancing.
- Every node has a kube-proxy process (pod) running

Services

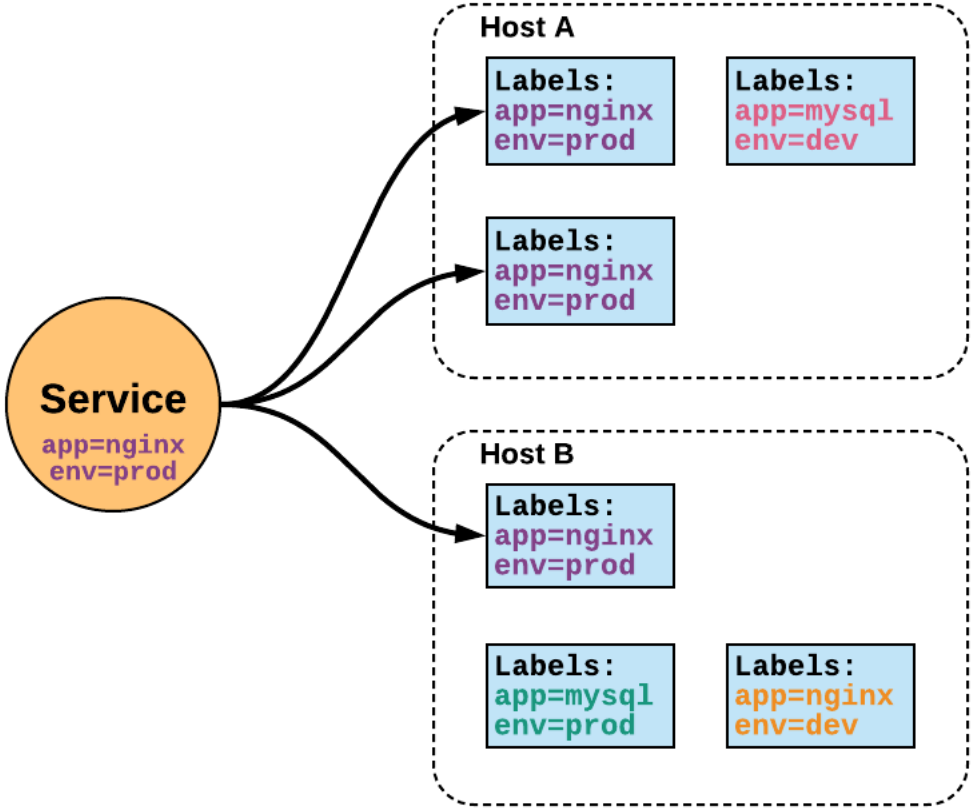


Image source :
https://github.com/mrbobbytables/diagrams

Service Implementation…

A Kubernetes service can be implemented as one of the following types:

ClusterIP: This type will make the service only visible to the internal network for routing internal traffic.(default)

Load Balancer: If services need to be exposed via a dynamic load balancer the service type can be set to Load Balancer. This feature is enabled by the underlying cloud provider (example: GCE).

NodeIP: NodePort services extend the ClusterIP service. This type will expose the service via node ports to the external network. Each port in a service will be mapped to a port and those will be accessible via <node-ip>:<node-port>. It exposes the port on every node in the cluster. Port can either be statically defined, or dynamically taken from a range between 30000-32767.

External Name: External Name is used to reference endpoints OUTSIDE the cluster. It creates an internal CNAME DNS entry that aliases another.

Ingress Router: In true sense, Ingress is actually NOT a type of service. It sits in front of multiple services and act as a “smart router” or entry point into your cluster. We will explore more details of this later.

Service Implementation - **ClusterIP**

A ClusterIP service is the default Kubernetes service. It gives you a service inside your cluster that other apps inside your cluster can access. There is no external access.

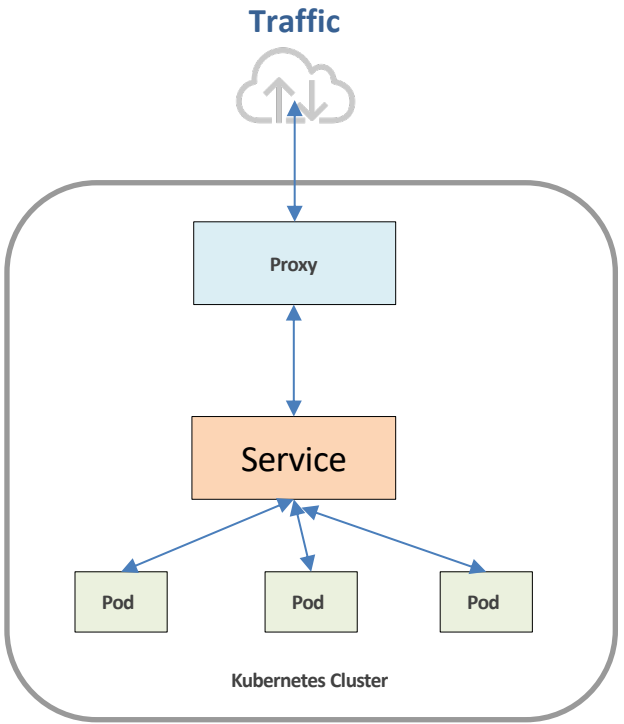
In case you want to access the ClusterIP from outside the cluster, you can access it using the Kubernetes proxy ("**kubectl proxy**") and navigate through the Kubernetes API to access this service using this scheme

http://localhost:8080/api/v1/proxy/namespaces/<NAMESPACE>/services/<SERVICE-NAME>:<PORT-NAME>/

You may use it for debugging your services, or connecting to them directly or to displaying internal dashboards, etc.

Since this method requires you to run kubectl as an authenticated user, you should NOT use this to expose your service to the internet or use it for production services.

Service Implementation - **ClusterIP Simple**



Service Implementation – **ClusterIP Detailed**

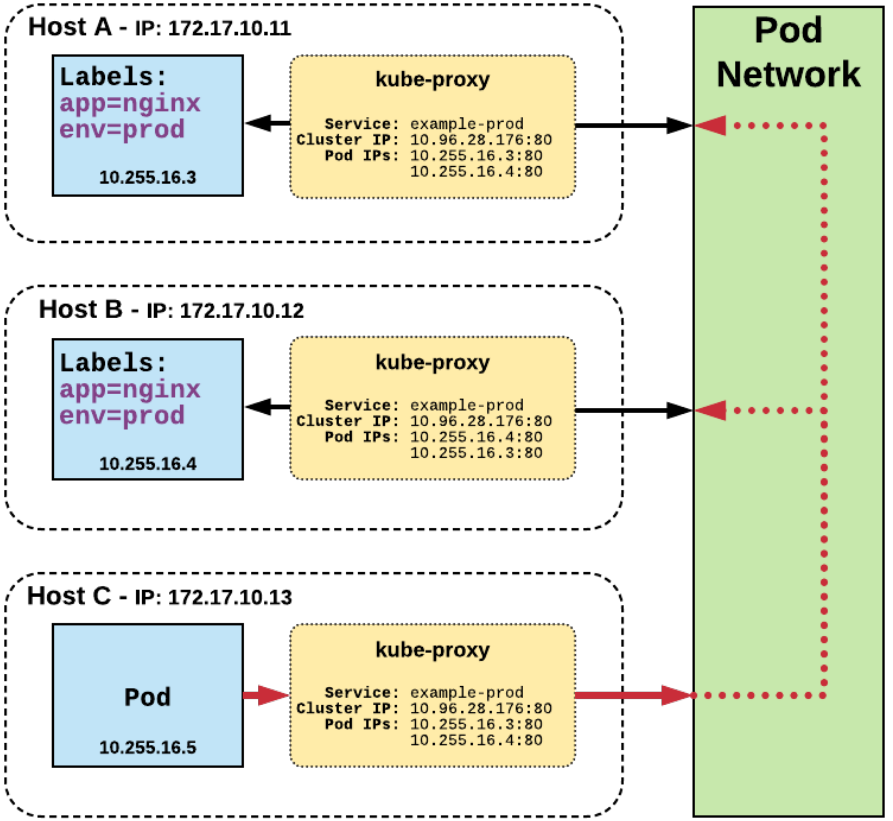


Image source : https://github.com/mrbobbytables/diagrams

Service Implementation - **LoadBalancer**

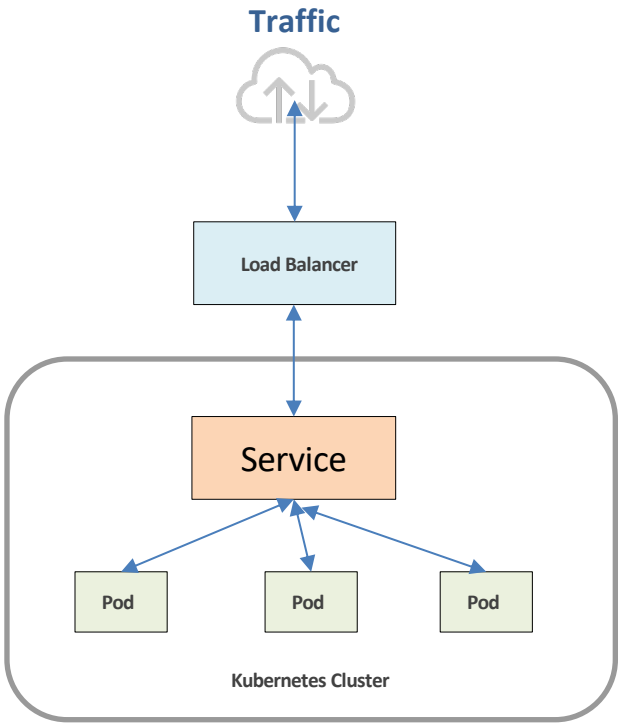
A LoadBalancer service is the standard way to expose a service to the internet. This is typically used by Cloud service providers like AWS, GCP and Azure. We can also implement this by making use of F5 loadbalancer.

On a Cloud provider, this will spin up a Network Load Balancer that will give you a single IP address that will forward all traffic to your service.

This is the best way to directly expose a service to the internet. All traffic on the port you specify will be forwarded to the service. There is no filtering, no routing, etc and you can send almost any kind of traffic to it, like HTTP, TCP, UDP, Websockets, gRPC, etc.

In the long run this will be expensive since each service you expose with a LoadBalancer will get its own IP address and a Cloud service provider LoadBalancer.

Service Implementation – **LoadBalancer Simple**



Service Implementation – **LoadBalancer Detailed**

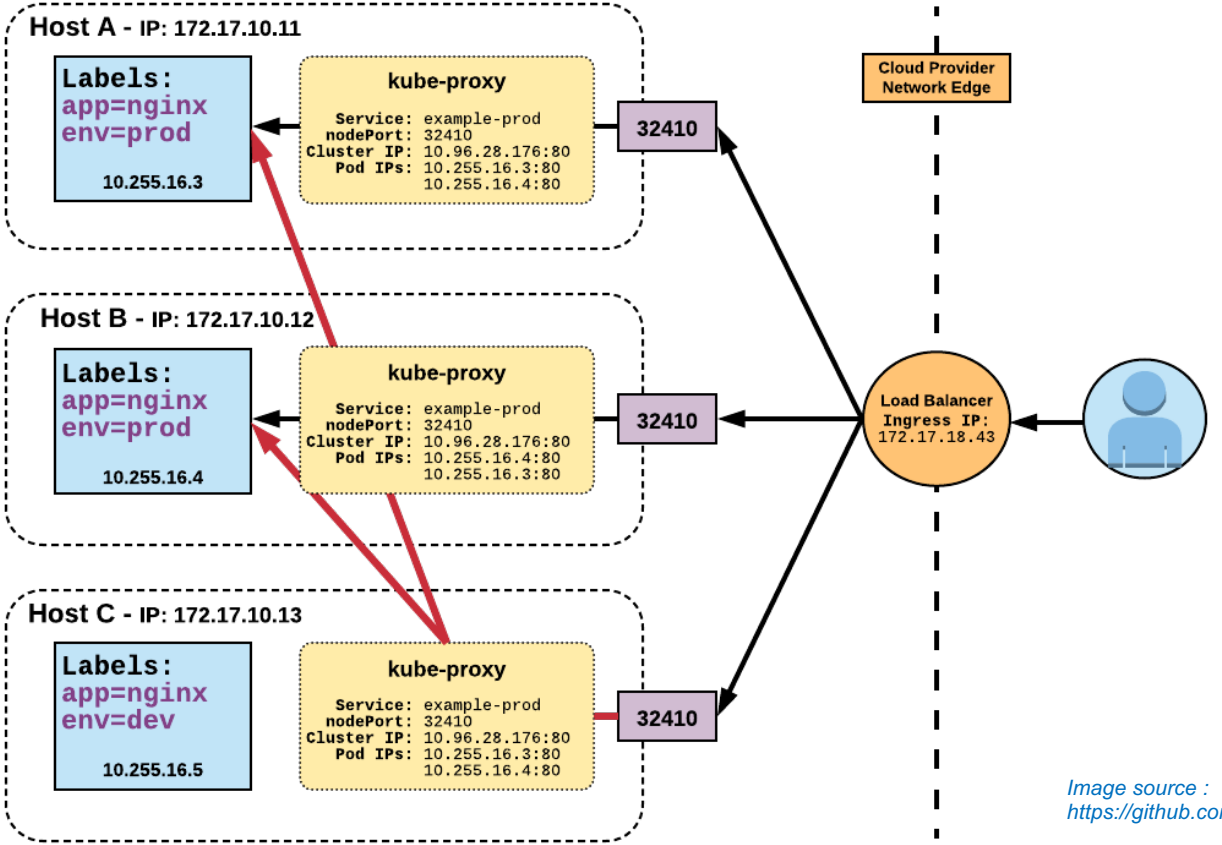


Image source :
https://github.com/mrbobbytables/diagrams

Service Implementation - **NodePort**

A NodePort service is the most basic way to get external traffic directly to your service.

NodePort, as the name implies, opens a specific port on all the Nodes (all the master and work node VMs), and any traffic that is sent to this port is forwarded to the service via the internal network.

Service Implementation – **NodePort - Detailed**

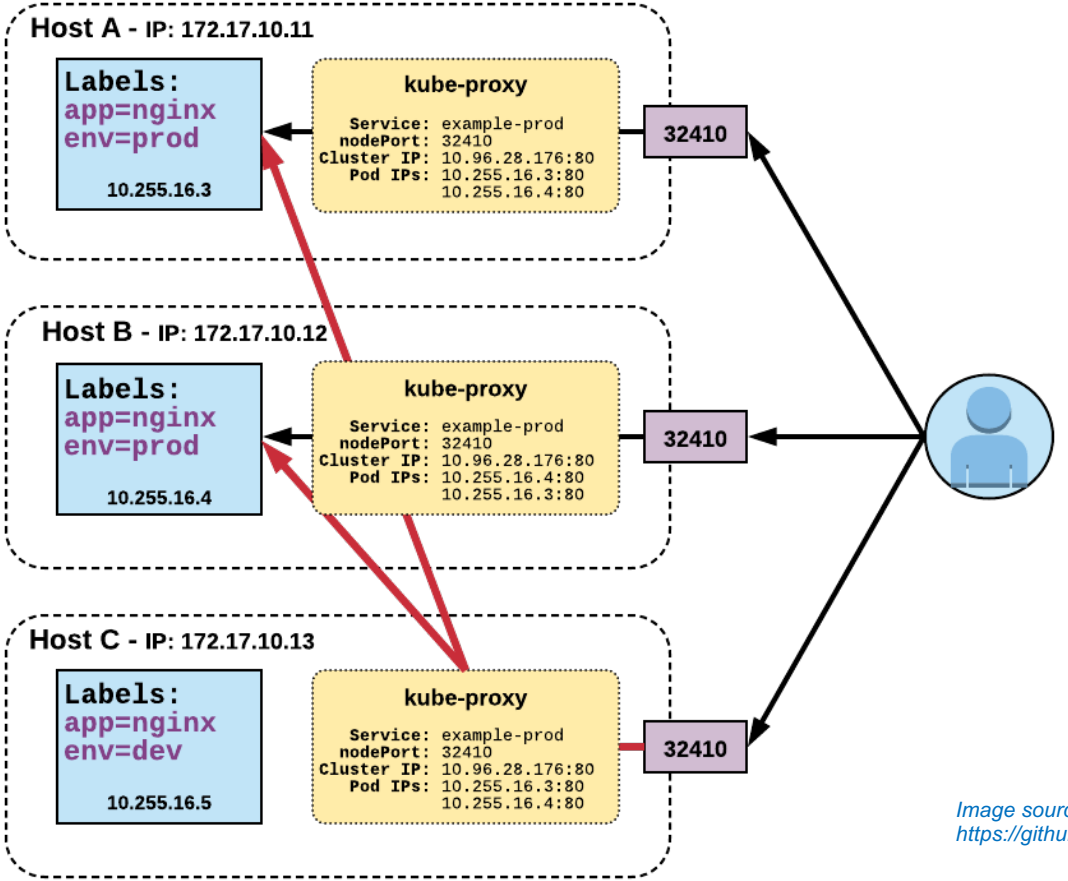


Image source : https://github.com/mrbobbytables/diagrams

Services

Let’s create a pod supervised by an RC and a service along with it:

```
[root@k8s-master|/root]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# ls -l create-*
services-rc.yaml  services-svc.yaml

[root@k8s-master|/root/labs/yamls]# kubectl create -f create-rc-for-service-impl-lab.yaml
replicationcontroller/service-impl-lab created


[root@k8s-master|/root/labs/yamls]# kubectl create -f create-service-for-service-impl-lab.yaml
service/simple-impl-service created
```

Services - checking the service details

Let’s check the details of the pods and access the pod using the internal IP:

```
[root@k8s-master|/root]# kubectl get pods -l app=service-impl-lab -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP            NODE
service-impl-lab-6sj8s   1/1     Running   0          48s   10.244.2.18   worker-node2.pod7.example.com

[root@k8s-master|/root]# kubectl describe pod service-impl-lab-6sj8s
Name:         service-impl-lab-6sj8s
Namespace:    default
Priority:     0
Node:         worker-node2.pod7.example.com/192.168.122.199
Start Time:   Mon, 04 Nov 2019 13:49:11 +0000
Labels:       app=service-impl-lab
Annotations:  <none>
Status:       Running
IP:           10.244.2.18
IPs:
  IP:           10.244.2.18
Controlled By:  ReplicationController/service-impl-lab

[root@k8s-master|/root]# export FIRST_POD_IP=10.244.2.18
[root@k8s-master|/root]# curl -s ${FIRST_POD_IP}:8080 | grep service-impl-lab
<p>This is served from POD : <b> service-impl-lab-6sj8s </b> </p>
```

Services - accessing using the service IP

Earlier method is not advisable since the IPs assigned to pods may change. Let see a better method. First, find the service IP that was assigned to the pod:

```
[root@k8s-master|/root]# kubectl get service
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
simple-impl-service   NodePort    10.105.68.247    <none>        80:31401/TCP   4m22s

[root@k8s-master|/root]# kubectl describe svc simple-impl-service
Name:                     simple-impl-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=service-impl-lab
Type:                     NodePort
IP:                       10.105.68.247
Port:                     <unset>  80/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  31401/TCP
Endpoints:                10.244.2.18:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Services - accessing using the service IP

The service keeps track of the pods it forwards traffic to through the label, in our case app=sise.

From within the cluster we can now access simpleservice like so:

```
[root@k8s]# export CLUSTER_IP=$(kubectl get services/simple-impl-service  -o go-template='{{(index .spec.clusterIP)}}')

[root@k8s]# export NODE_PORT=$(kubectl get services/simple-impl-service  -o go-template='{{(index .spec.ports 0).nodePort}}')

[root@k8s]# curl -s ${CLUSTER_IP}/ | grep service-impl-lab

<p>This is served from POD : <b> service-impl-lab-6sj8s </b> </p>

[root@k8s]# echo ${NODE_PORT}

[root@k8s]# echo ${PODURL}:${NODE_PORT}/

http://k8s-master.podXX.example.com:31401/
```

Use chrome and connect to the URL :: `http://k8s-master.podXX.example.com:31401/`

Services - How does this work?

We make use of IPtables to make the VIP 10.105.68.247 forward the traffic to the pod. There is a long list of long list of rules that tells the Linux kernel what to do with a certain IP package.

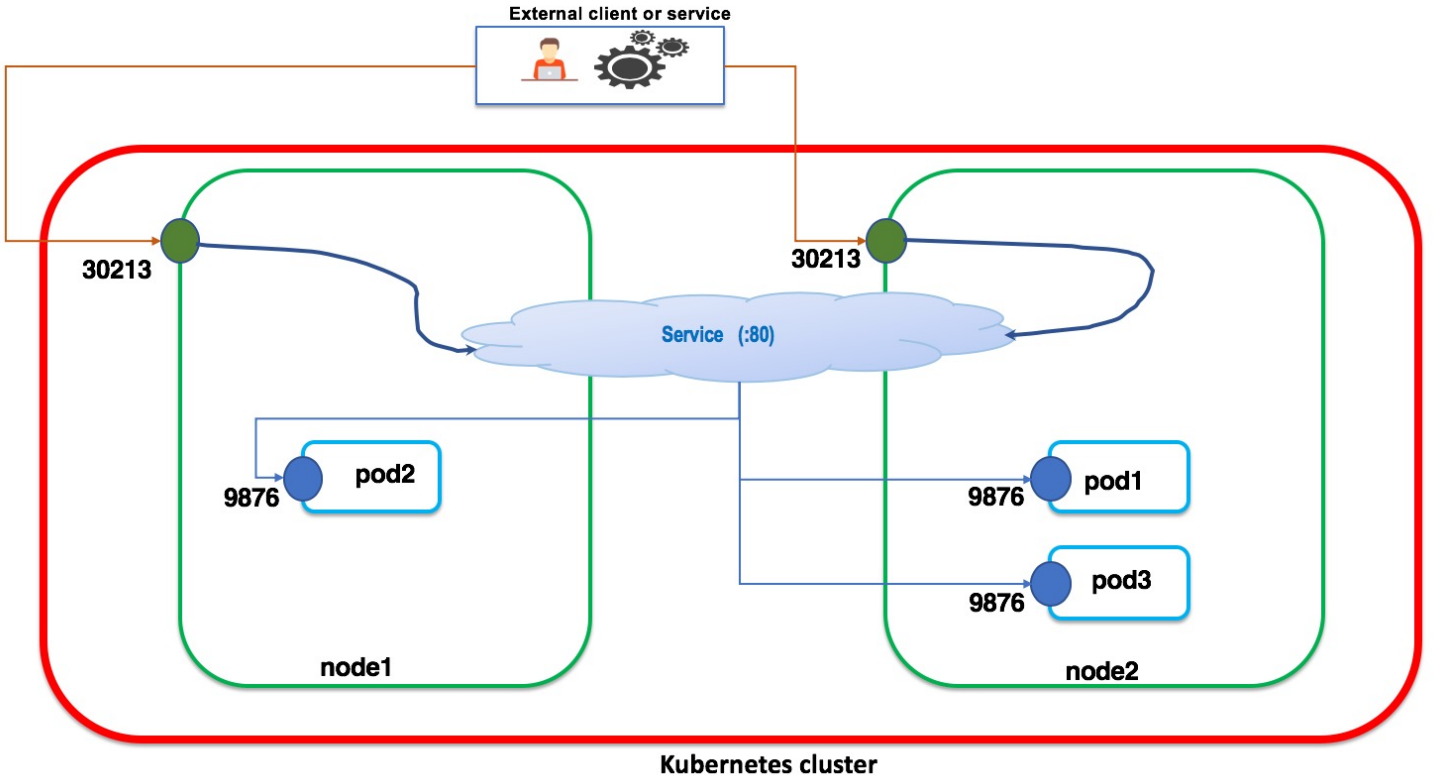
Let's have a look at the rules for our service:

```
[root@k8s-master|/root]# iptables-save | egrep "(simple-impl-service|${FIRST_POD_IP})"
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/simple-impl-service" -m tcp --dport 31401 -j KUBE-SVC-ZCHNKBZ4KJNU6GQE
-A KUBE-SERVICES -d 10.105.68.247/32 -p tcp -m comment --comment "default/simple-impl-service cluster IP" -m tcp --dport 80 -j KUBE-SVC-
ZCHNKBZ4KJNU6GQE
-A KUBE-SVC-ZCHNKBZ4KJNU6GQE -m comment --comment "default/simple-impl-service" -j KUBE-SEP-OIR4GADCMHQ75LY2
-A KUBE-SVC-ZCHNKBZ4KJNU6GQE -p tcp -m comment --comment "default/simple-impl-service" -m tcp --dport 31401 -j KUBE-MARK-MASQ
-A KUBE-SVC-ZCHNKBZ4KJNU6GQE ! -s 10.244.0.0/16 -d 10.105.68.247/32 -p tcp -m comment --comment "default/simple-impl-service cluster IP" -m tcp --
dport 80 -j KUBE-MARK-MASQ
-A KUBE-SEP-OIR4GADCMHQ75LY2 -p tcp -m comment --comment "default/simple-impl-service" -m tcp -j DNAT --to-destination 10.244.2.18:8080
-A KUBE-SEP-OIR4GADCMHQ75LY2 -s 10.244.2.18/32 -m comment --comment "default/simple-impl-service" -j KUBE-MARK-MASQ
```

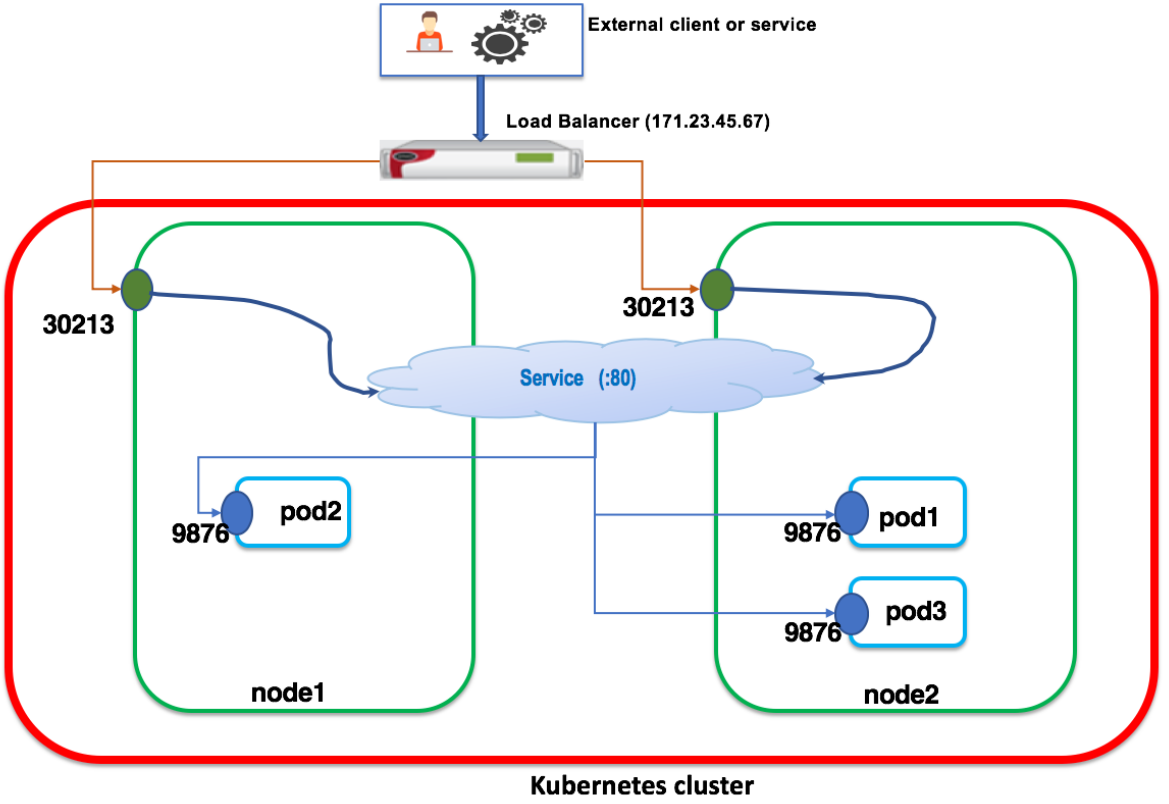
Services - How are the ports mapped and traffic flows?

```
apiVersion: v1
kind: Service
metadata:
  name: simple-impl-service
spec:
  type: NodePort    <<--  Set the service type to NodePort.
  ports:
    - port: 80      <<--  This is the port of the service's internal cluster IP
      targetPort: 9876 <<--  This is the target port of the backing pods.
      nodePort: 30213  <<--  service will be accessible through each of nodes
  selector:
    app: service-impl-lab
```

Services - How are the ports mapped and the traffic flows (For direct connect to NodePort):



Services - How are the ports mapped and traffic flows (Via Load Balancer):



Services - Scaling and checking the service behavior

Let's now add a second pod by scaling up the RC supervising it:

```
[root@k8s-master|/root]# kubectl scale --replicas=3 rc/service-impl-lab
replicationcontroller/service-impl-lab scaled

[root@k8s-master|/root]# kubectl get pods -o wide

[root@k8s-master|/root]# kubectl describe svc simple-impl-service

[root@k8s-master|/root]# echo ${PODURL}:${NODE_PORT}/
```

Do the following:

- Check the iptables rules again; what do you see?
- Check the output of **`kubectl describe svc simpleservice`**; what do you see?

Use chrome and connect to the URL above **(output of echo above)** to access service and keep refreshing:

Or use from k8s-master :::

```
while [ 1 ]; do curl -s ${CLUSTER_IP}|grep service-impl-lab ; echo ; sleep 1; done
```

Services - accessing using the service IP

Check the service IP after scaling to three pods:

```
[root@k8s-master|/root]# kubectl get service
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
simple-impl-service   NodePort    10.105.68.247    <none>        80:31401/TCP   25m

[root@k8s-master|/root]# kubectl describe svc simple-impl-service
Name:                     simple-impl-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=service-impl-lab
Type:                     NodePort
IP:                       10.105.68.247
Port:                     <unset>  80/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  31401/TCP
Endpoints:                10.244.1.21:8080,10.244.1.22:8080,10.244.2.18:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Clean up the lab

You can now clean up the lab:

```
[root@k8s-master|/root]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# kubectl delete -f create-rc-for-service-impl-lab.yaml
replicationcontroller/service-impl-lab deleted


[root@k8s-master|/root/labs/yamls]# kubectl delete -f create-service-for-service-impl-lab.yaml
service/simple-impl-service deleted
```

Ingress

Ingress

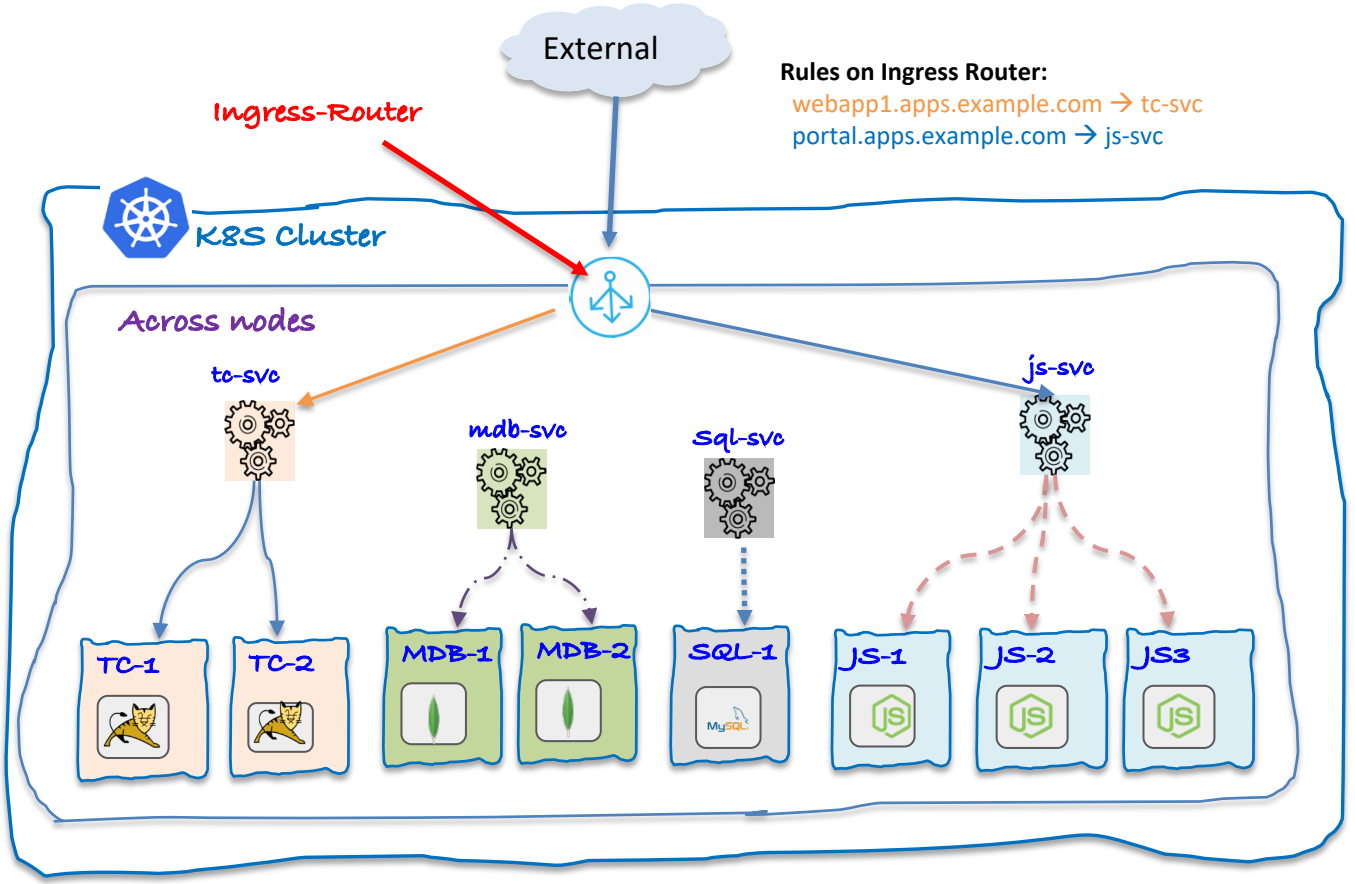
An Ingress is a collection of rules that allow inbound connections to reach the cluster services. It can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, offer name based virtual hosting, and more.

Ingress exposes ~~HTTP~~ and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

Whenever you want to expose your applications that reside in the k8s cluster, you configure an ingress resource that defines where the inbound traffic will be go.

Ingress is a powerful way to expose the services. There are many type of Ingress controllers like Load balancers from Cloud service providers, Nginx, Contour, Istio, Traefik and more. We will see an implementation using Nginx in our lab.

Ingress


External
Rules on Ingress Router:
webapp1.apps.example.com → tc-svc
portal.apps.example.com → js-svc
Ingress-Router
K8S Cluster
Across nodes
tc-svc
mdb-svc
sql-svc
js-svc
TC-1
TC-2
MDB-1
MDB-2
SQL-1
JS-1
JS-2
JS3

Setting up Ingress service

Now, we will setup an Ingress controller using Nginx.

Check the link below for complete details:

```
https://docs.nginx.com/nginx-ingress-controller/installation/installation-with-manifests/
https://github.com/nginxinc/kubernetes-ingress
```

Setting up Ingress service

Step 1: Configure RBAC

```
Make sure that you are in the correct directory

[root@k8s-master|]# cd /root/labs/kubernetes-ingress/deployments



Create a namespace and a service account for the Ingress controller:

[root@k8s-master|]# kubectl apply -f common/ns-and-sa.yaml


Create a cluster role and cluster role binding for the service account:

[root@k8s-master|]# kubectl apply -f rbac/rbac.yaml
```

Setting up Ingress service

Step 2: Create Common Resources

```
Create a secret with a TLS certificate and a key for the default server in NGINX:

$ kubectl apply -f common/default-server-secret.yaml

Create a config map for customizing NGINX configuration:

$ kubectl apply -f common/nginx-config.yaml


Create an IngressClass resource (for Kubernetes >= 1.18):

$ kubectl apply -f common/ingress-class.yaml
```

Setting up Ingress service

Step 3: Create Custom Resources

```
Create custom resource definitions for VirtualServer and VirtualServerRoute, TransportServer and Policy
resources:

$ kubectl apply -f common/crds/k8s.nginx.org_virtualservers.yaml
$ kubectl apply -f common/crds/k8s.nginx.org_virtualserverroutes.yaml
$ kubectl apply -f common/crds/k8s.nginx.org_transportservers.yaml
$ kubectl apply -f common/crds/k8s.nginx.org_policies.yaml


Create a custom resource definition for GlobalConfiguration resource:

$ kubectl apply -f common/crds/k8s.nginx.org_globalconfigurations.yaml
```

Setting up Ingress service

Step 4: Deploy the Ingress Controller using DaemonSet

```
Use a DaemonSet: When you run the Ingress Controller by using a DaemonSet, Kubernetes will create an Ingress
controller pod on every node of the cluster

$ kubectl apply -f daemon-set/nginx-ingress.yaml


Run the following command to make sure that the Ingress controller pods are running:

$ kubectl get pods --namespace=nginx-ingress
```

Testing up Ingress feature using a webserver

We can quickly launch a Nginx webserver and see if the Ingress controller is working fine.

```
[root@k8s-master|]# cd /root/labs/nginx-webserver-with-ingress

[root@k8s-master|]# kubectl apply -f nginx-webserver-deployment.yaml
deployment.apps/nginx-dm created
service/nginx-svc created

[root@k8s-master|]# kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
nginx-dm-74db7f4c-7w9r7       1/1     Running   0          6s
nginx-dm-74db7f4c-gmtkx       1/1     Running   0          6s
nginx-dm-74db7f4c-wq6n2       1/1     Running   0          6s

[root@k8s-master|]# kubectl get service

NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
nginx-svc    ClusterIP   10.103.78.253   <none>        80/TCP    62s

[root@k8s-master|]# curl http://10.103.78.253
```

Testing up Ingress feature using a webserver

Setup the ingress rules so that the controller can route it to the ”nginx-svc” and we will see the webpage

```
[root@k8s-master|]# cat ingress-for-nginx-webserver.yaml

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-webserver-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: "nginx.podXX.example.com"
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: nginx-svc
            port:
              number: 80
```

Testing up Ingress feature using a webserver

Setup the ingress rules so that the controller can route it to the”nginx-svc” and we will see the webpage

```
[root@k8s-master|]# kubectl apply -f ingress-for-nginx-webserver.yaml
ingress.networking.k8s.io/nginx-webserver-ingress created

[root@k8s-master|]# kubectl get ingress
NAME            HOSTS                          ADDRESS   PORTS   AGE
nginx-ingress   nginx.apps.podX.example.com              80      2m51s


[root@k8s-master|]# curl http://nginx.apps.podX.example.com


Now, use you browser and try accessing the site: http://nginx.apps.podX.example.com

Make sure that the ”podX” is replaced by you pod number.
```

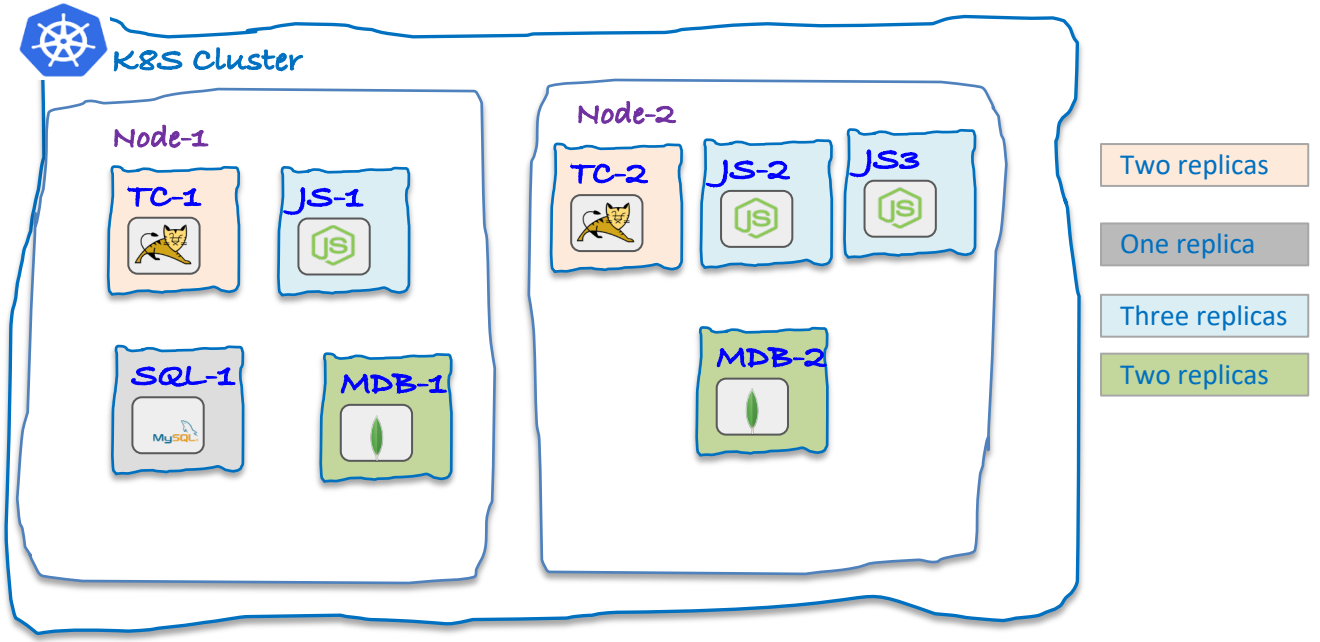
What happens if we hit a Route that is not defined..

```
[root@k8s-master|]# curl http://hithere.apps.podX.example.com


Now, use you browser and try accessing the site: http://hithere.apps.podX.example.com

Make sure that the ”podX” is replaced by you pod number.
```

Deployments

Deployment / Replica Sets

A deployment is a supervisor for pods and replica sets, giving you fine-grained control over how an application is deployed, how many instances and when a new version is rolled out as well as rolled back to a previous state. Each iteration creates a unique label that is assigned to both the Replica Set and subsequent Pods. It is Kubernetes's job to ensure that your cluster will host the replicas of your Pods at any given time and **always ensure the desired number of pods are running**.



RepliaSet Yaml Manifest

```yaml
apiVersion:
kind:
metadata:





spec:
```

RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
```

(defines what is inside the object you are creating - In this case multiple instances of a Pod and the selectors)

RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
```

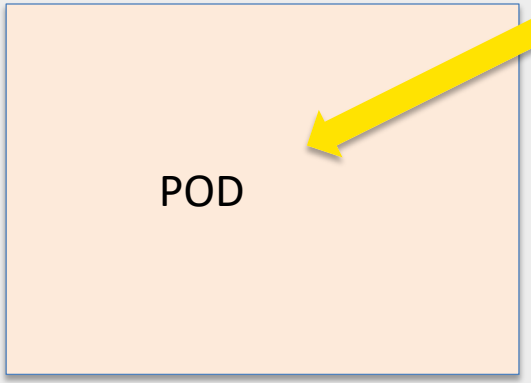
RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: my-nginx
    tier: front-end
    env: development
spec:
  containers:
  - name: nginx-container
    image: nginx
```

RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
```




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: my-nginx
    tier: front-end
    env: development
spec:
  containers:
  - name: nginx-container
    image: nginx
```

RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
    metadata:
      name: my-nginx-pod
      labels:
        app: my-nginx
        tier: front-end
        env: development
    spec:
      containers:
      - name: nginx-container
        image: bitnami/nginx
```

RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
     metadata:
       name: my-nginx-pod
       labels:
         app: my-nginx
         tier: front-end
         env: development
     spec:
       containers:
       - name: nginx-container
         image: bitnami/nginx
 replicas: 3
```

Up to this step, it is same as ReplicationController. Let us see what is new in ReplicaSet

RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
     metadata:
       name: my-nginx-pod
       labels:
         app: my-nginx
         tier: front-end
         env: development
     spec:
       containers:
       - name: nginx-container
         image: bitnami/nginx
 replicas: 3
 selector:
```

“selector: ” Tells what Pods fall under it.

Why do you need it when we have all the labels under the definition already?

The main difference between RC and RS is that RS provides complex set-based selection which is not possible in RC.

RepliaSet Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
     metadata:
       name: my-nginx-pod
       labels:
         app: my-nginx
         tier: front-end
         env: development
     spec:
       containers:
       - name: nginx-container
         image: bitnami/nginx
 replicas: 3
 selector:
    matchLabels:
      app: my-nginx-rc
      tier: front-end
      env: development
```

```
>> kubectl create -f replicaset-manifest.yaml
>> kubectl get pods
>> kubectl get replicaset
>> kubectl describe replicaset my-replicaset
>> kubectl scale --replicas=5 replicasset my-replicaset
```

Deployment Yaml Manifest

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
     metadata:
       name: my-nginx-pod
       labels:
         app: my-nginx
         tier: front-end
         env: development
     spec:
       containers:
       - name: nginx-container
         image: bitnami/nginx
 replicas: 3
 selector:
    matchLabels:
      app: my-nginx-rc
      tier: front-end
      env: development
```

We will start with ReplicaSet manifest file and you will see that the Deployment Yaml manifest is like “ReplicaSet” other than the “kind: “ which is now “Deployment”

Deployment Yaml Manifest

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: my-nginx-rc
    tier: front-end
    env: development
spec:
  template:
     metadata:
       name: my-nginx-pod
       labels:
         app: my-nginx
         tier: front-end
         env: development
     spec:
       containers:
       - name: nginx-container
         image: bitnami/nginx
 replicas: 3
 selector:
    matchLabels:
      app: my-nginx-rc
      tier: front-end
      env: development
```

```
>> kubectl create -f deployment-manifest.yaml
>> kubectl get pods
>> kubectl get deployments
>> kubectl get replicaset
>> kubectl describe deployment my-deployment
```

Deployments

A deployment is a supervisor for pods and replica sets, giving you fine-grained control over how and when a new pod version is rolled out as well as rolled back to a previous state.

Let’s create a deployment called sise-deploy that supervises two replicas of a pod as well as a replica set:

```
[root@k8s-master|/root]# cd /root/labs/yamls

[root@k8s-master|/root/labs/yamls]# ls -l deployment-simple-webapp-v1.yaml
-rw-r--r-- 1 root root 402 Jul 18 04:12 deployment-simple-webapp-v1.yaml


[root@k8s-master|/root/labs/yamls]# cat deployment-simple-webapp-v1.yaml


[root@k8s-master|/root/labs/yamls]# kubectl create -f deployment-simple-webapp-v1.yaml
deployment "simpleservice-deploy" created
```

Deployments

You can see the deployment, the replica set and the pods as below::

```
[root@k8s-master|/root]# kubectl get deploy
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
simpleservice-deploy   2         2         2            2           2m


[root@k8s-master|/root]# kubectl get rs
NAME                              DESIRED   CURRENT   READY     AGE
simpleservice-deploy-65d86896cd   2         2         2         56s

[root@k8s-master|/root]# kubectl get pods -o wide
NAME                                   READY     STATUS    RESTARTS   AGE       IP           NODE
simpleservice-deploy-65d86896cd-5snrv   1/1       Running   0          1m        10.244.2.6   worker-node2.lab.com
simpleservice-deploy-65d86896cd-zgxjq   1/1       Running   0          1m        10.244.1.3   worker-node1.lab.com
```

Note: Naming of the pods and replica set, derived from the deployment name. Also, please make a note of the IPs. We will need it shortly.

Deployments - connecting to the service

At this point, sise containers running in the pods are configured to return the version 0.9. Let's verify that from within the cluster (using kubectl describe first to get the IP of one of the pods):

```
[root@k8s-master|/root]# curl 10.244.1.3:8080

[root@k8s-master|/root]# curl -s 10.244.1.3:8080 | grep "div class="

<div class="box"><h1>v1</h1><h2></h2>
```

You can check the other IP address also

Deployments – exposing the service and accessing it from outside

Let’s expose the service and access it from outside the Cluster.

```
[root@k8s-master]# echo $PODURL
http://k8s-master.pod0.example.com

[root@k8s-master]# kubectl expose  deployment simpleservice-deploy --type=NodePort --name=webapp

[root@k8s-master]# echo $PODURL
http://k8s-master.pod0.example.com

[root@k8s-master]# kubectl get svc
http://k8s-master.pod0.example.com
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
webapp       NodePort    10.106.198.22   <none>        8080:31543/TCP   8s

[root@k8s-master]# export NODE_PORT=$(kubectl get services/webapp  -o go-template='{{(index .spec.ports 0).nodePort}}')

[root@k8s-master]# echo ${PODURL}:${NODE_PORT}/
http://k8s-master.pod0.example.com:31543/

[[
-- use you browser and check the application details
-- it will take sometime for the page to be displayed
-- What is the Version and the Background color you see
]]
```

Deployments - Verify your deployment

Deployments - updating the version from v1.0 to v2.0

Let's now see what happens if we change that version to 2.0 in an updated deployment:

Start a new Putty session and execute the below so that we can keep monitoring the pod status:

```
[root@k8s-master|/root]# kubectl get pods -o wide -w
simpleservice-deploy-65d86896cd-5snrv   1/1       Running   0          29m       10.244.2.6   worker-node2.lab.com
simpleservice-deploy-65d86896cd-zgxjq   1/1       Running   0          29m       10.244.1.3   worker-node1.lab.com
```

Apply the new deployment: (ignore the warning)

```
[root@k8s-master|/root/labs/yamls]# kubectl apply -f deployment-simple-webapp-v2.yaml

Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl
apply
deployment "simpleservice-deploy" configured
```

Deployments - output of ‘`kubectl get pods -o wide -w`’

What we now see is the rollout of two new pods with the updated version 2.0 as well as the two old pods with version 1.0 being terminated:

```
[root@k8s-master|/root]# kubectl get pods -o wide -w
simpleservice-deploy-6874bf8978-jpfhn   0/1       Pending   0         0s        <none>    <none>
simpleservice-deploy-6874bf8978-jpfhn   0/1       Pending   0         0s        <none>    worker-node2.lab.com
simpleservice-deploy-6874bf8978-jpfhn   0/1       ContainerCreating   0         0s        <none>    worker-node2.lab.com
simpleservice-deploy-6874bf8978-jpfhn   1/1       Running   0         2s        10.244.2.7   worker-node2.lab.com
simpleservice-deploy-65d86896cd-5snrv   1/1       Terminating   0         30m       10.244.2.6   worker-node2.lab.com
simpleservice-deploy-6874bf8978-k8bvl   0/1       Pending   0         0s        <none>    <none>
simpleservice-deploy-6874bf8978-k8bvl   0/1       Pending   0         0s        <none>    worker-node1.lab.com
simpleservice-deploy-6874bf8978-k8bvl   0/1       ContainerCreating   0         0s        <none>    worker-node1.lab.com
simpleservice-deploy-6874bf8978-k8bvl   1/1       Running   0         1s        10.244.1.4   worker-node1.lab.com
simpleservice-deploy-65d86896cd-zgxjq   1/1       Terminating   0         30m       10.244.1.3   worker-node1.lab.com
simpleservice-deploy-65d86896cd-5snrv   0/1       Terminating   0         30m       <none>    worker-node2.lab.com
simpleservice-deploy-65d86896cd-zgxjq   0/1       Terminating   0         30m       <none>    worker-node1.lab.com
simpleservice-deploy-65d86896cd-zgxjq   0/1       Terminating   0         30m       <none>    worker-node1.lab.com
simpleservice-deploy-65d86896cd-zgxjq   0/1       Terminating   0         30m       <none>    worker-node1.lab.com
simpleservice-deploy-65d86896cd-5snrv   0/1       Terminating   0         30m       <none>    worker-node2.lab.com
simpleservice-deploy-65d86896cd-5snrv   0/1       Terminating   0         30m       <none>    worker-node2.lab.com
```

Deployments - output of ‘`kubectl get rs`’

Also, a new replica set has been created by the deployment:

```
[root@k8s-master|/root]# kubectl get rs

NAME                                DESIRED   CURRENT   READY     AGE
simpleservice-deploy-65d86896cd     0         0         0         32m
simpleservice-deploy-6874bf8978     2         2         2         1m
```

Deployments - Checking the deployment status and history

During the deployment you can check the progress using below command:

```
[root@k8s-master|/root]# kubectl rollout status deploy/simpleservice-deploy

deployment "simpleservice-deploy" successfully rolled out
```

Similarly, you can get history of all deployments via:

```
[root@k8s-master|/root]# kubectl rollout history deploy/simpleservice-deploy
deployments "sise-deploy”
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

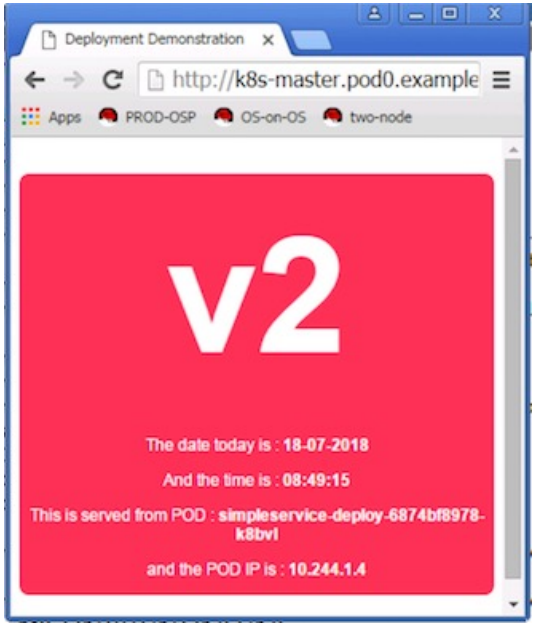
Deployments - Verify you deployment

To verify that if the new 1.0 version is really available, we execute from within the cluster (again using the IP of one of the pods from the new pods):

```
[root@k8s-master|/root]# curl -s 10.244.2.7:8080 | grep "div class="

<div class="box"><h1>v2</h1><h2></h2>

[root@k8s-master]# export NODE_PORT=$(kubectl get services/webapp  -o go-template='{{(index .spec.ports 0).nodePort}}')

[root@k8s-master]# echo ${PODURL}:${NODE_PORT}/
http://k8s-master.pod0.example.com:31543/

[[
-- use you browser and check the application details
-- it will take sometime for the page to be displayed
-- What is the Version and the Background color you see
]]
```

If there are any problems in the deployment Kubernetes new version, the deployment will automatically roll back to the previous version.

Deployments - Verify you deployment



Deployments - Manual roll back

You can also explicitly roll back to a specific revision, as in our case to revision 1 (the original pod version that returns **v1**) as below:

```
kubectl rollout undo deploy/simpleservice-deploy --to-revision=1
deployment "simpleservice-deploy" rolled back

[root@k8s-master|/root]# kubectl rollout history deploy/simpleservice-deploy
deployments "sise-deploy"
REVISION  CHANGE-CAUSE
2         <none>
3         <none>

[root@k8s-master|/root]# kubectl get pods -o wide -w
NAME                          READY     STATUS        RESTARTS   AGE       IP            NODE
simpleservice-deploy-6874bf8978-k8bvl   1/1       Terminating   0         10m       10.244.1.4   worker-node1.lab.com
simpleservice-deploy-6874bf8978-jpfhn   1/1       Terminating   0         10m       10.244.2.7   worker-node2.lab.com
simpleservice-deploy-65d86896cd-9t2lk   1/1       Running   0          51s       10.244.2.8   worker-node2.lab.com
simpleservice-deploy-65d86896cd-swgkf   1/1       Running   0          49s       10.244.1.5   worker-node1.lab.com

[[
-- use you browser and check the application details
-- it will take sometime for the page to be displayed
-- What is the Version and the Background color you see
]]
```

Deployments - Clean up…

To clean up, as we have seen earlier, we will remove the deployment and this in turn it will remove all the the replica sets and pods it supervises:

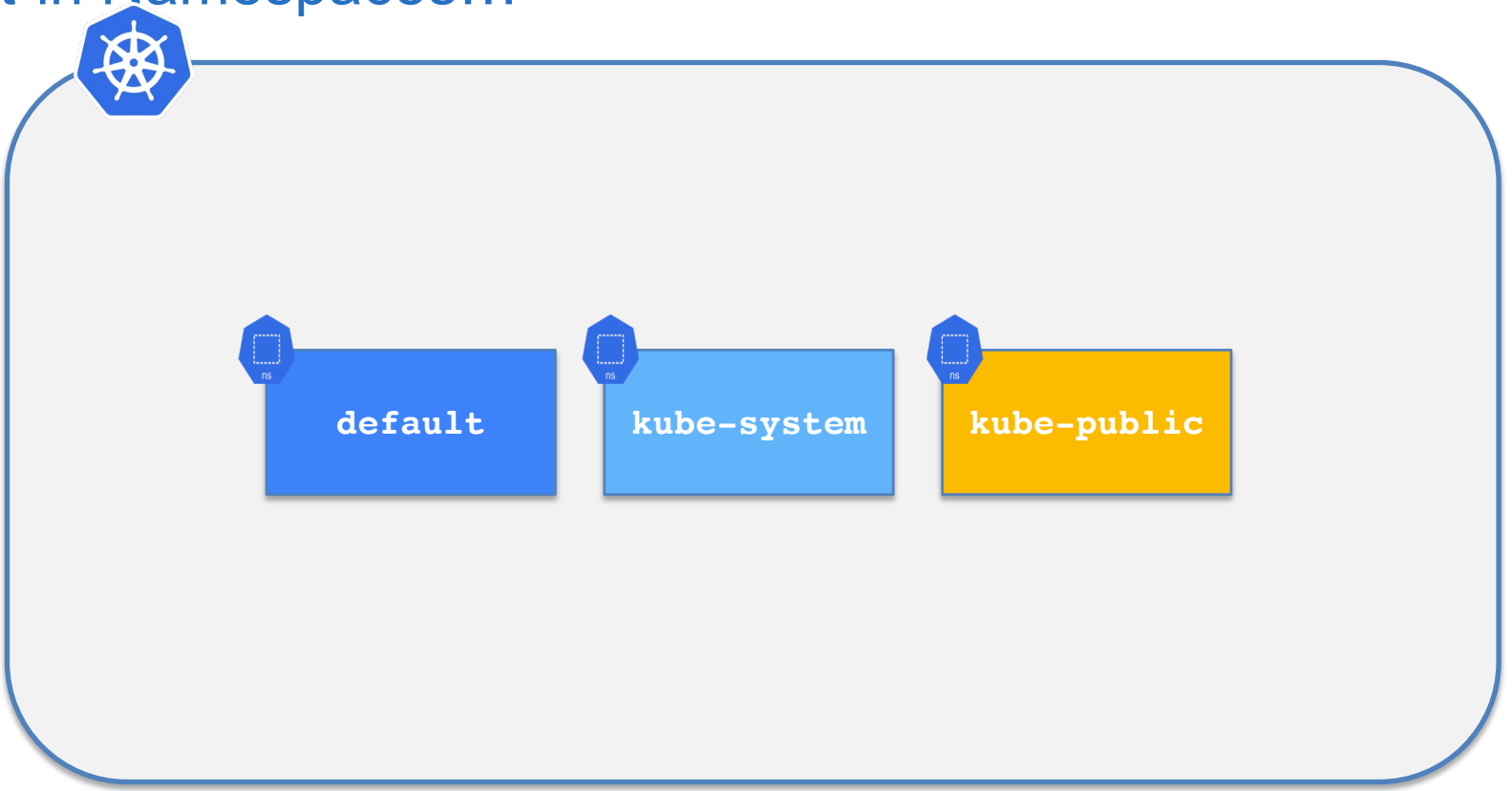
```
[root@k8s-master|/root]# kubectl delete deploy simpleservice-deploy

deployment "sise-deploy" deleted
```

Namespaces

Namespaces…

- Namespaces provide for a scope of Kubernetes objects.
- Many objects such as pods and services are isolated in a namespace, while some (like nodes, PV) are not
- When a Kubernetes cluster is initialized, it has three namespace: kube-system, kube-public and default
- You can have multiple namespaces inside a single Kubernetes cluster
- All namespaces are logically isolated from each other
- Namespace help you organize your projects and application and apply access, security and quotas across them

Built-in Namespaces…



Built-in Namespaces…

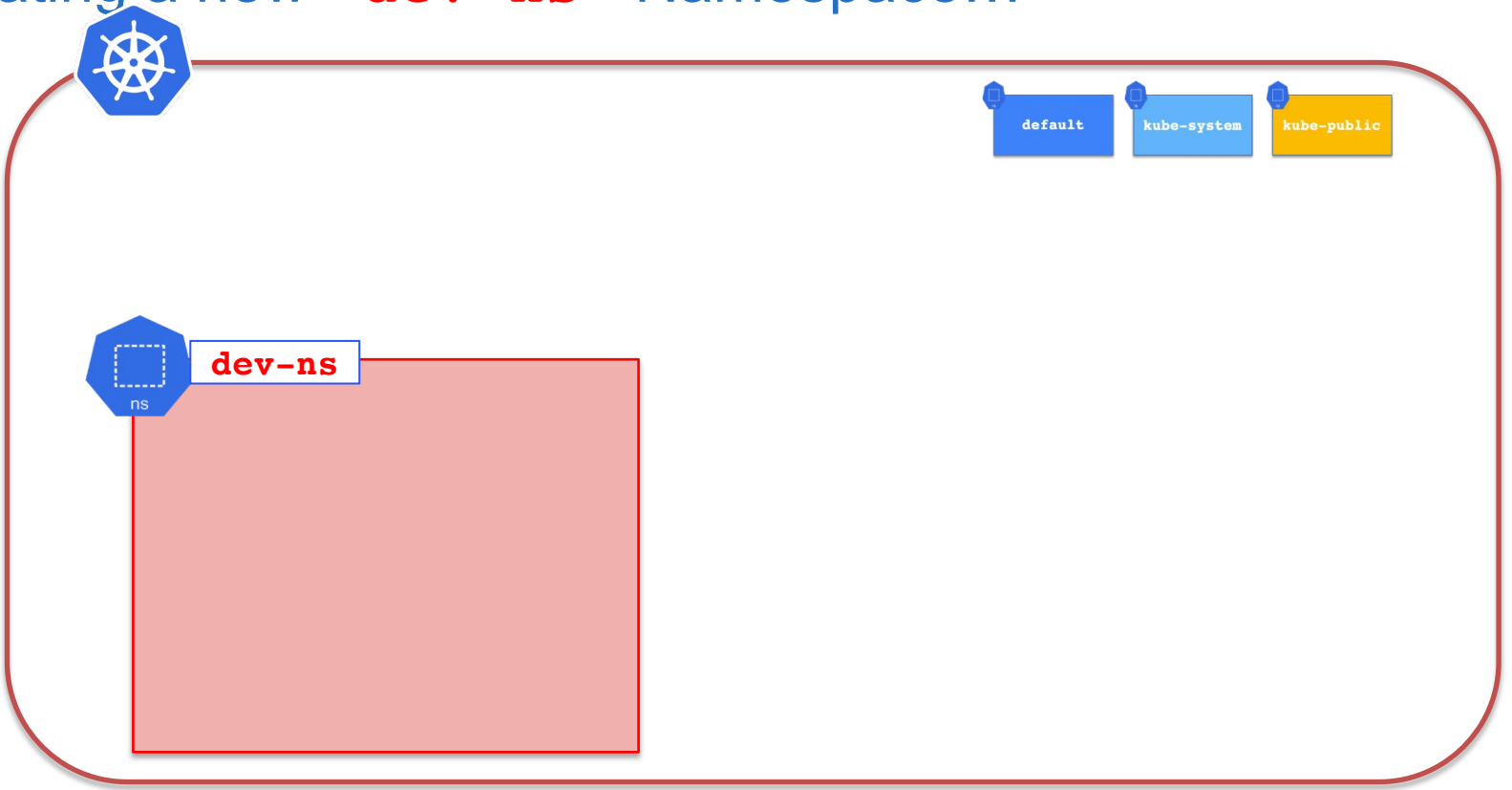
```
[root@k8s-master|/root]# kubectl get ns
NAME          STATUS    AGE
default       Active    6d
kube-public   Active    6d
kube-system   Active    6d



[root@k8s-master|/root]# kubectl describe ns kube-system
Name:         kube-system
Labels:       <none>
Annotations:  <none>
Status:       Active
No resource quota.
No resource limits.
```

Namespaces - Create a new "dev-ns" Namespace

Let's now create a new namespace called **dev-ns**:

```
[root@k8s-master|/root]# cd /root/labs/yamls/


[root@k8s-master|/root/labs/yamls]# kubectl apply -f dev-namespace.yaml
namespace "my-nspace" created

[root@k8s-master|/root]# kubectl get ns
NAME          STATUS    AGE
default       Active    6d
kube-public   Active    6d
kube-system   Active    6d
dev-ns        Active    9s
```

Creating a new "dev-ns" Namespace…

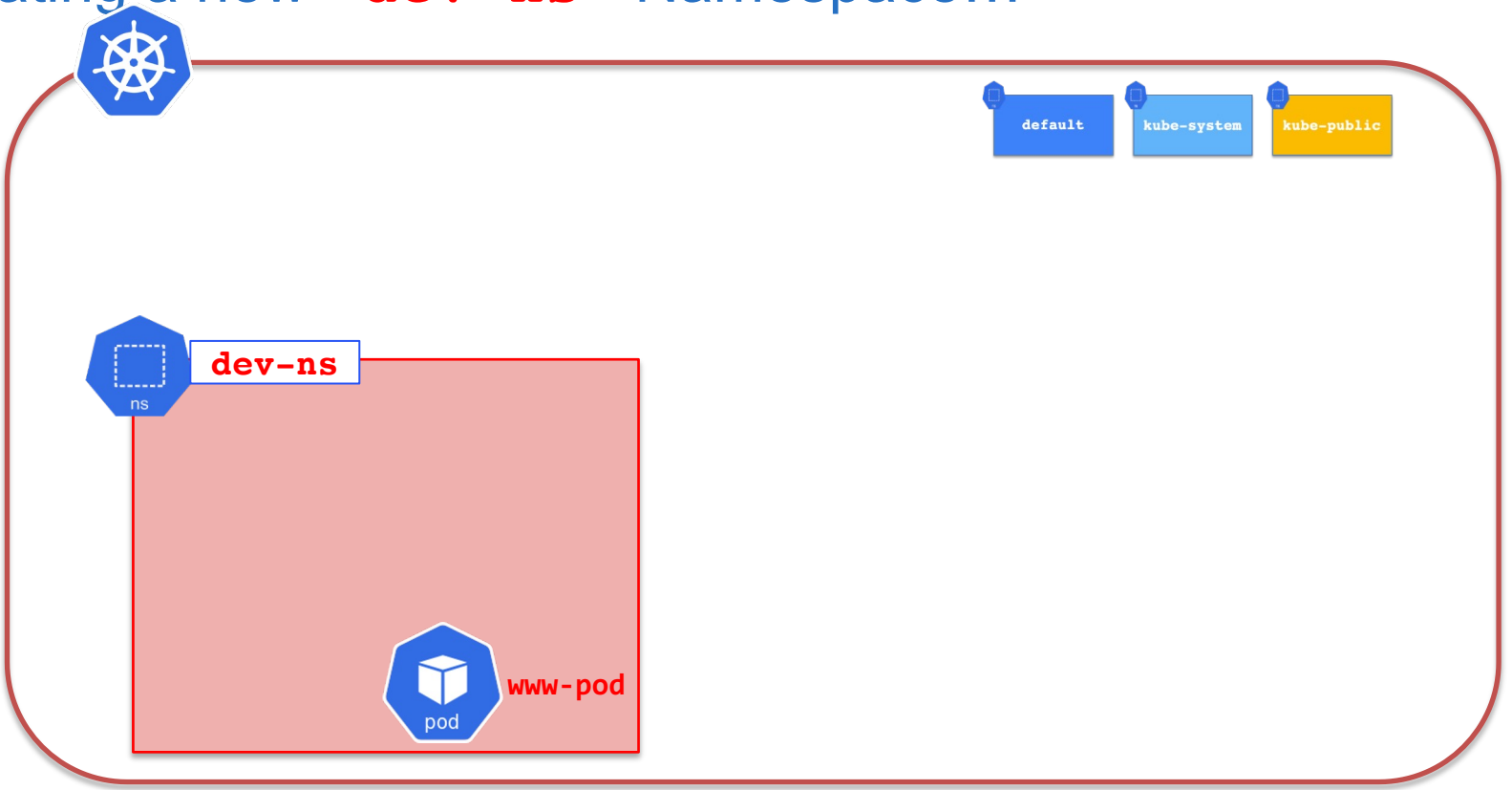


Namespaces - Create a pod in the new Namespace ‘dev-ns’

Let’s now create a new pod in the namespace **dev-ns** we created earlier:

```
[root@k8s-master|/root/labs/yamls]# kubectl create --namespace=dev-ns -f create-www-pod.yaml
pod/www-pod created
```

Creating a new "dev-ns" Namespace…

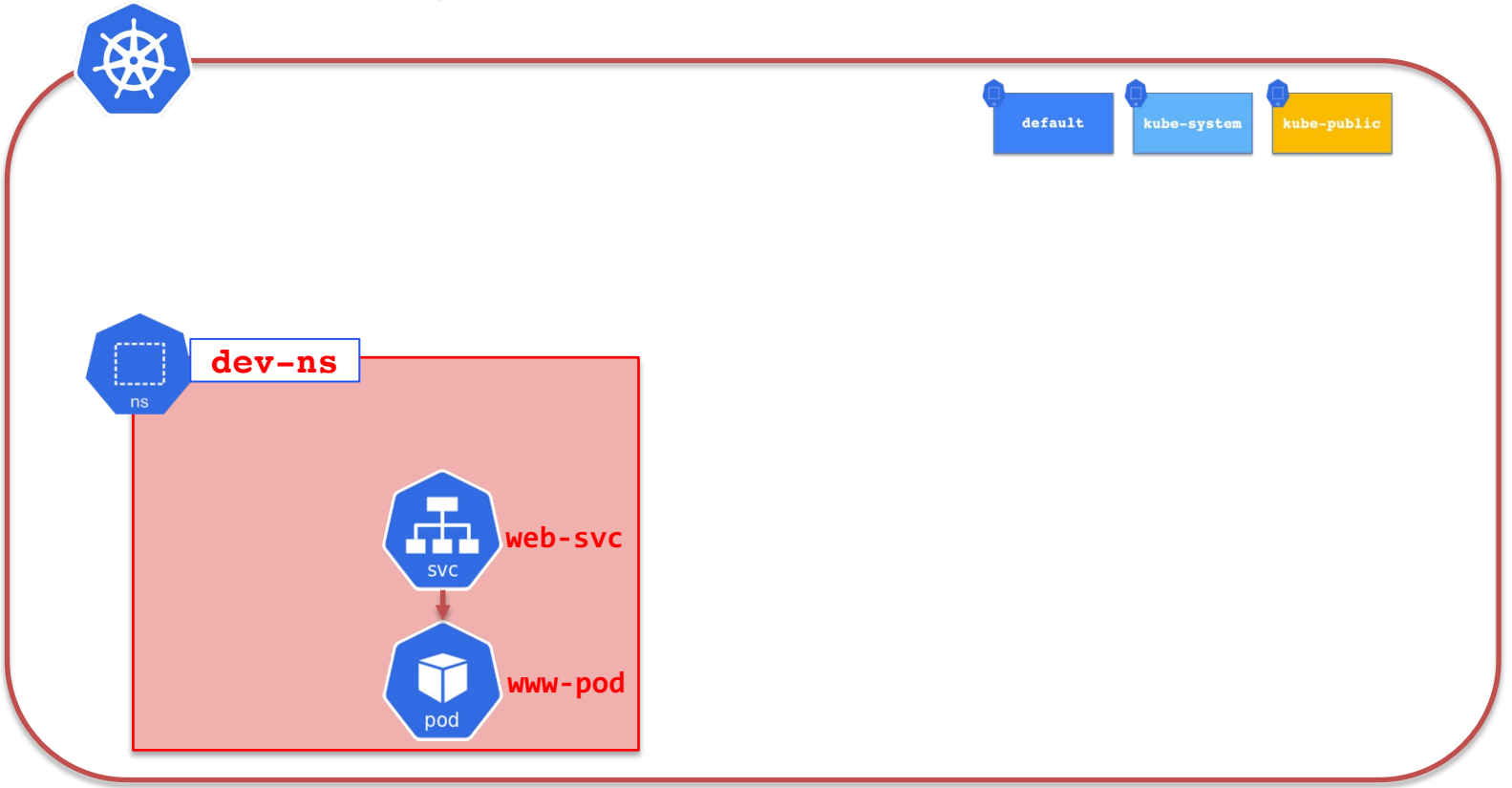


Namespaces - Create a pod in the new Namespace ‘dev-ns’

Let’s now create a new pod in the namespace **dev-ns** we created earlier:

```
[root@k8s-master|/root]# kubectl get pod

<<check the output… do you see the newly created Pod? >>

[root@k8s-master|/root]# kubectl get pod --namespace=dev-ns
NAME                READY     STATUS     RESTARTS    AGE
www-pod    1/1      Running    0           2m12s


<<Try to run the command to create the pod once more; what do you see in the output>>

[root@k8s-master|/root/labs/yamls]# kubectl create --namespace=dev-ns -f create-www-pod.yaml

Error from server (AlreadyExists): error when creating "create-www-pod.yaml": pods "www-pod" already exists
```

You can create other objects in **"dev-ns"** Namespace…

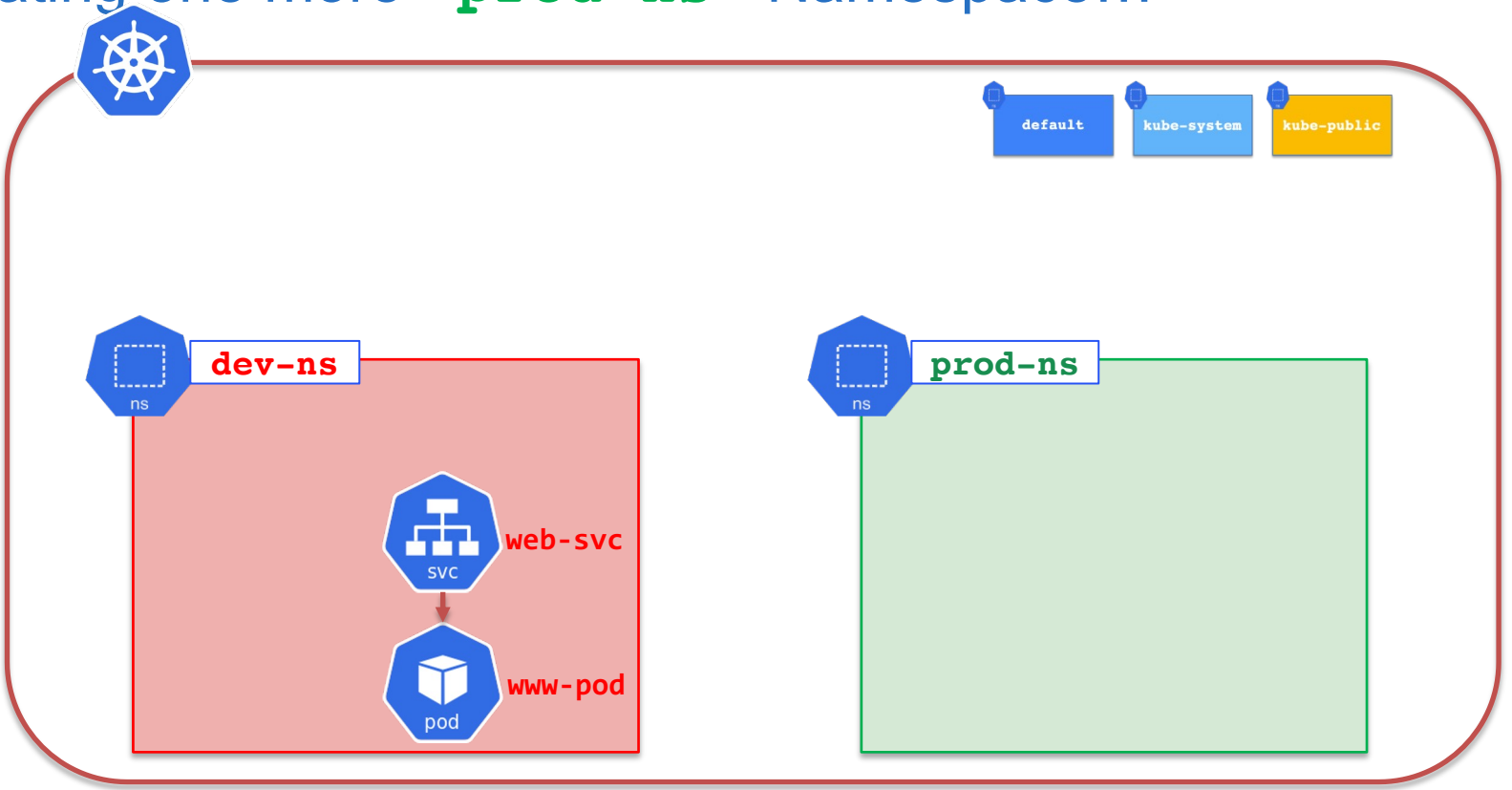


Namespaces - Create a new "prod-ns" Namespace

Let's now create a new namespace called **prod-ns from the command line**:

```
[root@k8s-master|/root/labs/yamls]# kubectl create namespace prod-ns
namespace "my-nspace" created

[root@k8s-master|/root]# kubectl get ns
NAME          STATUS    AGE
default       Active    6d
kube-public   Active    6d
kube-system   Active    6d
dev-ns        Active    25m
prod-ns       Active    15s
```

Creating one more "prod-ns" Namespace…

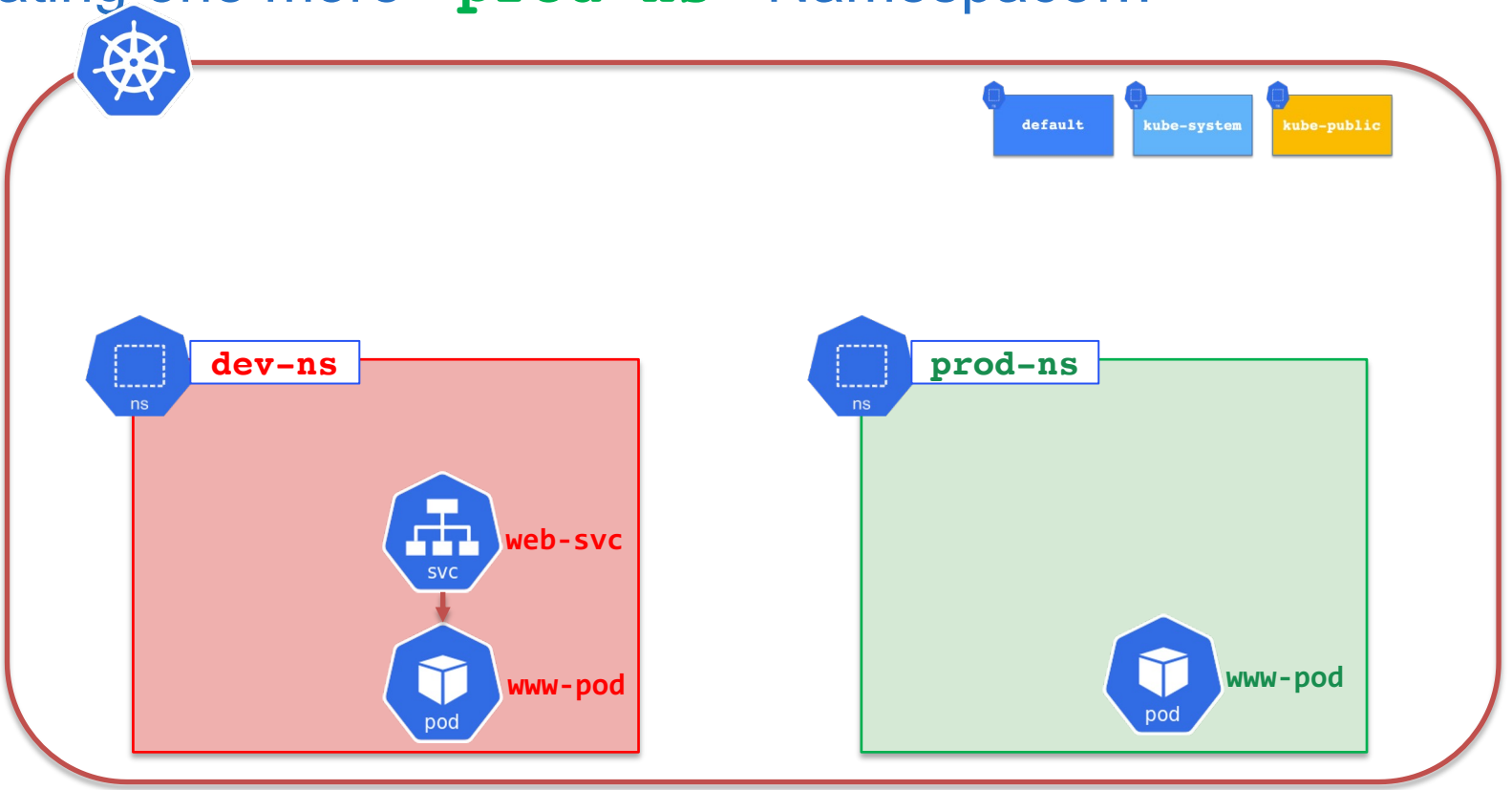


Namespaces - Create a pod in the new Namespace ‘prod-ns’

Let’s now create a new pod in the namespace **prod-ns** we created earlier:

```
[root@k8s-master|/root/labs/yamls]# kubectl create --namespace=prod-ns -f create-www-pod.yaml
pod/www-pod created


[root@k8s-master|/root/labs/yamls]# kubectl get pod -n prod-ns
NAME      READY   STATUS    RESTARTS   AGE
www-pod   1/1     Running   0          34s
```

Creating one more "prod-ns" Namespace…



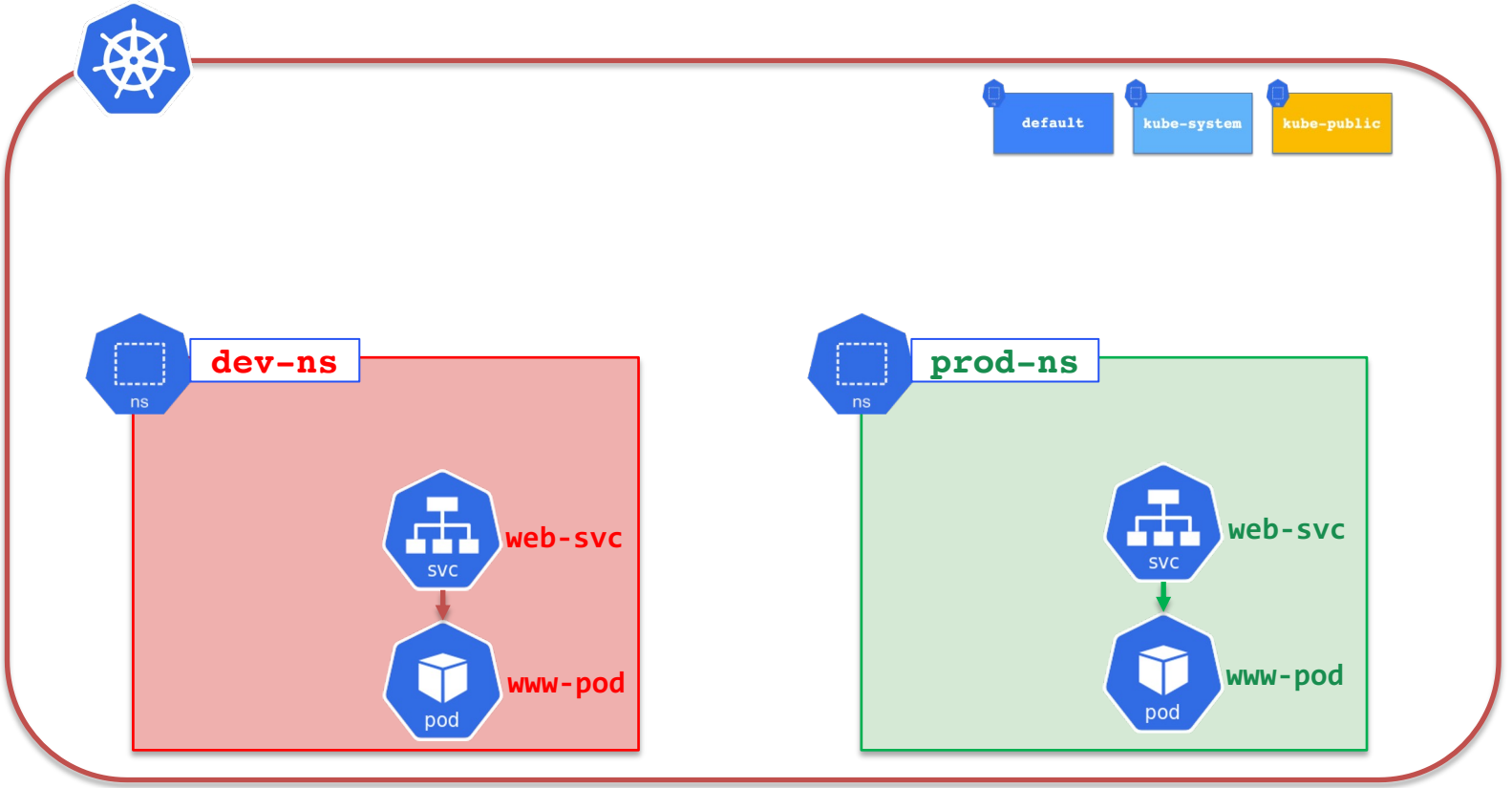
Namespaces - Create a pod with same name in another namespace

Why did it allow to create pods with the same name in prod-ns namespace?

```
[root@k8s-master|/root/labs/yamls]# kubectl get pods --all-namespaces | grep www-pod
dev-ns        www-pod                                    1/1     Running            0          15m
prod-ns       www-pod                                    1/1     Running            0          2m21s
```

Namespaces allows us to use the same object names in different namespaces. There is no problem in this. Within the same namespace, the object name must be unique.

You can create other objects in with same name in "prod-ns" Namespace…



Connecting to these ‘www-pod’ directly…

You can access these pods directly using ‘curl’ or ‘exec’ as below:

```
[root@k8s-master|/root/labs/yamls]# kubectl get pods --all-namespaces -o wide | grep www-pod
dev-ns        www-pod                                    1/1     Running            0          20m     10.244.2.52
prod-ns       www-pod                                    1/1     Running            0          7m7s    10.244.2.53


[root@k8s-master|/root/labs/yamls]# curl http://10.244.2.52:8080/


[root@k8s-master|/root/labs/yamls]# curl http://10.244.2.53:8080/

[root@k8s-master|/root/labs/yamls]# kubectl exec -it -n dev-ns www-pod -- sh

[root@k8s-master|/root/labs/yamls]# kubectl exec -it -n prod-ns www-pod -- sh
```

Deleting objects in namespace…

You can specify namespace with kubectl to delete objects from that namespace:

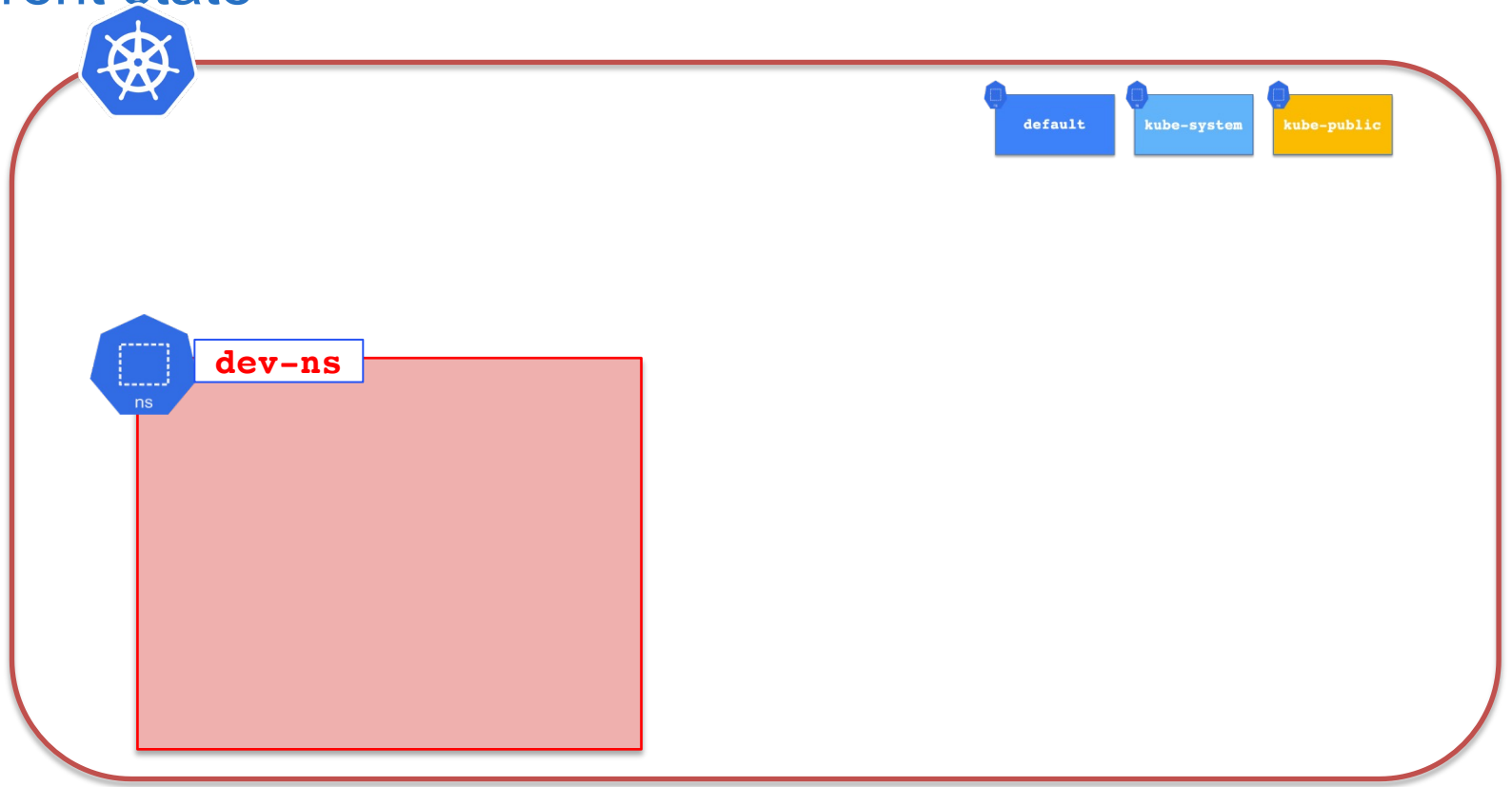
```
[root@k8s-master|/root/labs/yamls]# kubectl get pods --all-namespaces -o wide | grep www-pod
dev-ns        www-pod                                    1/1     Running            0          20m     10.244.2.52
prod-ns       www-pod                                    1/1     Running            0          7m7s    10.244.2.53


[root@k8s-master|/root/labs/yamls]# kubectl delete pod -n dev-ns www-pod

[root@k8s-master|/root/labs/yamls]# kubectl get pods --all-namespaces -o wide | grep www-pod
prod-ns       www-pod                                    1/1     Running            0          7m7s    10.244.2.53
```

Deleting the namespace (will delete all objects in it!!!)

You can delete a namespace with kubectl; however it will delete all the objects in it without any warning.

```
[root@k8s-master|/root/labs/yamls]# kubectl delete namespace prod-ns
```

Current state

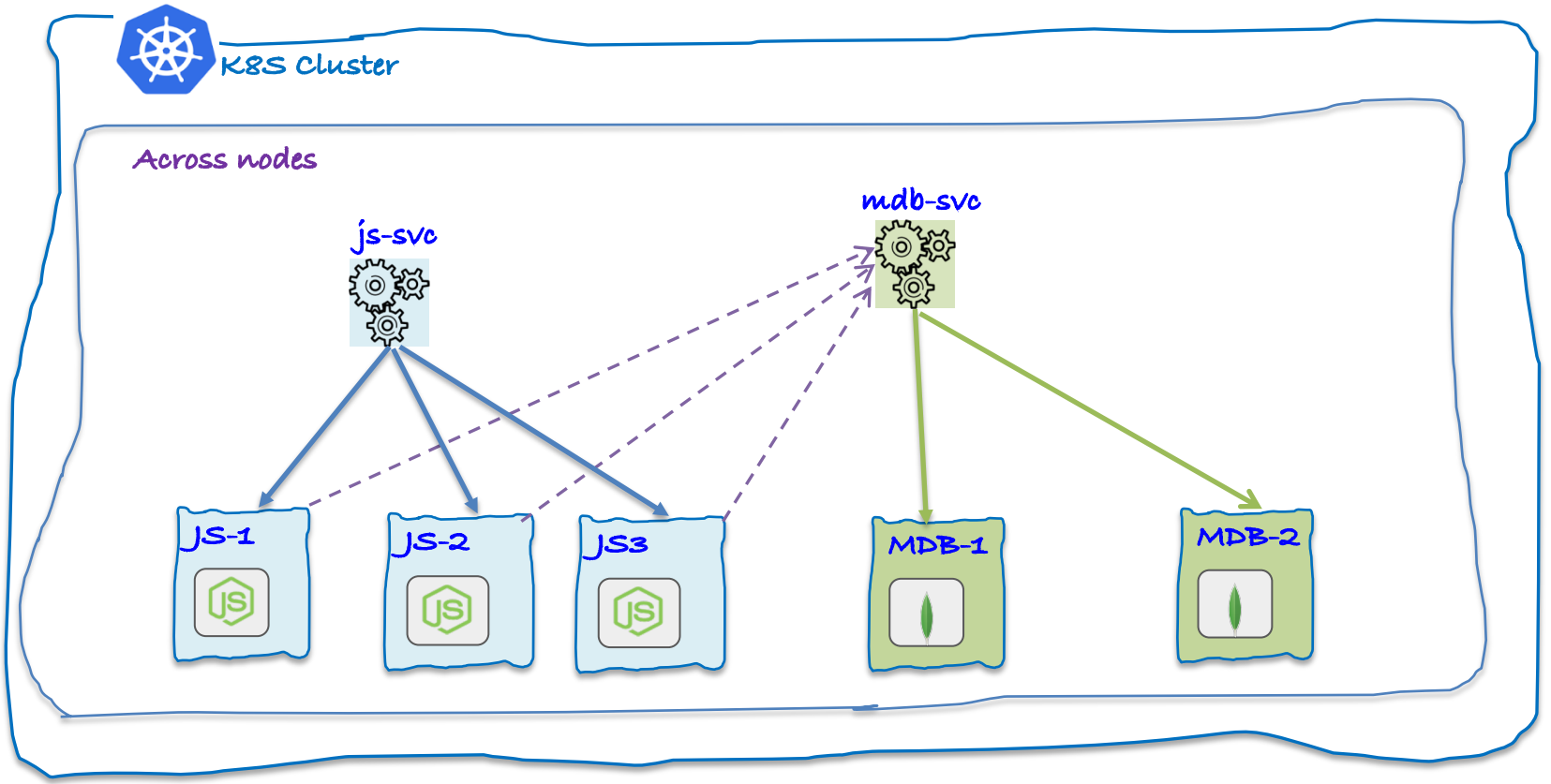


Service Discovery and Load Balancing

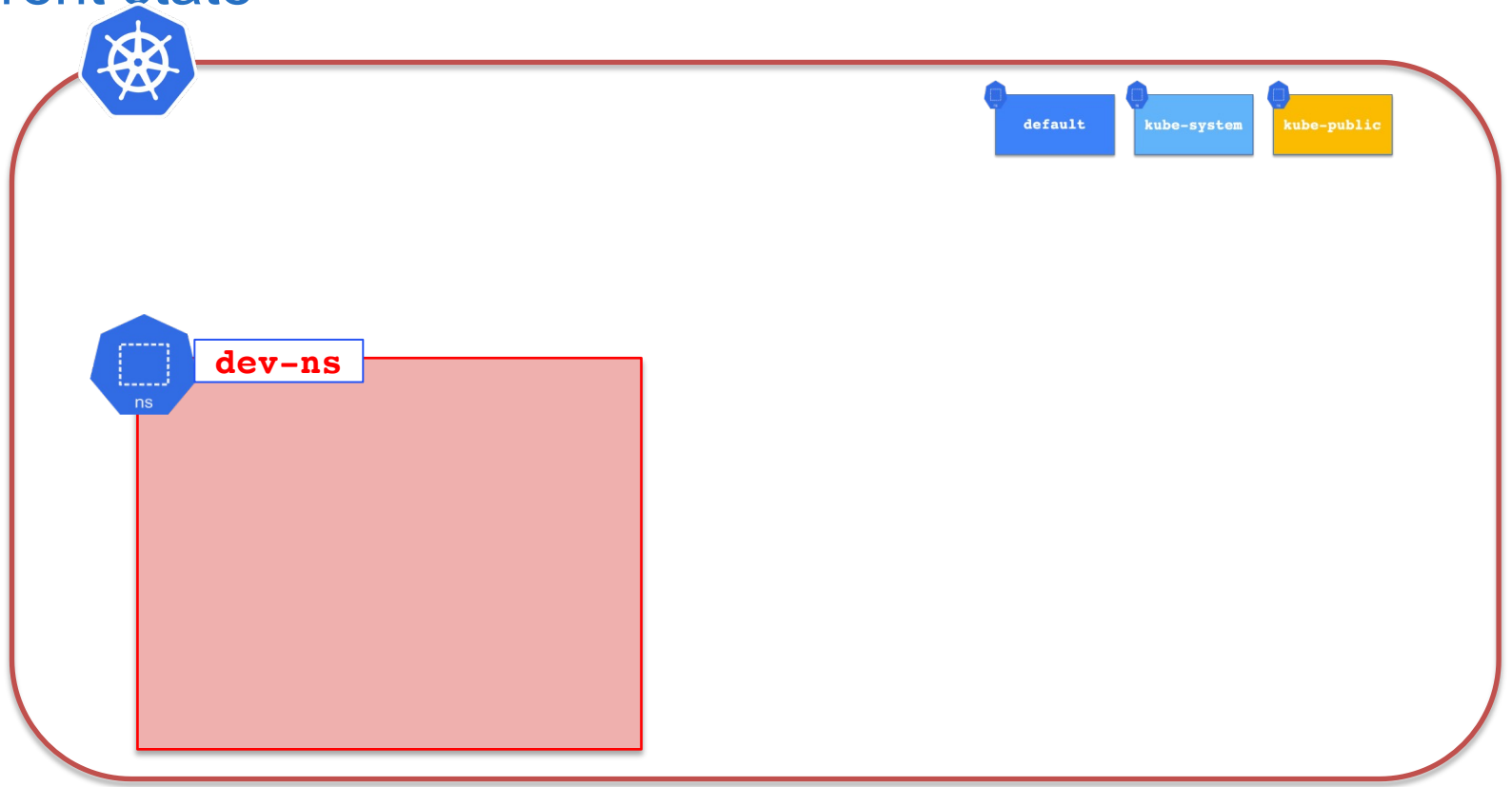
Service Discovery and Load Balancing

- Kubernetes service provides a mechanism for reaching pods and load balancing across pods.
- Service discovery is the process of figuring out how to connect to a service.
- It is implemented using kube-proxy
- Internally uses iptable rules for load balancing at the network layer.
- It is recommended to use the DNS-based service discovery.
- Each Kubernetes service exposes a DNS entry for accessing the services within the Kubernetes internal network
- For DNS based service discovery to work, we need to make sure that DNS cluster add-on is up and running
- Typically all Kubernetes distribution provides this install out of the box.

Service Discovery and Load Balancing



Current state

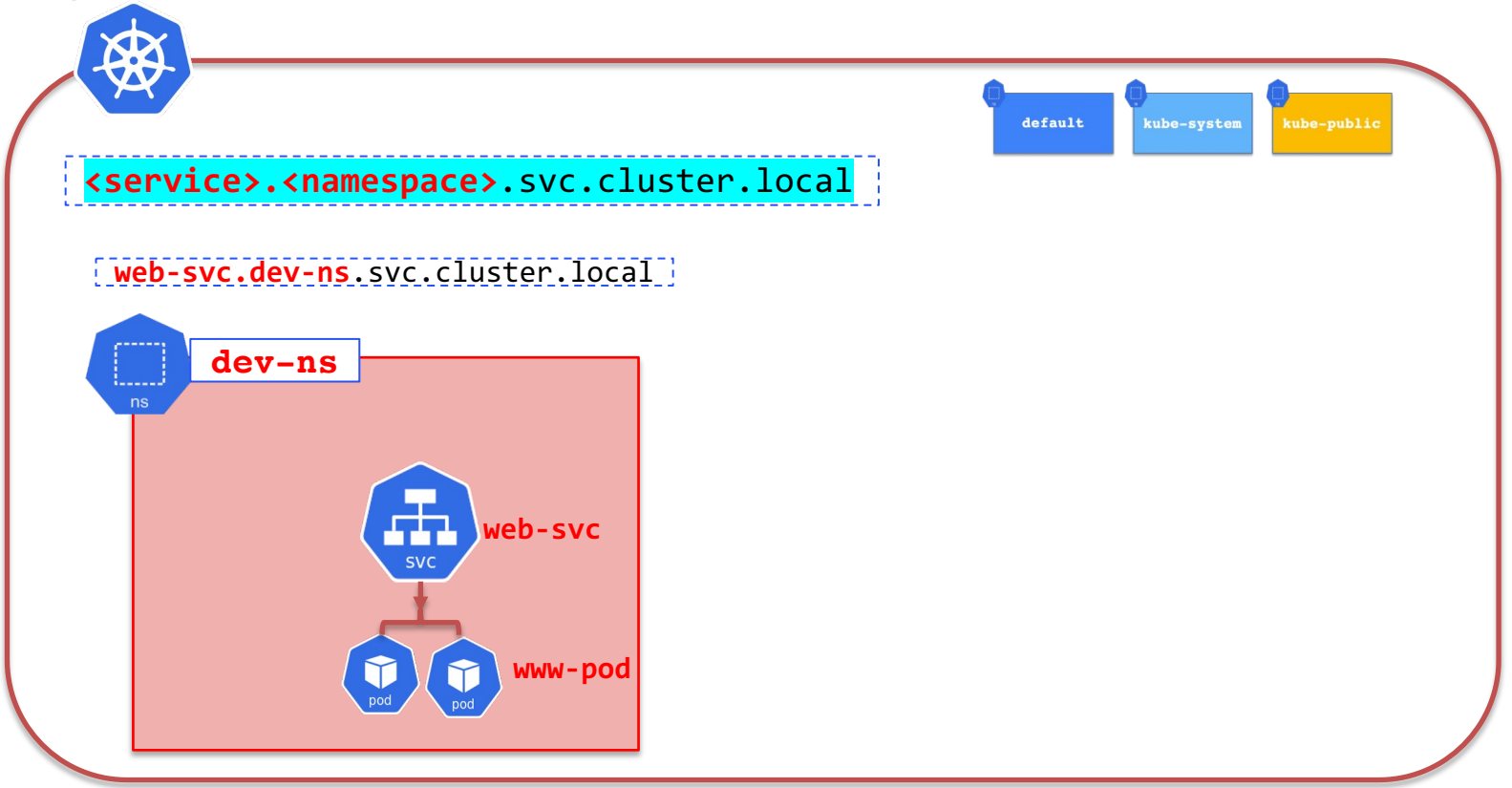


Service Discovery

Let’s see how service discovery works. We will start pods in each of the namespaces.

```
[root@k8s-master|/root]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# kubectl -n dev-ns create -f  service-discovery-rc.yaml
replicationcontroller/www-pod created

[root@k8s-master|/root/labs/yamls]# kubectl -n dev-ns create -f  service-discovery-svc.yaml
service/web-svc created
```

Creating a service also creates a FQDN in "dev-ns" namespace…



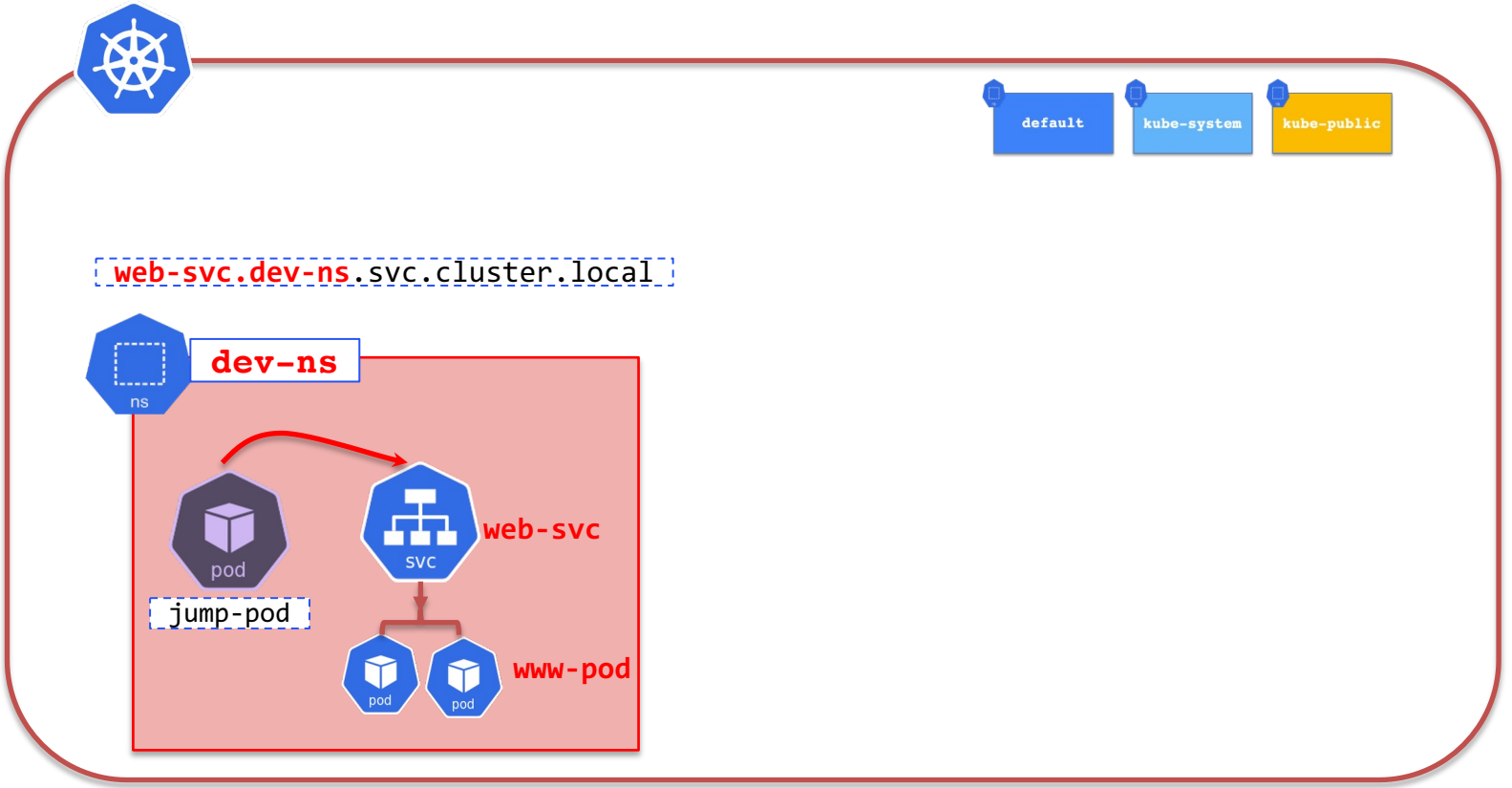
Service Discovery - accessing the service by name

The DNS add-on will make sure that our service **`web-svc`** is available via the FQDN **`web-svc.default.svc.cluster.local`** from other pods in the cluster.

Let's launch a jump-pod and try accessing it:

```
# kubectl -n dev-ns create -f service-discovery-jump-pod.yaml
pod/jump-pod created
```

Create 'jump-pod' in "dev-ns" namespace to access 'web-svc'...



Service Discovery - accessing the service by name

Try to ping the FQDN **`web-svc.default.svc.cluster.local`** from jump-pod in the same namespace.

```
# kubectl -n dev-ns exec jump-pod -i -t -- ping -c 2 web-svc.dev-ns.svc.cluster.local
PING web-svc.dev-ns.svc.cluster.local (10.108.188.56) 56(84) bytes of data.

--- web-svc.dev-ns.svc.cluster.local ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1008ms

command terminated with exit code 1
```

Did we get a ping response? If not, why did we not get the ping response? What do you think is the reason? Let see the next slide and test and see if we can access using ‘curl’.

Service Discovery - connecting to the service using curl

We can directly connect to and consume the service (in the same namespace) like so:

```
# kubectl -n dev-ns exec jump-pod -i -t -- curl http://web-svc/ | grep POD
This is served from POD : www-pod-8zsnb
and the POD IP is : 10.244.1.57

# kubectl -n dev-ns exec jump-pod -i -t -- curl http://web-svc.dev-ns/ | grep POD
This is served from POD : www-pod-8zsnb
and the POD IP is : 10.244.1.57

# kubectl -n dev-ns exec jump-pod -i -t -- curl http://web-svc.dev-ns.svc.cluster.local/ | grep POD
This is served from POD : www-pod-8zsnb
and the POD IP is : 10.244.1.57
```

In the first two cases, how is the pod able to resolve these curl URLs? Let us take a closer look at how DNS is working.

Service Discovery – and the DNS connection….

```
# kubectl -n dev-ns exec jump-pod -i -t -- cat /etc/resolv.conf
nameserver 10.96.0.10
search dev-ns.svc.cluster.local svc.cluster.local cluster.local example.com
```

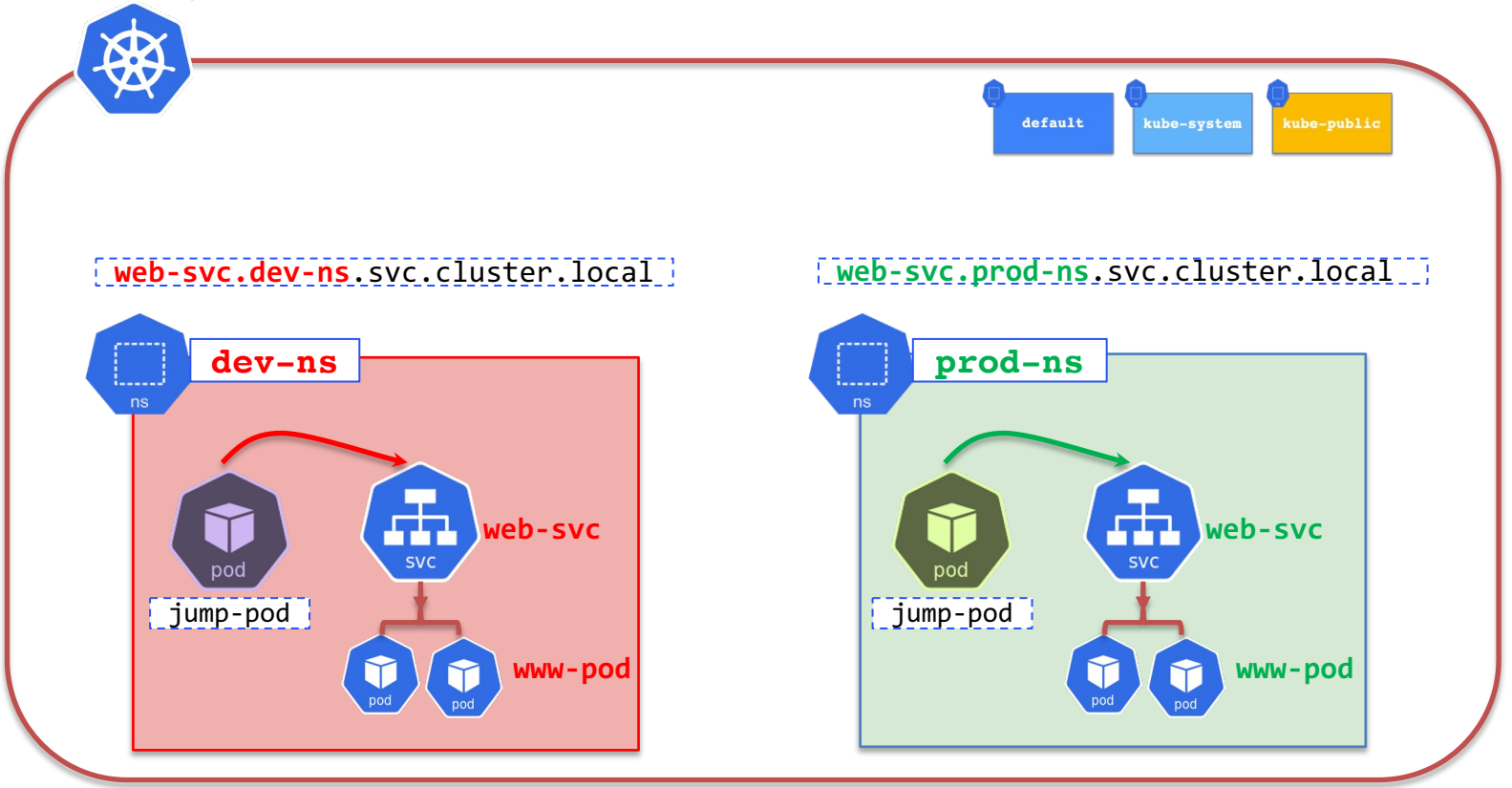
You can see that the `/etc/resolv.conf` has the `"search"` keyword populated with the search domains correctly. This is the reason why we can resolve the service names correctly. This is same as using any other DNS services and settings. You can check using `'nslookup'` tool.

```
# kubectl -n dev-ns exec jump-pod -i -t -- nslookup web-svc

# kubectl -n dev-ns exec jump-pod -i -t -- nslookup web-svc.dev-ns

# kubectl -n dev-ns exec jump-pod -i -t -- nslookup web-svc.dev-ns.svc.cluster.local
```

Now let's create a new namespace (prod-ns) and create the same objects…

```
[root@k8s-master|/root]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# kubectl create ns prod-ns

[root@k8s-master|/root/labs/yamls]# kubectl -n prod-ns create -f  service-discovery-rc.yaml
replicationcontroller/www-pod created

[root@k8s-master|/root/labs/yamls]# kubectl -n prod-ns create -f  service-discovery-svc.yaml
service/web-svc created

[root@k8s-master|/root/labs/yamls]# kubectl -n prod-ns create -f service-discovery-jump-pod.yaml
pod/jump-pod created
```

Additional objects in "prod-ns" namespace…



Service Discovery - accessing the service by name from another namespace

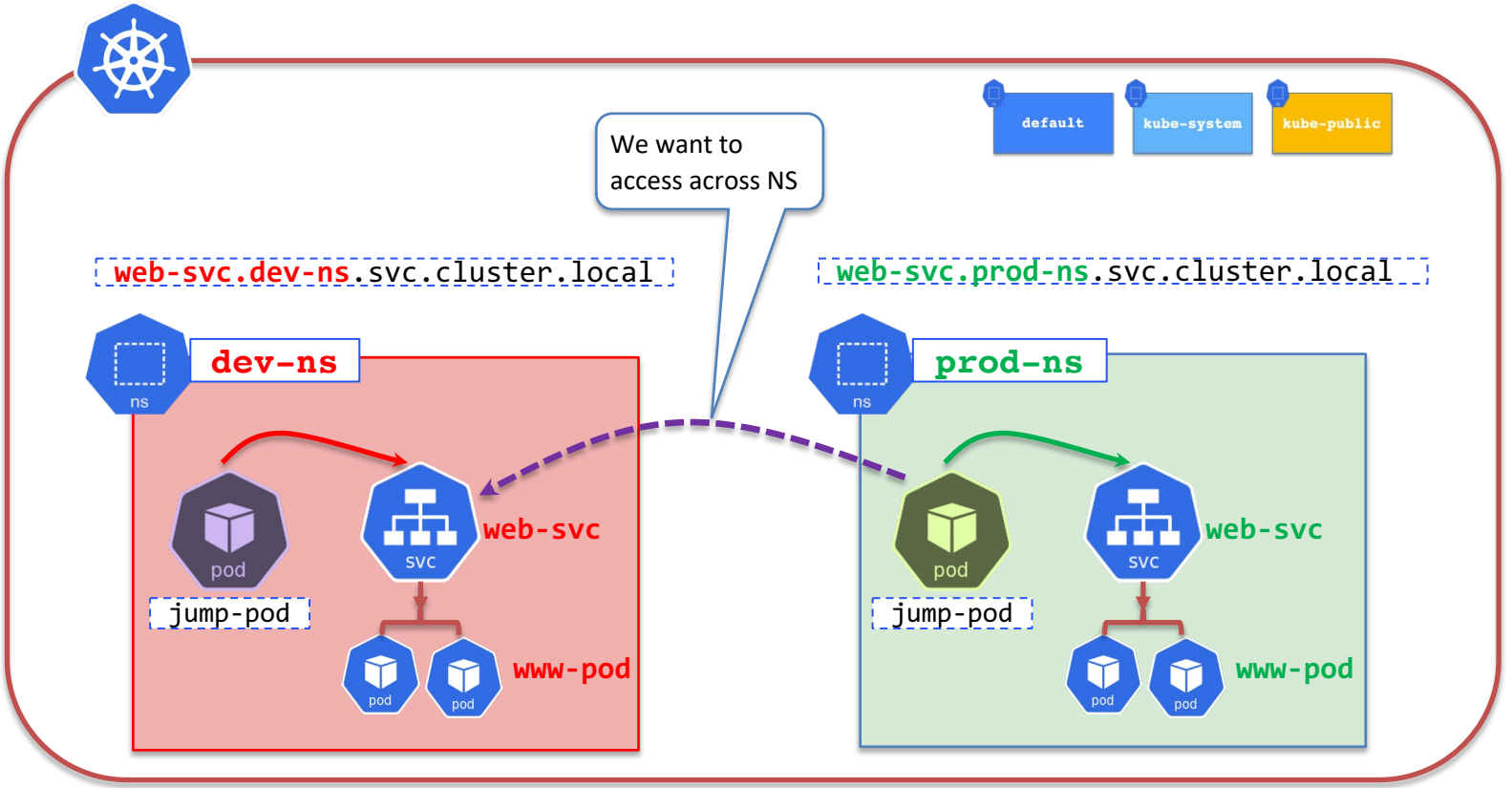
Let's try accessing the **web-svc** service across the namespace (from **prod-ns** to **dev-ns**)…

```
# kubectl -n prod-ns exec jump-pod -i -t -- curl http://web-svc/ | grep POD
<p>This is served from POD : <b> www-pod-hbvhv </b> </p>
<p>and the POD IP is :<b> 10.244.2.74 </b> </p>
```

what is the output? What do you think would have happened? Is this what you wanted?

```
# kubectl get pods --all-namespaces -o wide | grep www-pod
dev-ns        www-pod-7wc65                              1/1     Running   0          21m     10.244.2.69   hk8s-node3
dev-ns        www-pod-8zsnb                              1/1     Running   0          21m     10.244.1.57   hk8s-node2
prod-ns       www-pod-hbvhv                              1/1     Running   0          111s    10.244.2.74   hk8s-node3
prod-ns       www-pod-hlgqq                              1/1     Running   0          111s    10.244.1.58   hk8s-node2
```

Connect from prod-ns to dev-ns



Connect from **prod-ns** to **dev-ns**

To access a service that is deployed in a different namespace than the one you're accessing it from:

Partial (with Namespace) :: **<SERVICE>.<NAMESPACE>**
Or
FQDN (recommended) :: **<SERVICE>.<NAMESPACE>.svc.cluster.local.**

Else, you will have undesired results.

```
# kubectl -n prod-ns exec jump-pod -i -t -- curl http://web-svc.dev-ns/ | grep POD
<p>This is served from POD : <b> www-pod-7wc65 </b> </p>
<p>and the POD IP is :<b> 10.244.2.69 </b> </p>

## IT IS ALWAYS RECOMMENDED TO USE FQDN AS BELOW

# kubectl -n prod-ns exec jump-pod -i -t -- curl http://web-svc.dev-ns.svc.cluster.local/ | grep POD
<p>This is served from POD : <b> www-pod-8zsnb </b> </p>
<p>and the POD IP is :<b> 10.244.1.57 </b> </p>

# kubectl -n dev-ns get pods -o wide | grep pod
www-pod-7wc65   1/1     Running   0          27m   10.244.2.69   hk8s-node3   <none>           <none>
www-pod-8zsnb   1/1     Running   0          27m   10.244.1.57   hk8s-node2   <none>           <none>
```

Clean up….

Clean up all the resources that was created….

```
[root@k8s-master:/]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# kubectl delete ns dev-ns


[root@k8s-master|/root/labs/yamls]# kubectl delete ns prod-ns
```

Volumes

Volumes

- Volumes in Kubernetes is essentially a directory accessible to all containers running in a pod.
- Unlike container-local filesystem, the data in volumes is preserved across container restarts.
- We have seen the similar behavior when we learned Docker volumes

Let’s create a pod with two containers that use an directory to exchange data:

```
[root@k8s-master|/root]# cd /root/labs/yamls


[root@k8s-master|/root/labs/yamls]# kubectl create -f volumes-pod.yaml

pod "sharevol" created

[root@k8s-master|/root]# kubectl describe pod sharevol | grep EmptyDir
 Type:    EmptyDir (a temporary directory that shares a pod's lifetime)
```

Volumes -- accessing the volumes from first container

```
[root@k8s-master|/root]# kubectl exec sharevol -c container1 -i -t -- bash


[root@sharevol ~]# mount | grep xchang
/dev/mapper/centos-root on /tmp/xchange type xfs (rw,relatime,attr2,inode64,noquota)


[root@sharevol ~]# echo "Hello from container1...." > /tmp/xchange/hello_from_c1.txt
[root@sharevol ~]#
```

Volumes -- verifying the contents from second container

```
[root@k8s-master|/root]# kubectl exec sharevol -c container2 -i -t -- bash


[root@sharevol /]# mount | grep /tmp/data
/dev/mapper/centos-root on /tmp/data type xfs (rw,relatime,attr2,inode64,noquota)

[root@sharevol /]# cat /tmp/data/hello_from_c1.txt
Hello from C1....
```

PersistentVolume and PersistentVolumeClaim

PersistentVolume and PersistentVolumeClaim - Introduction

As discussed earlier, Pods, at the back, use Docker images; these are immutable and it is not possible to simply store persistent data within containers. When applications write to the Docker union file system, that data is lost as soon as the container crashes or deleted. K8S provides a solution for storing persisting data, that allows users to mount a volume inside the Pod. All containers within a pod share same resources, including storage.

A **`PersistentVolume (PV)`** object is a storage resource in an K8S cluster. Storage is provisioned by your cluster administrator by creating **`PersistentVolume (PV)`** objects from sources such as GCE Persistent Disk, AWS Elastic Block Store (EBS), and NFS mounts.

High-availability of storage in the infrastructure is left to the underlying storage provider.

Storage can be made available to the pods by laying claims to the resource. You can make a request for storage resources using a **`PersistentVolumeClaim (PVC)`** object; the claim is paired with a volume that generally matches your request.

PersistentVolume and PersistentVolumeClaim - Scope

PVCs are **specific to a namespace** and are created and used by users as a means to use a PV. PV resources on their own are not scoped to any single namespace; they can be shared across the entire K8S cluster and claimed from any project. After a PV has been bound to a PVC, however, that PV cannot then be bound to additional PVCs. This has the effect of scoping a bound PV to a single namespace (that of the binding namespace).

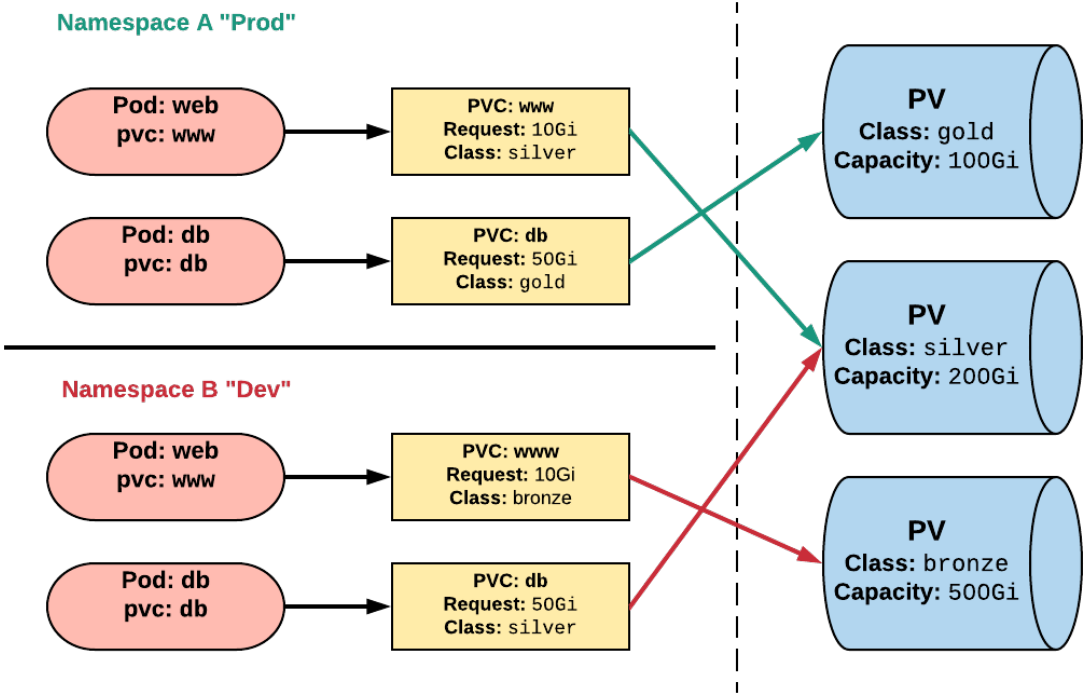
Using persistent volume claims (PVCs), users can request PV resources without having specific knowledge of the underlying storage infrastructure.

A typical lifecycle of a Volume and Claim is a as below:

Provisioning-Binding-Using-Releasing-Reclaiming

PV/PVC and Namespaces

PV is not tied to any namespace; however, PVC are tied to the namespace from where it is claimed.



PersistentVolume and PersistentVolumeClaim - Properties

Access Modes:

Access Mode	CLI Abbreviation	Description
ReadWriteOnce	RWO	The volume can be mounted as read-write by a single node.
ReadOnlyMany	ROX	The volume can be mounted read-only by many nodes.
ReadWriteMany	RWX	The volume can be mounted as read-write by many nodes.

Recycle Policy:

Recycling Policy	Description
Retain	Manual reclamation
Recycle	Basic scrub (e.g, rm -rf /<volume>/*)

Phases of Volume:

Phase	Description
Available	A free resource that is not yet bound to a claim.
Bound	The volume is bound to a claim.
Released	The claim has been deleted, but the resource is not yet reclaimed by the cluster.
Failed	The volume has failed its automatic reclamation.

Volumes -- Configure a Pod to Use a PersistentVolume

In this lab we will configure a Pod to use a PersistentVolumeClaim for storage.

- A cluster administrator creates a PersistentVolume that is backed by physical storage. The administrator does not associate the volume with any Pod.
- A cluster user creates a PersistentVolumeClaim, which gets automatically bound to a suitable PersistentVolume
- The user creates a Pod that uses the PersistentVolumeClaim as storage.

```
[root@worker-node1|/root]# mkdir -p /var/tmp/data

[root@worker-node1|/root]# echo 'Hello from Kubernetes storage on NODE1' > /var/tmp/data/hello.txt

[root@worker-node2|/root]# mkdir -p /var/tmp/data

[root@worker-node2|/root]# echo 'Hello from Kubernetes storage on NODE2' > /var/tmp/data/hello.txt
```

Credits: https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/

Volumes -- Create a PersistentVolume

Check the 'task-pv-volume.yaml' file; the configuration file specifies that the volume is at /var/tmp/data on the the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of ReadWriteOnce, which means the volume can be mounted as read-write by a single Node. It defines the StorageClass name manual for the PersistentVolume, which will be used to bind PersistentVolumeClaim requests to this PersistentVolume.

Run the below command to create the PersistentVolume:

```
[root@k8s-master|/root/labs/yamls]# kubectl create -f task-pv-volume.yaml
persistentvolume "task-pv-volume" created


[root@k8s-master|/root/labs/yamls]# kubectl get pv task-pv-volume
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM     STORAGECLASS   REASON    AGE
task-pv-volume   10Gi       RWO            Retain           Available             manual                   26s
```

The output shows that the PersistentVolume has a STATUS of Available. This means it has not yet been bound to a PersistentVolumeClaim

Volumes -- Create a PersistentVolumeClaim

The next step is to create a PersistentVolumeClaim. Pods use PersistentVolumeClaims to request physical storage. In this exercise, you create a PersistentVolumeClaim that requests a volume of at least three gibibytes that can provide read-write access for at least one Node.

Create the PersistentVolumeClaim:

```
[root@k8s-master|/root/labs/yamls]# kubectl create -f  task-pv-claim.yaml
persistentvolumeclaim "task-pv-claim" created
```

After you create the PersistentVolumeClaim, the Kubernetes control plane looks for a PersistentVolume that satisfies the claim’s requirements. If the control plane finds a suitable PersistentVolume with the same StorageClass, it binds the claim to the volume.

Volumes -- Create a PersistentVolumeClaim

Look again at the PersistentVolume:

```
[root@k8s-master|/root/labs/yamls]# kubectl get pv task-pv-volume
NAME            CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS    CLAIM                   STORAGECLASS   REASON    AGE
task-pv-volume  10Gi       RWO            Retain           Bound     default/task-pv-claim   manual                   6m
```

Look at the PersistentVolumeClaim:

```
[root@k8s-master|/root/labs/yamls]# kubectl get pvc task-pv-claim
NAME            STATUS    VOLUME           CAPACITY   ACCESS MODES   STORAGECLASS   AGE
task-pv-claim   Bound     task-pv-volume   10Gi       RWO            manual         5m
```

Volumes -- Finally Create a Pod to use the PersistentVolumeClaim

The next step is to create a Pod that uses your PersistentVolumeClaim as a volume.

Here is the configuration file for the Pod: `[[cat task-pv-pod.yaml]]`

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
       claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx:run
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Volumes -- Finally Create a Pod to use the PersistentVolumeClaim

Create the pod

```
[root@k8s-master|/root/labs/yamls]# kubectl create -f task-pv-pod.yaml
pod "task-pv-pod" created

[root@k8s-master|/root/labs/yamls]# kubectl get pod task-pv-pod -o wide
NAME          READY     STATUS              RESTARTS   AGE       IP        NODE
task-pv-pod   1/1       Running   0          24s       10.244.1.3   worker-node1
```

Volumes -- Testing the PersistentVolume Claim

Use Curl to test if the Volumes are working:

```
[root@k8s-master|/root/labs/yamls]# curl 10.244.1.4/hello.txt
Hello from Kubernetes storage on NODE1

[root@k8s-master|/root/labs/yamls]# kubectl delete pod task-pv-pod
pod "task-pv-pod" deleted

[root@k8s-master|/root/labs/yamls]# kubectl create -f task-pv-pod.yaml
pod "task-pv-pod" created

[root@k8s-master|/root/labs/yamls]# kubectl get pod task-pv-pod -o wide
task-pv-pod   1/1       Running   0          2s        10.244.2.5   worker-node2

[root@k8s-master|/root/labs/yamls]# curl 10.244.2.5/hello.txt
Hello from Kubernetes storage on NODE2
```

Use Curl to test if the Volumes are working:

Volumes -- Notes

In this exercise, we create a hostPath PersistentVolume. As you saw, hostPath PersistentVolume uses a file or directory on the Node to emulate network-attached storage. You did see that the patch/folder needs to exits on all nodes and needs to be synced.

So, in a production cluster, you would not use hostPath. Instead you would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume, or a iSCSI, etc. Cluster administrators can also use StorageClasses to set up dynamic provisioning.

We will see in the OpenShift lab how this is done using NFS Shares and more advanced use of PersistentVolume.

ConfigMaps

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable and are lightweight.

A ConfigMap is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB.

Caution: ConfigMap does not provide secrecy or encryption. If the data you want to store are confidential, use a Secret rather than a ConfigMap, or use additional (third party) tools to keep your data private.

ConfigMaps

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

- Inside a container command and args
- Environment variables for a container
- Add a file in read-only volume, for the application to read
- Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

Creating and Accessing variables using ConfigMap

Let’s create a ConfigMap and store environment variables in it and later access if from a POD.

```
[root@k8s-master ~]# cd /root/new-add-on-Labs/config-maps


[root@k8s-master ~]# cat config-map.yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
data:
  # Configuration values can be set as key-value properties
  database: mongodb
  database_uri: mongodb://localhost:27017

  # Or set as complete file contents (even JSON!)
  keys: |
    image.public.key=771
    rsa.public.key=42
```

Creating and Accessing variables using ConfigMap

Let’s create a ConfigMap and store environment variables in it and later access if from a POD.

```
[root@k8s-master ~] # kubectl create -f config-map.yaml
configmap/example-configmap created


[root@k8s-master # kubectl get configmap
NAME                 DATA   AGE
example-configmap    3      52s
```

Creating and Accessing variables using ConfigMap

Verify the details...

```
[root@k8s-master ~] # kubectl describe configmap  example-configmap
Name:         example-configmap
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
database:
----
mongodb
database_uri:
----
mongodb://localhost:27017
keys:
----
image.public.key=771
rsa.public.key=42


BinaryData
====

Events:  <none>
```

Using the ConfigMap in a pod via environment variables

Now let's use the ConfigMap in a pod via a environment variables:

```
[root@k8s-master yamls]# cat pod-env-variable-from-config-map.yaml

[root@k8s-master yamls]# kubectl create -f pod-env-variable-from-config-map.yaml
pod/pod-env-var created
```

Login to the container and check the environment to validate:

```
[root@k8s-master ~]# kubectl get pods
NAME                    READY   STATUS      RESTARTS   AGE
pod-env-var             1/1     Running     0          40s

[root@k8s-master # kubectl exec pod-env-var -i -t -- /bin/sh
/ # env | egrep "(database|database_uri|image.public.key|rsa.public.key)"
database=mongodb
database_uri=mongodb://localhost:27017
keys=image.public.key=771
rsa.public.key=42
```

Secrets

Secrets

Secrets provide a mechanism to keep a password or a sensitive API key information in a safe and reliable way and access it when required.

Secrets has the following properties:

- Secrets are name-spaced objects, that is, exist in the context of a namespace
- You can access them via a volume or an environment variable from a container running in a pod
- The secret data on nodes is stored in tmpfs volumes
- A per-secret size limit of 1MB exists
- The API server stores secrets as plaintext in etcd

Secrets -- Creating and Accessing the using secrets

Let's create a secret **`apitoken`** that holds a (made-up) API key:

```
[root@k8s-master ~]# echo -n "a355e6.cd4749aa8de249cd" > ./apitoken.txt


[root@k8s-master ~]# kubectl create secret generic apitoken --from-file=./apitoken.txt
secret "apitoken" created

[root@k8s-master ~]# kubectl describe secrets/apitoken
Name:         apitoken
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
apitoken.txt:  23 bytes
```

Secrets -- Using the secret in a pod via a volume

Now let's use the secret in a pod via a volume:

```
[root@k8s-master ~]# cd  /root/labs/yamls


[root@k8s-master yamls]# cat secrets-pod.yaml

[root@k8s-master yamls]# kubectl create -f secrets-pod.yaml
pod/consume-secret created
```

Login to the container we see the secret mounted at `/tmp/apitoken`:

```
[root@k8s-master ~]# kubectl exec consume-secret -i -t -- bash
[root@consumesec /]#

[root@consumesec /]#  mount | grep apitoken
tmpfs on /tmp/apitoken type tmpfs (ro,relatime)


[root@consumesec /]# cat /tmp/apitoken/apitoken.txt
a355e6.cd4749aa8de249cd
```

Jobs

Jobs

A Kubernetes job is a workload controller object that performs one or more finite tasks in a cluster. The finite nature of jobs differentiates them from most controller objects, such as deployments, replica sets, stateful sets, and daemon sets (which are long running in nature).

Job objects again maintain the desired state and number of pods in the cluster, jobs run until they complete the task and then terminate the associated pods.

Typical Kubernetes Job Use Cases:

- Maintenance tasks (such as performing backups)
- Batch tasks such as sending emails
- Batch jobs to processing files and transfer to other endpoints
- Monitoring node behaviors
- Checking and probing endpoints
- Shipping/Cleaning up of logs

Jobs

Let’s run a simple demo:

```
[root@k8s-master ~]# cd  /root/new-add-on-Labs/jobs

[root@k8s-master ~]# kubectl create namespace jobs-demo

[root@k8s-master ~]# kubectl  -n jobs-demo apply -f job-demo1.yaml

[[run this in a different Putty session]]
[root@k8s-master ~]# kubectl  -n jobs-demo get pod -w
NAME                 READY   STATUS      RESTARTS   AGE
job-demo1--1-9gr68   1/1     Running   0          108s


[root@k8s-master ~]# kubectl  -n jobs-demo logs job-demo1--1-9gr68 --follow
9

[[run this in another new Putty session]]
[root@k8s-master ~]# kubectl  -n jobs-demo get job -w
NAME        COMPLETIONS   DURATION   AGE
job-demo1   1/1           15s        2m44s

[root@k8s-master ~]# kubectl -n jobs-demo describe job job-demo1

[root@k8s-master ~]# kubectl  -n jobs-demo get pods
NAME                 READY   STATUS      RESTARTS   AGE
job-demo1--1-9gr68   0/1     Completed   0          5m29s
```

Jobs

Execute job-demo2-multiple Job :

```
[root@k8s-master ~]# kubectl -n jobs-demo apply -f job-demo2-multiple.yaml

[root@k8s-master ~]# kubectl -n jobs-demo get pod

[[run this in a different Putty session]]
[root@k8s-master ~]# kubectl -n jobs-demo get pods -w


[root@k8s-master ~]# kubectl -n jobs-demo get job -w

[root@k8s-master ~]# kubectl -n jobs-demo describe job job-demo2-multiple
<<<check the details...>>>>>
```

Jobs

Output of job-demo2-multiple Job:

```
[root@k8s-master ~]# kubectl -n jobs-demo get pods
NAME                          READY   STATUS      RESTARTS   AGE
job-demo2-multiple--1-f45j4   0/1     Completed   0          12s
job-demo2-multiple--1-gmgdh   0/1     Completed   0          2s
job-demo2-multiple--1-hnhvl   0/1     Completed   0          14s
job-demo2-multiple--1-l7gk5   0/1     Completed   0          9s
job-demo2-multiple--1-ljspv   0/1     Completed   0          10s
job-demo2-multiple--1-qf275   0/1     Completed   0          5s
job-demo2-multiple--1-tvws6   0/1     Completed   0          8s
job-demo2-multiple--1-vxlhv   0/1     Completed   0          4s
job-demo2-multiple--1-xjqvd   0/1     Completed   0          11s
job-demo2-multiple--1-z2zt8   0/1     Completed   0          13s


[root@k8s-master ~]# kubectl -n jobs-demo get job
NAME                 COMPLETIONS   DURATION   AGE
job-demo2-multiple   10/10         13s        28s

[root@k8s-master ~]# kubectl -n jobs-demo describe job job-demo2-multiple
<<<check the details...>>>>>
```

Jobs

Execute Job in parallel:

```
[root@k8s-master ~]# kubectl -n jobs-demo apply -f job-demo3-parallel.yaml

[root@k8s-master ~]# kubectl -n jobs-demo get pod

[[run this in a different Putty session]]
[root@k8s-master ~]# kubectl -n jobs-demo get pods -w


[root@k8s-master ~]# kubectl -n jobs-demo get job -w

[root@k8s-master ~]# kubectl -n jobs-demo describe job job-demo3-parallel
```

Jobs

Limit Time for Job Completion using ‘activeDeadlineSeconds’:

```
[root@k8s-master ~]# kubectl -n jobs-demo apply -f job-demo4-with-deadline.yaml

[root@k8s-master ~]# kubectl  -n jobs-demo get job

[[run this in a different Putty session]]
[root@k8s-master ~]# kubectl  -n jobs-demo get pods -w


[root@k8s-master ~]# kubectl -n jobs-demo get job -w

[root@k8s-master ~]# kubectl -n jobs-demo describe job job-demo4-deadline
<<<check the details...>>>>>
```

Jobs

Output of job-demo4-with-deadline Job:

```
[root@k8s-master ~]# kubectl -n jobs-demo get pods
NAME                          READY   STATUS      RESTARTS   AGE
job-demo4-deadline--1-5nmg8   0/1     Completed   0          49s
job-demo4-deadline--1-6k5sv   0/1     Completed   0          45s
job-demo4-deadline--1-9pxcg   0/1     Completed   0          51s
job-demo4-deadline--1-dlxwq   0/1     Completed   0          50s
job-demo4-deadline--1-fkjp5   0/1     Completed   0          42s
job-demo4-deadline--1-gt87z   0/1     Completed   0          44s
job-demo4-deadline--1-jlmcz   0/1     Completed   0          46s
job-demo4-deadline--1-n96xz   0/1     Completed   0          47s


[root@k8s-master ~]# kubectl -n jobs-demo describe job job-demo4-deadline | tail -5
  Normal   SuccessfulCreate  2m26s  job-controller  Created pod: job-demo4-deadline--1-jlmcz
  Normal   SuccessfulCreate  2m25s  job-controller  Created pod: job-demo4-deadline--1-6k5sv
  Normal   SuccessfulCreate  2m24s  job-controller  Created pod: job-demo4-deadline--1-gt87z
  Normal   SuccessfulCreate  2m22s  job-controller  Created pod: job-demo4-deadline--1-fkjp5
  Warning  DeadlineExceeded  2m21s  job-controller  Job was active longer than specified deadline
```

Clean up….

Clean up all the resources that was created….

```
[root@k8s-master ~]# kubectl -n jobs-demo delete -f job-demo1.yaml

[root@k8s-master ~]# kubectl -n jobs-demo delete -f job-demo2-multiple.yaml

[root@k8s-master ~]# kubectl -n jobs-demo delete -f job-demo3-parallel.yaml

[root@k8s-master ~]# kubectl -n jobs-demo delete -f job-demo4-with-deadline.yaml
```

CronJob

CronJob

As seen earlier, a regular Kubernetes job is a workload controller object that performs one or more finite tasks in a cluster. Regular Kubernetes Jobs can run multiple times if the number of completions declared in the YAML file is larger than one. However, each job instance runs either simultaneously with others or immediately after the previous one completes.

To schedule job instances for a later time, use the CronJob controller. The CronJob is a Kubernetes controller that creates Jobs on a repeating schedule. It utilizes the Cron scheduling format used in Linux for script and command scheduling.

Typical Kubernetes CronJob Use Cases:

- Regular Schedule Maintenance tasks (such as performing backups)
- Repetitive Batch tasks such as sending emails
- Repetitive Batch jobs to processing files and transfer to other endpoints
- Regular Shipping/Cleaning up of logs on schedule times

CronJobs

How to Configure CronJob

*	*	*	*	*
Minutes (0-59)	Hours (0-23)	Days in a month (1-31)	Months (1-12)	Weekdays (0-6) (Sunday to Saturday)

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
      name: cron-job-demo
spec:
      # run every two minutes...
      schedule: "*/2 * * * *"
```

CronJobs

Let’s run a simple demo:

```
[root@k8s-master|/root]# cd /root/new-add-on-Labs/cronjob

[root@k8s-master ~]# kubectl apply -f cronjob-demo1.yaml

[[run this in a different Putty session]]
[root@k8s-master ~]# kubectl get pod -w

[root@k8s-master ~]# kubectl describe pod cron-job-demo


[[run this in a different Putty session]]
[root@k8s-master ~]# kubectl get cronjob --watch
```

CronJobs

CronJob demo output….

```
[root@k8s-master|/root]# kubectl get cronjobs -w
NAME              SCHEDULE       SUSPEND    ACTIVE    LAST SCHEDULE    AGE
cron-job-demo   */1 * * * *   False     0        <none>          1s
cron-job-demo   */1 * * * *   False     1        0s              22s
cron-job-demo   */1 * * * *   False     0        1s              23s
cron-job-demo   */1 * * * *   False     0        1s              23s
------------
cron-job-demo   */1 * * * *   False     1        0s              82s
cron-job-demo   */1 * * * *   False     0        2s              84s
cron-job-demo   */1 * * * *   False     0        2s              84s


[root@k8s-master|/root]# kubectl get pods -w
NAME                            READY   STATUS    RESTARTS   AGE
cron-job-demo-28061927--1-j5mvx   0/1     Pending   0          0s
cron-job-demo-28061927--1-j5mvx   0/1     Pending   0          0s
cron-job-demo-28061927--1-j5mvx   0/1     ContainerCreating   0          0s
cron-job-demo-28061927--1-j5mvx   0/1     Completed           0          1s
-----------
cron-job-demo-28061928--1-qbk85   0/1     Pending             0          0s
cron-job-demo-28061928--1-qbk85   0/1     Pending             0          0s
cron-job-demo-28061928--1-qbk85   0/1     ContainerCreating   0          0s
cron-job-demo-28061928--1-qbk85   0/1     Completed           0          2s
```

Clean up….

Clean up all the resources that was created….

```
[root@k8s-master|/root/labs/yamls]# kubectl delete -f cronjob-demo1.yaml
```

Lifecycle of a Pod

Lifecycle of a Pod

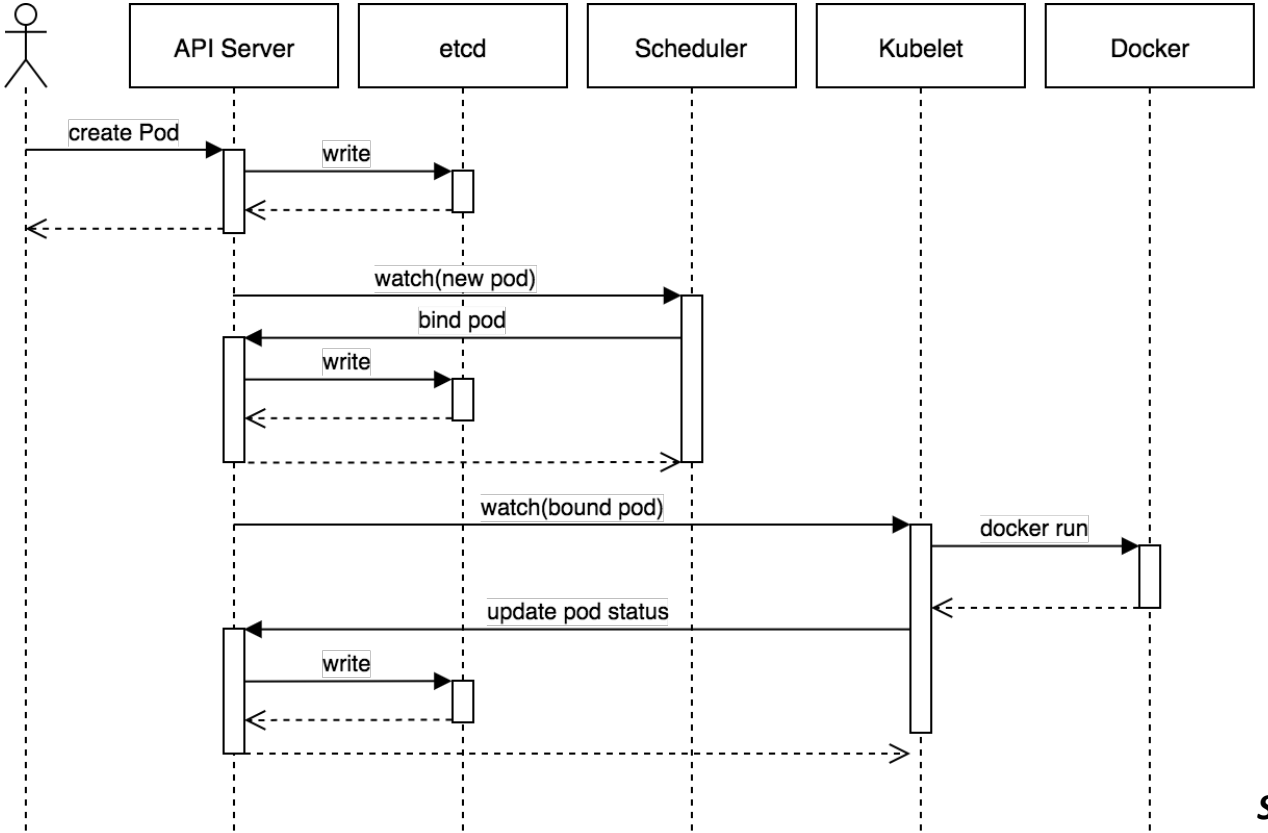
Understanding the lifecycle of Pods is important and it will help us while interacting and troubleshooting issues

States of a Pod

Through its lifecycle, a Pod can attain following states

- **Pending**: The pod is accepted by the Kubernetes system but its container(s) is/are not created yet.
- **Running**: The pod is scheduled on a node and all its containers are created and at-least one container is in Running state.
- **Succeeded**: All container(s) in the Pod have exited with status 0 and will not be restarted.
- **Failed**: All container(s) of the Pod have exited and at least one container has returned a non-zero status.
- **CrashLoopBackoff**: The container fails to start and is tried again and again.

Flow - Creation of a Pod



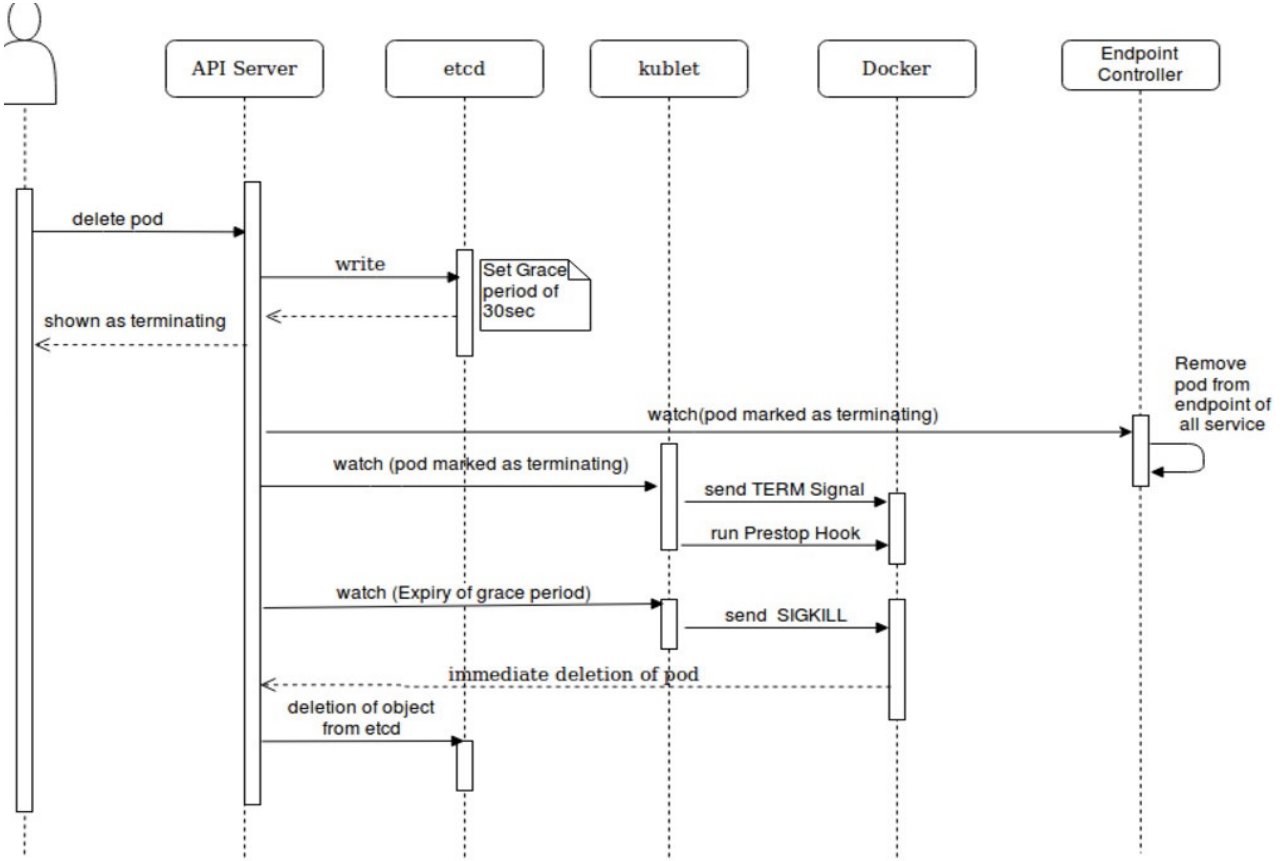
Source: heptio.com

Creation of a Pod

When you create a pod, the following sequence of events happens (summarized)

1. kubectl or any other API client submits the Pod definition to the API server.
2. The API server writes the Pod object to the etcd data store.
3. Once the write is successful, an acknowledgment is sent back to API server and to the client.
4. The API server now writes the change in state of etcd.
5. All Kubernetes components use watches to keep checking API server for relevant changes.
6. In this case, the kube-scheduler (via its watcher) sees that a new Pod object is created on API server but is not bound to any node.
7. kube-scheduler assigns a node to the pod and updates the API server.
8. This change is then propagated to the etcd data store. The API server also reflects this node assignment on its Pod object.
9. Kubelet on every node also runs watchers who keep watching API server. Kubelet on the destination node sees that a new Pod is assigned to it.
10. Kubelet starts the pod on its node by calling Docker and updates the container state back to the API server.
11. The API server persists the pod state into etcd.
12. Once etcd sends the acknowledgment of a successful write, the API server sends an acknowledgment back to kubelet indicating that the event is accepted.

Flow - Termination of a Pod



Source: heptio.com

Termination of a Pod

When you terminate a pod, the following sequence of events heppens (summarized)

1. kubectl or any other API client submits a command to delete a Pod.
2. The Pod object in the API server is updated with the time beyond which the Pod is considered “dead” (default of 30 seconds) along with the grace period.
3. The below actions happen in parallel:
   - The pod shows up as “Terminating” when listed in client commands.
   - When the Kubelet sees that a Pod has been marked as terminating because the time of 30 seconds has been set, it begins the pod shutdown process.
   - The endpoint controller watches the pod is about to be deleted and hence removes the pod from all the endpoints which were serviced by the pod.
4. If the pod has defined a preStop hook, it is invoked inside of the pod. If the preStop hook is still running after the grace period of 30 seconds expires, it then invoked with a small (2 second) extended grace period.
5. The processes in the Pod are sent the TERM signal.
6. When the grace period expires, any processes still running in the Pod are killed with SIGKILL.
7. The Kubelet will finish deleting the Pod on the API server by setting grace period 0 (immediate deletion). The Pod disappears from the API and is no longer visible from the client.

Init Containers

Init Container Concepts

- Init containers are those that will be first executed before the main ones are started/initiated
- A Pod can have more than one Init container
- These pods will be executed in order (instead of in parallel)
- All Init containers must complete successfully before the main containers can be started
- These properties gives us what exactly can be done when a pod comes up

https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

Uses of Init Container

- Load content
- Generate configuration (or certificates)
- Database initialization or migrations
- Waiting for other services to be up (e.g. test probe to backend)
- Avoid connection errors in main container because of depended services are not available.

Example #1 - Simple example…

Create a pod with two init-container and main application container….

```
[root@k8s-master|/root]# cd  /root

[root@k8s-master|/root]# wget http://172.20.11.97/repos/Kubernetes-1.22.2/new-add-on-Labs.tar

[root@k8s-master|/root]# tar xvf new-add-on-Labs.tar

[root@k8s-master|/root]# cd  /root/new-add-on-Labs/init-containers

[root@k8s-master|/root/new-add-on-Labs/init-containers]# cat init-containers-simple.yaml
apiVersion: v1
kind: Pod
metadata:
  name: init-containers-simple
  labels:
    app: init-containers-demo
spec:
  containers:
  - name: my-main-container
    image: registry.example.com:5000/busybox:lab
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-container-10-seconds
    image: registry.example.com:5000/busybox:lab
    command: ['sh', '-c', "echo Started init-container-10-seconds...; sleep 10; echo done with init-container-10-seconds..."]
  - name: init-container-another-20-seconds
    image: registry.example.com:5000/busybox:lab
    command: ['sh', '-c', "echo Started init-container-another-20-seconds...; sleep 10; echo done with init-container-another-20-seconds..."]
```

Simple example…

Check the details using below commands:

```
[root@k8s-master|/root]# kubectl create -f init-containers-simple.yaml

[root@k8s-master|/root]# kubectl  get pods -w

[root@k8s-master|/root]# kubectl  describe pod init-containers-simple

<<check the init container logs>>

[root@k8s-master|/root]# kubectl logs init-containers-simple -c init-container-10-seconds

[root@k8s-master|/root]# kubectl logs init-containers-simple -c init-container-20-seconds


* How long did it take for the main container to come online?

* How many “Init” containers were there?
```

Simple example…

Some standard outputs…

```
[root@k8s-master|/root]# kubectl get pods -w
NAME                                    READY   STATUS      RESTARTS   AGE
init-containers-simple                  0/1     Pending     0          0s
init-containers-simple                  0/1     Pending     0          0s
init-containers-simple                  0/1     Init:0/2    0          0s
init-containers-simple                  0/1     Init:1/2    0          11s
init-containers-simple                  0/1     PodInitializing   0          22s
init-containers-simple                  1/1     Running           0          23s


[root@k8s-master|/root]# kubectl logs init-containers-simple -c init-container-10-seconds
Started init-container-10-seconds...
done with init-container-10-seconds...

[root@k8s-master|/root]# kubectl logs init-containers-simple -c init-container-another-20-seconds
Started init-container-another-20-seconds...
done with init-container-another-20-seconds...

[root@k8s-master|/root]# kubectl  describe pod init-containers-simple


[root@k8s-master|/root]# kubectl delete  -f init-containers-simple.yaml
```

Example #2 – Web server with site initialization……

Create a pod with one init-container that updated the website and then starts the main nginx container….

```
[root@k8s-master|/root/new-add-on-Labs/init-containers]# cat website-preload-with-init-container.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-website1
spec:
  volumes:
  - name: www
  containers:
  - name: website1-container
    image: registry.example.com:5000/nginx:1.21.1-alpine
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html/
  initContainers:
  - name: clone-from-git-init-container
    image: registry.example.com:5000/alpine:lab
    #command: [ "sh", "-c", "apk add --no-cache git && git clone https://github.com/octocat/Spoon-Knife /www" ]
    command: [ "sh", "-c", "cd /www && wget http://172.20.13.125/Spoon-Knife/Spoon-Knife-main.tar && cd /www && tar xf Spoon-Knife-main.tar" ]
    volumeMounts:
    - name: www
      mountPath: /www/
```

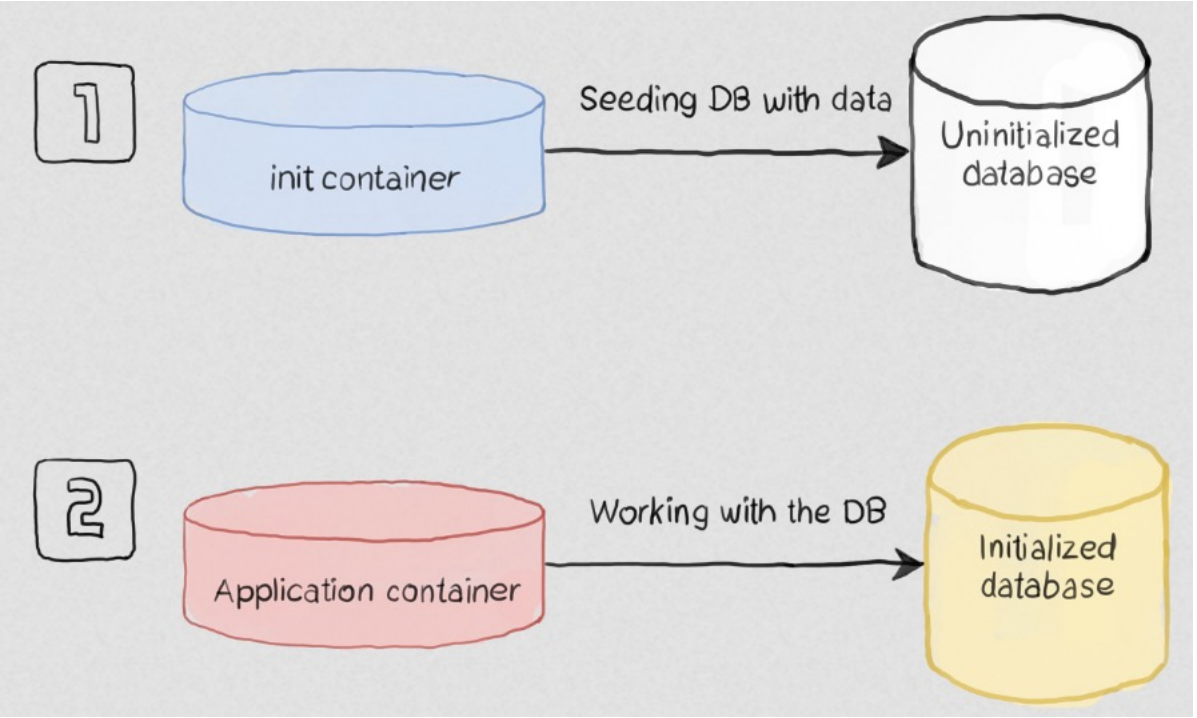
Web server example…

Check the details using below commands:

```
[root@k8s-master|/root]# kubectl create -f website-preload-with-init-container.yaml

[root@k8s-master|/root]# kubectl  get pods -w

[root@k8s-master|/root]# kubectl  describe pod my-website1

<<check the init container logs>>

[root@k8s-master|/root]# kubectl logs my-website1 -c website1-container

[root@k8s-master|/root]# kubectl logs my-website1 -c clone-from-git-init-container


* How long did it take for the main container to come online?

* How many “Init” containers were there?

[[You can check the website content using curl and pod-IP. You can also check with browser after you create a server and/or ingress
and exposed it. These tasks are left as an exercise to the students. Please try to do the same]]

[root@k8s-master|/root]# kubectl delete -f website-preload-with-init-container.yaml
```

Example #3 – Clone/Initialize MySQL DB….

We will make use of init-containers to clone/initialize the MySQL database when we perform the MySQL-HA lab



Creating PVs using NFS Server

Creating PVs

Let's create some PVs for our environment. In the lab, we will use **NFS storage as the backend for PVs**.

First create the **NFS shares**:

```
[root@ks8-master ~]$ mkdir -p /srv/nfs/user-vols/pv{1..10}

[root@ks8-master ~]$ mkdir -p /etc/exports.d

[root@ks8-master ~]$ cd /root/new-add-on-Labs/create-pvs

[root@ks8-master ~]$ cat create-nfs.sh

for pvnum in {1..10} ; do
    echo "\"/srv/nfs/user-vols/pv${pvnum}\" *(rw,sync,no_subtree_check,no_root_squash)" >> /etc/exports.d/user-vols.exports
    chown -R nfsnobody.nfsnobody  /srv/nfs/user-vols/pv${pvnum}
    chmod -R 777 /srv/nfs/user-vols/pv${pvnum}
done


[root@ks8-master ~]$ sh create-nfs.sh
```

Creating PVs

Restart NFS and confirm the NFS Shares are available:

```
[root@ks8-master ~]$ systemctl restart nfs-server

[root@ks8-master ~]$ exportfs
/srv/nfs/user-vols/pv1
                <world>
/srv/nfs/user-vols/pv2
                <world>
/srv/nfs/user-vols/pv3
                <world>
/srv/nfs/user-vols/pv4
                <world>
/srv/nfs/user-vols/pv5
                <world>
/srv/nfs/user-vols/pv6
                <world>
/srv/nfs/user-vols/pv7
                <world>
/srv/nfs/user-vols/pv8
                <world>
/srv/nfs/user-vols/pv9
                <world>
/srv/nfs/user-vols/pv10
                <world>
```

Creating PVs

Let's create some PVs for our environment. Create an empty directory to hold the manifest files:

```
[root@ks8-master ~]$ mkdir /root/pvs
```

Creating PVs

Create the manifest files:

```
[root@ks8-master ~]$ cat create-pvs.sh
for volume in pv{1..10} ; do
cat << EOF > /root/pvs/${volume}
{
  "apiVersion": "v1",
  "kind": "PersistentVolume",
  "metadata": {
    "name": "${volume}"
  },
  "spec": {
    "capacity": {
        "storage": "10Gi"
    },
    "accessModes": [ "ReadWriteOnce", "ReadWriteMany" ],
    "nfs": {
        "path": "/srv/nfs/user-vols/${volume}",
        "server": "k8s-master.podx.example.com"
    },
    "persistentVolumeReclaimPolicy": "Recycle"
  }
}
EOF
echo "Created def file for ${volume}";
done;
```

Note: change the podx to your pod-ID. It must look like k8s-master.podx.example.com

Creating PVs

Create the manifest files:

```
[root@ks8-master ~]$ sh create-pvs.sh


Created def file for pv1
Created def file for pv2
Created def file for pv3
Created def file for pv4
Created def file for pv5
Created def file for pv6
Created def file for pv7
Created def file for pv8
Created def file for pv9
Created def file for pv10
```

Creating PVs

Create the PVs using the manifest files as the OpenShift Admin user:

```
<<<Verify one of the file and make sure it has your POD number correctly>>>

[root@ks8-master ~]$ cat /root/pvs/pv5 | grep `hostname`

[[run the below command to finally create the PVs]]
[root@ks8-master ~]$ cat /root/pvs/* | kubectl create -f -
```

You will see the output as below:

```
persistentvolume "pv1" created
persistentvolume "pv10" created
persistentvolume "pv2" created
persistentvolume "pv3" created
persistentvolume "pv4" created
persistentvolume "pv5" created
persistentvolume "pv6" created
persistentvolume "pv7" created
persistentvolume "pv8" created
persistentvolume "pv9" created
```

Creating PVs

List the PVs and verify. This output should match as below.

Please confirm this as many labs depend on these PVs that you just created.

```
[root@ks8-master ~]$ kubectl get pv | egrep "(NAME|^pv)"

NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
pv1    10Gi       RWO            Recycle          Available                                   10s
pv10   10Gi       RWO            Recycle          Available                                   10s
pv2    10Gi       RWO            Recycle          Available                                   10s
pv3    10Gi       RWO            Recycle          Available                                   10s
pv4    10Gi       RWO            Recycle          Available                                   10s
pv5    10Gi       RWO            Recycle          Available                                   10s
pv6    10Gi       RWO            Recycle          Available                                   10s
pv7    10Gi       RWO            Recycle          Available                                   10s
pv8    10Gi       RWO            Recycle          Available                                   10s
pv9    10Gi       RWO            Recycle          Available                                   10s
```

Replicated Stateful Application with MySQL

Credits: *https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/*

This example is modified to run on the Kubernets environment using kubectl commands and reduced CPU/Memory requirements. All are requested to read this article to get a deeper understanding of the lab. In the interest of time, we will be covering only important parts of this article.

Replicated Stateful Application with MySQL

In this lab you will learn how to run a replicated stateful application using a StatefulSet controller. The example is a **MySQL single-master topology with multiple slaves** running asynchronous replication.

Note that this is not a production configuration. In particular, MySQL settings remain on insecure defaults to keep the focus on general patterns for running stateful applications in Kubernetes.

We will see the following scenarios:

- Deploy a replicated MySQL topology with a StatefulSet controller.
- Send MySQL client traffic and access it back.
- Observe resistance to pod outages.

Deploy Replicated MySQL

The example MySQL deployment consists of a ConfigMap, two Services, and a StatefulSet:

```
[root@master ~]$ kubectl create ns mysql-ha


[root@master ~]$ cd /root/new-add-on-Labs/mysql-HA

[root@master mysql-HA]$ ls -l *yaml
-rw-r--r--. 1 podxuser podxuser  260 Jun  5 11:21 mysql-configmap.yaml
-rw-r--r--. 1 podxuser podxuser  530 Jun  5 11:21 mysql-services.yaml
-rw-r--r--. 1 podxuser podxuser 5459 Jun 17 09:54 mysql-statefulset.yaml
```

Deploy Replicated MySQL - `ConfigMap`

This ConfigMap provides **`my.cnf`** overrides that let you independently control configuration on the MySQL master and slaves. In this case, you want the master to be able to serve replication logs to slaves and you want slaves to reject any writes that don't come via replication.

There's nothing special about the ConfigMap itself that causes different portions to apply to different Pods. Each Pod decides which portion to look at as it's initializing, based on information provided by the StatefulSet controller.

Create the ConfigMap from the following YAML configuration file:

```
[root@master mysql-HA]$ kubectl -n mysql-ha  create -f mysql-configmap.yaml
configmap "mysql" created
```

Deploy Replicated MySQL - `Services`

The Headless Service provides a home for the DNS entries that the StatefulSet controller creates for each Pod that's part of the set. Because the Headless Service is named **`mysql`**, the Pods are accessible by resolving **`<pod-name>.mysql`** from within any other Pod in the same OpenShift cluster and namespace.

The Client Service, called **`mysql-read`**, is a normal Service with its own cluster IP that distributes connections across all MySQL Pods that report being Ready. The set of potential endpoints includes the MySQL master and all slaves.

Note that only read queries can use the load-balanced Client Service. Because there is only one MySQL master, clients should connect directly to the MySQL master Pod (through its DNS entry within the Headless Service) to execute writes.

Create the Services from the following YAML configuration file:

```
[root@master mysql-HA]$ kubectl -n mysql-ha create -f mysql-services.yaml
service "mysql" created
service "mysql-read" created
```

Deploy Replicated MySQL - `StatefulSet`

This manifest uses a variety of techniques for managing stateful Pods as part of a StatefulSet.

The StatefulSet controller starts Pods one at a time, in order by their ordinal index. It waits until each Pod reports being Ready before starting the next one.

In addition, the controller assigns each Pod a unique, stable name of the form **`<statefulset-name>-<ordinal-index>`**. In this case, that results in Pods named **`mysql-0`**, **`mysql-1`**, and **`mysql-2`**.

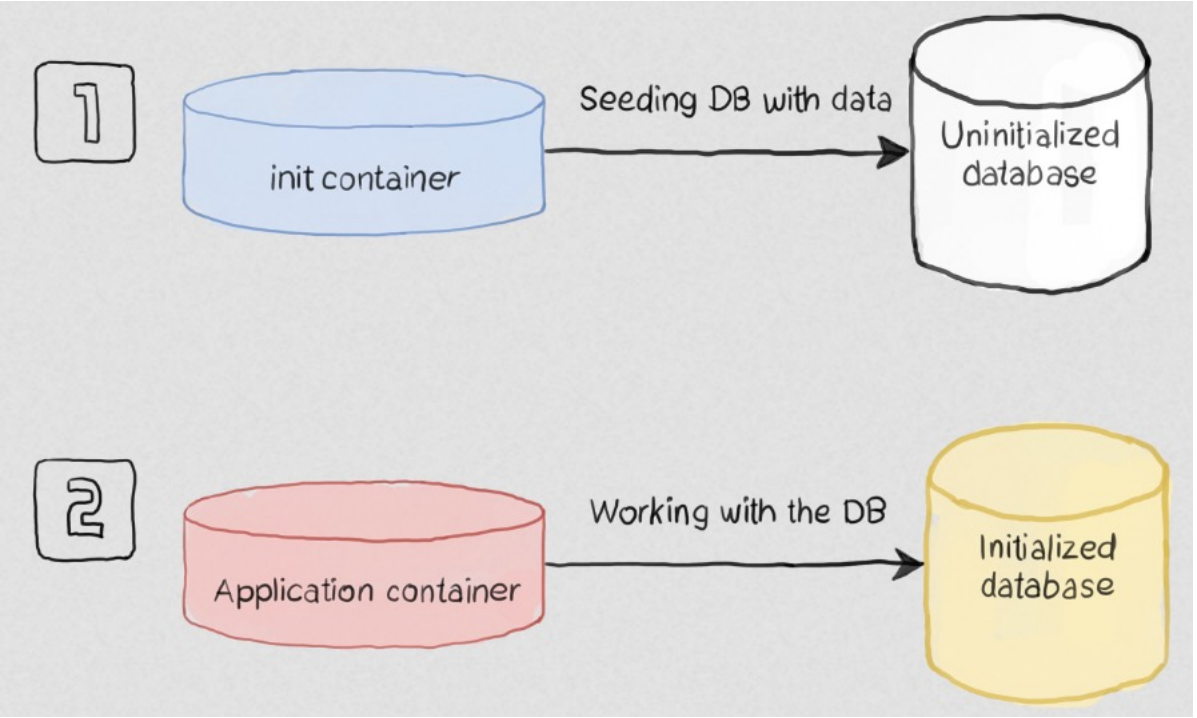
The Pod template in the above StatefulSet manifest takes advantage of these properties to perform orderly startup of MySQL replication.

Before starting any of the containers in the Pod spec, the Pod first runs any Init Containers in the order defined.

The first Init Container, named init-mysql, generates special MySQL config files based on the ordinal index.

Initialize/Configure/Clone MySQL DB….

The MySQL stateful set will make use of init-containers to initialize/configure/clone the MySQL database.



Deploy Replicated MySQL - `StatefulSet`

The script in the **`init-mysql`** container also applies either **`master.cnf or slave.cnf`** from the ConfigMap by copying the contents into **`conf.d`**.

Because the example topology consists of a **single MySQL master and any number of slaves**, the script simply assigns ordinal **`0`** to be the master, and everyone else to be slaves. Combined with the StatefulSet controller's deployment order guarantee, this ensures the MySQL master is Ready before creating slaves, so they can begin replicating

The second Init Container, named **`clone-mysql`**, performs a clone operation on a slave Pod the first time it starts up on an empty **`PersistentVolume`**. That means it copies all existing data from another running Pod, so its local state is consistent enough to begin replicating from the master.

Deploy Replicated MySQL - `StatefulSet`

MySQL itself does not provide a mechanism to do this, so the example uses a popular open-source tool called **XtraBackup**. During the clone, the source MySQL server might suffer reduced performance. To minimize impact on the MySQL master, the script instructs each Pod to clone from the Pod whose ordinal index is one lower. This works because the StatefulSet controller always ensures **Pod N** is Ready before starting **Pod N+1.**

Deploy Replicated MySQL - `StatefulSet` – Starting replication

After the Init Containers complete successfully, the regular containers run. The MySQL Pods consist of a **`mysql`** container that runs the actual **`mysqld`** server, and an **`xtrabackup`** container that acts as a sidecar.

The **`xtrabackup`** sidecar looks at the cloned data files and determines if it's necessary to initialize MySQL replication on the slave. If so, it waits for **`mysqld`** to be ready and then executes the **`CHANGE MASTER TO`** and **`START SLAVE`** commands with replication parameters extracted from the XtraBackup clone files.

Once a slave begins replication, it remembers its MySQL master and reconnects automatically if the server restarts or the connection dies. Also, because slaves look for the master at its stable DNS name (**`mysql-0.mysql`**), they automatically find the master even if it gets a new Pod IP due to being rescheduled.

Lastly, after starting replication, the **`xtrabackup`** container listens for connections from other Pods requesting a data clone. This server remains up indefinitely in case the StatefulSet scales up, or in case the next Pod loses its PersistentVolumeClaim and needs to redo the clone.

Deploy Replicated MySQL - `Creating StatefulSet`

Create the StatefulSet from the following YAML configuration file:

```
[root@master mysql-HA]$ kubectl -n mysql-ha create -f mysql-statefulset.yaml
statefulset "mysql" created

[root@master mysql-HA]$ kubectl -n mysql-ha get pods
NAME      READY     STATUS            RESTARTS   AGE
mysql-0   2/2       Running           0          30s
mysql-1   0/2       PodInitializing   0          13s

[root@master mysql-HA]$ kubectl -n mysql-ha get pods
NAME      READY     STATUS     RESTARTS   AGE
mysql-0   2/2       Running    0          44s
mysql-1   2/2       Running    1          27s
mysql-2   0/2       Init:1/2   0          7s

[[you can  keep watching all the stages discussed in the earlier slides using the '-w' option]]

[root@master mysql-HA]$ kubectl -n mysql-ha get pods -o wide -w
```

Deploy Replicated MySQL - `Creating StatefulSet`

You can see the progress as below:

```
[root@master1 ~]$ kubectl -n mysql-ha get pods -o wide -w
NAME      READY     STATUS    RESTARTS   AGE       IP           NODE
mysql-0   1/2       Running   0          18s       10.1.2.113   node1.example.com
mysql-0   2/2       Running   0         19s       10.1.2.113   node1.example.com
mysql-1   0/2       Pending   0         0s        <none>    <none>
mysql-1   0/2       Pending   0         0s        <none>    node2.example.com
mysql-1   0/2       Init:0/2   0         0s        <none>    node2.example.com
mysql-1   0/2       Init:1/2   0         3s        10.1.4.120   node2.example.com
mysql-1   0/2       Init:1/2   0         4s        10.1.4.120   node2.example.com
mysql-1   0/2       PodInitializing   0         12s       10.1.4.120   node2.example.com
mysql-1   1/2       Running   0         13s       10.1.4.120   node2.example.com
mysql-1   0/2       Error     0         15s       10.1.4.120   node2.example.com
mysql-1   1/2       Running   1         16s       10.1.4.120   node2.example.com
mysql-1   2/2       Running   1         18s       10.1.4.120   node2.example.com
mysql-2   0/2       Pending   0         0s        <none>    <none>
mysql-2   0/2       Pending   0         0s        <none>    node1.example.com
mysql-2   0/2       Init:0/2   0         0s        <none>    node1.example.com
mysql-2   0/2       Init:1/2   0         3s        10.1.2.114   node1.example.com
mysql-2   0/2       Init:1/2   0         4s        10.1.2.114   node1.example.com
mysql-2   0/2       PodInitializing   0         13s       10.1.2.114   node1.example.com
mysql-2   1/2       Running   0         14s       10.1.2.114   node1.example.com
mysql-2   2/2       Running   0         18s       10.1.2.114   node1.example.com
```

Deploy Replicated MySQL - `Creating StatefulSet`

Finally, after sometime, you should see all 3 Pods become Running:

```
[root@master mysql-HA]$ kubectl -n mysql-ha get pods -o wide -w
NAME      READY     STATUS    RESTARTS   AGE       IP          NODE
mysql-0   2/2       Running   0          1m        10.1.2.96   node2.example.com
mysql-1   2/2       Running   1          1m        10.1.0.89   node1.example.com
mysql-2   2/2       Running   0          1m        10.1.2.97   node2.example.com
```

Deploy Replicated MySQL - `Accessing the Database`

You can send test queries to the MySQL master (hostname **`mysql-0.mysql`**) by running a temporary container with the **`mysql:5.7`** image and running the **`mysql client`** binary. We will create a **`'test'`** database and insert two rows into it.

```
[root@master mysql-HA]$ cat create.sh
kubectl -n mysql-ha run mysql-client --image=mysql:5.7 -i --rm --restart=Never -- mysql -u root  -h mysql-
0.mysql <<EOF
CREATE DATABASE test;
CREATE TABLE test.messages (message VARCHAR(250));
INSERT INTO test.messages VALUES ('hello');
EOF

[root@master mysql-HA]$ sh create.sh
.
.
test
.
.
```

Deploy Replicated MySQL - `Accessing the Database`

Use the hostname **`mysql-read`** to send test queries to any server that reports being **Ready state**:

You should see the two rows that you inserted earlier.

```
[root@master mysql-HA]$ cat select.sh
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never -- mysql -u root -h mysql-read -e
"SELECT * FROM test.messages"


[root@master mysql-HA]$ sh select.sh
+---------+
| message |
+---------+
| Hello   |
| World   |
+---------+
```

Deploy Replicated MySQL - `Select across servers`

To demonstrate that the **`mysql-read`** Service distributes connections across servers, you can run **`SELECT @@server_id`** in a loop. You can use **Ctrl+C** to stop

```
[root@master mysql-HA]$ cat dist-select0.sh
kubectl run mysql-client-loop-0 --image=mysql:5.7 -i -t --rm --restart=Never -- bash -ic "while sleep 1; do mysql -u root -h
mysql-read -e 'SELECT @@server_id,NOW()'; done"

[root@master mysql-HA]$ sh dist-select0.sh
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         102 | 2018-06-17 07:05:47 |
+-------------+---------------------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         101 | 2018-06-17 07:05:48 |
+-------------+---------------------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         100 | 2018-06-17 07:05:51 |
+-------------+---------------------+
```

Deploy Replicated MySQL - `Select across servers`

We will use another select query which will show the data also -- **keep it running for next lab]]**

```
[root@master mysql-HA]$ sh dist-select-message.sh
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         102 | 2018-06-17 07:05:47 |
+-------------+---------------------+
+---------+
| message |
+---------+
| Hello   |
| World   |
+---------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         101 | 2018-06-17 07:05:48 |
+-------------+---------------------+
+---------+
| message |
+---------+
| Hello   |
| World   |
+---------+
```

Deploy Replicated MySQL - `Simulating Pod outage`

To demonstrate the increased availability of reading from the pool of slaves instead of a single server, keep the **`SELECT @@server_id`** loop from previous lab running while you delete a POD.

```
[root@master ~]$ kubectl -n mysql-ha delete pod mysql-2
pod "mysql-2" deleted

[[Keep the below command running in another Putty Session]]
[root@master ~]$ kubectl -n mysql-ha get pods -o wide -w
```

Since the POD is now deleted, you will see that the select loop only return queries from the first two pods with ID-100 and ID-101 as below:

```
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         100 | 2018-06-17 07:15:51 |
+-------------+---------------------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         101 | 2018-06-17 07:15:52 |
+-------------+---------------------+
```

Deploy Replicated MySQL - `Simulating Pod recovery`

If you have been watching the pod status in another Putty session you will see that the deleted POD went though a cycle like:

`Terminating → Pending → Init → PodInitializing → Running`

It will recover itself in sometime and the queries will start being served from POD-2 which we had deleted.

The StatefulSet controller notices that no **`mysql-2`** Pod exists anymore, and creates a new one with the same name and linked to the same **`PersistentVolumeClaim`**. In the select loop Putty session, you should see server **`ID 102`** disappear from the loop output for a while and then return on its own.

```
podx-mysql-ha   mysql-2   2/2       Terminating   0         1h        10.1.2.97   node2.example.com
podx-mysql-ha   mysql-2   0/2       Terminating   0         1h        10.1.2.97   node2.example.com
podx-mysql-ha   mysql-2   0/2       Terminating   0         1h        10.1.2.97   node2.example.com
podx-mysql-ha   mysql-2   0/2       Terminating   0         1h        10.1.2.97   node2.example.com
podx-mysql-ha   mysql-2   0/2       Pending   0         0s        <none>    <none>
podx-mysql-ha   mysql-2   0/2       Pending   0         0s        <none>    node2.example.com
podx-mysql-ha   mysql-2   0/2       Init:0/2   0         0s        <none>    node2.example.com
podx-mysql-ha   mysql-2   0/2       Init:1/2   0         4s        10.1.2.98   node2.example.com
podx-mysql-ha   mysql-2   0/2       PodInitializing   0         5s        10.1.2.98   node2.example.com
podx-mysql-ha   mysql-2   1/2       Running   0         7s        10.1.2.98   node2.example.com
podx-mysql-ha   mysql-2   2/2       Running   0         11s       10.1.2.98   node2.example.com
```

OpenShift - Clean up

Clean up all the objects:

```
[root@master ~]$ kubectl delete namespace mysql-ha
```

WordPress and MySQL Example

Credits: *https://github.com/brennv/wordpress-ex*

This example is modified to run in our lab.

WordPress and MySQL - An example WordPress set-up for Kubernetes

This is a set of configuration files adapted from OpenShift/origin examples that work with Kubernetes to create a WordPress application with MySQL as the backend database.

This lab show how to use all that we have learned to setup a small blog site using WordPress. We will do the following in this lab:

- Create a project
- Create PVC for MySQL and WordPress
- Launch A MySQL POD
- Expose MySQL as a service so that WordPress can access and create a database in it
- Launch a WordPress POD
- Expose the WordPress Service
- Expose the WordPress Service as a Route so that we can access it from outside the cluster
- Finally, start Blogging

This lab will show how to put all that we have learned to good use.

Deploy WordPress and MySQL - Create a WordPress Project

Login and create the the project and explore the files.

```
[root@master ~]$ kubectl create ns wordpress

[root@master ~]$ cd /root/new-add-on-Labs/wordpress-mysql-example

[root@master wordpress-mysql-example]$ ls -l *yaml
-rw-r--r--. 1 podxuser podxuser 219 Jun  5 11:21 create-route.yaml
-rw-r--r--. 1 podxuser podxuser 730 Jun  5 11:21 pod-mysql.yaml
-rw-r--r--. 1 podxuser podxuser 891 Jun  5 11:21 pod-wordpress.yaml
-rw-r--r--. 1 podxuser podxuser 182 Jun  5 11:21 pvc-mysql.yaml
-rw-r--r--. 1 podxuser podxuser 200 Jun  5 11:21 pvc-wp.yaml
-rw-r--r--. 1 podxuser podxuser 244 Jun  5 11:21 pv-mysql.yaml
-rw-r--r--. 1 podxuser podxuser 264 Jun  5 11:21 pv-wp.yaml
-rw-r--r--. 1 podxuser podxuser 275 Jun  5 11:21 service-mysql.yaml
-rw-r--r--. 1 podxuser podxuser 340 Jun  5 11:21 service-wp.yaml
```

Deploy WordPress and MySQL - Establish Persistent Volumes Claims

Create claims for storage for each of the Pods. For this claim to work, the administrator should have already provisioned the storage and they PVs should be in ‘Available’ Status.

Create the MySQL PVC and the WP PVC and check the status; it should be in ‘Bound’ Status.:

```
[root@master wordpress-mysql-example]$ kubectl -n wordpress create -f pvc-mysql.yaml
persistentvolumeclaim "claim-mysql" created

[root@master wordpress-mysql-example]$ kubectl -n wordpress create -f pvc-wp.yaml
persistentvolumeclaim "claim-wp" created

[root@master wordpress-mysql-example]$ kubectl -n wordpress get pvc
NAME          STATUS    VOLUME            CAPACITY   ACCESS MODES   STORAGECLASS   AGE
claim-mysql   Bound     pv12              10G        RWO                           1m
claim-wp      Bound     pv110             10Gi       RWX                           1m
```

Deploy WordPress and MySQL - Launch MySQL Pod

Launch the MySQL Pod. After a few moments, MySQL will be running and accessible via the pod's IP address.

```
[root@master wordpress-mysql-example]$ kubectl -n wordpress create -f pod-mysql.yaml
pod "mysql" created

[root@master wordpress-mysql-example]$ kubectl -n wordpress get pods -o wide -w
NAME      READY     STATUS              RESTARTS   AGE       IP           NODE
mysql     0/1       ContainerCreating   0          8s        <none>       node2.example.com
mysql     1/1       Running             0          7m        10.1.2.119   node2.example.com

[[This may take some time if the images needs to be pulled.]]
```

Deploy WordPress and MySQL - Create MySQL Service

We don't know what the IP address will be and we wouldn't want to hard-code that value in any pod that wants access to MySQL. For this we will use a service.

Create a service in front of MySQL that allows other pods to connect to it by name. This allows the pod to access MySQL via a service name instead of hard-coded host address

```
[root@master wordpress-mysql-example]$ kubectl -n wordpress create -f service-mysql.yaml
service "mysql" created

[root@master wordpress-mysql-example]$ kubectl -n wordpress get svc
NAME      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
mysql     ClusterIP   172.30.123.117   <none>        3306/TCP   10s

[root@master wordpress-mysql-example]$ kubectl -n wordpress describe service mysql
Name:              mysql
Namespace:         podx-wordpress
Labels:            name=mysql
Selector:          name=mysql
Type:              ClusterIP
IP:                172.30.123.117
Port:              <unset>  3306/TCP
TargetPort:        3306/TCP
Endpoints:         10.1.2.119:3306
```

Deploy WordPress and MySQL - Launch the WordPress Pod

Check the "**pod-wordpress.yaml**" file. We use the MySQL service defined previously in our "**pod-wordpress.yaml**" file. The variable **WORDPRESS_DB_HOST** is set to the name of our MySQL service.

Because the WordPress pod and MySQL service are running in the same namespace, we can reference the service by name. We can also access a service in another namespace by using the name and namespace: **mysql.another_namespace**. The fully qualified name of the service would also work: **mysql.<namespace>.svc.cluster.local**

Launch the Wordpress pod and make sure that it is running:

```
[root@master wordpress-mysql-example]$ kubectl -n wordpress create -f pod-wordpress.yaml
pod "wordpress" created

[root@master wordpress-mysql-example]$ kubectl -n wordpress get pods -o wide -w
NAME        READY     STATUS              RESTARTS   AGE       IP           NODE
mysql       1/1       Running             0          24m       10.1.2.119   node2.example.com
wordpress   0/1       ContainerCreating   0          3s        <none>       node1.example.com
wordpress   1/1       Running             0          3s        10.1.0.109   node1.example.com

[[Note: Check the README file for Adjusting Security Constraints for Third-Party Docker Images. This is taken
care in the lab and the WordPress Pod should start without any issues.]]
```

Deploy WordPress and MySQL - Create the WordPress service

Just like the MySQL Pod, we don't want to use IPs to access the WP site. We will use the names to access it.

Create a service in front of WordPress Pod so that we can expose it the outside world by name:

```
[root@master wordpress-mysql-example]$ kubectl -n wordpress create -f service-wp.yaml
service "wpfrontend" created

[root@master wordpress-mysql-example]$ kubectl -n wordpress get svc
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP                     PORT(S)        AGE
mysql        ClusterIP      172.30.123.117   <none>                          3306/TCP       21m
wpfrontend   NodePort   172.30.144.41    172.29.255.160,172.29.255.160   80:30965/TCP   11s
```

Deploy WordPress and MySQL - Expose the WordPress service

As a last step, we need to expose the WordPress service that we created in the previous slide as a ingress so that it can be accessed externally via a browser from outside the Kubernetes cluster.

Check the manifest file and make sure that the namespace points to you POD namespace. In this case for **podX**; it should point to "**podX**" as below:

```
[root@master wordpress-mysql-example]$ cat create-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: wordpress-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: "wordpress.podX.example.com"
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: wpfrontend
            port:
              number: 80
```

← Change this as per your POD Name

Deploy WordPress and MySQL - Expose the WordPress service

Create the route and check the details:

```
[root@master wordpress-mysql-example]$ kubectl -n wordpress create -f create-ingress.yaml
route "wpfrontend" created

[root@master wordpress-mysql-example]$ kubectl -n wordpress get ingress
NAME                CLASS   HOSTS                        ADDRESS   PORTS   AGE
wordpress-ingress   nginx   wordpress.podX.example.com             80      5s
```

Point you web browser to `http://wordpress.podx.example.com/wp-admin/` access the WordPress blog site and to start the configuration.

Once it is configured, you can use the `http://wordpress.podx.example.com/`

Your own WordPress Blog site…

wpfrontend-pod0-wordpress.apps.example.com/w 50%

Welcome

Welcome to the famous five-minute WordPress installation process! Just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

Information needed

Please provide the following information. Don't worry, you can always change these settings later.

Site Title MyBlog

Username admin

Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods, and the @ symbol.

Password admin Hide

Very weak

Important: You will need this password to log in. Please store it in a secure location.

Confirm Password Confirm use of weak password

Your Email binnyr@example.com

Double-check your email address before continuing.

Search Engine Visibility Discourage search engines from indexing this site

It is up to search engines to honor this request.

Install WordPress

OpenShift - Clean up

Clean up all the objects:

```
[root@master ~]$ kubectl delete ns wordpress
```

Basic Kubernetes Troubleshooting

Kubernetes Troubleshooting – CPU and Memory Resource issues

In Kubernetes, a Container is guaranteed to have as much CPU or Memory as it requests, but is not allowed to use more CPU or Memory than its limit. Sometimes, the assigned values may not be thought after properly and defining a request that is too big for your Nodes will give you issues.

The CPU or Memory request for a Pod is the sum of the CPU or Memory requests for all the Containers in the Pod. Likewise, the CPU or Memory limit for a Pod is the sum of the CPU or Memory limits for all the Containers in the Pod.

Pod scheduling is based on requests. A Pod is scheduled to run on a Node only if the Node has enough CPU **and** Memory resources available to satisfy the Pod CPU request.

In this lab, we will create a Pod that has a CPU and Memory request so big that it exceeds the capacity of any Node in your cluster and see what happens.

CPU and Memory Resource issues

```
[root@k8s-master|/root]# cd /root/labs/yamls/

[root@k8s-master yamls]# kubectl create -f webapp-cpu-memory.yaml

[root@k8s-master|/root/labs/yamls]# kubectl get pods  -o wide
NAME                              READY     STATUS    RESTARTS   AGE       IP        NODE
simple-webapp-69cf679cfd-7hqlc    0/1       Pending   0          48s       <none>    <none>
simple-webapp-69cf679cfd-rqdtv    0/1       Pending   0          48s       <none>    <none>


[root@k8s-master|/root/labs/yamls]# kubectl logs simple-webapp-69cf679cfd-7hqlc

[root@k8s-master|/root/labs/yamls]# kubectl describe pod  simple-webapp-69cf679cfd-7hqlc
.
.
Events:
  Type     Reason            Age               From               Message
  ----     ------            ----              ----               -------
  Warning  FailedScheduling  8s (x14 over 2m)  default-scheduler  No nodes are available that match all of the
predicates: Insufficient cpu (3), Insufficient memory (3), PodToleratesNodeTaints (1).
```

CPU and Memory Resource issues

```
[root@k8s-master|/root/labs/yamls]# cat webapp-cpu-memory.yaml

>> Edit the file so that the resources request/limits are as below:


resources:
           requests:
              memory: "50Mi"
              cpu: "50m"
           limits:
              memory: "64Mi"
              cpu: "100m”

((Tip:: You can use “kubectl edit deploy/simple-webapp” and edit directly))
```

CPU and Memory Resource issues

```
>> Now apply the new definition

[root@k8s-master|/root/labs/yamls]# kubectl apply -f webapp-cpu-memory.yaml
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl
apply deployment "simple-webapp" configured

[root@k8s-master|/root/labs/yamls]# kubectl get pods  -o wide
NAME                              READY     STATUS    RESTARTS   AGE       IP           NODE
simple-webapp-598b8746dd-8q76w    1/1       Running   0          47s       10.244.1.5   worker-node1
simple-webapp-598b8746dd-dtg7x    1/1       Running   0          48s       10.244.2.6   worker-node2

[root@k8s-master|/root/labs/yamls]# kubectl describe pod  simple-webapp-598b8746dd-8q76w
Events:
  Type    Reason                 Age   From                   Message
  ----    ------                 ----  ----                   -------
  Normal  Scheduled              1m    default-scheduler      Successfully assigned simple-webapp-598b8746dd-8q76w to worker-node1
  Normal  Created                1m    kubelet, worker-node1  Created container
  Normal  Started                1m    kubelet, worker-node1  Started container
```

Checking Resource Usage

```
[root@k8s-master|/root]# kubectl top node
NAME            CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
worker-node2    51m          2%        665Mi           35%
k8s-master      147m         7%        1069Mi          57%
worker-node1    51m          2%        716Mi           38%

[root@k8s-master|/root]# kubectl get nodes --no-headers | awk '{print $1}' | xargs -I {} sh -c 'echo {};
kubectl describe node {} | grep Allocated -A 5 | grep -ve Event -ve Allocated -ve percent -ve -- ; echo'
k8s-master
CPU Requests  CPU Limits  Memory Requests  Memory Limits
  810m (40%)    0 (0%)      110Mi (5%)       170Mi (9%)

worker-node1
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  50m (2%)      100m (5%)   50Mi (2%)        64Mi (3%)

worker-node2
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  50m (2%)      100m (5%)   50Mi (2%)        64Mi (3%)

[root@k8s-master|/root]# kubectl top pod
NAME                              CPU(cores)   MEMORY(bytes)
simple-webapp-598b8746dd-8q76w    0m           0Mi
simple-webapp-598b8746dd-dtg7x    0m           0Mi
```

Check HTML Guide

CPU and Memory Resource issues

```
Clean up the pods for the net lab...


[root@k8s-master|/root/labs/yamls]# kubectl delete deploy/simple-webapp

deployment "simple-webapp" deleted
```

Node Labelling and Draining

Node Labelling

In Kubernetes, you can constrain a pod to only be able to run on particular nodes. There are several ways to do this, and the recommended approaches all use label selectors to make the selection. There are some circumstances where you may want more control on a node where a pod lands, e.g. to ensure that a pod ends up on a "production" node, or a node with an SSD attached to it, etc.

In this lab we will see how to label and constrain a pod to a particular node. We will also see in this lab how to watch and troubleshoot for such errors.

Deploying a pod with “nodeSelector”

```
[root@k8s-master|/root/labs/yamls]# ls -l simple-webapp.yaml
-rw-r--r-- 1 root root 440 Apr 30 15:52 simple-webapp.yaml

[root@k8s-master|/root/labs/yamls]# kubectl create -f simple-webapp.yaml
deployment "simple-webapp" created

[root@k8s-master|/root]# kubectl get pods  -o wide -w
NAME                              READY     STATUS    RESTARTS   AGE       IP        NODE
simple-webapp-6cbcb44cdc-jh96x    0/1       Pending   0          18s       <none>    <none>
simple-webapp-6cbcb44cdc-sfzd7    0/1       Pending   0          18s       <none>    <none>

[root@k8s-master|/root/labs/yamls]# kubectl describe pod  simple-webapp-6cbcb44cdc-jh96x
Events:
  Type     Reason            Age               From               Message
  ----     ------            ----              ----               -------
  Warning  FailedScheduling  6s (x13 over 2m)  default-scheduler  No nodes are available that match all of the
predicates: MatchNodeSelector (3), PodToleratesNodeTaints (1).
```

Checking the value of “nodeSelector”

```
[root@k8s-master|/root/labs/yamls]# kubectl describe pod  simple-webapp-6cbcb44cdc-jh96x | grep Node-Selectors

Node-Selectors:  nodetype=prod


[root@k8s-master|/root/labs/yamls]# cat simple-webapp.yaml  | grep -A 3 nodeSelector

      nodeSelector:
         nodetype: prod
```

Checking the node labels

```
[root@k8s-master|/root/labs/yamls]# kubectl get nodes --show-labels

NAME           STATUS    ROLES     AGE       VERSION   LABELS
k8s-master     Ready     master    2h        v1.8.5
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=k8s-master,node-role.kubernetes.io/master=
worker-node1   Ready     <none>    2h        v1.8.5
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=worker-node1
worker-node2   Ready     <none>    2h        v1.8.5
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=worker-node2
```

Applying the label “prod” to “worker-node2”

```
[root@k8s-master|/root/labs/yamls]# kubectl label nodes worker-node2 nodetype=prod

node "worker-node2" labeled


[root@k8s-master|/root/labs/yamls]# kubectl get nodes worker-node2 --show-labels

NAME           STATUS    ROLES     AGE       VERSION   LABELS
worker-node2   Ready     <none>    2h        v1.8.5
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=worker-node2,nodetype=prod
```

Check the pod status

```
[root@k8s-master|/root/labs/yamls]# kubectl get pods  -o wide

NAME                              READY     STATUS          RESTARTS   AGE       IP           NODE
simple-webapp-6cbcb44cdc-jh96x    0/1       ErrImagePull    0          15m       10.244.2.8   worker-node2
simple-webapp-6cbcb44cdc-sfzd7    0/1       ErrImagePull    0          15m       10.244.2.7   worker-node2
```

At least now it got scheduled to a node which has the label **“prod”**

Again check and see why it is failing…

```
[root@k8s-master|/root/labs/yamls]# kubectl describe pod  simple-webapp-6cbcb44cdc-jh96x | grep -A 5 Events

Events:
  Type     Reason                 Age                From                   Message
  ----     ------                 ----               ----                   -------
  Warning  FailedScheduling       8m (x37 over 18m)  default-scheduler      No nodes are available that match all of the
predicates: MatchNodeSelector (3), PodToleratesNodeTaints (1).
  Normal   SuccessfulMountVolume  4m                 kubelet, worker-node2  MountVolume.SetUp succeeded for volume
"default-token-5kkx4"
  Normal   Pulling                2m (x4 over 4m)    kubelet, worker-node2  pulling image
"registry.example.com:5000/simple-webapp:v11"
  Warning  Failed                 2m (x4 over 4m)    kubelet, worker-node2  Failed to pull image
"registry.example.com:5000/simple-webapp:v11": rpc error: code = Unknown desc = Error: image simple-webapp:v11 not found
  Warning  FailedSync             2m (x10 over 4m)   kubelet, worker-node2  Error syncing pod
  Normal   BackOff                2m (x6 over 4m)    kubelet, worker-node2  Back-off pulling image
"registry.example.com:5000/simple-webapp:v11"
```

Lets check the registry…

```
[root@k8s-master|/root/labs/yamls]# curl -s  -X GET http://registry.example.com:5000/v2/_catalog  | grep simple-webapp

[root@k8s-master|/root/labs/yamls]# curl -s -X GET http://registry.example.com:5000/v2/_catalog?n=200 | grep "simple-
webapp"

....,"simple-webapp”,...

[root@k8s-master|/root/labs/yamls]# curl -s -X GET http://registry.example.com:5000/v2/simple-webapp/tags/list

{"name":"simple-webapp","tags":["v1","v2"]}
```

Edit the deployment directly and fix the image “tag” issue

```
[root@k8s-master|/root/labs/yamls]# kubectl edit deploy/simple-webapp

deployment "simple-webapp" edited



[root@k8s-master|/root/labs/yamls]# kubectl get pods  -o wide

NAME                              READY     STATUS    RESTARTS   AGE       IP            NODE
simple-webapp-79b549cc4-9pc9f     1/1       Running   0          35s       10.244.2.9    worker-node2
simple-webapp-79b549cc4-kxhzq     1/1       Running   0          34s       10.244.2.10   worker-node2
```

Node Draining

In Kubernetes, you can use **`kubectl drain`** to safely evict all of your pods from a node before you perform maintenance on the node (e.g. kernel upgrade, hardware maintenance, etc.). Safe evictions allows the pod's containers to gracefully terminate.

In this lab, we will drain "worker-node2" and then add it back to the cluster.

```
Note: If you are using a single node cluster these labs may not work properly.

If you have at least one worker node, you can taint the master node as below:

# kubectl taint nodes --all node-role.kubernetes.io/master-
```

Let’s deploy a new application to the cluster.

We will deploy a new application and watch its behavior. If you have completed the previous lab successfully, you should have the “`simple-webapp`” in “`Running`” state on worker-node2

```
[root@k8s-master|/root/labs/yamls]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# kubectl create -f another-application.yaml
deployment "another-webapp" created

[root@k8s-master|/root/labs/yamls]# kubectl get pods  -o wide -w
NAME                              READY     STATUS    RESTARTS   AGE       IP            NODE
another-webapp-65d86896cd-9hxj5   1/1       Running   0          2m        10.244.2.11   worker-node2
another-webapp-65d86896cd-bcbng   1/1       Running   0          2m        10.244.2.12   worker-node2
another-webapp-65d86896cd-nbzvl   1/1       Running   0          2m        10.244.1.6    worker-node1
another-webapp-65d86896cd-v2829   1/1       Running   0          2m        10.244.1.7    worker-node1
another-webapp-65d86896cd-wg4bf   1/1       Running   0          2m        10.244.1.9    worker-node1
another-webapp-65d86896cd-wmsdx   1/1       Running   0          2m        10.244.1.8    worker-node1
simple-webapp-79b549cc4-9pc9f     1/1       Running   0          19m       10.244.2.9    worker-node2
simple-webapp-79b549cc4-kxhzq     1/1       Running   0          19m       10.244.2.10   worker-node2
```

“Cordon” off worker-node2

As a first step in draining a node, we will “cordon” the node so that new pods are NOT scheduled on it.

```
[root@k8s-master|/root/labs/yamls]# kubectl cordon worker-node2
node "worker-node2" cordoned


[root@k8s-master|/root/labs/yamls]# kubectl get nodes
NAME           STATUS                     ROLES     AGE       VERSION
k8s-master     Ready                      master    3h        v1.8.5
worker-node1   Ready                      <none>    3h        v1.8.5
worker-node2   Ready,SchedulingDisabled   <none>    3h        v1.8.5
```

Deploy one more application and see it behavior...

We will one more application and watch its behavior.
Please keep "`kubectl get pods  -o wide -w`" in another Putty session

```
[root@k8s-master|/root/labs/yamls]# cd /root/labs/yamls/

[root@k8s-master|/root/labs/yamls]# kubectl create -f new-application.yaml
deployment "new-webapp" created

[root@k8s-master|/root/labs/yamls]# kubectl get pods  -o wide -w
NAME                              READY     STATUS    RESTARTS   AGE       IP            NODE
another-webapp-65d86896cd-9hxj5   1/1       Running   0          2m        10.244.2.11   worker-node2
.
simple-webapp-79b549cc4-kxhzq     1/1       Running   0          19m       10.244.2.10   worker-node2
new-webapp-65d86896cd-fwz25   1/1       Running   0         2s        10.244.1.10   worker-node1
new-webapp-65d86896cd-zn5lc   1/1       Running   0         3s        10.244.1.15   worker-node1
new-webapp-65d86896cd-9mdj8   1/1       Running   0         3s        10.244.1.14   worker-node1
new-webapp-65d86896cd-dzx5t   1/1       Running   0         4s        10.244.1.12   worker-node1
new-webapp-65d86896cd-mlbbn   1/1       Running   0         5s        10.244.1.11   worker-node1
new-webapp-65d86896cd-4d2p6   1/1       Running   0         6s        10.244.1.13   worker-node1
```

“Drain” worker-node2 node from the cluster…

Now we will “drain” worker-node2.
Please keep watching the “`kubectl get pods -o wide -w`” in another Putty session

```
[root@k8s-master|/root/labs/yamls]# kubectl drain worker-node2 --ignore-daemonsets --delete-local-data
node "worker-node2" already cordoned
WARNING: Ignoring DaemonSet-managed pods: kube-flannel-ds-ldjr4, kube-proxy-sjrzd;
Deleting pods with local storage: monitoring-grafana-5bccc9f786-p5dvp, monitoring-influxdb-85cb4985d4-6bh7t
pod "simple-webapp-79b549cc4-kxhzq" evicted
pod "simple-webapp-79b549cc4-9pc9f" evicted
pod "monitoring-influxdb-85cb4985d4-6bh7t" evicted
pod "monitoring-grafana-5bccc9f786-p5dvp" evicted
pod "another-webapp-65d86896cd-9hxj5" evicted
pod "another-webapp-65d86896cd-bcbng" evicted
node "worker-node2" drained

>>> All the old monitoring data will be lost <<<
```

“Drain” worker-node2 node from the cluster…

What just happened?

```
[root@k8s-master|/root]# kubectl get pods  -o wide -w
NAME                              READY     STATUS    RESTARTS   AGE       IP            NODE
another-webapp-65d86896cd-nbzvl   1/1       Running   0          14m       10.244.1.6    worker-node1
another-webapp-65d86896cd-phjtr   1/1       Running   0          1m        10.244.1.17   worker-node1
another-webapp-65d86896cd-r8jgk   1/1       Running   0          1m        10.244.1.19   worker-node1
another-webapp-65d86896cd-v2829   1/1       Running   0          14m       10.244.1.7    worker-node1
another-webapp-65d86896cd-wg4bf   1/1       Running   0          14m       10.244.1.9    worker-node1
another-webapp-65d86896cd-wmsdx   1/1       Running   0          14m       10.244.1.8    worker-node1
new-webapp-65d86896cd-4d2p6       1/1       Running   0          5m        10.244.1.13   worker-node1
new-webapp-65d86896cd-9mdj8       1/1       Running   0          5m        10.244.1.14   worker-node1
new-webapp-65d86896cd-dzx5t       1/1       Running   0          5m        10.244.1.12   worker-node1
new-webapp-65d86896cd-fwz25       1/1       Running   0          5m        10.244.1.10   worker-node1
new-webapp-65d86896cd-mlbbn       1/1       Running   0          5m        10.244.1.11   worker-node1
new-webapp-65d86896cd-zn5lc       1/1       Running   0          5m        10.244.1.15   worker-node1
simple-webapp-79b549cc4-mw7c6     0/1       Pending   0          1m        <none>        <none>
simple-webapp-79b549cc4-ncsdh     0/1       Pending   0          1m        <none>        <none>
```

Node Draining

Since the worker-node2 is out of the cluster:

- you should now "stop" the "kubelet" service
- you can now do you maintenance worker (update/patch/ etc)
- once it is completed, you can reboot

In an cloud provider environment, you can delete the node or add a new node back to the cluster (typical with cloud providers).

However, in a cloud provider environment, it is recommended to first add a new node, label it correctly so that all pods that will be evicted have new node and we do not have a production outage like we just saw. Finally, we can cordon and delete the old node (rather than patching, etc)

Reboot the worker-node2

```
[root@worker-node2 ~]# systemctl stop kubelet

[root@worker-node2 ~]# systemctl status kubelet

[root@worker-node2 ~]# init 6
Connection to node2 closed by remote host.
Connection to node2 closed.
```

Check and enable scheduling on worker-node2

Please keep watching the "`kubectl get pods  -o wide -w`" in another Putty session

```
[root@worker-node2 ~]# uptime
 16:13:24 up 2 min,  1 user,  load average: 3.21, 2.91, 1.99

[root@worker-node2 ~]# systemctl status kubelet

[root@k8s-master|/root/labs/yamls]# kubectl get nodes
NAME           STATUS                     ROLES     AGE       VERSION
k8s-master     Ready                      master    3h        v1.8.5
worker-node1   Ready                      <none>    3h        v1.8.5
worker-node2   Ready,SchedulingDisabled   <none>    3h        v1.8.5


[root@k8s-master|/root/labs/yamls]# kubectl uncordon worker-node2
node "worker-node2" uncordoned

[root@k8s-master|/root/labs/yamls]# kubectl get nodes
NAME           STATUS    ROLES     AGE       VERSION
k8s-master     Ready     master    3h        v1.8.5
worker-node1   Ready     <none>    3h        v1.8.5
worker-node2   Ready     <none>    3h        v1.8.5
```

Check the status if the pods…

```
[root@k8s-master|/root]# kubectl get pods  -o wide -w
NAME                                READY     STATUS    RESTARTS   AGE       IP            NODE
another-webapp-65d86896cd-nbzvl     1/1       Running   0          27m       10.244.1.6    worker-node1
another-webapp-65d86896cd-phjtr     1/1       Running   0          14m       10.244.1.17   worker-node1
another-webapp-65d86896cd-r8jgk     1/1       Running   0          14m       10.244.1.19   worker-node1
another-webapp-65d86896cd-v2829     1/1       Running   0          27m       10.244.1.7    worker-node1
another-webapp-65d86896cd-wg4bf     1/1       Running   0          27m       10.244.1.9    worker-node1
another-webapp-65d86896cd-wmsdx     1/1       Running   0          27m       10.244.1.8    worker-node1
new-webapp-65d86896cd-4d2p6         1/1       Running   0          18m       10.244.1.13   worker-node1
new-webapp-65d86896cd-9mdj8         1/1       Running   0          18m       10.244.1.14   worker-node1
new-webapp-65d86896cd-dzx5t         1/1       Running   0          18m       10.244.1.12   worker-node1
new-webapp-65d86896cd-fwz25         1/1       Running   0          18m       10.244.1.10   worker-node1
new-webapp-65d86896cd-mlbbn         1/1       Running   0          18m       10.244.1.11   worker-node1
new-webapp-65d86896cd-zn5lc         1/1       Running   0          18m       10.244.1.15   worker-node1
simple-webapp-79b549cc4-mw7c6       1/1       Running   0          14m       10.244.2.14   worker-node2
simple-webapp-79b549cc4-ncsdh       1/1       Running   0          14m       10.244.2.13   worker-node2
```

What is the status of “`new-webapp`” and “`another-webapp`”. Did it rebalance?

Sometimes you will see issues with the old deployment..

If you are in a situation, when the application like “simple-webapp” is still not scheduling on the node after the ”reboot” for a long time, you can either redeploy or scale to ‘0’ and back. Please exercise **caution** while doing this when you have multiple node with the same label as it will cause outage. You can always delete a pod and let the scheduler do the job in such cases.

```
[root@k8s-master|/root]# kubectl scale --replicas=0 deploy/simple-webapp
deployment "simple-webapp" scaled


[root@k8s-master|/root]# kubectl scale --replicas=2 deploy/simple-webapp
deployment "simple-webapp" scaled
```

For other application Pods that are only scheduled on one node, you can use ‘scale’ option to reduce to 50% (for a two node cluster) and scale up again and **hopefully** it will rebalance. Again, exercise **caution.**

```
[root@k8s-master|/root/labs/yamls]# kubectl scale --replicas=3 deploy/new-webapp
deployment "new-webapp" scaled

[root@k8s-master|/root/labs/yamls]# kubectl scale --replicas=6 deploy/new-webapp
deployment "new-webapp" scaled
```

Horizontal Pod Autoscaler

Credits: *https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/*

Horizontal Pod Autoscaler

Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization.

For this lab to work you will need **Heapster** monitoring; it needs to be deployed in the cluster as Horizontal Pod Autoscaler uses it to collect metrics. Please make sure that the Heapster is working correctly.

Horizontal Pod Autoscaler - Step One: Run & expose php-apache server

To demonstrate Horizontal Pod Autoscaler we will use a custom docker image based on the php-apache image.

[[Please make sure that gcr.io/google_containers/hpa-example container image is present on node1 & node2]]

First, we will start a deployment running the image and expose it as a service:

```
## Install metrics-server is needed. It is a must ##

[root@k8s-master|/root]# cd /root/labs/
[root@k8s-master|/root]# kubectl apply -f metrics-server-0.3.6-1.8+/

## Start the HPA demo pods ##

[root@k8s-master|/root]# cd /root/labs/yamls/

[root@k8s-master|/root]# kubectl apply -f php-apache-hpa-demo.yaml
deployment.apps/php-apache-hpa-demo created
service/php-apache-hpa-svc created
```

Horizontal Pod Autoscaler - Step Two: Create Horizontal Pod Autoscaler

Now that the server is running, we will create the autoscaler using kubectl autoscale.

The following command will create a Horizontal Pod Autoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache deployment we created in the first step of these instructions.

Roughly speaking, HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 50% (since each pod requests 200 milli-cores by kubectl run, this means average CPU usage of 100 milli-cores).

```
[root@k8s-master|/root]# kubectl autoscale deployment php-apache-hpa-demo --cpu-percent=50 --min=2 --max=10
deployment "php-apache" autoscaled
```

Horizontal Pod Autoscaler - Step Two: Create Horizontal Pod Autoscaler

We may check the current status of autoscaler by running:

```
[root@k8s-master|/root]# kubectl get hpa
NAME          REFERENCE                 TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache    Deployment/php-apache     0% / 50%   1         10        1          1m
```

Please note that the current CPU consumption is 0% as we are not sending any requests to the server (the CURRENT column shows the average across all the pods controlled by the corresponding deployment). You may have to wait for a minute or two.

Horizontal Pod Autoscaler - Step Three: Increase load

Now, we will see how the autoscaler reacts to increased load. We will start a container, and send an infinite loop of queries to the php-apache service (please run it in a different terminal):

```
[root@k8s-master|/root]# kubectl run -i --tty load-generator --image=busybox /bin/sh

If you don't see a command prompt, try pressing enter.

# while true; do wget -q -O- http://php-apache-hpa-svc.default.svc.cluster.local; done

OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
```

Horizontal Pod Autoscaler - Step Three: Increase load

From another Putty session check the status:

```
[root@k8s-master|/root]# kubectl get hpa
NAME          REFERENCE                TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
php-apache    Deployment/php-apache    72% / 50%   1         10        2          7m
[root@k8s-master|/root]#

[root@k8s-master|/root]# kubectl get hpa
NAME          REFERENCE                TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
php-apache    Deployment/php-apache    360% / 50%   1         10        2          8m
[root@k8s-master|/root]#


OR you can use 'watch'
[root@k8s-master ~]# watch  kubectl get hpa
```

Horizontal Pod Autoscaler - Step Three: Increase load

Here, CPU consumption has increased to 360% of the request. As a result, the deployment was resized to 2 replicas:

```
[root@k8s-master|/root]# kubectl get deployment php-apache
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   2         2         2            2           12m

[[after two more minutes of waiting]]

[root@k8s-master|/root]# kubectl get deployment php-apache
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   4         4         4            4           14m

[root@k8s-master|/root]# kubectl get hpa
NAME         REFERENCE                TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache    119% / 50%   1         10        8          15m
```

Note Sometimes it may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example. For our lab it may stay at 8

Horizontal Pod Autoscaler - Step Three: Increase load

Check the pods:

```
[root@k8s-master|/root]# kubectl get pods | grep php
php-apache-6b749bb748-2wl5j       1/1       Running   0          56s
php-apache-6b749bb748-459fc       1/1       Running   0          19m
php-apache-6b749bb748-48xth       1/1       Running   0          4m
php-apache-6b749bb748-cjsz6       1/1       Running   0          56s
php-apache-6b749bb748-fvt8z       1/1       Running   0          8m
php-apache-6b749bb748-hmhvr       1/1       Running   0          4m
php-apache-6b749bb748-qhnts       1/1       Running   0          56s
php-apache-6b749bb748-rmc7w       1/1       Running   0          56s
```

Note Sometimes it may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example. For our lab it may stay at 8

Horizontal Pod Autoscaler - Step Four: Stop load

We will finish our example by stopping the user load. In the terminal where we created the container with busybox image, terminate the load generation by typing <Ctrl> + C.

Then we will verify the result state (after a minute or so):

```
[root@k8s-master|/root]# kubectl get hpa
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache    0% / 50%   1         10        1          20m


[root@k8s-master|/root]# kubectl get deployment php-apache
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   1         1         1            1           24m
```

Here CPU utilization dropped to 0, and so HPA autoscaled the number of replicas back down to 1.Note autoscaling the replicas may take a few minutes.

Kubernetes Summary – what we learned

- Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services
- Support and facilitates both declarative configuration and automation
- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks
- Automatic bin packing
- Self-healing
- Secret and configuration management
- Remember, Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system

Beginning of a new journey…

What we have tried to do:

- Helped you jumpstart your container journey
- Tried to cover most important building blocks
- Tried to give you an easy-to-follow material
- Tried to give you ample hand-on lab exercise
- Tried to cover both Docker & Kubernetes to make realize the power of container and container orchestration.
- Tried to cover Red Hat OpenShift

What we want you do:

- Feedback - what to add/change/remove - what was good or bad
- Feedback - What do you want as a next level in this series of course
- Spread the word around about this training
- Inform us about other projects who use Docker, Kubernetes & OpenShift, etc.
- Keep us updated on what you do in your projects
- Share use cases with us