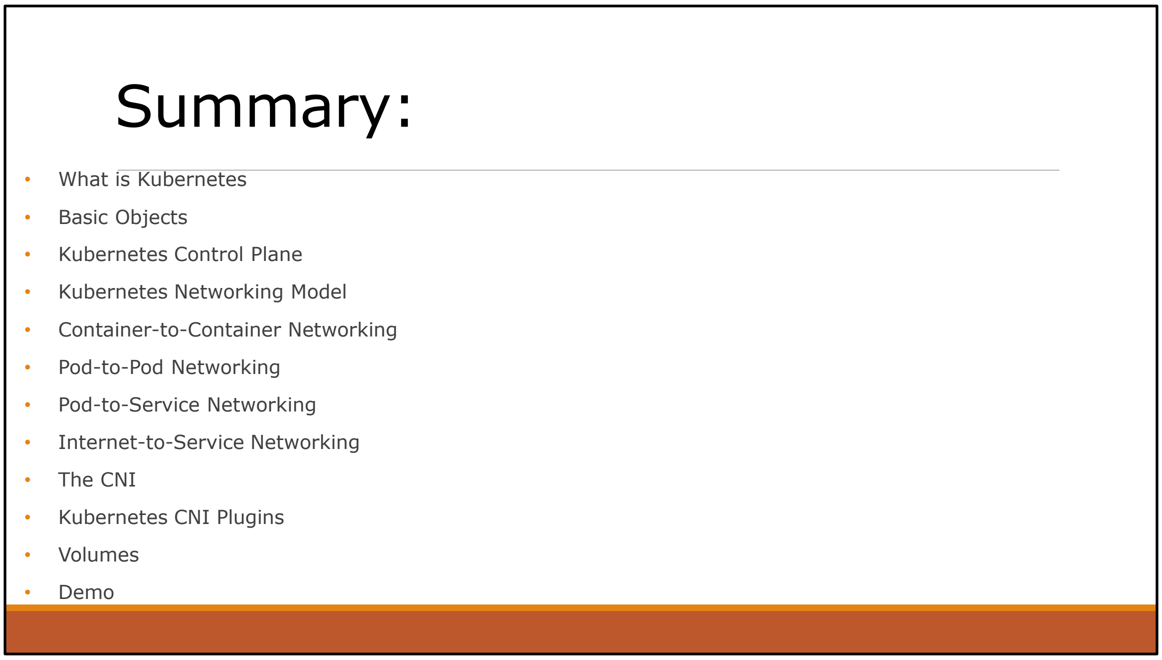
Kubernetes: An Introduction

Gianluca Reali



Summary:

- What is Kubernetes
- Basic Objects
- Kubernetes Control Plane
- Kubernetes Networking Model
- Container-to-Container Networking
- Pod-to-Pod Networking
- Pod-to-Service Networking
- Internet-to-Service Networking
- The CNI
- Kubernetes CNI Plugins
- Volumes
- Demo

What is Kubernetes

- Open source and Production-Grade container orchestration platform
- Google spawns billions of containers per week with these systems
- Based on Borg and Omega
- Google donated the Kubernetes project to the newly formed Cloud Native Computing Foundation (CNCF) in 2015.

"κυβερνήτης" (kubernetes) is Greek for "pilot" or "helmsman of ship"

What is Kubernetes

With Kubernetes you can:

Orchestrate containers across multiple hosts.

Control and automate application deployments and updates

Mount and add storage to run stateful apps

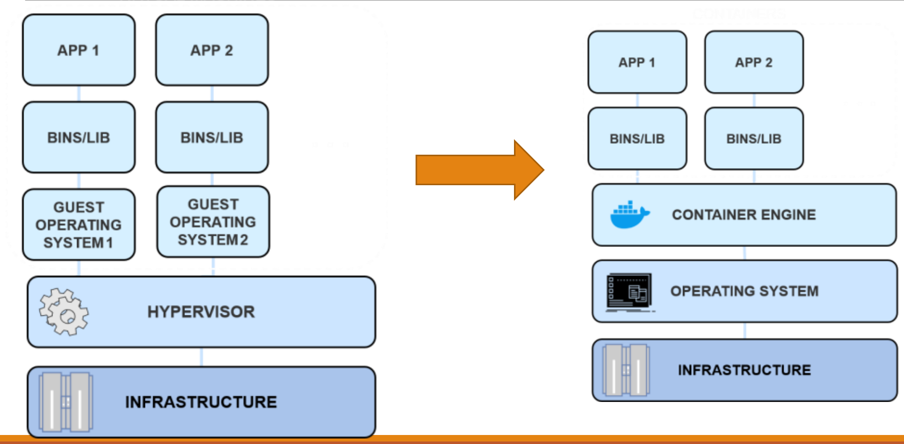
Scale containerized applications and their resources on the fly

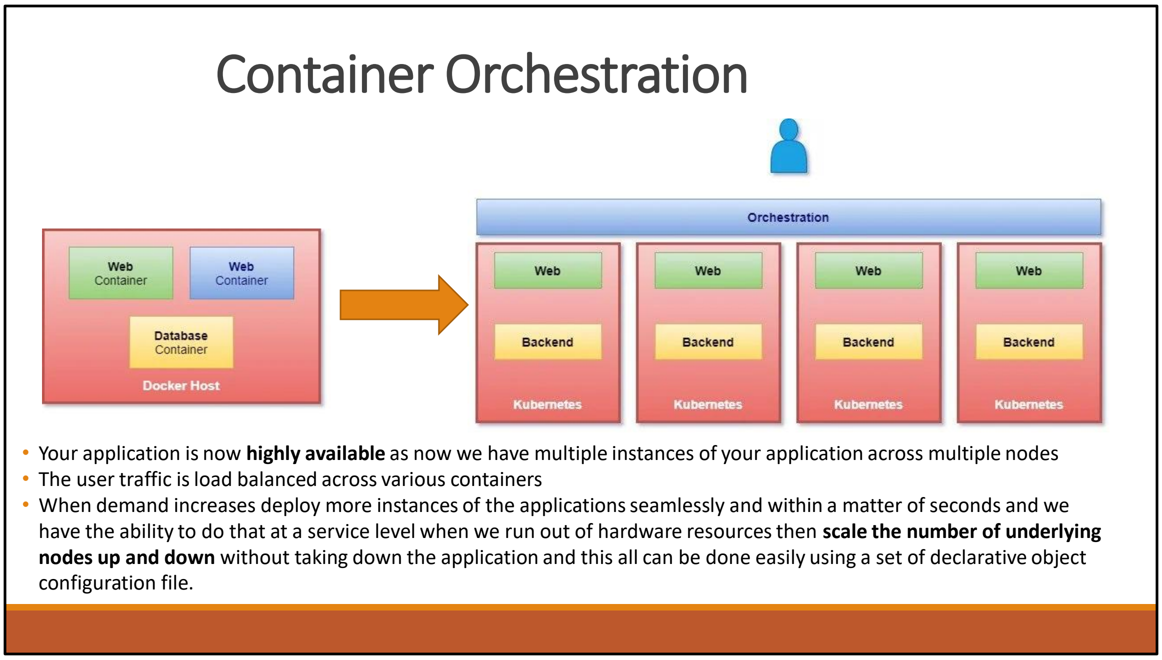
- All services within Kubernetes are natively Load Balanced.
- Can scale up and down dynamically.

Declaratively manage services (applications are always running the way you intended them to run)

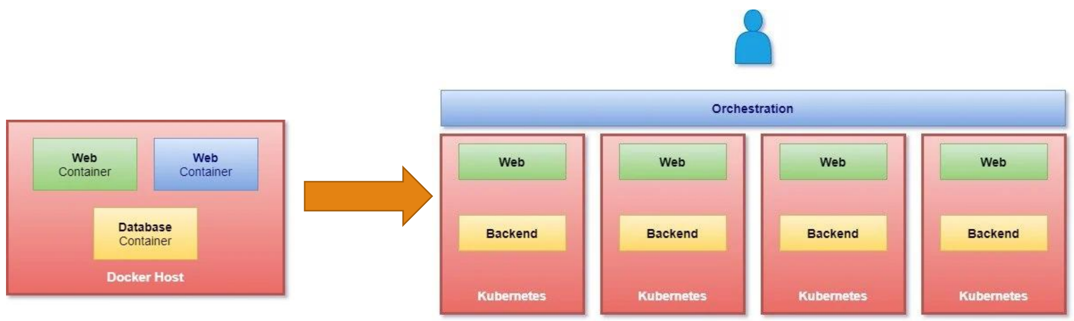
Health-check and self-heal your apps with autoplacement, autorestart, autoreplication, and autoscaling

… what is a container?



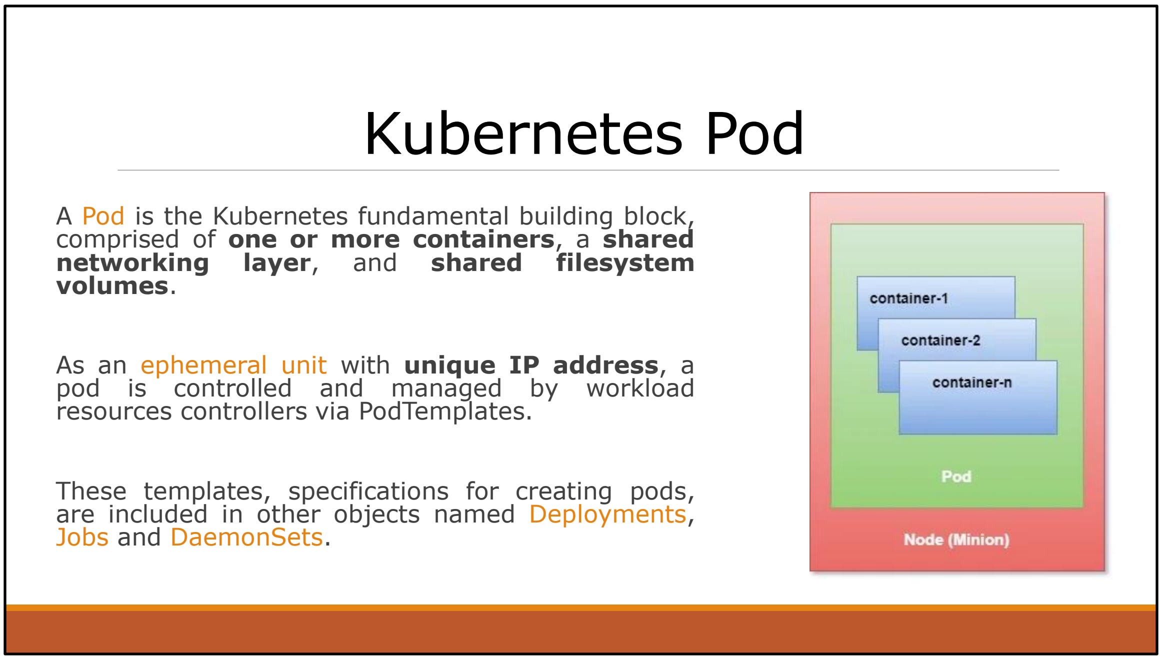


Container Orchestration



- Your application is now **highly available** as now we have multiple instances of your application across multiple nodes
- The user traffic is load balanced across various containers
- When demand increases deploy more instances of the applications seamlessly and within a matter of seconds and we have the ability to do that at a service level when we run out of hardware resources then **scale the number of underlying nodes up and down** without taking down the application and this all can be done easily using a set of declarative object configuration file.

It is possible that your application from container 1 is dependent on some other application from another container such as database, message, logging service in the production environment. You may also need the ability to scale up the number of containers during peak time, for example I am sure you must be familiar with Amazon sale during holidays when they have a bunch of extra offers on all products. In such case they need to **scale up** their resources for applications to be able to handle more number of users. Once the festive offer is finished then they would again need to **scale down** the amount of containers with applications. To enable this functionality we need an underlying platform with a set of resources and capabilities. The platform needs to **orchestrate** the connectivity between the containers and automatically scale up or down based on the load. This while process of deploying and managing containers is known as **container orchestration Kubernetes is thus a container orchestration technology** used to orchestrate the deployment and management of hundreds and thousands of containers in a cluster environment. There are multiple similar technologies available today, docker has it's own orchestration software i.e. Docker Swarm, Kubernetes from Google and Mesos from Apache.

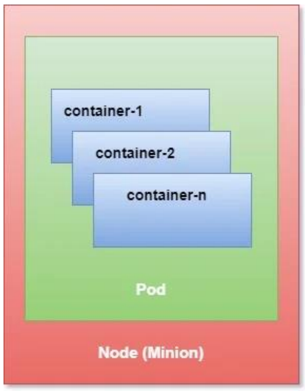


Kubernetes Pod

A Pod is the Kubernetes fundamental building block, comprised of **one or more containers**, a **shared networking layer**, and **shared filesystem volumes**.

As an ephemeral unit with **unique IP address**, a pod is controlled and managed by workload resources controllers via PodTemplates.

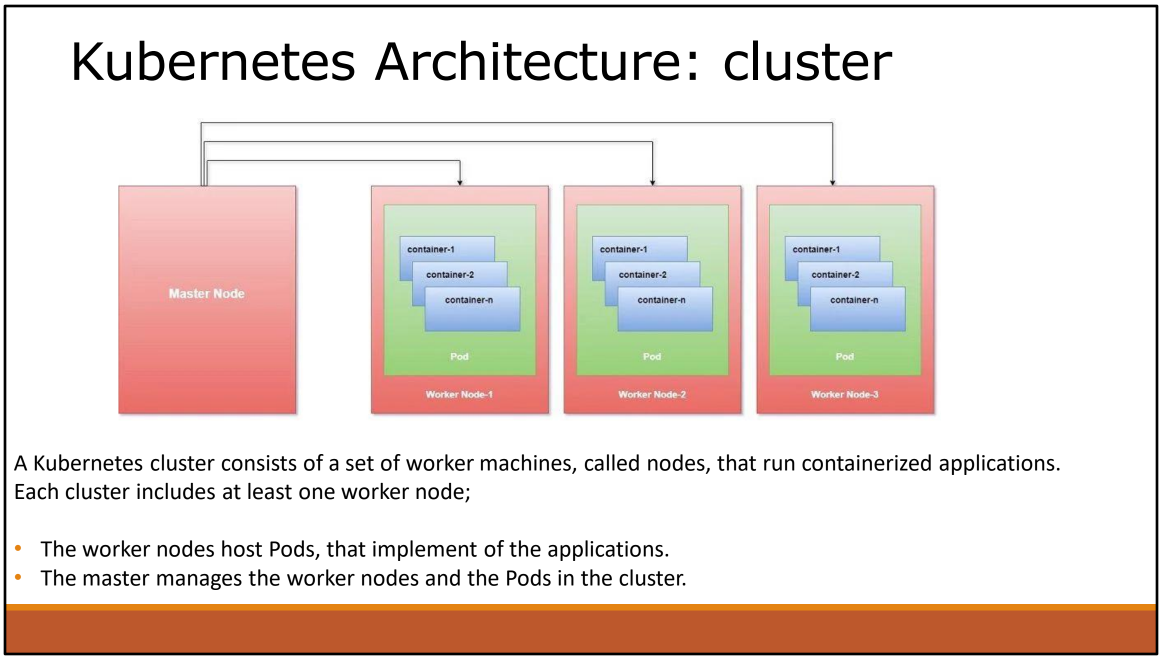
These templates, specifications for creating pods, are included in other objects named Deployments, Jobs and DaemonSets.



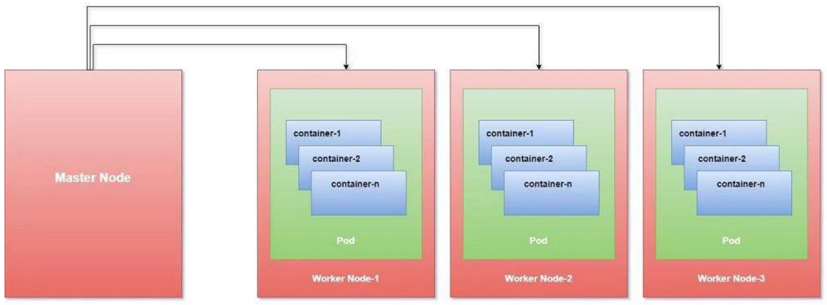
A basic architectural diagram of Kubernetes and the relationship between containers, pods, and physical worker nodes which were referred as Minion in past.

Nodes (Minion)
•You can think of these as container clients.
•These are individual hosts (physical or virtual) on which Docker would be installed to host different containers within your managed cluster
•Each Node will run ETCD (key pair management and communication service, used by Kubernetes for exchanging messages and reporting Cluster status) as well as the Kubernetes proxy

A Pod is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers. We're not implying that a pod always includes more than one container—it's common for pods to contain only a single container. The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node—it never spans multiple worker nodes These containers are guaranteed (by the cluster controller) to be located on the same host machine in order to facilitate sharing of resources Pods are assigned unique IP address within each cluster. These allow an application to use ports without having to worry about conflicting port utilization Pods can contain definitions of disk volumes or share and then provide access from those to all the members (containers) with the pod.



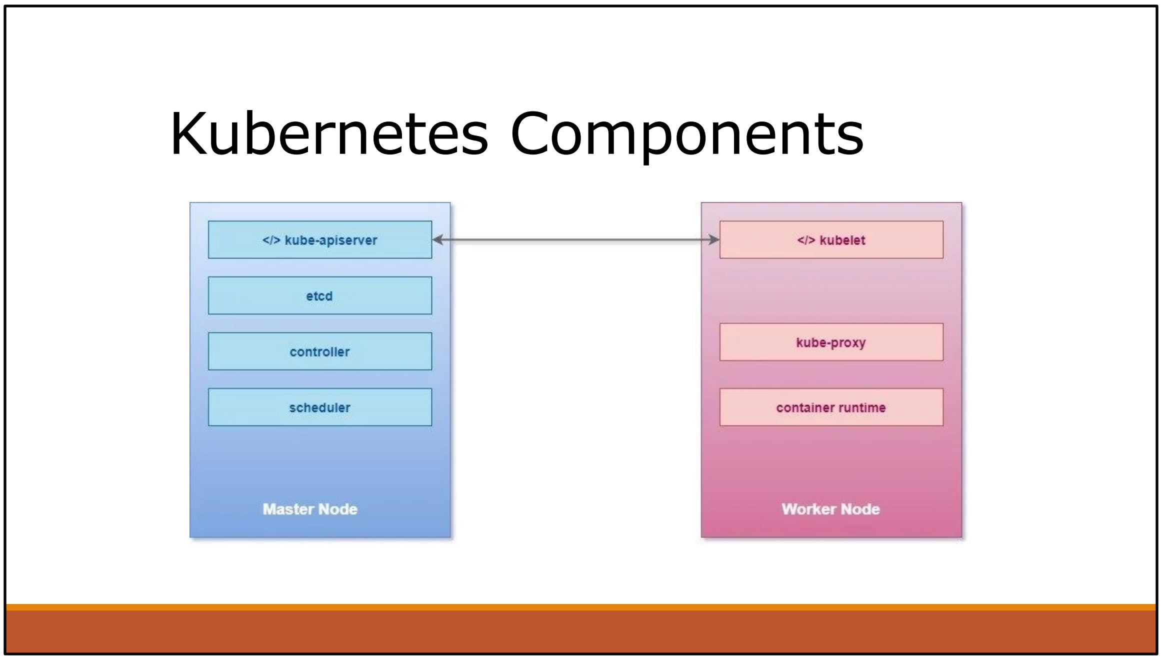
Kubernetes Architecture: cluster



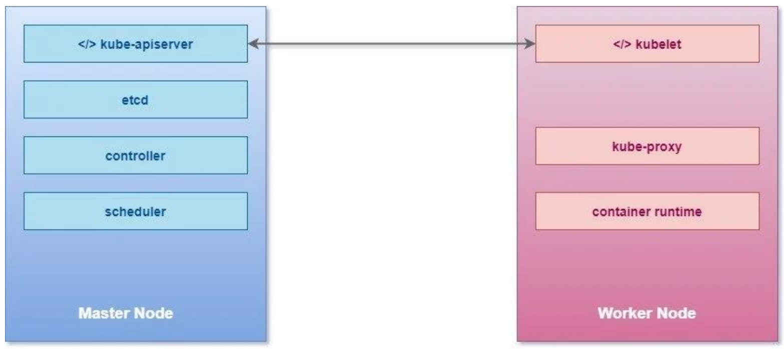
A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Each cluster includes at least one worker node;

- The worker nodes host Pods, that implement of the applications.
- The master manages the worker nodes and the Pods in the cluster.

• A **Kubernetes Cluster** consists of Master and Client node setup where we will have one Master or **Controller** node along with multiple Client nodes also referred as **worker** nodes or in minions.
• A **Master** is a node with Kubernetes installed and is responsible for the actual orchestration of containers on the worker nodes. It will contain all the information of cluster nodes, monitor each node and if a worker node fails then it will move the workload from the failed node to another worker node.
• A **Node** is a worker machine which can be a physical or virtual machine on which Kubernetes is installed.
• Kubernetes does not deploy containers directly into the worker nodes, the containers are encapsulated into a Kubernetes object known as **Pods**.
• A pod is a single instance of an application and they are the smallest deployable units of computing that you can create and manage in Kubernetes.
• You will deploy containers inside these pods where you will deploy your application



Kubernetes Components



Master
•The master is the control plane of Kubernetes.
•It consists of several components, such as an API server, a scheduler, and a controller manager.
•The master is responsible for the global state of the cluster, cluster-level scheduling of pods, and handling of events. Usually, all the master components are set up on a single host.
•When considering high-availability scenarios or very large clusters, you will want to have master redundancy.

The master is in charge of:

- exposing the Kubernetes (REST) API,
- scheduling applications,
- managing the cluster,
- directing communications across the entire system,
- monitoring the containers running in each node as well as the health of all the registered nodes.

API Server
•The API server acts as a frontend for Kubernetes
•It can easily scale horizontally as it is stateless and stores all the data in the etcd cluster.
•The users, management devices, command line interfaces, all talk to API server to interact with Kubernetes cluster

etcd key store
•It is a distributed reliable key value store
•Kubernetes uses it to store the entire cluster state
•In a small, transient cluster a single instance of etcd can run on the same node with all the other master components, but for more substantial clusters, it is typical to have a **three-node or even five-node etcd cluster for redundancy and high availability**.
•It is responsible for implementing locks within the clusters to ensure that there are no conflicts between the masters

Scheduler
Kube-scheduler is responsible for scheduling pods into nodes. This is a very complicated task as it needs to consider multiple interacting factors, such as the following:
•**Resource requirements**
•**Service requirements**
•**Hardware/software policy constraints**
•**Node affinity and anti-affinity specifications**
•**Pod affinity and anti-affinity specifications**
•**Taints and tolerations** (*Node affinity* is a property of Pods that *attracts* them to a set of nodes (either as a preference or a hard requirement). *Taints* are the opposite -- they allow a node to repel a set of pods.)
•**Data locality**
•**Deadlines**

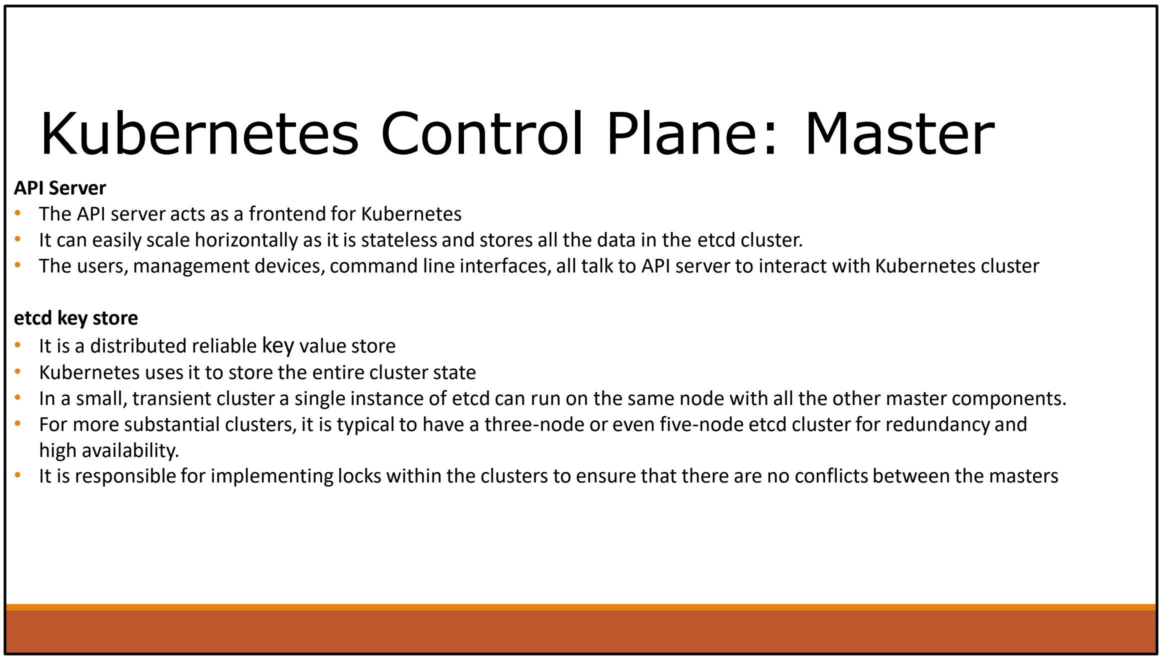
Controllers
•The controllers are the brain behind Orchestration.
•They are responsible for noticing and responding when Nodes, containers or end points goes down
•The controller makes decision to bring up new containers in such case

Container Runtime
It is the underlying software that is used to run containers. In our case we will be using Docker as the underlying container but there are other options as well such as:

•Docker (via a CRI shim)
•rkt (direct integration to be replaced with Rktlet)
•CRI-O
•Frakti (Kubernetes on the Hypervisor, previously Hypernetes)
•rktlet (CRI implementation for rkt)
•CRI-containerd
The major design policy is that Kubernetes itself should be completely decoupled from specific runtimes. The **Container Runtime Interface (CRI)** enables it.

kubelet
It is the agent that runs on each nodes in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected. That includes the following:
•Receiving pod specs
•Downloading pod secrets from the API server
•Mounting volumes
•Running the pod's containers (via the configured runtime)
•Reporting the status of the node and each pod
•Running the container startup, liveness, and readiness probes

kube-proxy
•Kube-proxy does low-level network housekeeping on each node
•Containers run on the server nodes, but they interact with each other as they are running in a unified networking setup.
•kube-proxy makes it possible for containers to communicate, although they are running on different nodes.
•It reflects the Kubernetes services locally and can perform TCP and UDP forwarding.
•It finds cluster IPs via environment variables or DNS.



Kubernetes Control Plane: Master

API Server
- The API server acts as a frontend for Kubernetes
- It can easily scale horizontally as it is stateless and stores all the data in the etcd cluster.
- The users, management devices, command line interfaces, all talk to API server to interact with Kubernetes cluster

etcd key store
- It is a distributed reliable key value store
- Kubernetes uses it to store the entire cluster state
- In a small, transient cluster a single instance of etcd can run on the same node with all the other master components.
- For more substantial clusters, it is typical to have a three-node or even five-node etcd cluster for redundancy and high availability.
- It is responsible for implementing locks within the clusters to ensure that there are no conflicts between the masters

Master

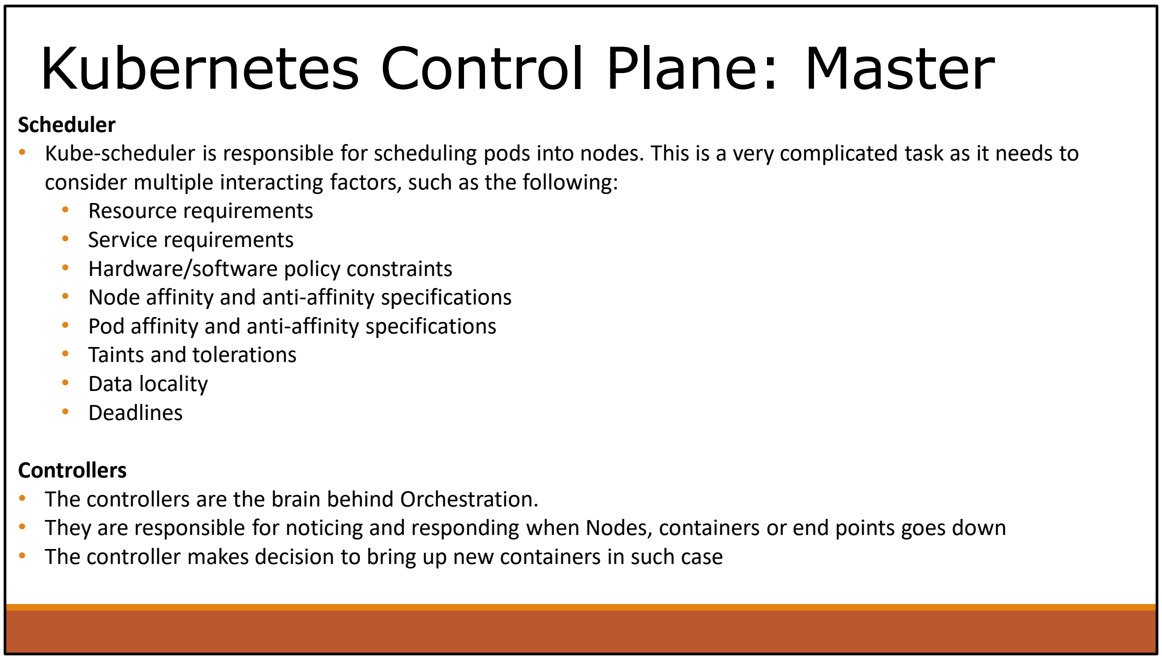
•The master is the control plane of Kubernetes.

•It consists of several components, such as an API server, a scheduler, and a controller manager.

•The master is responsible for the global state of the cluster, cluster-level scheduling of pods, and handling of events. Usually, all the master components are set up on a single host.

•When considering high-availability scenarios or very large clusters, you will want to have master redundancy.

The master is in charge of:
- exposing the Kubernetes (REST) API,
- scheduling applications,
- managing the cluster,
- directing communications across the entire system,
- monitoring the containers running in each node as well as the health of all the registered nodes.



Kubernetes Control Plane: Master

Scheduler

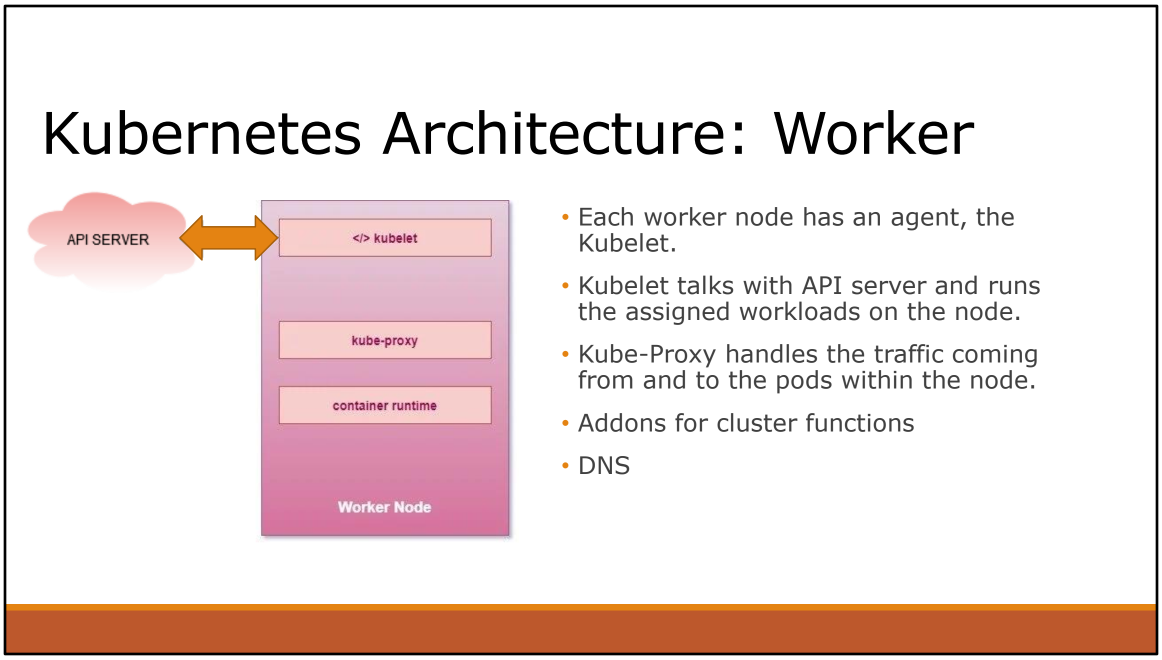
- Kube-scheduler is responsible for scheduling pods into nodes. This is a very complicated task as it needs to consider multiple interacting factors, such as the following:
  - Resource requirements
  - Service requirements
  - Hardware/software policy constraints
  - Node affinity and anti-affinity specifications
  - Pod affinity and anti-affinity specifications
  - Taints and tolerations
  - Data locality
  - Deadlines

Controllers

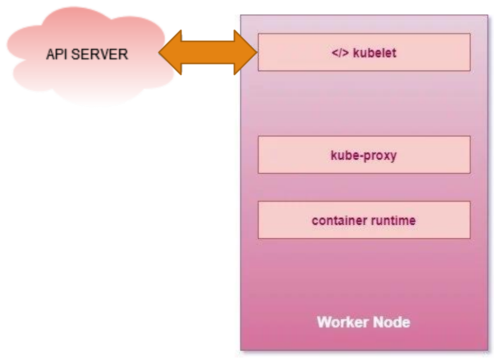
- The controllers are the brain behind Orchestration.
- They are responsible for noticing and responding when Nodes, containers or end points goes down
- The controller makes decision to bring up new containers in such case

- The Kubernetes master runs the following components that form the control plane:
- **API server**: the front-end for the Kubernetes control plane that exposes the Kubernetes API
  - kube-apiserver, component of Kubernetes designed to scale horizontally
- **etcd**: is a persistent, lightweight, distributed key-value data store that maintains the entire state of the cluster at time
- **scheduler**: manages new created Pods and selects nodes to host them
  - kube-scheduler
- **Controller manager**: control loop that checks the state of the cluster through the apiserver and makes changes attempting to transfer the current cluster state into the desired cluster state
  - kube-controller-manager

  - cloud-controller-manager
- **Node affinity** ensures that pods are hosted on particular node. **Pod affinity** ensures two pods to be co-located in a single node.
- ***Taints*** are the opposite -- they allow a node to repel a set of pods.
- **Tolerations** are applied to pods. Tolerations allow the scheduler to schedule pods with matching taints. Tolerations allow scheduling but don't guarantee scheduling: the scheduler also evaluates other parameters as part of its function.
  - Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.
- **Deadline**: When setting step **Timeout** settings, it's important to know that Kubernetes has an optional deadline parameter that specifies the number of seconds you want Kubernetes to wait for your Deployment to progress before the system reports back that the Deployment has failed progressing.



Kubernetes Architecture: Worker



- Each worker node has an agent, the Kubelet.
- Kubelet talks with API server and runs the assigned workloads on the node.
- Kube-Proxy handles the traffic coming from and to the pods within the node.
- Addons for cluster functions
- DNS

Node components run on each node, keeping the pods running and providing the Kubernetes runtime environment.

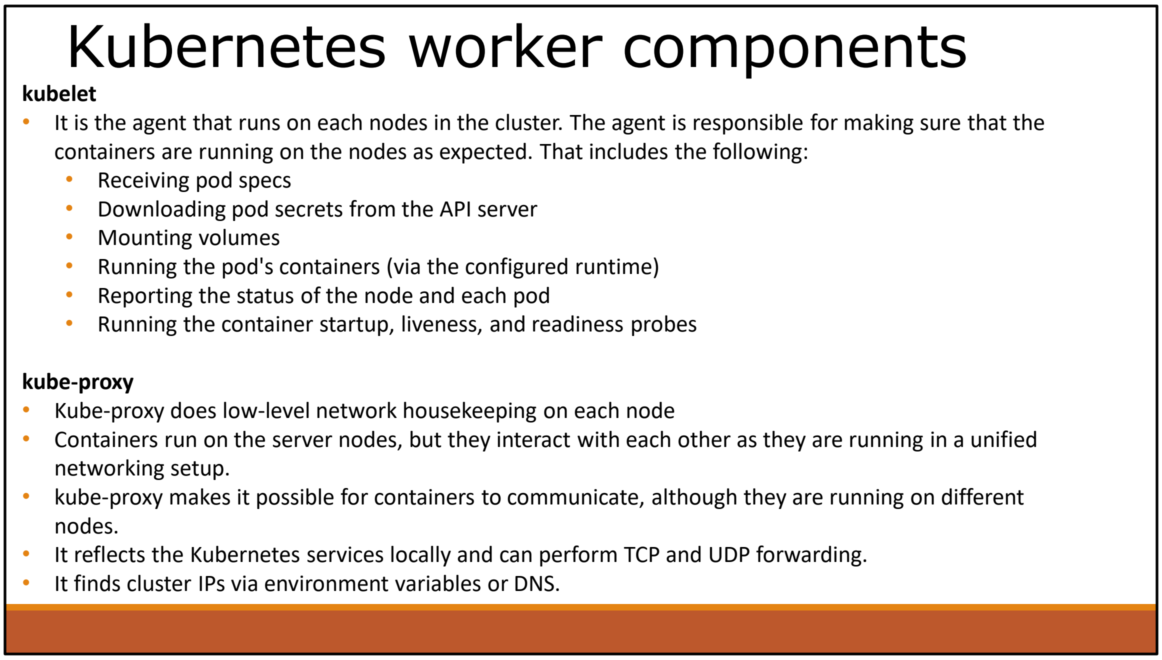
Kubelet: An agent that runs on every node of the cluster. It makes sure that the containers are running in a pod. The kubelet receives a set of PodSpecs which are provided through various mechanisms, and makes sure that the containers described in these PodSpecs are functioning properly and are healthy. The kubelet does not manage containers that were not created by Kubernetes.

kube-proxy: It is a proxy running on each node of the cluster, responsible for managing the Kubernetes Services. Kube proxies keep networking rules on nodes. These rules allow communication to the other nodes of the cluster or to the outside. The kube-proxy uses the operating system libraries whenever possible; otherwise the kube-proxy manages the traffic directly.

Container Runtime: The container runtime is the software that is responsible for running the containers. Kubernetes supports several container runtimes: Docker, containerd, cri-o, rktlet and all Kubernetes CRI (Container Runtime Interface) implementations.

Addons: Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster functionality. Since addons provide cluster-level functionality, resources that need a namespace are placed in the kube-system namespace.

DNS: While other addons are not strictly required, all Kubernetes clusters should be equipped with a cluster DNS, since many applications need it. Cluster DNS is an additional DNS server to other DNS servers on the network, and specifically takes care of DNS records for Kubernetes services. Containers run by Kubernetes automatically use this server for DNS resolution.



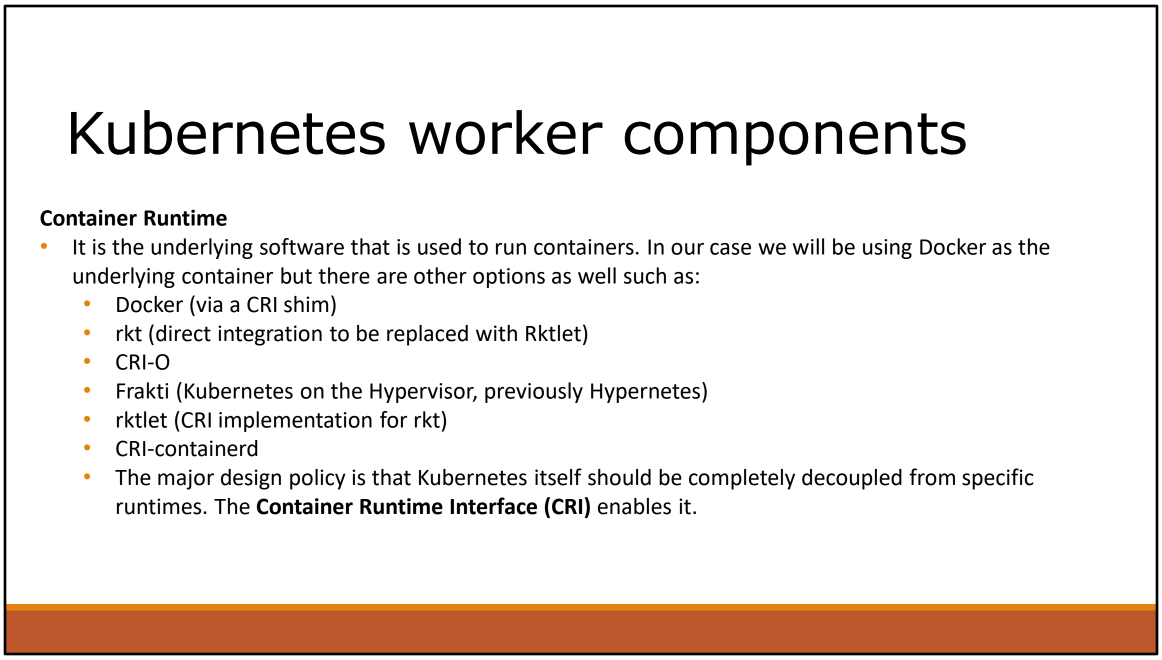
Kubernetes worker components

kubelet

- It is the agent that runs on each nodes in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected. That includes the following:
  - Receiving pod specs
  - Downloading pod secrets from the API server
  - Mounting volumes
  - Running the pod's containers (via the configured runtime)
  - Reporting the status of the node and each pod
  - Running the container startup, liveness, and readiness probes

kube-proxy

- Kube-proxy does low-level network housekeeping on each node
- Containers run on the server nodes, but they interact with each other as they are running in a unified networking setup.
- kube-proxy makes it possible for containers to communicate, although they are running on different nodes.
- It reflects the Kubernetes services locally and can perform TCP and UDP forwarding.
- It finds cluster IPs via environment variables or DNS.



Kubernetes worker components

Container Runtime

- It is the underlying software that is used to run containers. In our case we will be using Docker as the underlying container but there are other options as well such as:
  - Docker (via a CRI shim)
  - rkt (direct integration to be replaced with Rktlet)
  - CRI-O
  - Frakti (Kubernetes on the Hypervisor, previously Hypernetes)
  - rktlet (CRI implementation for rkt)
  - CRI-containerd
  - The major design policy is that Kubernetes itself should be completely decoupled from specific runtimes. The **Container Runtime Interface (CRI)** enables it.

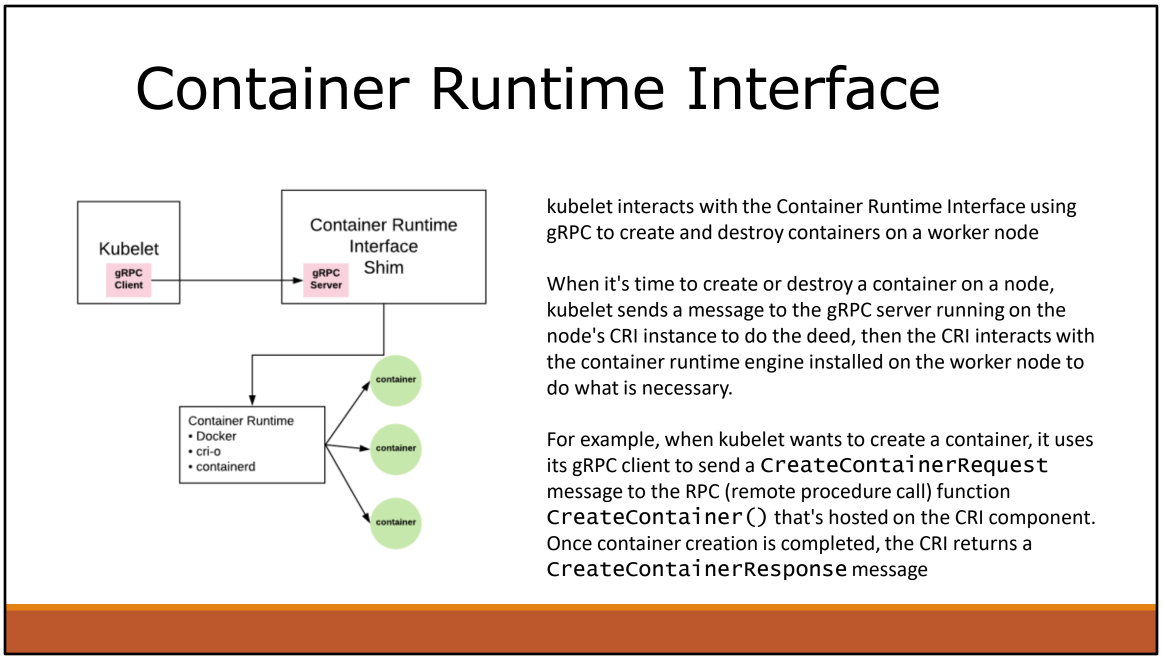
The CRI is a **plugin interface** which enables the kubelet to use a wide variety of container runtimes, without having a need to recompile the cluster components.

You need a working container runtime on each Node in your cluster, so that the kubelet can launch Pods and their containers.

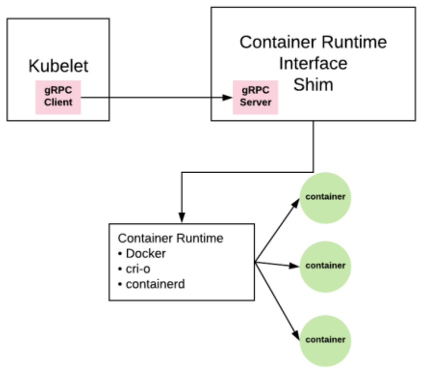
The Container Runtime Interface (CRI) is the main protocol for the communication between the kubelet and Container Runtime.

The Kubernetes Container Runtime Interface (CRI) defines the main gRPC protocol for the communication between the node components kubelet and container runtime.

gRPC is a modern open source high performance Remote Procedure Call (RPC) framework.



Container Runtime Interface



kubelet interacts with the Container Runtime Interface using gRPC to create and destroy containers on a worker node

When it's time to create or destroy a container on a node, kubelet sends a message to the gRPC server running on the node's CRI instance to do the deed, then the CRI interacts with the container runtime engine installed on the worker node to do what is necessary.

For example, when kubelet wants to create a container, it uses its gRPC client to send a `CreateContainerRequest` message to the RPC (remote procedure call) function `CreateContainer()` that's hosted on the CRI component. Once container creation is completed, the CRI returns a `CreateContainerResponse` message

See https://www.mulesoft.com/api-university/grpc-real-world-kubernetes-container-runtime-interface

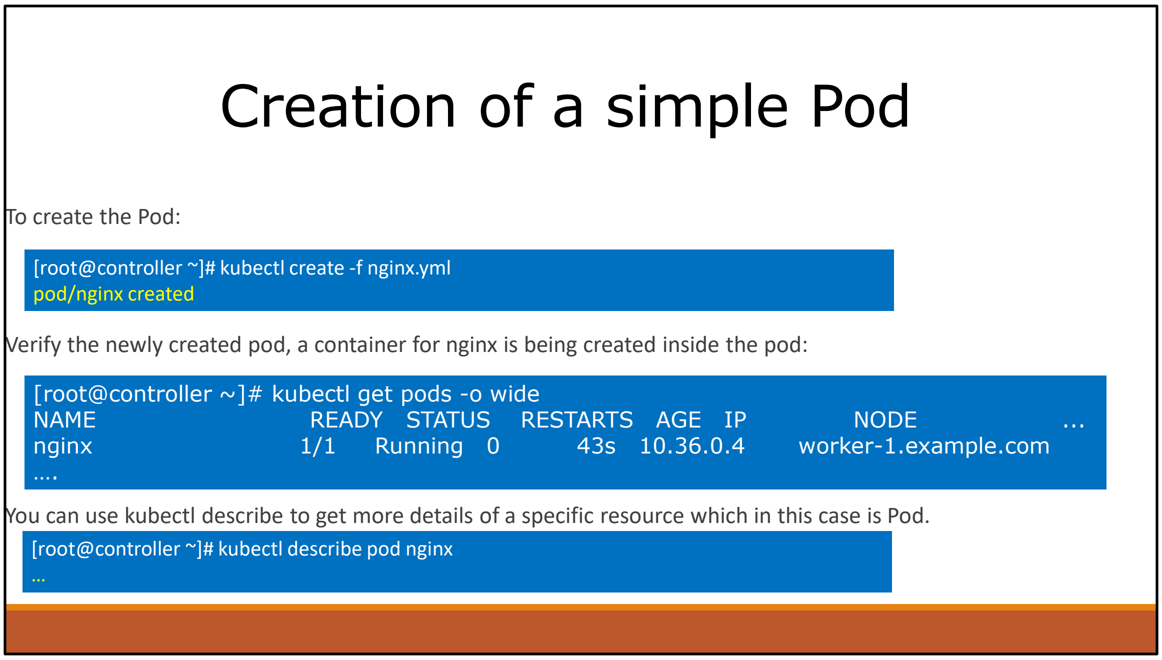
A container runtime shim is a lightweight daemon launching the *container runtime* and controlling the container process. The shim's process is tightly bound to the container's process but is completely detached from the manager's process. All the communications between the container and the manager happen through the shim.

Creation of a simple Pod

A **Creating Pods using YAML file**

- Pods and other Kubernetes resources are usually created by posting a JSON or YAML manifest to the Kubernetes REST API endpoint.

```
[root@controller ~]# cat nginx.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```



Creation of a simple Pod

To create the Pod:

```
[root@controller ~]# kubectl create -f nginx.yml
pod/nginx created
```

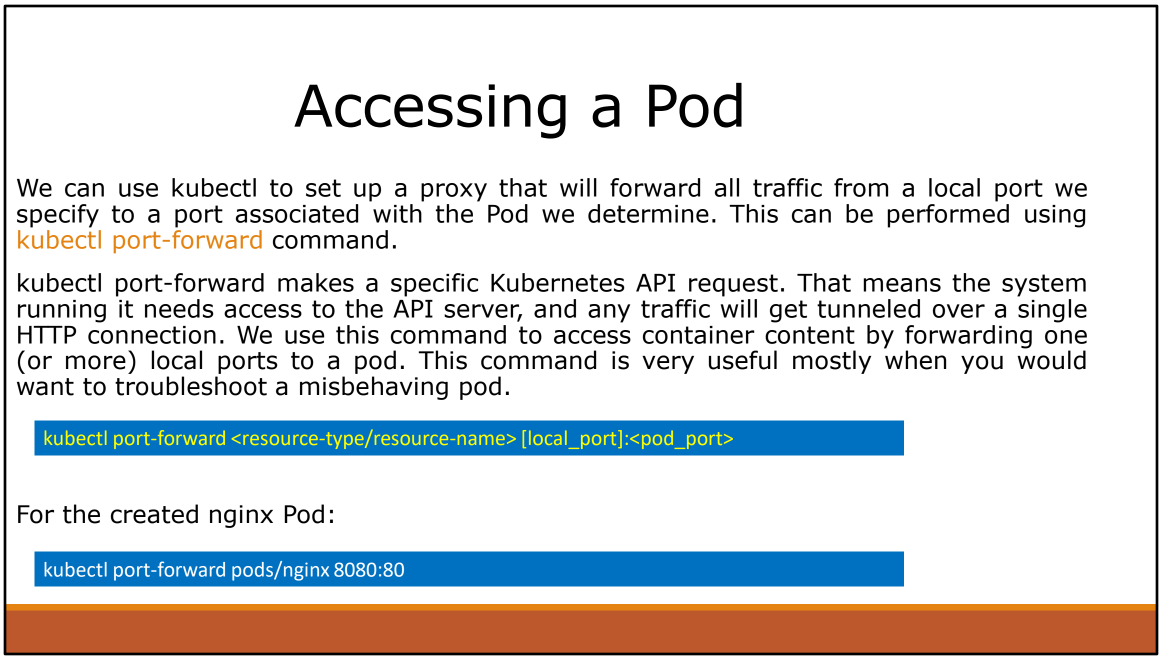
Verify the newly created pod, a container for nginx is being created inside the pod:

```
[root@controller ~]# kubectl get pods -o wide
NAME                   READY   STATUS    RESTARTS   AGE   IP           NODE                  …
nginx                  1/1     Running   0          43s   10.36.0.4    worker-1.example.com
….
```

You can use kubectl describe to get more details of a specific resource which in this case is Pod.

```
[root@controller ~]# kubectl describe pod nginx
…
```

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. For more information including a complete list of kubectl operations, see the kubectl reference documentation.



Accessing a Pod

We can use kubectl to set up a proxy that will forward all traffic from a local port we specify to a port associated with the Pod we determine. This can be performed using kubectl port-forward command.

kubectl port-forward makes a specific Kubernetes API request. That means the system running it needs access to the API server, and any traffic will get tunneled over a single HTTP connection. We use this command to access container content by forwarding one (or more) local ports to a pod. This command is very useful mostly when you would want to troubleshoot a misbehaving pod.

```
kubectl port-forward <resource-type/resource-name> [local_port]:<pod_port>
```

For the created nginx Pod:

```
kubectl port-forward pods/nginx 8080:80
```

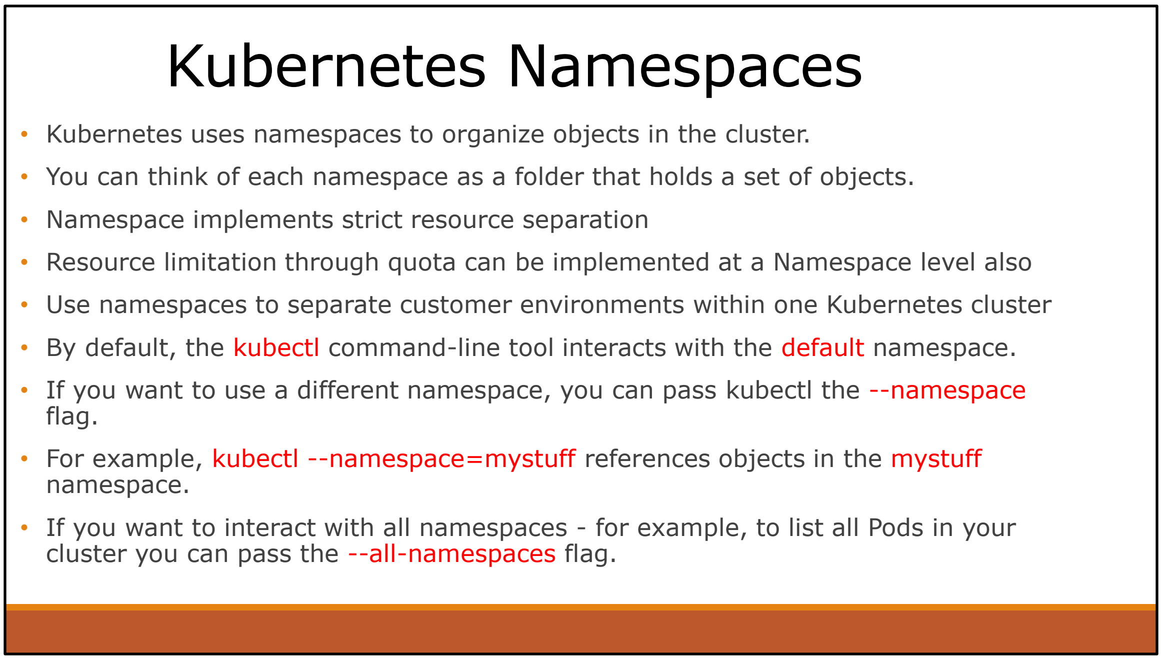
In this method we will forward the traffic to port **8080 on localhost** (on controller) to port **80 on worker node** based nginx container. This is forwarding any and all traffic that gets created on your local machine at TCP port 8080 to TCP port 80 on the Pod nginx.

The same port forwarding scheme can be used also for other Kuberbetes objects, that are described in this presentation

Sample command to perform port forwarding on **deployment**:
kubectl port-forward deployment/my-deployment 8080:80

Sample command to perform port forwarding on **replicaset**:
kubectl port-forward replicaset/my-replicaset 8080:80

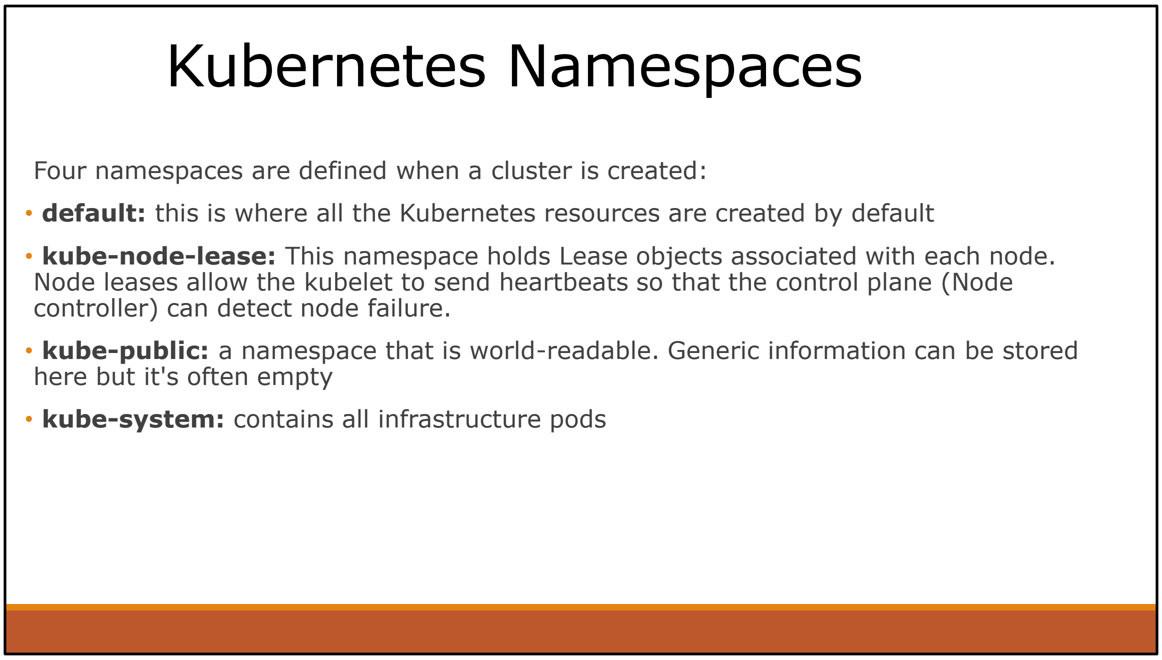
Sample command to perform port forwarding on **service**:
kubectl port-forward service/my-service 8080:80



Kubernetes Namespaces

- Kubernetes uses namespaces to organize objects in the cluster.
- You can think of each namespace as a folder that holds a set of objects.
- Namespace implements strict resource separation
- Resource limitation through quota can be implemented at a Namespace level also
- Use namespaces to separate customer environments within one Kubernetes cluster
- By default, the kubectl command-line tool interacts with the default namespace.
- If you want to use a different namespace, you can pass kubectl the --namespace flag.
- For example, kubectl --namespace=mystuff references objects in the mystuff namespace.
- If you want to interact with all namespaces - for example, to list all Pods in your cluster you can pass the --all-namespaces flag.

Using multiple namespaces allows you to split complex systems with numerous components into smaller distinct groups. This is useful in scenarios wherein you want to split and limit resources across different resources. Resource names only need to be unique within a namespace. Two different namespaces can contain resources of the same name.
Although namespaces allow you to isolate objects into distinct groups, which allows you to operate only on those belonging to the specified namespace, they don't provide any kind of isolation of running objects. For example, you may think that when different users deploy pods across different namespaces, those pods are isolated from each other and can't communicate, but that's not necessarily the case. Whether namespaces provide network isolation depends on which networking solution is deployed with Kubernetes. When the solution doesn't provide inter-namespace network isolation, if a pod in namespace foo knows the IP address of a pod in namespace bar, there is nothing preventing it from sending traffic, such as HTTP requests, to the other pod.



Kubernetes Namespaces

Four namespaces are defined when a cluster is created:

- **default:** this is where all the Kubernetes resources are created by default
- **kube-node-lease:** This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane (Node controller) can detect node failure.
- **kube-public:** a namespace that is world-readable. Generic information can be stored here but it's often empty
- **kube-system:** contains all infrastructure pods

Leases

Distributed systems often have a need for leases, which provide a mechanism to lock shared resources and coordinate activity between members of a set. In Kubernetes, the lease concept is represented by Lease objects in the coordination.k8s.io API Group, which are used for system-critical capabilities such as node heartbeats and component-level leader election.

Node heartbeats

Kubernetes uses the Lease API to communicate kubelet node heartbeats to the Kubernetes API server. For every Node , there is a Lease object with a matching name in the kube-node-lease namespace. Under the hood, every kubelet heartbeat is an update request to this Lease object, updating the spec.renewTime field for the Lease. The Kubernetes control plane uses the time stamp of this field to determine the availability of this Node.

Kubernetes Namespaces

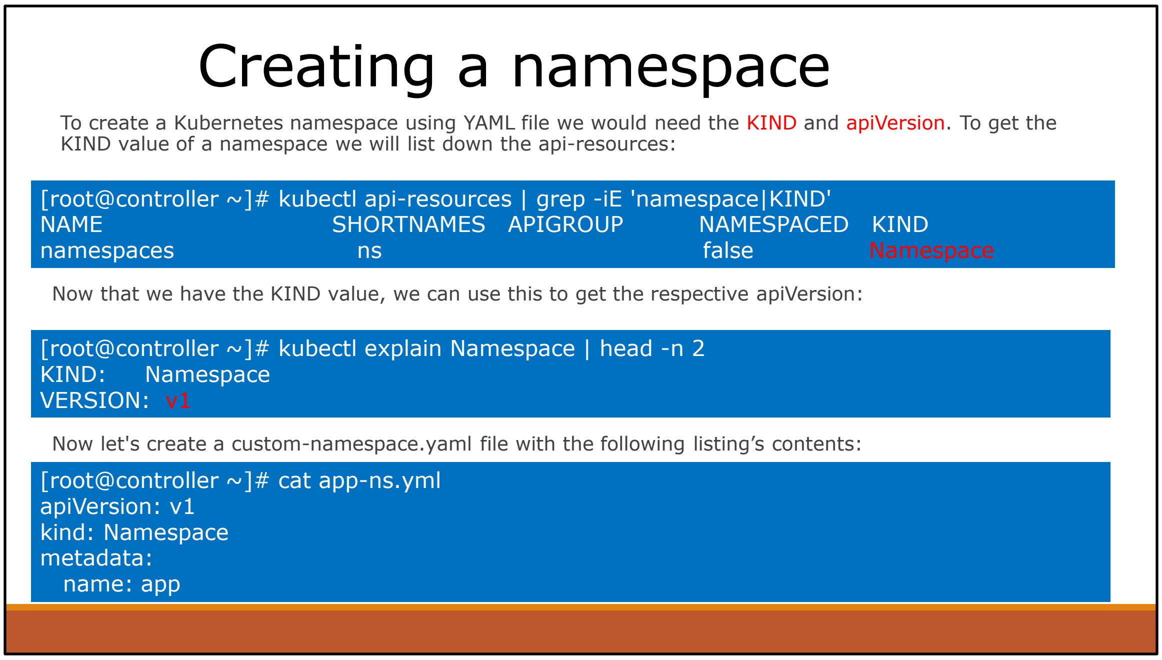
To get the list of available namespaces, operating in the default namespace:

```
[root@controller ~]# kubectl get ns
NAME              STATUS   AGE
default           Active   14d
kube-node-lease   Active   14d
kube-public       Active   14d
kube-system       Active   14d
```

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs.

Alternatively you can also use kubectl get all in all the available namespaces:

```
[root@controller ~]# kubectl get all --all-namespaces
NAMESPACE     NAME                            READY   STATUS    RESTARTS   AGE
kube-system   pod/coredns-f9fd979d6-nmsq5     1/1     Running   4          14d
…
NAMESPACE     NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
default       service/kubernetes   ClusterIP   10.96.0.1                  443/TCP   14d
…
```



Creating a namespace

To create a Kubernetes namespace using YAML file we would need the KIND and apiVersion. To get the KIND value of a namespace we will list down the api-resources:

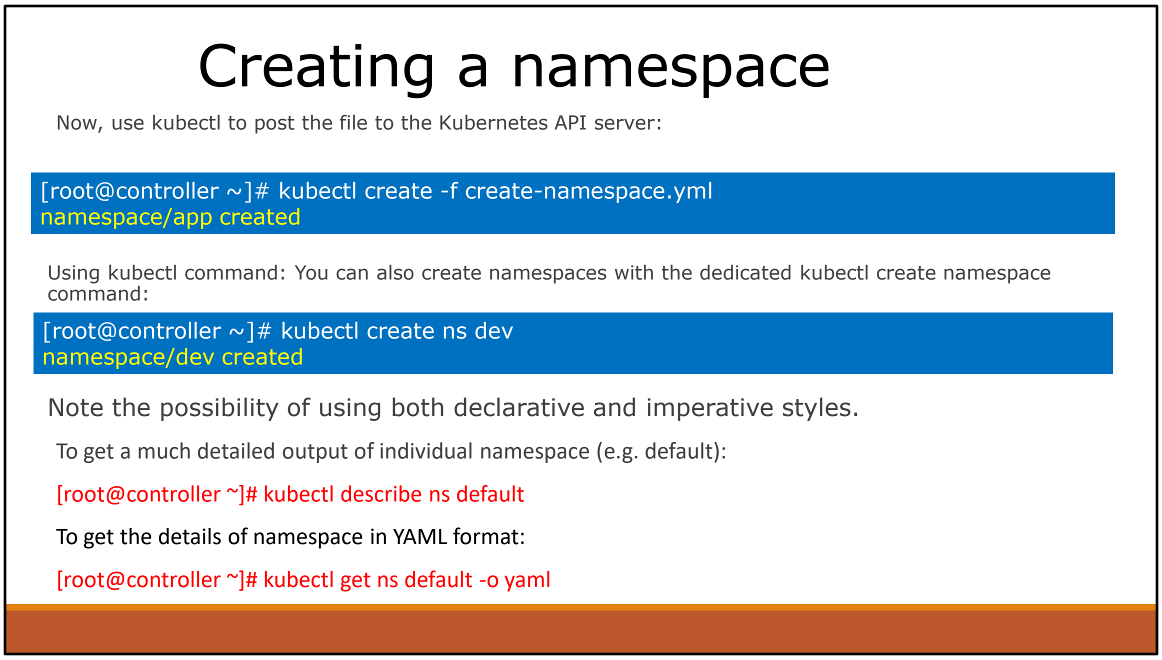
```
[root@controller ~]# kubectl api-resources | grep -iE 'namespace|KIND'
NAME                          SHORTNAMES   APIGROUP         NAMESPACED   KIND
namespaces                      ns                           false        Namespace
```

Now that we have the KIND value, we can use this to get the respective apiVersion:

```
[root@controller ~]# kubectl explain Namespace | head -n 2
KIND:     Namespace
VERSION:  v1
```

Now let's create a custom-namespace.yaml file with the following listing’s contents:

```
[root@controller ~]# cat app-ns.yml
apiVersion: v1
kind: Namespace
metadata:
  name: app
```



Creating a namespace

Now, use kubectl to post the file to the Kubernetes API server:

```
[root@controller ~]# kubectl create -f create-namespace.yml
namespace/app created
```

Using kubectl command: You can also create namespaces with the dedicated kubectl create namespace command:

```
[root@controller ~]# kubectl create ns dev
namespace/dev created
```

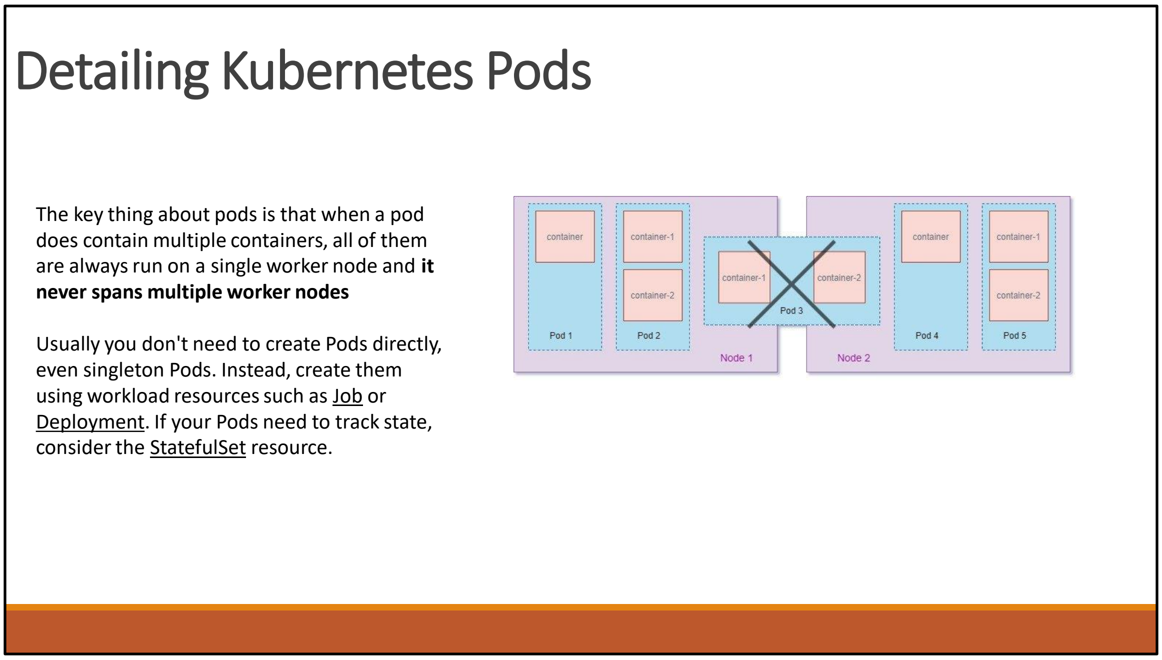
Note the possibility of using both declarative and imperative styles.

To get a much detailed output of individual namespace (e.g. default):

```
[root@controller ~]# kubectl describe ns default
```

To get the details of namespace in YAML format:

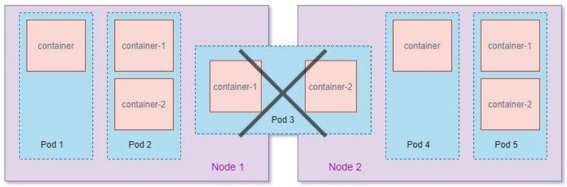
```
[root@controller ~]# kubectl get ns default -o yaml
```



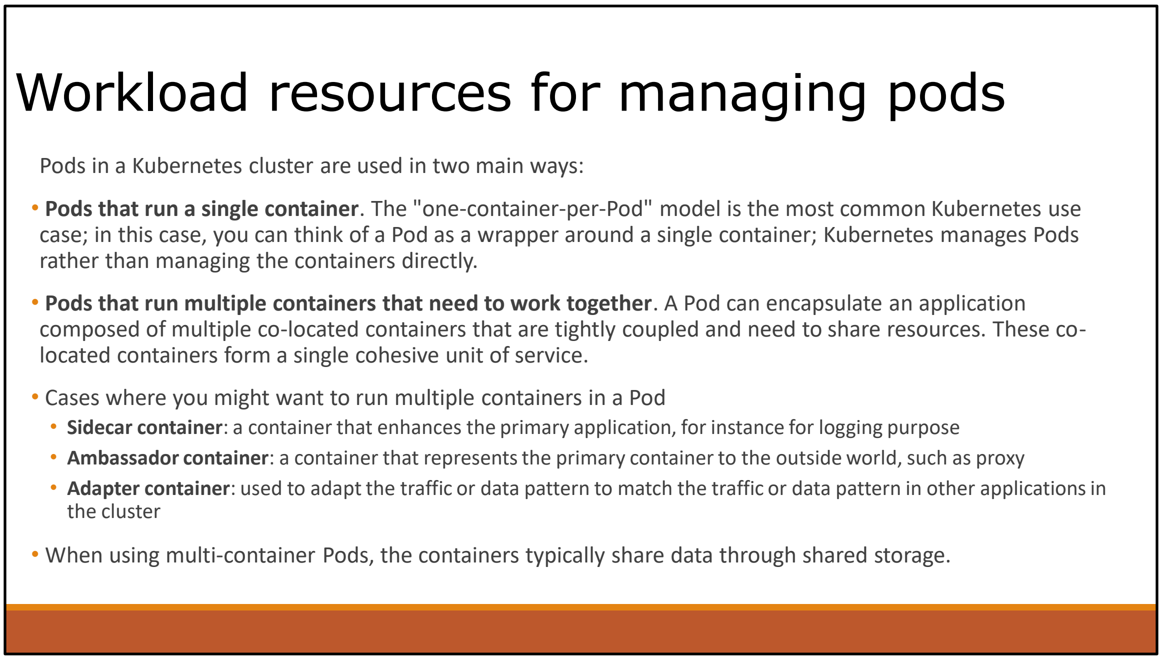
Detailing Kubernetes Pods

The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node and **it never spans multiple worker nodes**

Usually you don't need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as Job or Deployment. If your Pods need to track state, consider the StatefulSet resource.



•We already know that a pod is a co-located group of containers and represents the basic building block in Kubernetes.
•Instead of deploying containers individually, you always deploy and operate on a pod of containers.
•We're not implying that a pod always includes more than one container, it's common for pods to contain only a single container.

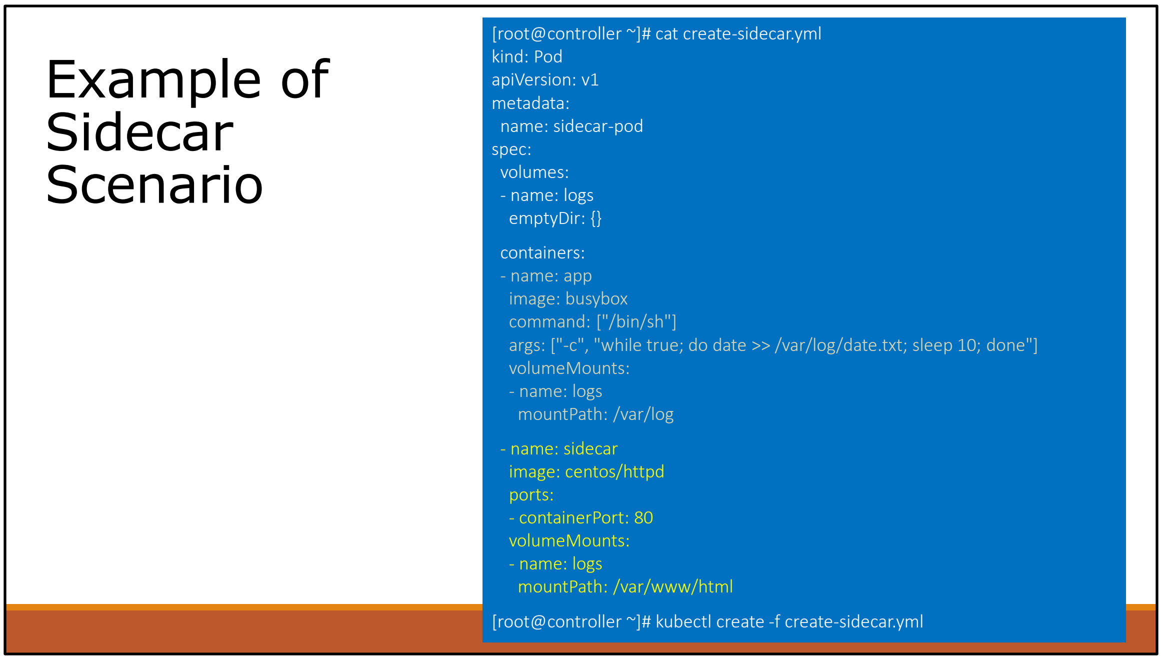


Workload resources for managing pods

Pods in a Kubernetes cluster are used in two main ways:

- **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
- **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service.
- Cases where you might want to run multiple containers in a Pod
  - **Sidecar container**: a container that enhances the primary application, for instance for logging purpose
  - **Ambassador container**: a container that represents the primary container to the outside world, such as proxy
  - **Adapter container**: used to adapt the traffic or data pattern to match the traffic or data pattern in other applications in the cluster
- When using multi-container Pods, the containers typically share data through shared storage.

Whenever we create a pod, a pause container image such as *gcr.io/google_containers/pause:0.8.0* is implicitly required. What is that pause container's purpose? The pause container essentially holds the network namespace for the pod. It does nothing useful and its container image (see its Dockerfile) basically contains a simple binary that goes to sleep and never wakes up (see its code). However, when the top container, such as nginx container used before, dies and gets restarted by kubernetes, all the network setup will still be there. Normally, if the last process in a network namespace dies, the namespace will be destroyed. Restarting nginx container without pause would require creating all new network setup. With pause, you will always have that one last thing in the namespace.



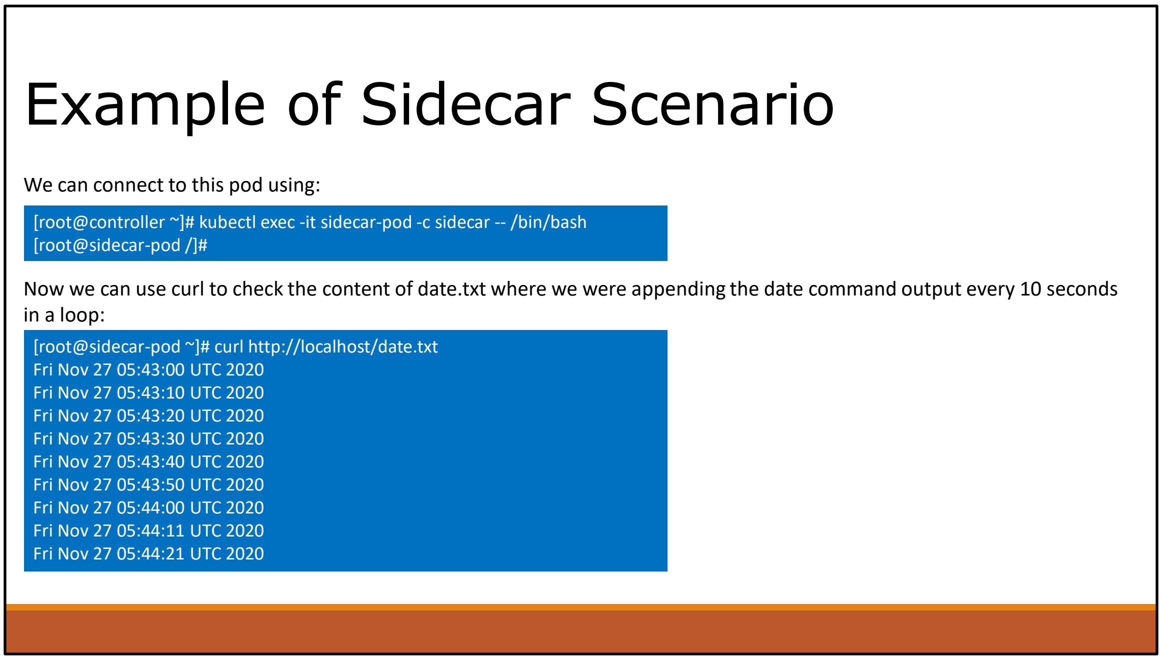
Example of Sidecar Scenario

```
[root@controller ~]# cat create-sidecar.yml
kind: Pod
apiVersion: v1
metadata:
  name: sidecar-pod
spec:
  volumes:
  - name: logs
    emptyDir: {}

  containers:
  - name: app
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do date >> /var/log/date.txt; sleep 10; done"]
    volumeMounts:
    - name: logs
      mountPath: /var/log

  - name: sidecar
    image: centos/httpd
    ports:
    - containerPort: 80
    volumeMounts:
    - name: logs
      mountPath: /var/www/html

[root@controller ~]# kubectl create -f create-sidecar.yml
```

For a Pod that defines an emptyDir volume, the volume is created when the Pod is assigned to a node. As the name says, the emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

The **{}** at the end means we do not supply any further requirements for the **emptyDir**
.

If the **-c** option is present in **sh**, then commands are read from *string*. If there are arguments after the *string*, they are assigned to the positional parameters, starting with **$0**.



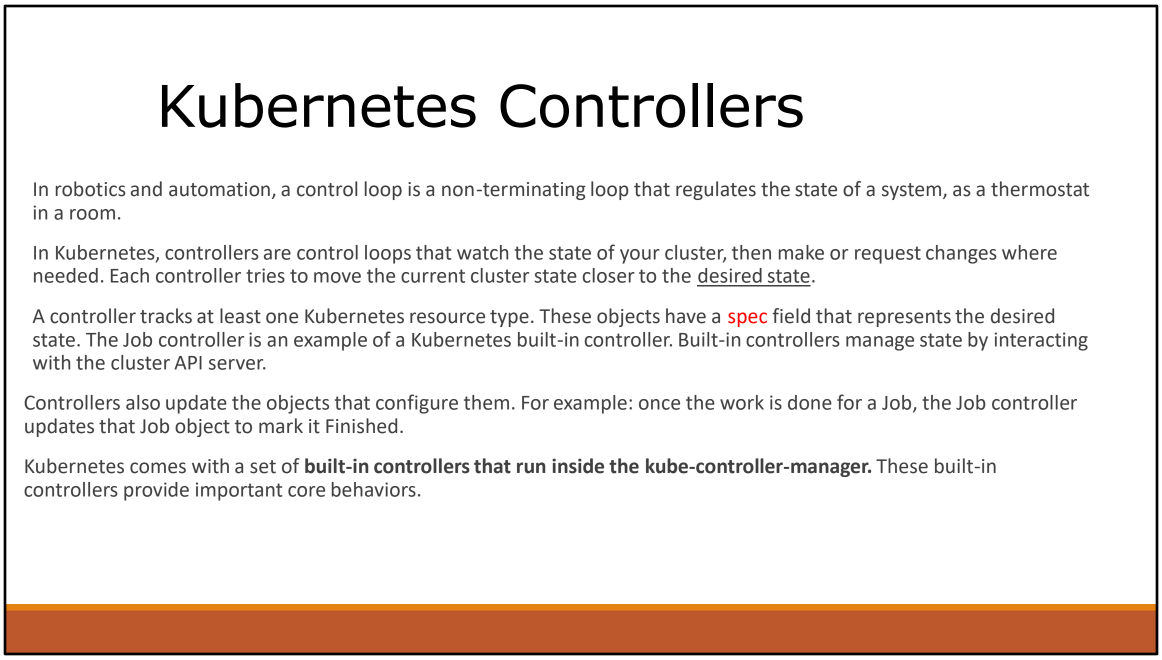
Example of Sidecar Scenario

We can connect to this pod using:

```
[root@controller ~]# kubectl exec -it sidecar-pod -c sidecar -- /bin/bash
[root@sidecar-pod /]#
```

Now we can use curl to check the content of date.txt where we were appending the date command output every 10 seconds in a loop:

```
[root@sidecar-pod ~]# curl http://localhost/date.txt
Fri Nov 27 05:43:00 UTC 2020
Fri Nov 27 05:43:10 UTC 2020
Fri Nov 27 05:43:20 UTC 2020
Fri Nov 27 05:43:30 UTC 2020
Fri Nov 27 05:43:40 UTC 2020
Fri Nov 27 05:43:50 UTC 2020
Fri Nov 27 05:44:00 UTC 2020
Fri Nov 27 05:44:11 UTC 2020
Fri Nov 27 05:44:21 UTC 2020
```



Kubernetes Controllers

In robotics and automation, a control loop is a non-terminating loop that regulates the state of a system, as a thermostat in a room.

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

A controller tracks at least one Kubernetes resource type. These objects have a spec field that represents the desired state. The Job controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it Finished.

Kubernetes comes with a set of **built-in controllers that run inside the kube-controller-manager.** These built-in controllers provide important core behaviors.

https://kubernetes.io/docs/concepts/architecture/controller/

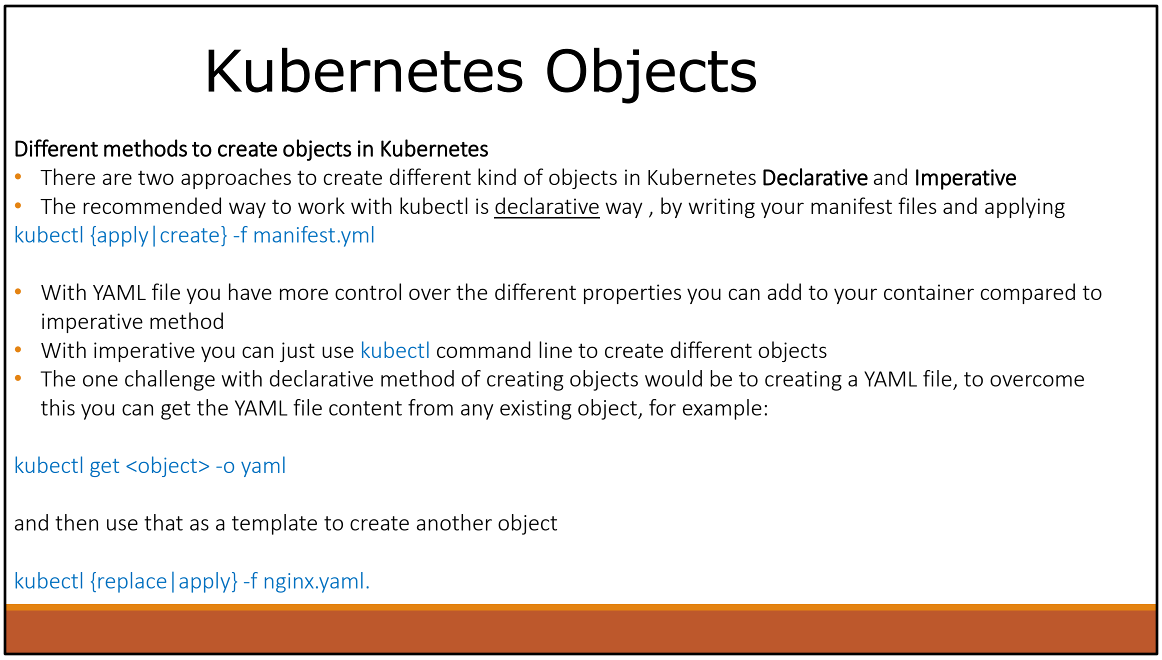
Kubernetes Objects

Kubernetes objects are persistent entities provided by Kubernetes for deploying, maintaining, and scaling applications.

Kubernetes uses these entities to represent the state of your cluster.

The objects can describe:

- Which containerized applications are being executed, and sometimes on which nodes.
- The resources made available to the executed applications
- The policies governing the executed applications, such as restart policies, upgrades, and fault-tolerance



Kubernetes Objects

Different methods to create objects in Kubernetes

- There are two approaches to create different kind of objects in Kubernetes **Declarative** and **Imperative**
- The recommended way to work with kubectl is <u>declarative</u> way , by writing your manifest files and applying

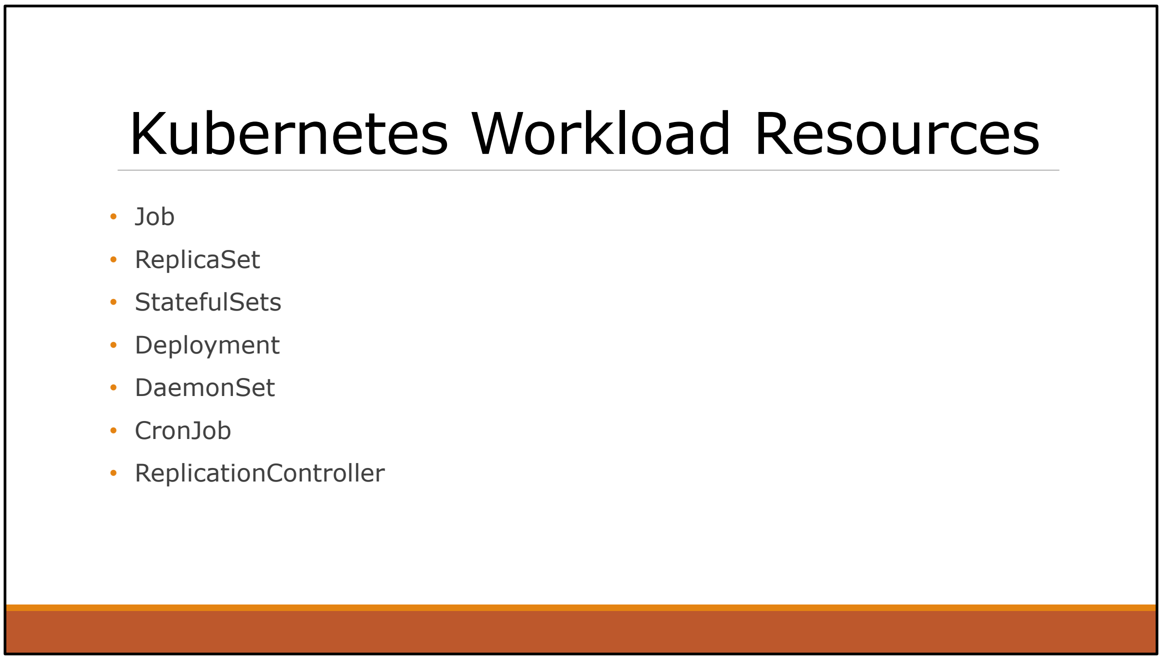
```
kubectl {apply|create} -f manifest.yml
```

- With YAML file you have more control over the different properties you can add to your container compared to imperative method
- With imperative you can just use kubectl command line to create different objects
- The one challenge with declarative method of creating objects would be to creating a YAML file, to overcome this you can get the YAML file content from any existing object, for example:

```
kubectl get <object> -o yaml
```

and then use that as a template to create another object

```
kubectl {replace|apply} -f nginx.yaml.
```



Kubernetes Workload Resources

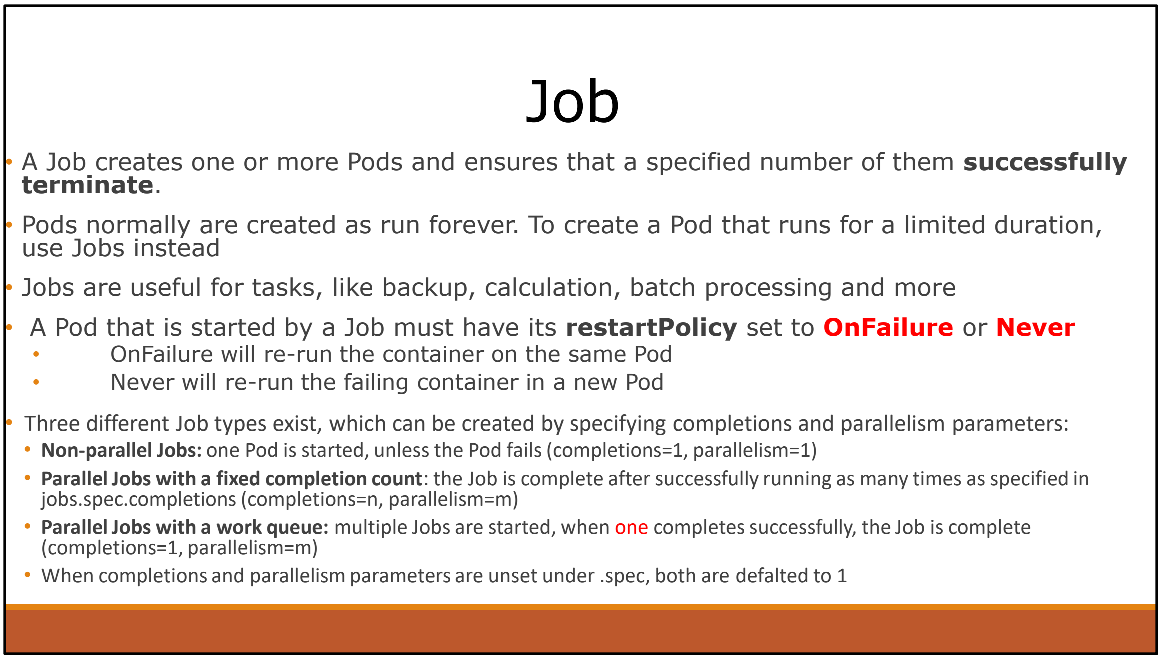
- Job
- ReplicaSet
- StatefulSets
- Deployment
- DaemonSet
- CronJob
- ReplicationController

Deployment is an object which can own ReplicaSets and update them and their Pods via declarative, server-side rolling updates. While ReplicaSets can be used independently, today they're mainly used by Deployments as a mechanism to orchestrate Pod creation, deletion and updates. When you use Deployments you don't have to worry about managing the ReplicaSets that they create. Deployments own and manage their ReplicaSets. As such, it is recommended to use Deployments when you want ReplicaSets.

Automatic Clean-up for Finished Jobs

TTL-after-finished controller provides a TTL (time to live) mechanism to limit the lifetime of resource objects that have finished execution. TTL controller only handles Jobs.

ReplicationController ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.
If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods.

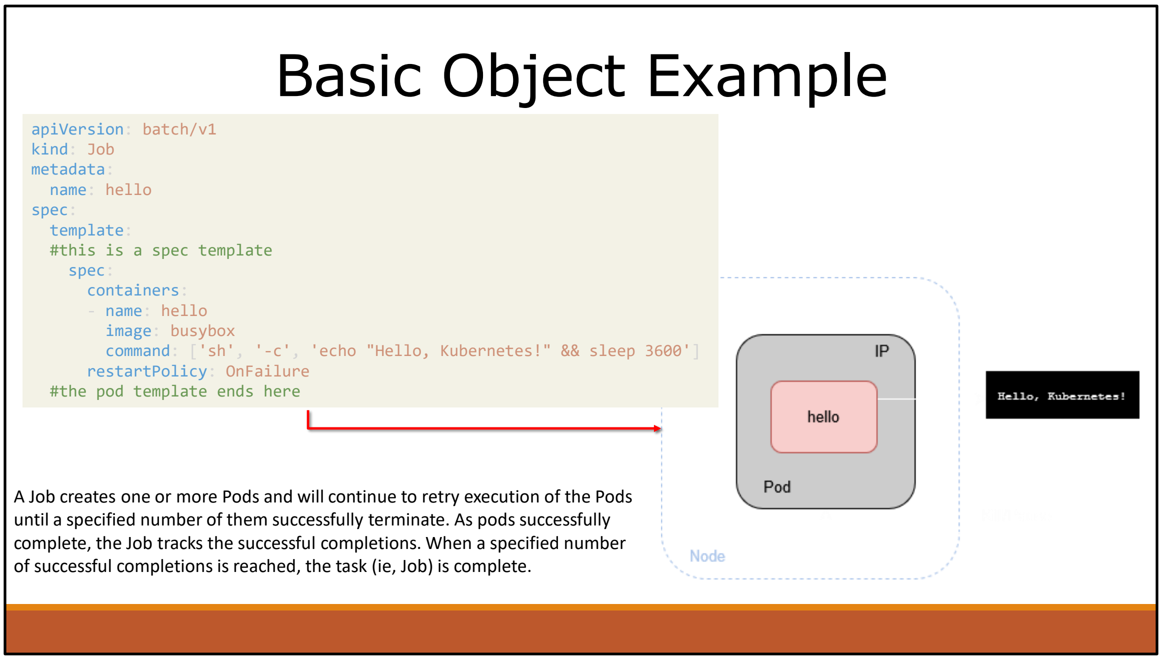


Job

- A Job creates one or more Pods and ensures that a specified number of them **successfully terminate**.
- Pods normally are created as run forever. To create a Pod that runs for a limited duration, use Jobs instead
- Jobs are useful for tasks, like backup, calculation, batch processing and more
- A Pod that is started by a Job must have its **restartPolicy** set to **OnFailure** or **Never**
  - OnFailure will re-run the container on the same Pod
  - Never will re-run the failing container in a new Pod
- Three different Job types exist, which can be created by specifying completions and parallelism parameters:
  - **Non-parallel Jobs:** one Pod is started, unless the Pod fails (completions=1, parallelism=1)
  - **Parallel Jobs with a fixed completion count**: the Job is complete after successfully running as many times as specified in jobs.spec.completions (completions=n, parallelism=m)
  - **Parallel Jobs with a work queue:** multiple Jobs are started, when one completes successfully, the Job is complete (completions=1, parallelism=m)
  - When completions and parallelism parameters are unset under .spec, both are defalted to 1

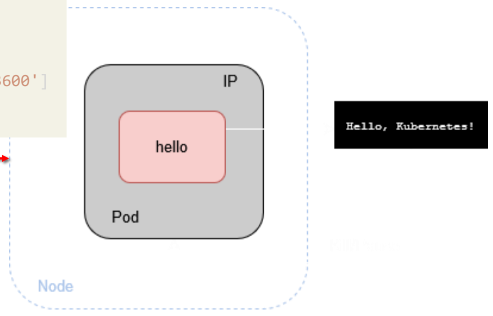
A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.
A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).
You can also use a Job to run multiple Pods in parallel.

See: **https://kubernetes.io/docs/concepts/workloads/controllers/job/**



Basic Object Example

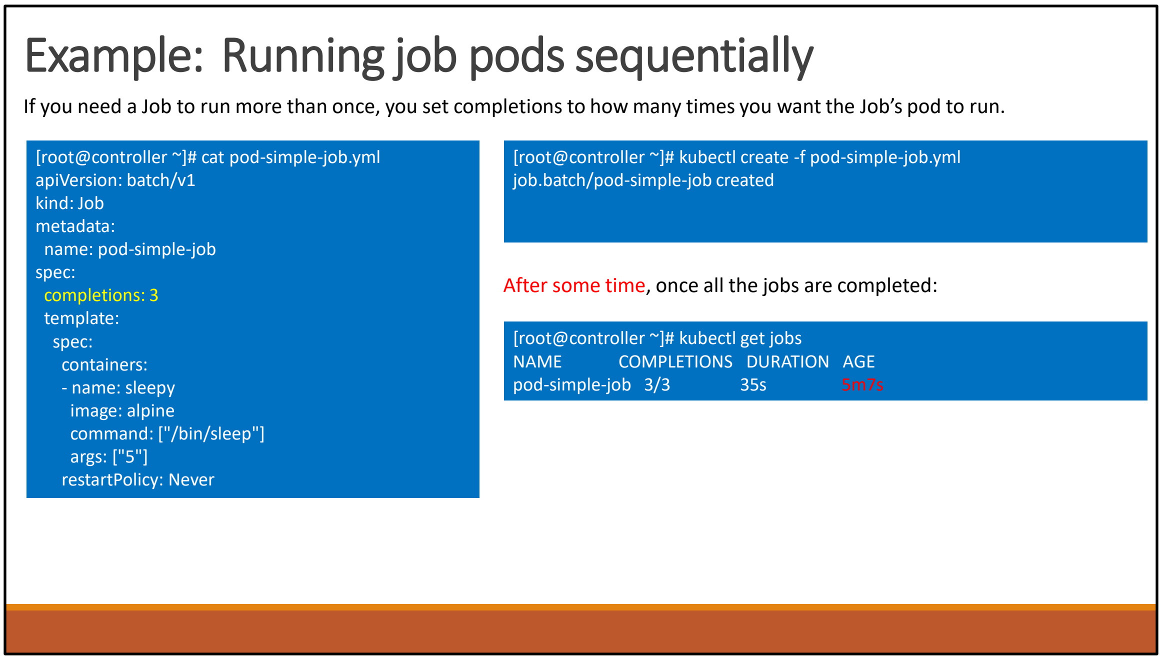
```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
  #this is a spec template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
  #the pod template ends here
```



```
Hello, Kubernetes!
```

A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete.

If you'd like your container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a restartPolicy of “Always” or “OnFailure”. Otherwise you can use “Never”
OnFailure means that **the container will only be restarted if it exited with a non-zero exit code** (i.e. something went wrong). This is useful when you want accomplish a certain task with the pod, and ensure that it completes successfully - if it doesn't it will be restarted until it does.



Example: Running job pods sequentially

If you need a Job to run more than once, you set completions to how many times you want the Job's pod to run.

```
[root@controller ~]# cat pod-simple-job.yml
apiVersion: batch/v1
kind: Job
metadata:
  name: pod-simple-job
spec:
  completions: 3
  template:
    spec:
      containers:
      - name: sleepy
        image: alpine
        command: ["/bin/sleep"]
        args: ["5"]
      restartPolicy: Never
```

```
[root@controller ~]# kubectl create -f pod-simple-job.yml
job.batch/pod-simple-job created
```

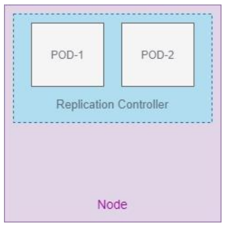
After some time, once all the jobs are completed:

```
[root@controller ~]# kubectl get jobs
NAME            COMPLETIONS   DURATION   AGE
pod-simple-job  3/3           35s        5m7s
```

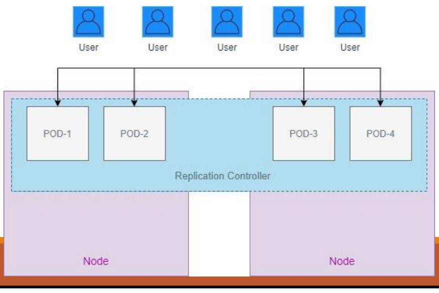
ReplicationController and ReplicaSet

A ReplicationController is a Kubernetes resource that ensures its pods are always kept running.

- If the pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the pod was evicted from the node, the ReplicationController notices the missing pod and creates a replacement pod.
- The ReplicationController in general, are meant to create and manage multiple copies (replicas) of a pod

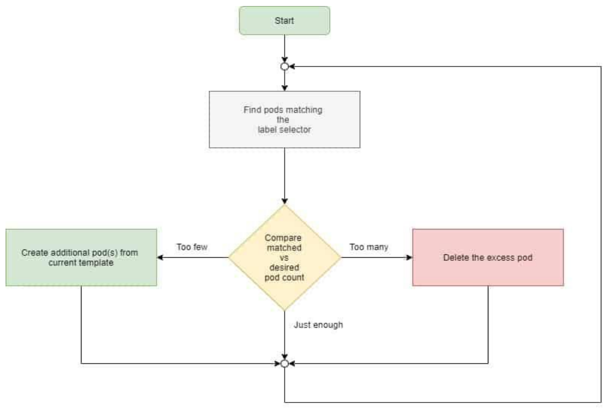


It is possible that your Node is out of resources while creating new pods with Replication controllers or replica sets, in such case it will automatically create new pods on another available cluster node



ReplicationController and ReplicaSet

A ReplicationController's task is to make sure that an **exact number of pods always matches its label selector**. If it doesn't, the ReplicationController takes the appropriate action to reconcile the actual with the desired number. The ReplicationController in general, are meant to create and manage multiple copies (replicas) of a pod



Start
Find pods matching the label selector
Compare matched vs desired pod count
Too few
Too many
Just enough
Create additional pod(s) from current template
Delete the excess pod

A ReplicationController has three essential parts:

- A ***label selector***, which determines what pods are in the ReplicationController's scope
- A ***replica count***, which specifies the desired number of pods that should be running
- A ***pod template***, which is used when creating new pod replicas

ReplicationController and ReplicaSet

```
[root@controller ~]# cat replication-controller.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: dev
spec:
  replicas: 3
  selector:
    app: myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: dev
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

Horizontally scaling pods

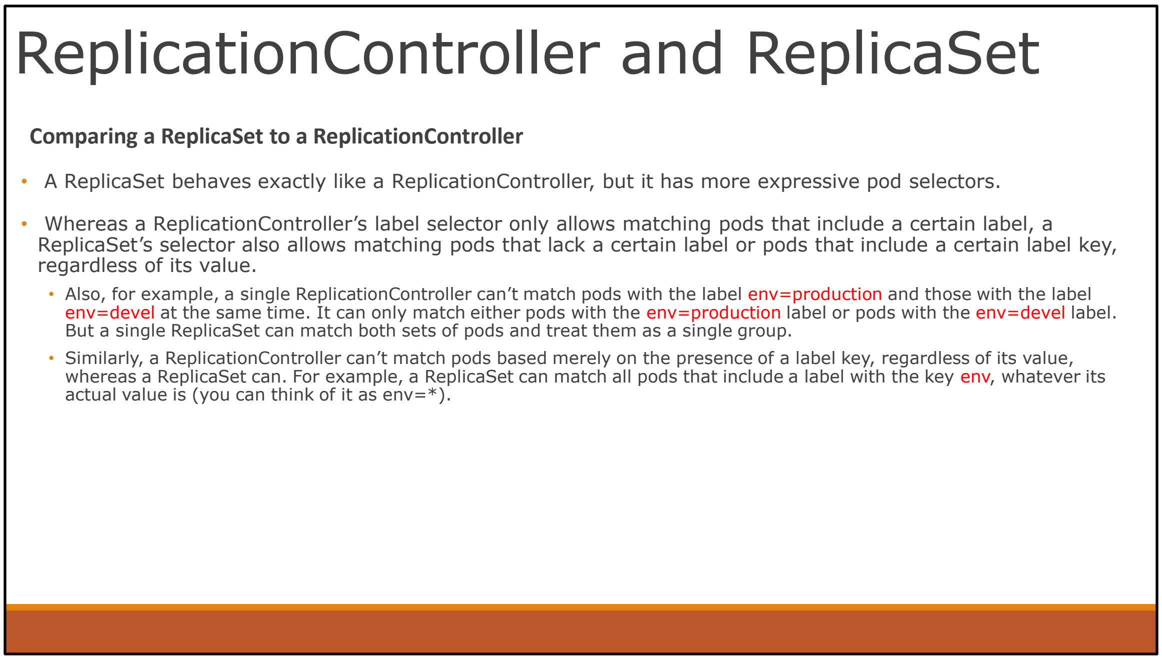
You've seen how ReplicationControllers make sure a specific number of pod instances is always running. Because it's incredibly simple to change the desired number of replicas, this also means scaling pods horizontally is trivial.

Assuming you suddenly expect that load on your application is going to increase so you must deploy more pods until the load is reduced, in such case you can easily scale up the number of pods runtime. For example, for having 6 replicas:

```
[root@controller ~]# kubectl scale rc myapp-rc --replicas=6
replicationcontroller/myapp-rc scaled
```

For downscaling to 3 replicas:

```
[root@controller ~]# kubectl scale rc myapp-rc --replicas=3
replicationcontroller/myapp-rc scaled
```



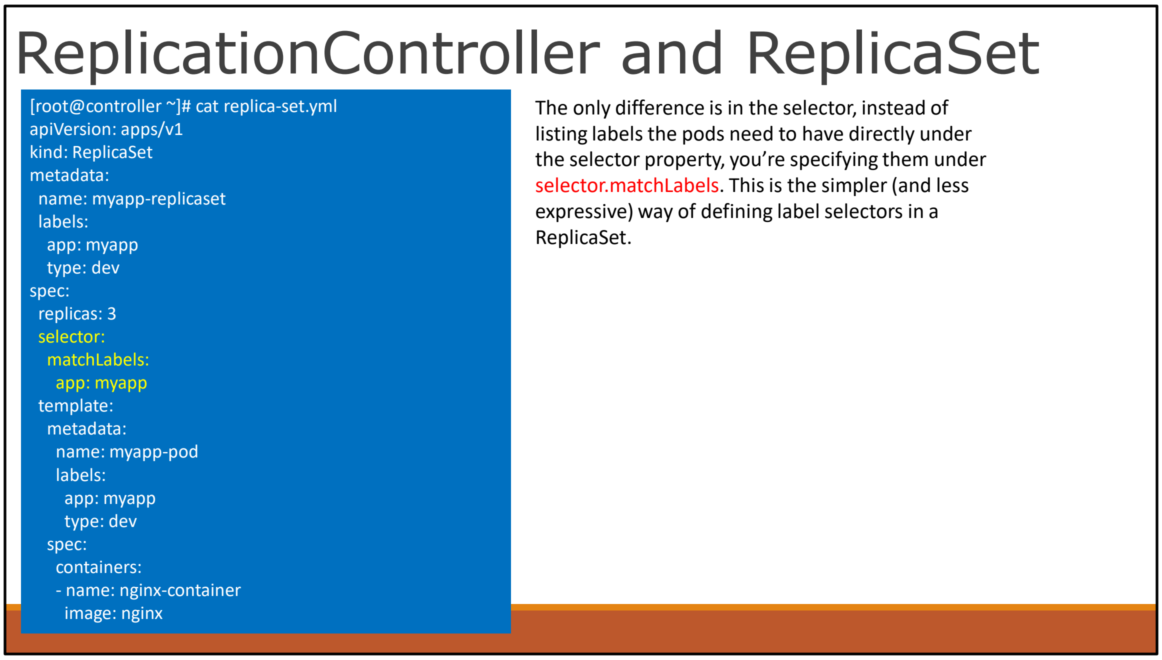
ReplicationController and ReplicaSet

Comparing a ReplicaSet to a ReplicationController

- A ReplicaSet behaves exactly like a ReplicationController, but it has more expressive pod selectors.
- Whereas a ReplicationController's label selector only allows matching pods that include a certain label, a ReplicaSet's selector also allows matching pods that lack a certain label or pods that include a certain label key, regardless of its value.
  - Also, for example, a single ReplicationController can't match pods with the label env=production and those with the label env=devel at the same time. It can only match either pods with the env=production label or pods with the env=devel label. But a single ReplicaSet can match both sets of pods and treat them as a single group.
  - Similarly, a ReplicationController can't match pods based merely on the presence of a label key, regardless of its value, whereas a ReplicaSet can. For example, a ReplicaSet can match all pods that include a label with the key env, whatever its actual value is (you can think of it as env=*).

Replica Controller is deprecated and replaced by ReplicaSets.
The replica set and the replication controller's key difference is that **the replication controller only supports equality-based selectors whereas the replica set supports set-based selectors**.

Deployments are recommended over ReplicaSets.



ReplicationController and ReplicaSet

```
[root@controller ~]# cat replica-set.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: dev
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

The only difference is in the selector, instead of listing labels the pods need to have directly under the selector property, you're specifying them under selector.matchLabels. This is the simpler (and less expressive) way of defining label selectors in a ReplicaSet.

Se al posto di myapp nel file si scrive pippo il replicaset viene creato lo stesso

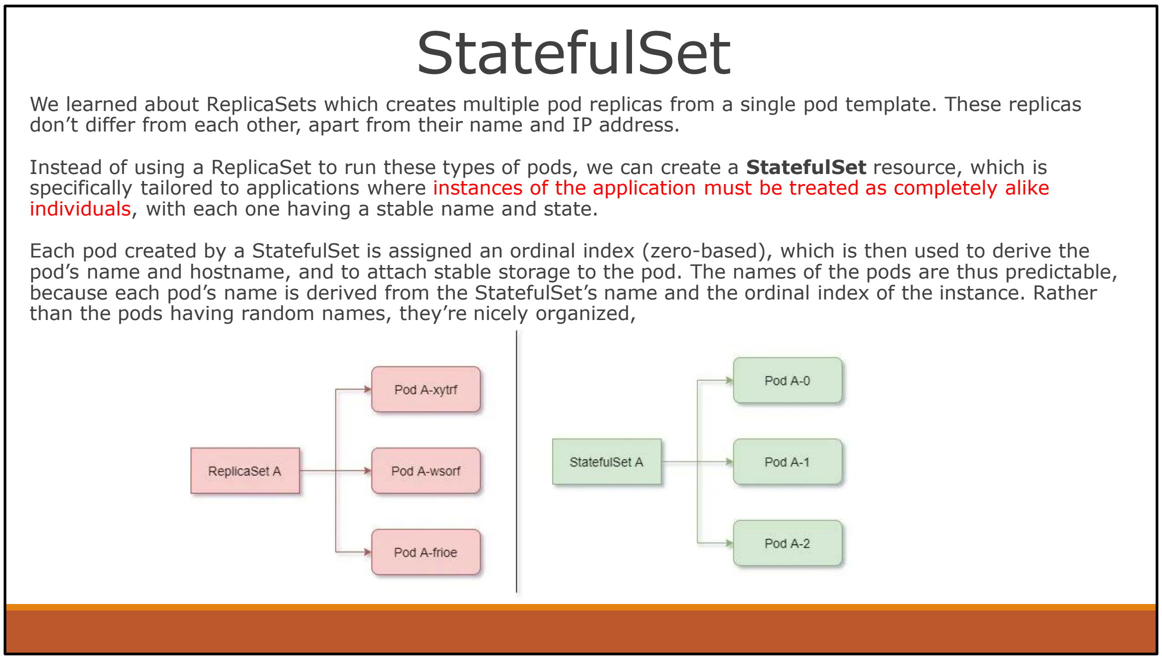
ReplicationController and ReplicaSet

```
[root@controller ~]# cat replica-set.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: dev
spec:
  replicas: 3
  selector:
    matchExpressions:
    - key: app
      operator: In
      values:
      - myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: dev
    spec:
 ....
```

Now, we will rewrite the selector to use the more powerful matchExpressions property:
Here, this selector requires the pod to contain a label with the "app" key and the label's value must be "myapp".

You can add additional expressions to the selector. As in the example, each expression must contain a key, an operator, and possibly (depending on the operator) a list of values. You'll see four valid operators:

- **In**: Label's value must match one of the specified values.
- **NotIn**: Label's value must not match any of the specified values.
- **Exists**: Pod must include a label with the specified key (the value isn't important). When using this operator, you shouldn't specify the values field.
- **DoesNotExist**: Pod must not include a label with the specified key. The values property must not be specified.



StatefulSet

We learned about ReplicaSets which creates multiple pod replicas from a single pod template. These replicas don’t differ from each other, apart from their name and IP address.

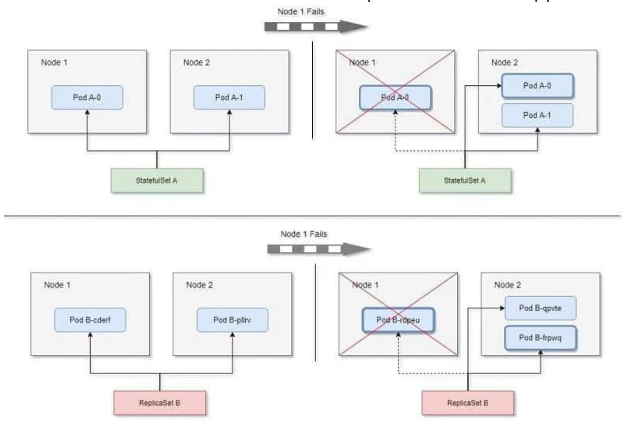
Instead of using a ReplicaSet to run these types of pods, we can create a **StatefulSet** resource, which is specifically tailored to applications where instances of the application must be treated as completely alike individuals, with each one having a stable name and state.

Each pod created by a StatefulSet is assigned an ordinal index (zero-based), which is then used to derive the pod’s name and hostname, and to attach stable storage to the pod. The names of the pods are thus predictable, because each pod’s name is derived from the StatefulSet’s name and the ordinal index of the instance. Rather than the pods having random names, they’re nicely organized,

StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.
If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.
Using StatefulSets

StatefulSet

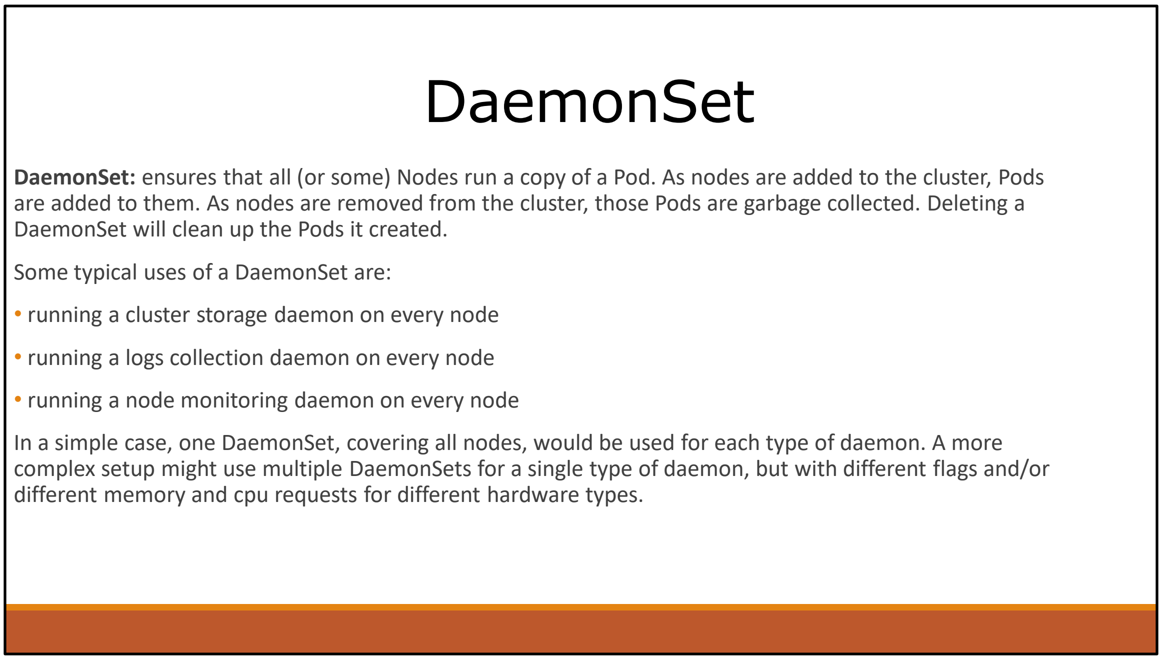
When a pod instance managed by a StatefulSet disappears (because the node the pod was running on has failed, it was evicted from the node, or someone deleted the pod object manually), the StatefulSet makes sure it’s replaced with a new instance—similar to how ReplicaSets do it. But in contrast to ReplicaSets, the replacement pod gets the same name and hostname as the pod that has disappeared.



To summarise, Kubernetes StatefulSet manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods.

Limitations

•The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner based on the requested storage class, or pre-provisioned by an admin.
•Deleting and/or scaling a StatefulSet down will not delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources.
•StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.
•StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the StatefulSet down to 0 prior to deletion.
•When using Rolling Updates with the default Pod Management Policy (OrderedReady), it's possible to get into a broken state that requires manual intervention to repair.



DaemonSet

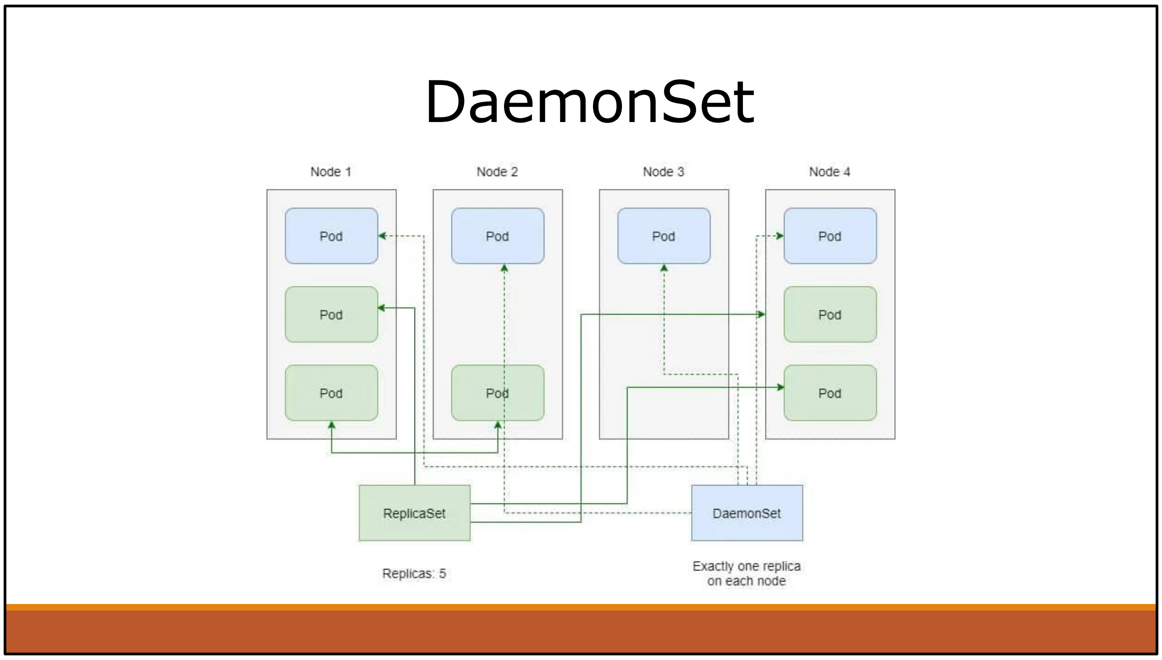
DaemonSet: ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

DaemonSets are used to ensure that some or all of your K8S nodes run a copy of a pod, which allows you to run a daemon on every node.
When you add a new node to the cluster, a pod gets added to match the nodes. Similarly, when you remove a node from your cluster, the pod is put into the trash. Deleting a DaemonSet cleans up the pods that it previously created.
A Daemonset is another controller that manages pods like Deployments, ReplicaSets, and StatefulSets. It was created for one particular purpose: ensuring that the pods it manages to run on all the cluster nodes. As soon as a node joins the cluster, the DaemonSet ensures that it has the necessary pods running on it. When the node leaves the cluster, those pods are garbage collected.
DaemonSets are used in Kubernetes when you need to run one or more pods on all (or a subset of) the nodes in a cluster. The typical use case for a DaemonSet is logging and monitoring for the hosts. For example, a node needs a service (daemon) that collects health or log data and pushes them to a central system or database (like ELK stack). DaemonSets can be deployed to specific nodes either by the nodes' user-defined labels or using values provided by Kubernetes like the node hostname.

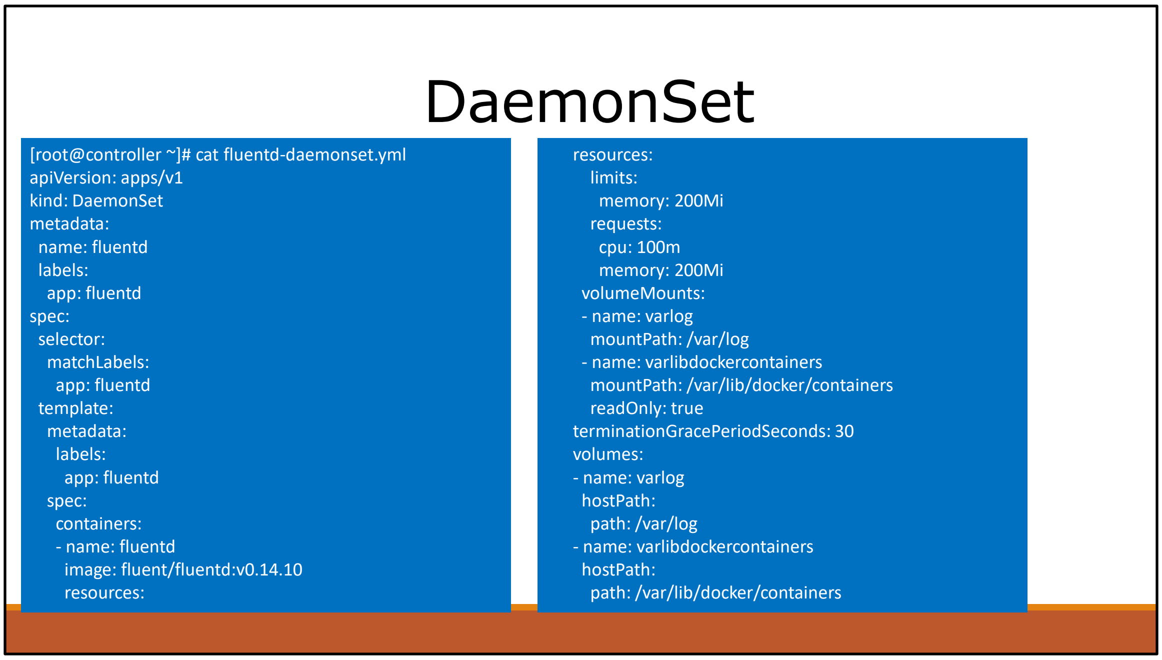

DaemonSet

A Kubernetes **DaemonSet** ensures a copy of a Pod is running across a set of nodes in a Kubernetes cluster. DaemonSets are used to deploy system daemons such as log collectors and monitoring agents, which typically must run on every node. DaemonSets share similar functionality with ReplicaSets; both create Pods that are expected to be long-running services and ensure that the desired state and the observed state of the cluster match.

When to use - DaemonSet and/or ReplicaSet?
Given the similarities between DaemonSets and ReplicaSets, it’s important to understand when to use one over the other.
•ReplicaSets should be used when your application is completely decoupled from the node and you can run multiple copies on a given node without special consideration.
•DaemonSets should be used when a single copy of your application must run on all or a subset of the nodes in the cluster.

Using a DaemonSet to run a pod on every node
A DaemonSet deploys pods to all nodes in the cluster, unless you specify that the pods should only run on a subset of all the nodes. This is done by specifying the node-Selector property in the pod template, which is part of the DaemonSet

definition (similar to the pod template in a ReplicaSet or ReplicationController).



DaemonSet

```
[root@controller ~]# cat fluentd-daemonset.yml
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: fluentd
 labels:
  app: fluentd
spec:
 selector:
  matchLabels:
   app: fluentd
 template:
  metadata:
   labels:
    app: fluentd
  spec:
   containers:
   - name: fluentd
    image: fluent/fluentd:v0.14.10
    resources:
```

```
    resources:
      limits:
       memory: 200Mi
      requests:
       cpu: 100m
       memory: 200Mi
     volumeMounts:
     - name: varlog
      mountPath: /var/log
     - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
    terminationGracePeriodSeconds: 30
    volumes:
    - name: varlog
     hostPath:
      path: /var/log
    - name: varlibdockercontainers
     hostPath:
      path: /var/lib/docker/containers
```

Resource limits are useful if don't want these daemons to eat up all system's resources.
Once you have a valid DaemonSet configuration in place, you can use the kubectl create command to submit the DaemonSet to the Kubernetes API. In this section we will create a DaemonSet to ensure the fluentd HTTP server is running on every node in our cluster.

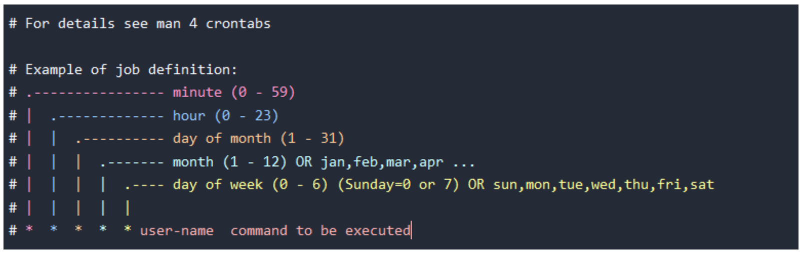
•**Ei = EiB = Exbibyte.** 1Ei = 2^{60} = 1,152,921,504,606,846,976 bytes
•**Pi = PiB = Pebibyte.** 1Pi = 2^{50} = 1,125,899,906,842,624 bytes
•**Ti = TiB = Tebibyte.** 1Ti = 2^{40} = 1,099,511,627,776 bytes
•**Gi = GiB = Gibibyte.** 1Gi = 2^{30} = 1,073,741,824 bytes
•**Mi = MiB = Mebibyte.** 1Mi = 2^{20} = 1,048,576 bytes
•**Ki = KiB = Kibibyte.** 1Ki = 2^{10} = 1,024 bytes

To deploy:
kubectl apply -f .\fluentd-daemonset.yml
To delete:
kubectl delete daemonset fluentd-elasticsearch --namespace=kube-system

The key difference between kubectl apply and create is that **apply creates Kubernetes objects through a declarative syntax, while the create command is imperative**. The important thing to understand about `kubectl create` vs. `kubectl apply` is that you use `kubectl create` to create Kubernetes resources imperatively at the command-line or declaratively against a manifest file. However, you can use `kubectl create` declaratively only to create a new resource.

CronJob

- At the configured time, Kubernetes will create a Job resource according to the Job template configured in the CronJob object. When the Job resource is created, one or more pod replicas will be created and started according to the Job's pod template
- Let us first understand the basics of cron job. You can get the structure of different fields in crontab in /etc/crontab file:

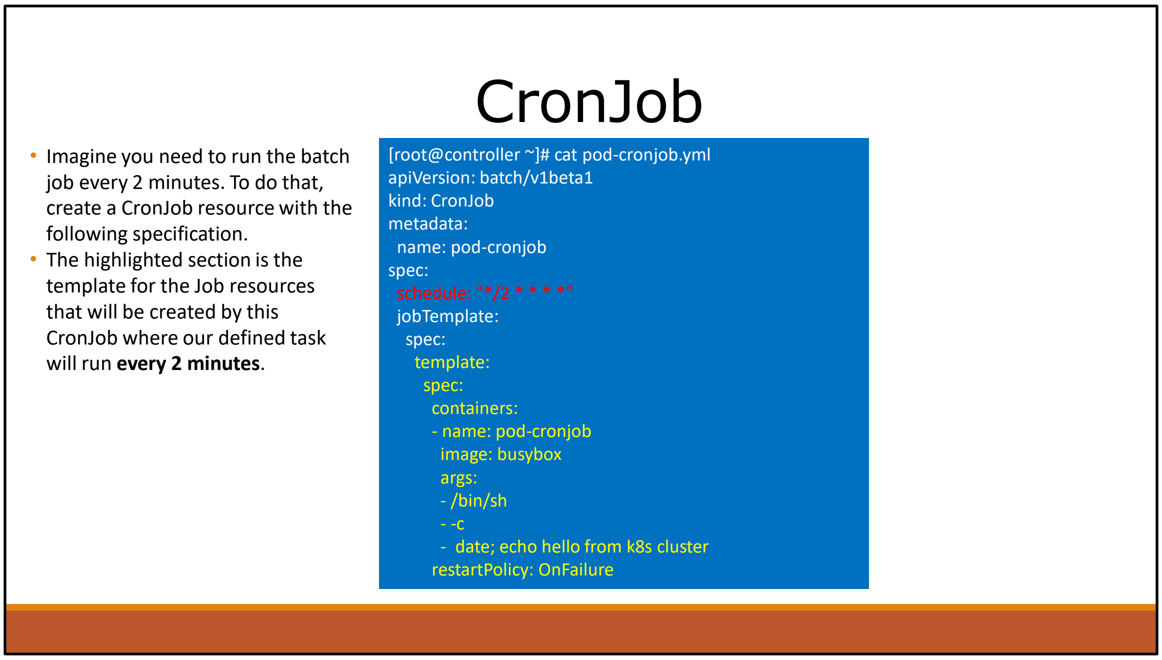


```
# For details see man 4 crontabs

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name  command to be executed
```

ob resources run their pods immediately when you create the Job resource. But many batch jobs need to be run at a specific time in the future or repeatedly in the specified interval. In Linux- and UNIX-like operating systems, these jobs are better known as cron jobs. Kubernetes supports them, too.
A cron job in Kubernetes is configured by creating a CronJob resource. The schedule for running the job is specified in the well-known cron format, so if you're familiar with regular cron jobs, you'll understand Kubernetes' CronJobs in a matter of seconds.

CronJob

- Example:
```
0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
```
- The first section of a cron job, a 0 in this example, refers to the minute that the job will happen. Since 0 is used here, the minutes section is :00.
- The next section is the hour; 5 in this case. Therefore, we can gather so far that the command will be run at 5:00 a.m. (cron uses military time).
- The third section refers to the day of the month, but we have an asterisk (*) here. The asterisk, in this case, means that it doesn't matter what day of the month it is; just run it.
- The fourth section is another asterisk. In this example, it means we don't care which month it is. So far, we have a command we want to run at 5:00 a.m., regardless of the date.
- The fifth field is set to 1 in our example, and this field represents the day of the week. Since we have 1 here, it means that we want to run the command on Mondays. (Sunday would have been 0).
- Finally, the command that we want to execute is listed.
- If we put it all together, this line means we want to run the following command **every Monday at 5:00 a.m**:



CronJob

- Imagine you need to run the batch job every 2 minutes. To do that, create a CronJob resource with the following specification.
- The highlighted section is the template for the Job resources that will be created by this CronJob where our defined task will run **every 2 minutes**.

```
[root@controller ~]# cat pod-cronjob.yml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pod-cronjob
spec:
  schedule: "*/2 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: pod-cronjob
            image: busybox
            args:
            - /bin/sh
            - -c
            -  date; echo hello from k8s cluster
          restartPolicy: OnFailure
```

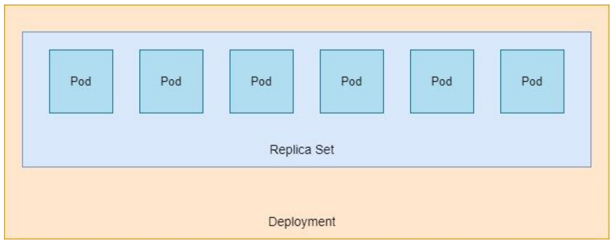
Step values can be used in conjunction with ranges. Following a range with /<number> specifies skips of the number's value through the range. For example, 0-23/2 can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is 0,2,4,6,8,10,12,14,16,18,20,22). Steps are also permitted after an asterisk, so if you want to say "every two hours", just use */2.

Deployment

Deployment: *Deployment* provides declarative updates for Pods and ReplicaSets (multiple replicas of pods).

- When running Pods in datacenter, additional features may be needed such as scalability, updates and rollback etc which are offered by Deployments
- A Deployment is a higher-level resource meant for deploying applications and updating them declaratively, instead of doing it through a ReplicationController or a ReplicaSet, which are both considered lower-level concepts.
- When you create a Deployment, a ReplicaSet resource is created underneath. Replica-Sets replicate and manage pods, as well. Thus, when a Deployment is used, the actual pods are created and managed by the Deployment's ReplicaSets, not by the Deployment directly
- The *desired state* of ReplicaSet is described in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Deployment

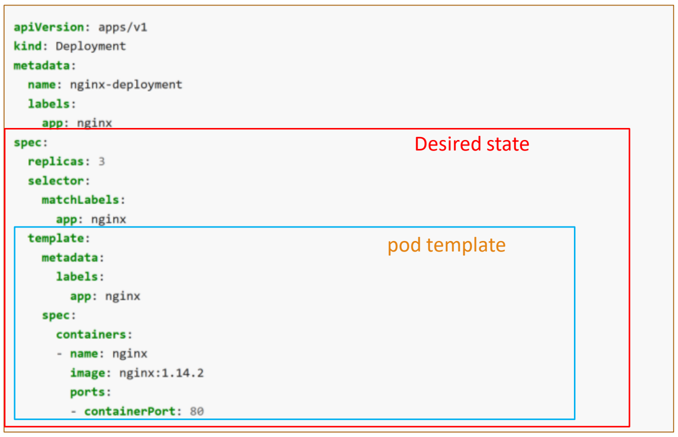


In the deployment spec, following properties are managed:

- **replicas:** explains how many copies of each Pod should be running
- **strategy:** explains how Pods should be updated
- **selector:** uses matchLabels to identify how labels are matched against the Pod
- **template:** contains the pod specification and is used in a deployment to create Pods

Example:

Creation of the Deployment:
kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml

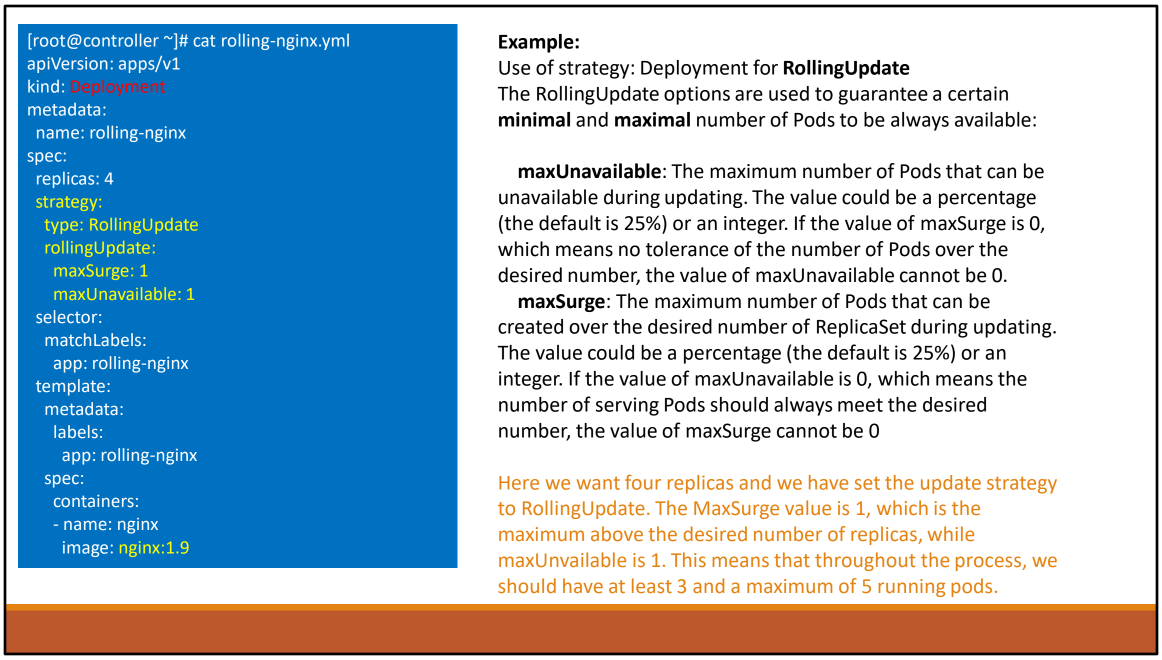


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Deployment

The name of a ReplicaSet object is composed of the name of the controller plus a randomly generated string (for example, nginx-v1-m33mv). The pods created by the Deployment include an additional numeric value in the middle of their names. What is that exactly?
The number corresponds to the hashed value of the pod template in the Deployment and the ReplicaSet managing these pods.

```
[[root@controller ~]# kubectl get pods
NAME                           READY  STATUS   RESTARTS   AGE
nginx                          1/1    Running  5          4d1h
nginx-deploy-d98cc8bdb-48ppw   1/1    Running  0          80s
nginx-deploy-d98cc8bdb-nvcb5   1/1    Running  0          80s
```



```
[root@controller ~]# cat rolling-nginx.yml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: rolling-nginx
spec:
 replicas: 4
 strategy:
  type: RollingUpdate
  rollingUpdate:
   maxSurge: 1
   maxUnavailable: 1
 selector:
  matchLabels:
   app: rolling-nginx
 template:
  metadata:
   labels:
    app: rolling-nginx
  spec:
   containers:
   - name: nginx
    image: nginx:1.9
```

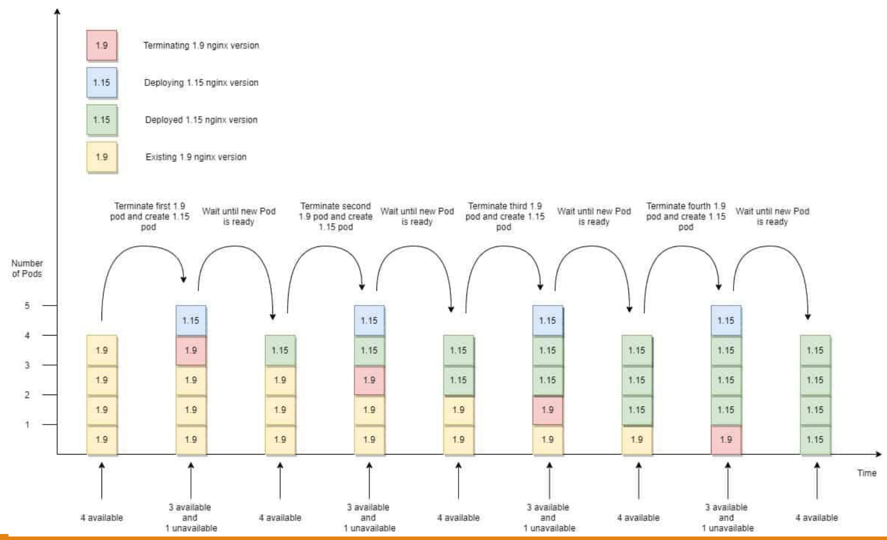
Example:
Use of strategy: Deployment for **RollingUpdate**
The RollingUpdate options are used to guarantee a certain **minimal** and **maximal** number of Pods to be always available:

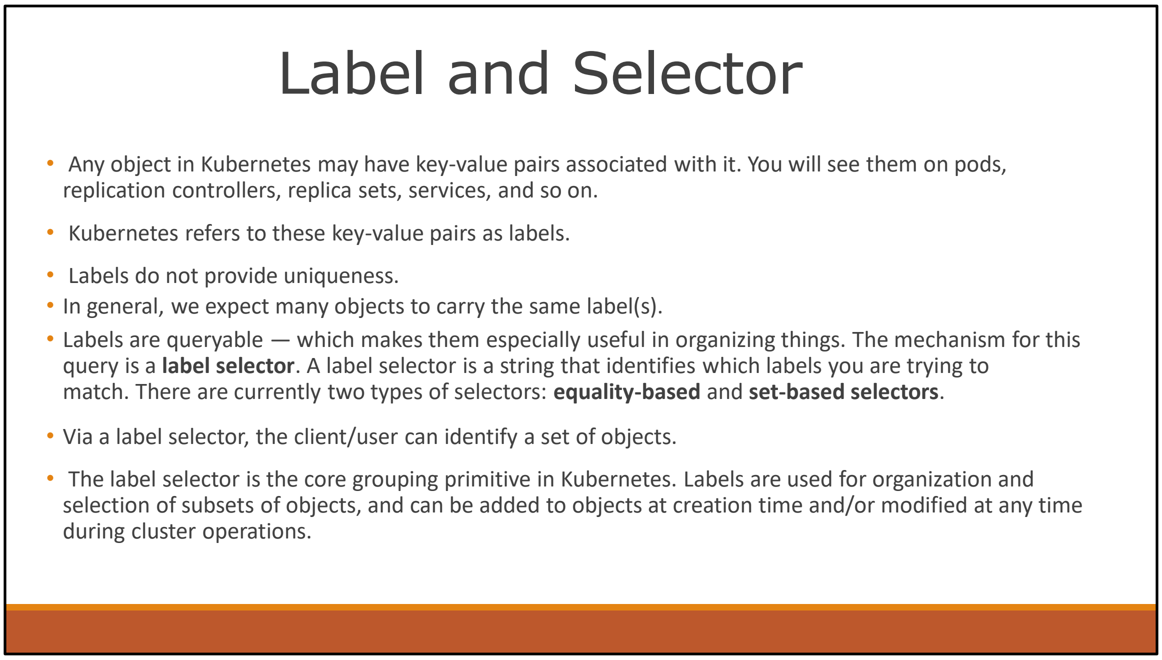
maxUnavailable: The maximum number of Pods that can be unavailable during updating. The value could be a percentage (the default is 25%) or an integer. If the value of maxSurge is 0, which means no tolerance of the number of Pods over the desired number, the value of maxUnavailable cannot be 0.
maxSurge: The maximum number of Pods that can be created over the desired number of ReplicaSet during updating. The value could be a percentage (the default is 25%) or an integer. If the value of maxUnavailable is 0, which means the number of serving Pods should always meet the desired number, the value of maxSurge cannot be 0

Here we want four replicas and we have set the update strategy to RollingUpdate. The MaxSurge value is 1, which is the maximum above the desired number of replicas, while maxUnvailable is 1. This means that throughout the process, we should have at least 3 and a maximum of 5 running pods.

Initially, the pods run the first version of your application, let's suppose its image is tagged as **v1**. You then develop a newer version of the app and push it to an image repository as a new image, tagged as **v2**. You'd next like to replace all the pods with this new version.
You have two ways of updating all those pods. You can do one of the following:
•**Recreate:** Delete all existing pods first and then start the new ones. This will lead to a temporary unavailability.
•**Rolling Update:** Updates Pod one at a time to guarantee availability of the application. This is the preferred approach and you can further tune its behaviour.

Rollout history

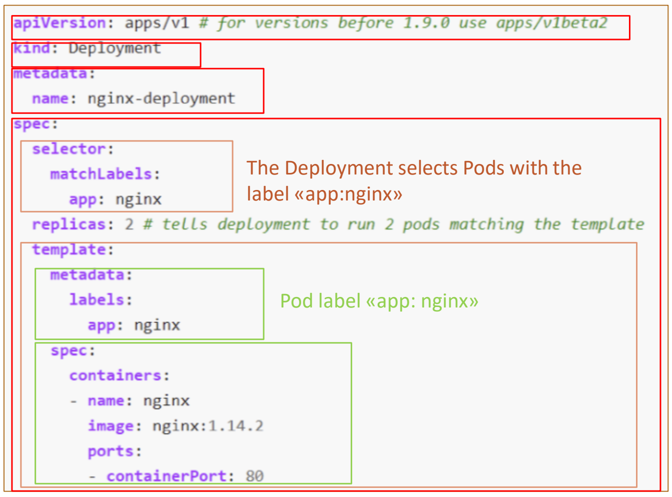

1.9 Terminating 1.9 nginx version
1.15 Deploying 1.15 nginx version
1.15 Deployed 1.15 nginx version
1.9 Existing 1.9 nginx version
Terminate first 1.9 pod and create 1.15 pod
Wait until new Pod is ready
Terminate second 1.9 pod and create 1.15 pod
Wait until new Pod is ready
Terminate third 1.9 pod and create 1.15 pod
Wait until new Pod is ready
Terminate fourth 1.9 pod and create 1.15 pod
Wait until new Pod is ready
Number of Pods
Time
4 available
3 available and 1 unavailable
4 available
3 available and 1 unavailable
4 available
3 available and 1 unavailable
4 available
3 available and 1 unavailable
4 available



Label and Selector

- Any object in Kubernetes may have key-value pairs associated with it. You will see them on pods, replication controllers, replica sets, services, and so on.
- Kubernetes refers to these key-value pairs as labels.
- Labels do not provide uniqueness.
- In general, we expect many objects to carry the same label(s).
- Labels are queryable — which makes them especially useful in organizing things. The mechanism for this query is a **label selector**. A label selector is a string that identifies which labels you are trying to match. There are currently two types of selectors: **equality-based** and **set-based selectors**.
- Via a label selector, the client/user can identify a set of objects.
- The label selector is the core grouping primitive in Kubernetes. Labels are used for organization and selection of subsets of objects, and can be added to objects at creation time and/or modified at any time during cluster operations.

Example



```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

The Deployment selects Pods with the label «app:nginx»

Pod label «app: nginx»

version of the Kubernetes API used to create this object

type of object you want to create

data that helps uniquely identifying the object, including a name string, UID, etc

specific format of the object spec

Label and Selector

Equality-based selector

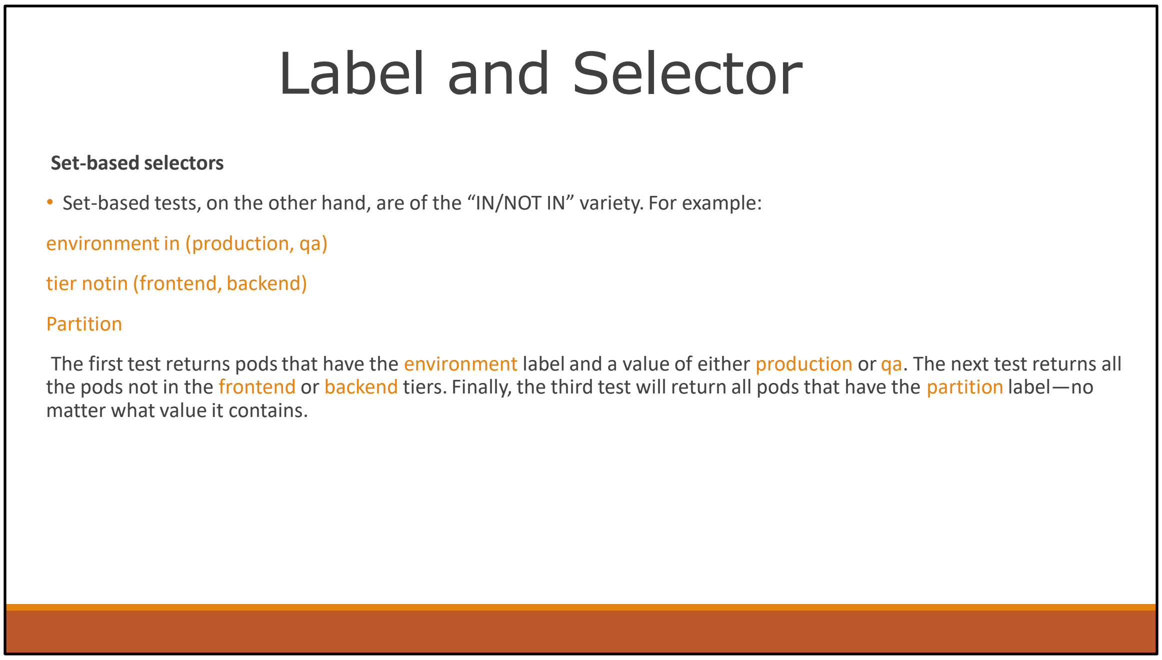
- An equality-based test is just a “IS/IS NOT” test.
- For example: tier = frontend will return all pods that have a label with the key “tier” and the value “frontend”. On the other hand, if we wanted to get all the pods that were not in the frontend tier, we would say:

tier != frontend

- You can also combine requirements with commas like so:

tier != frontend, game = super-shooter-2

This would return all pods that were part of the game named super-shooter-2 but were not in its frontend tier.



Label and Selector

Set-based selectors

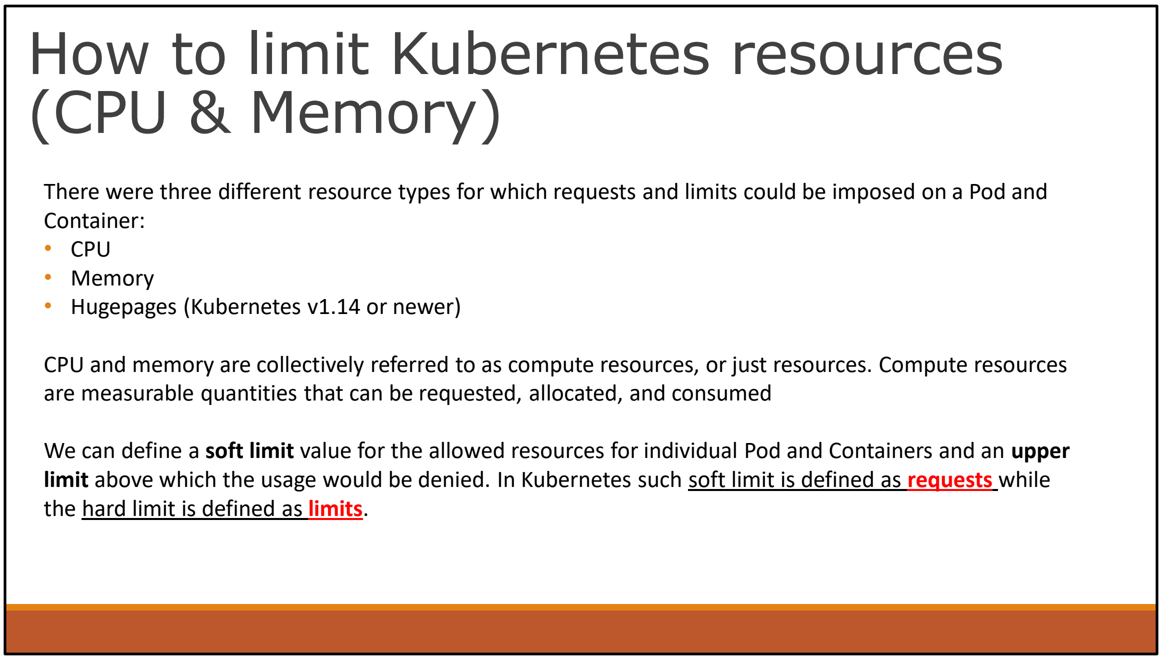
- Set-based tests, on the other hand, are of the “IN/NOT IN” variety. For example:

environment in (production, qa)

tier notin (frontend, backend)

Partition

The first test returns pods that have the environment label and a value of either production or qa. The next test returns all the pods not in the frontend or backend tiers. Finally, the third test will return all pods that have the partition label—no matter what value it contains.



How to limit Kubernetes resources (CPU & Memory)

There were three different resource types for which requests and limits could be imposed on a Pod and Container:

- CPU
- Memory
- Hugepages (Kubernetes v1.14 or newer)

CPU and memory are collectively referred to as compute resources, or just resources. Compute resources are measurable quantities that can be requested, allocated, and consumed

We can define a **soft limit** value for the allowed resources for individual Pod and Containers and an **upper limit** above which the usage would be denied. In Kubernetes such soft limit is defined as **requests** while the hard limit is defined as **limits**.

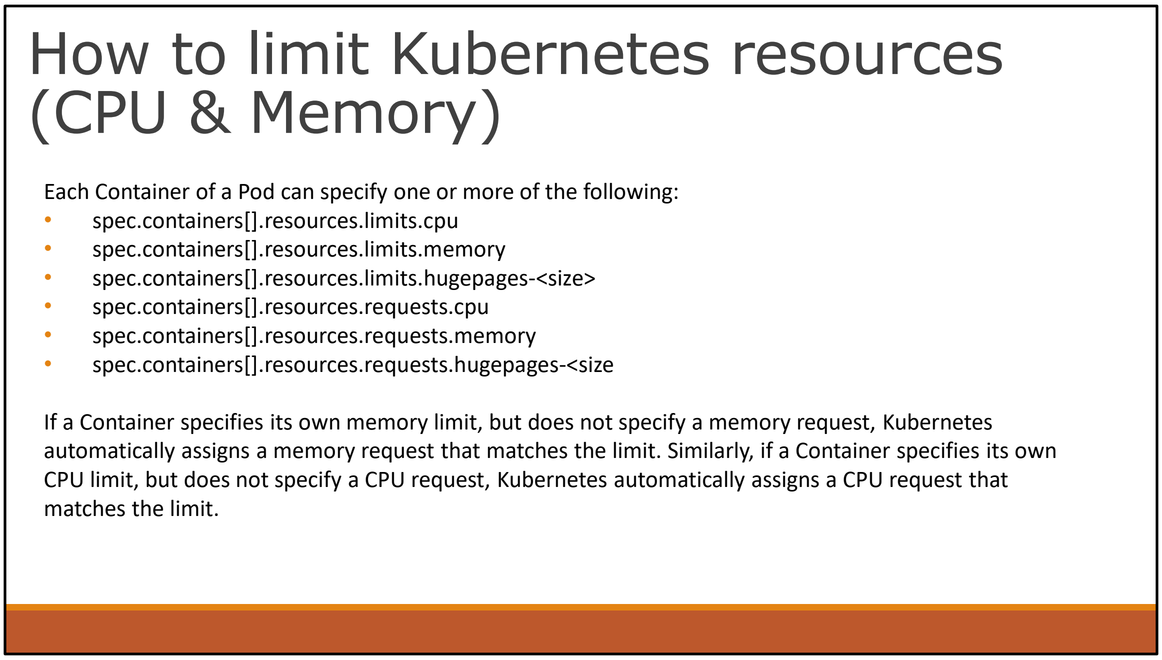
If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its request for that resource specifies. However, a container is not allowed to use more than its resource limit.

Huge pages can be consumed via container level resource requirements using the resource name hugepages-<size>, where <size> is the most compact binary notation using integer values supported on a particular node. For example, if a node supports 2048KiB and 1048576KiB page sizes, it will expose a schedulable resources hugepages-2Mi and hugepages-1Gi. Unlike CPU or memory, huge pages do not support overcommit. Note that when requesting hugepage resources, either memory or CPU resources must be requested as well.

A pod may consume multiple huge page sizes in a single pod spec. In this case it must use medium: HugePages-<hugepagesize> notation for all volume mounts.

```
apiVersion: v1
kind: Pod
metadata:
  name: huge-pages-example
```

```
spec:
  containers:
  - name: example
    image: fedora:latest
    command:
    - sleep
    - inf
    volumeMounts:
    - mountPath: /hugepages-2Mi
      name: hugepage-2mi
    - mountPath: /hugepages-1Gi
      name: hugepage-1gi
    resources:
      limits:
        hugepages-2Mi: 100Mi
        hugepages-1Gi: 2Gi
        memory: 100Mi
      requests:
        memory: 100Mi
  volumes:
  - name: hugepage-2mi
    emptyDir:
      medium: HugePages-2Mi
  - name: hugepage-1gi
    emptyDir:
      medium: HugePages-1Gi
```



How to limit Kubernetes resources (CPU & Memory)

Each Container of a Pod can specify one or more of the following:

- spec.containers[].resources.limits.cpu
- spec.containers[].resources.limits.memory
- spec.containers[].resources.limits.hugepages-<size>
- spec.containers[].resources.requests.cpu
- spec.containers[].resources.requests.memory
- spec.containers[].resources.requests.hugepages-<size

If a Container specifies its own memory limit, but does not specify a memory request, Kubernetes automatically assigns a memory request that matches the limit. Similarly, if a Container specifies its own CPU limit, but does not specify a CPU request, Kubernetes automatically assigns a CPU request that matches the limit.

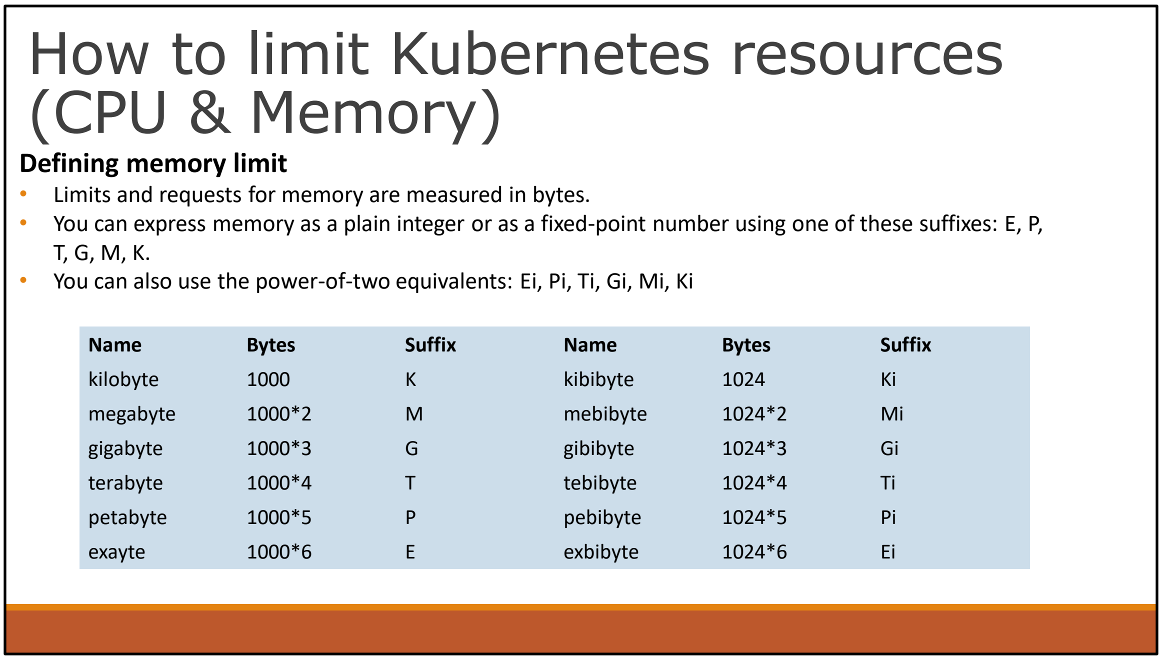
If you do not specify a CPU limit for a Container, then one of these situations applies:

- The Container has no upper bound on the CPU resources it can use. The Container could use all of the CPU resources available on the Node where it is running.
- The Container is running in a namespace that has a default CPU limit, and the Container is automatically assigned the default limit. Cluster administrators can use a LimitRange to specify a default value for the CPU limit.

How to limit Kubernetes resources (CPU & Memory)

Defining CPU limit

- Limits and requests for CPU resources are measured in cpu units.
- **One cpu**, in Kubernetes, is equivalent to **1 vCPU/Core** for cloud providers and **1 hyperthread** on bare-metal Intel processors.
- Whenever we specify CPU requests or limits, we specify them in terms of CPU cores.
- Because often we want to request or limit the use of a pod to some fraction of a whole CPU core, we can either specify this fraction of a CPU as a decimal or as a millicore value.
- For example, a value of 0.5 represents half of a core.
- It is also possible to configure requests or limits with a millicore value. As there are 1,000 millicores to a single core, we could specify half a CPU as 500 m.
- The smallest amount of CPU that can be specified is 1 m or 0.001.



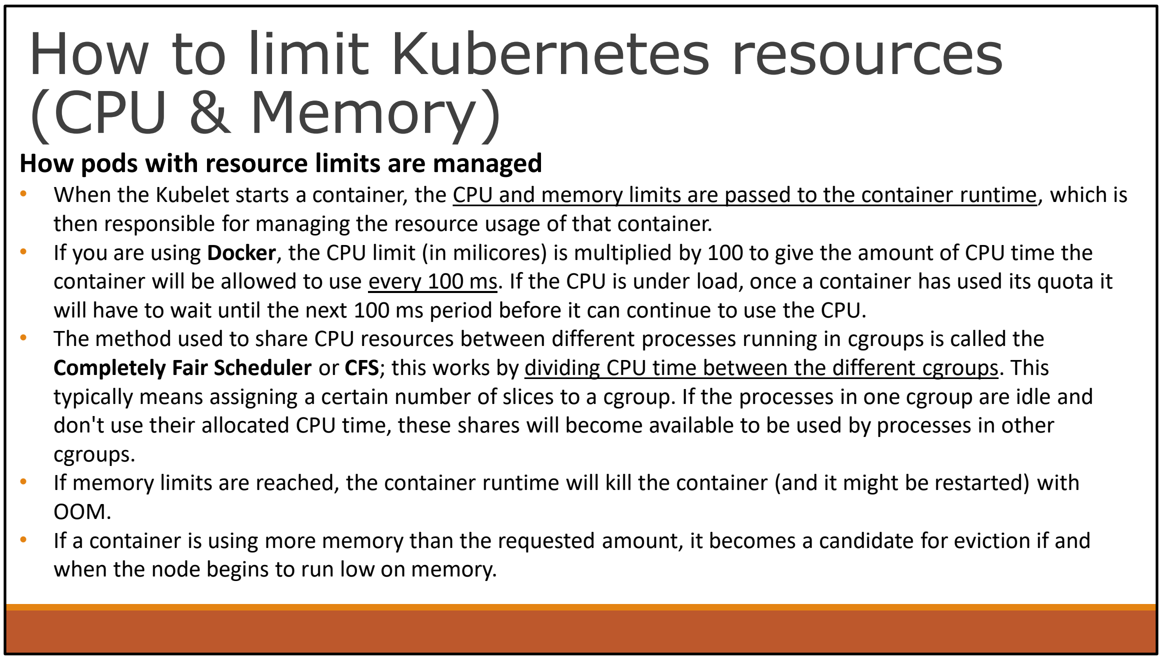
How to limit Kubernetes resources (CPU & Memory)

Defining memory limit

- Limits and requests for memory are measured in bytes.
- You can express memory as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, K.
- You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki

Name	Bytes	Suffix	Name	Bytes	Suffix
kilobyte	1000	K	kibibyte	1024	Ki
megabyte	1000*2	M	mebibyte	1024*2	Mi
gigabyte	1000*3	G	gibibyte	1024*3	Gi
terabyte	1000*4	T	tebibyte	1024*4	Ti
petabyte	1000*5	P	pebibyte	1024*5	Pi
exayte	1000*6	E	exbibyte	1024*6	Ei

Memory units supported by Kubernetes



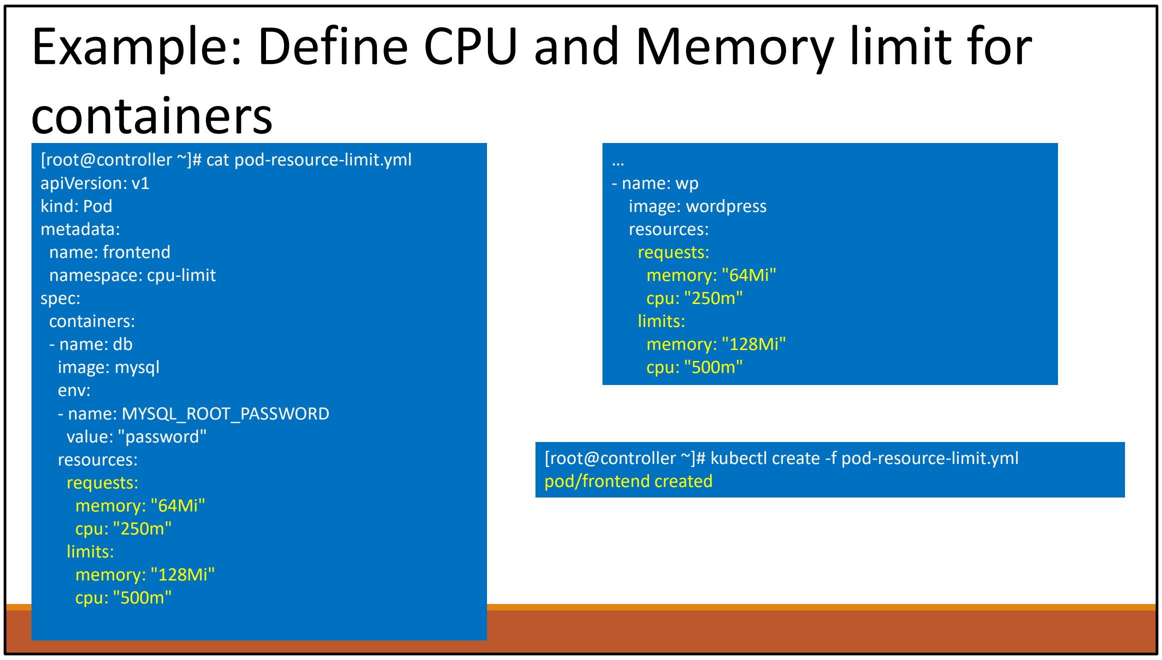
How to limit Kubernetes resources (CPU & Memory)

How pods with resource limits are managed

- When the Kubelet starts a container, the CPU and memory limits are passed to the container runtime, which is then responsible for managing the resource usage of that container.
- If you are using **Docker**, the CPU limit (in milicores) is multiplied by 100 to give the amount of CPU time the container will be allowed to use every 100 ms. If the CPU is under load, once a container has used its quota it will have to wait until the next 100 ms period before it can continue to use the CPU.
- The method used to share CPU resources between different processes running in cgroups is called the **Completely Fair Scheduler** or **CFS**; this works by dividing CPU time between the different cgroups. This typically means assigning a certain number of slices to a cgroup. If the processes in one cgroup are idle and don't use their allocated CPU time, these shares will become available to be used by processes in other cgroups.
- If memory limits are reached, the container runtime will kill the container (and it might be restarted) with OOM.
- If a container is using more memory than the requested amount, it becomes a candidate for eviction if and when the node begins to run low on memory.

cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

When working with Kubernetes, Out of Memory (OOM) errors and CPU throttling are typica issues in handling cloud applications. When a process runs Out Of Memory (OOM), it’s killed since it doesn’t have the required resources.
In case CPU consumption is higher than the actual limits, the process will start to be throttled.

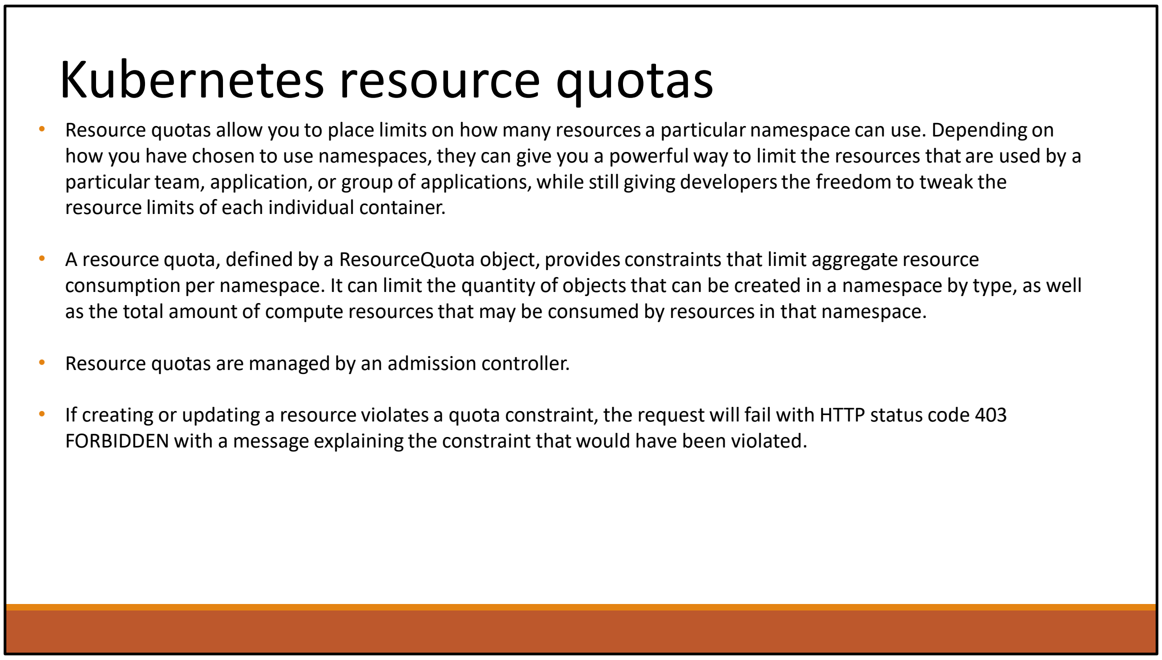


Example: Define CPU and Memory limit for containers

```
[root@controller ~]# cat pod-resource-limit.yml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  namespace: cpu-limit
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

```
…
- name: wp
  image: wordpress
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
```

```
[root@controller ~]# kubectl create -f pod-resource-limit.yml
pod/frontend created
```

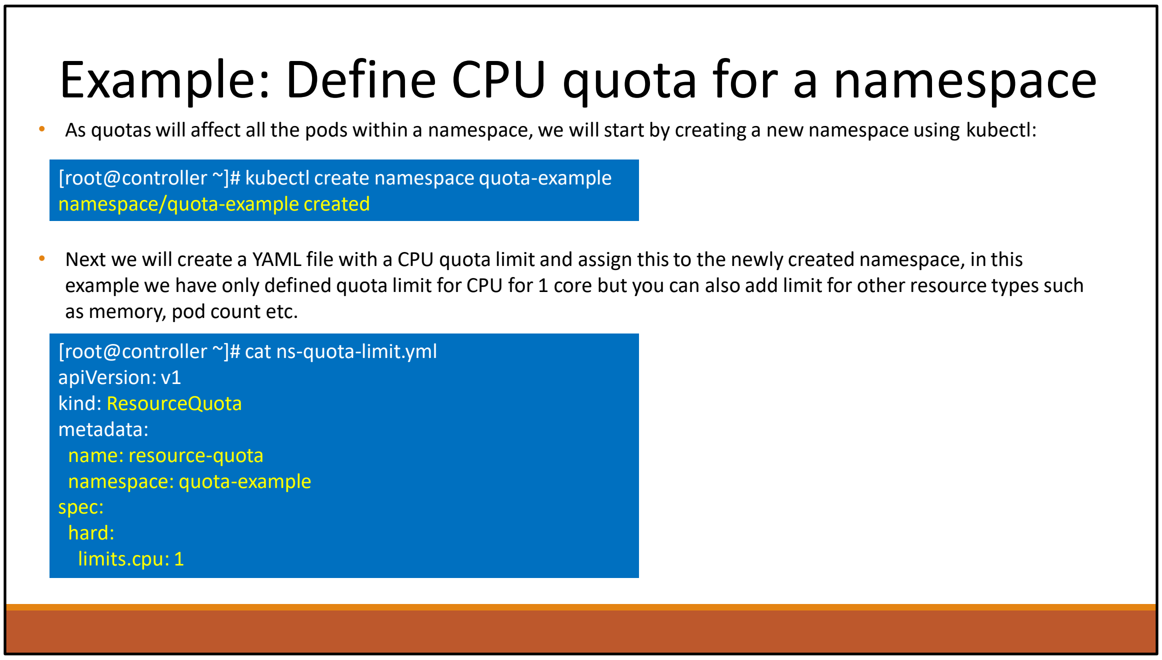


Kubernetes resource quotas

- Resource quotas allow you to place limits on how many resources a particular namespace can use. Depending on how you have chosen to use namespaces, they can give you a powerful way to limit the resources that are used by a particular team, application, or group of applications, while still giving developers the freedom to tweak the resource limits of each individual container.
- A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that namespace.
- Resource quotas are managed by an admission controller.
- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code 403 FORBIDDEN with a message explaining the constraint that would have been violated.

The following resource types are supported:

Resource Name	Description
limits.cpu	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
limits.memory	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
requests.cpu	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
requests.memory	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
hugepages-<size>	Across all pods in a non-terminal state, the number of huge page requests of the specified size cannot exceed this value.
cpu	Same as requests.cpu
memory	Same as requests.memory



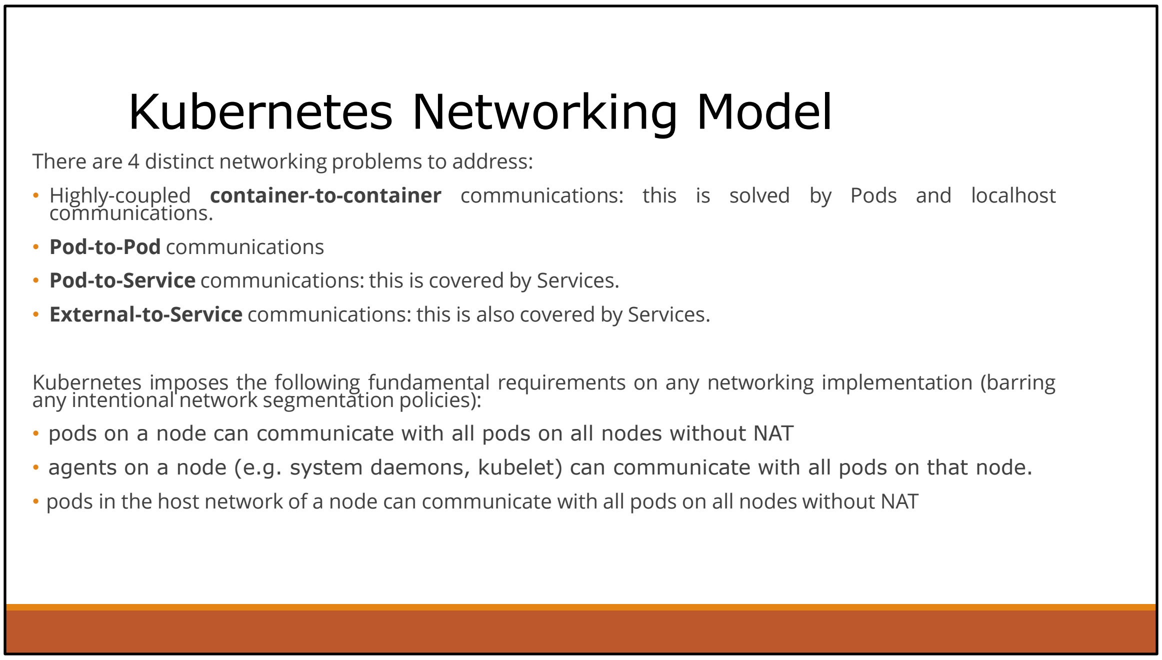
Example: Define CPU quota for a namespace

- As quotas will affect all the pods within a namespace, we will start by creating a new namespace using kubectl:

```
[root@controller ~]# kubectl create namespace quota-example
namespace/quota-example created
```

- Next we will create a YAML file with a CPU quota limit and assign this to the newly created namespace, in this example we have only defined quota limit for CPU for 1 core but you can also add limit for other resource types such as memory, pod count etc.

```
[root@controller ~]# cat ns-quota-limit.yml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: resource-quota
  namespace: quota-example
spec:
  hard:
    limits.cpu: 1
```



Kubernetes Networking Model

There are 4 distinct networking problems to address:

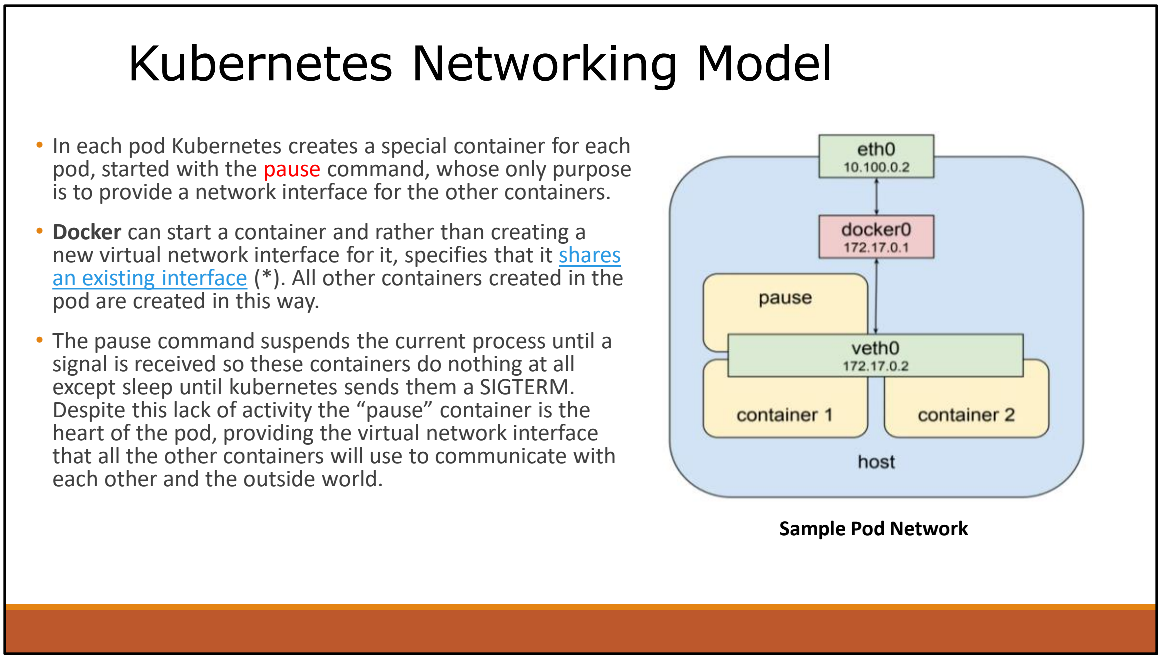
- Highly-coupled **container-to-container** communications: this is solved by Pods and localhost communications.
- **Pod-to-Pod** communications
- **Pod-to-Service** communications: this is covered by Services.
- **External-to-Service** communications: this is also covered by Services.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- pods on a node can communicate with all pods on all nodes without NAT
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node.
- pods in the host network of a node can communicate with all pods on all nodes without NAT

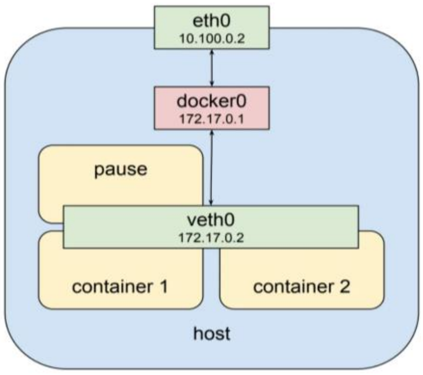
If a pod runs in the host network of the node where the pod is deployed, the pod can use the network namespace and network resources of the node. In this case, the pod can access loopback devices, listen to addresses, and monitor the traffic of other pods on the node.
If the IP address of the pod is the same as that of the node, this indicates that the pod runs in the host network of the node.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  hostNetwork: true
  containers:
    - name: nginx
      image: nginx
```



Kubernetes Networking Model

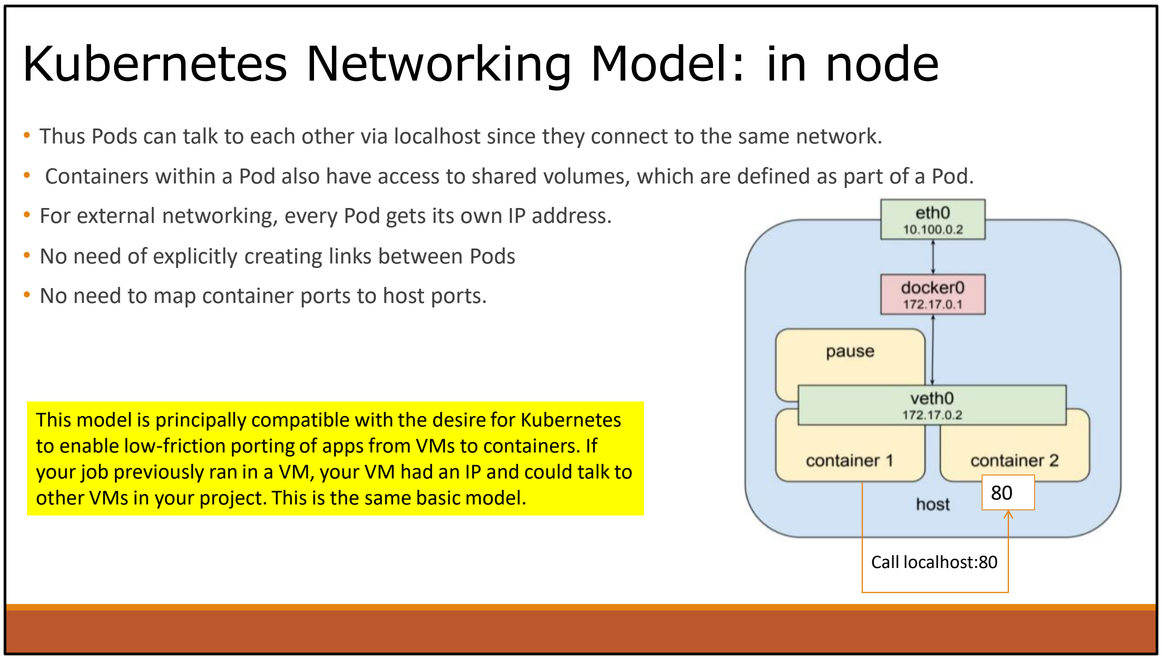
- In each pod Kubernetes creates a special container for each pod, started with the pause command, whose only purpose is to provide a network interface for the other containers.
- **Docker** can start a container and rather than creating a new virtual network interface for it, specifies that it shares an existing interface (*). All other containers created in the pod are created in this way.
- The pause command suspends the current process until a signal is received so these containers do nothing at all except sleep until kubernetes sends them a SIGTERM. Despite this lack of activity the "pause" container is the heart of the pod, providing the virtual network interface that all the other containers will use to communicate with each other and the outside world.



Sample Pod Network

(*) https://docs.docker.com/engine/reference/run/#network-settings

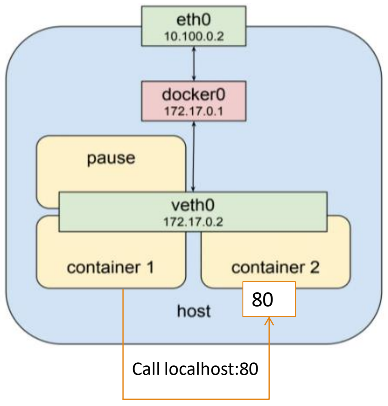
Kubernetes deviates from the default Docker networking model (though as of Docker 1.8 their network plugins are getting closer). The goal is for each pod to have an IP in a flat shared networking namespace that has full communication with other physical computers and containers across the network. IP-per-pod creates a clean, backward-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.



Kubernetes Networking Model: in node

- Thus Pods can talk to each other via localhost since they connect to the same network.
- Containers within a Pod also have access to shared volumes, which are defined as part of a Pod.
- For external networking, every Pod gets its own IP address.
- No need of explicitly creating links between Pods
- No need to map container ports to host ports.

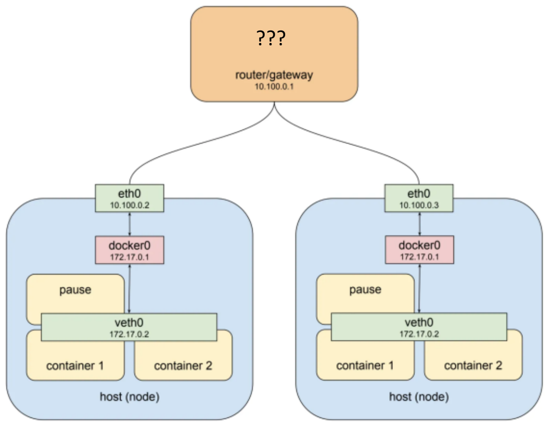
This model is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.



(*) https://docs.docker.com/engine/reference/run/#network-settings

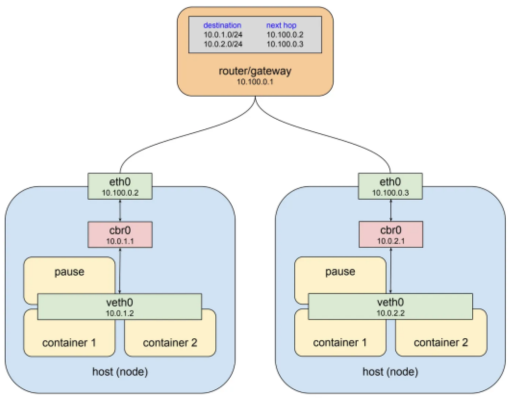
Kubernetes deviates from the default Docker networking model (though as of Docker 1.8 their network plugins are getting closer). The goal is for each pod to have an IP in a flat shared networking namespace that has full communication with other physical computers and containers across the network. IP-per-pod creates a clean, backward-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes Networking Model: out node



- Each pod on a node has its own network namespace. Each pod has its own IP address, and each pod thinks it has a totally normal ethernet device called eth0 to make network requests through.
- Both Pods can legitimately enclose the same 172.17.0.0/24 subnet.
- It is worst case, and it might very well work out that way if you just installed docker and let it do its thing.
- Even if the used network is different this highlights the more fundamental problem which is that one node typically has no idea what private address space was assigned to a bridge on another node, and we need to know that if we're going to send packets to it and have them arrive at the right place.
- Clearly some managing structure, as an **overlay network**, is required

Pod-to-Pod Networking: out node



- Kubernetes has to assigns an overall address space for the bridges on each node, and then assigns the bridges addresses within that space, based on the node the bridge is built on.
- Secondly it adds routing rules to the gateway.
- The changed of name of the bridges from “docker0” to “cbr0,” short for “custom bridge,” indicates that Kubernetes does not use the standard docker bridge.
- Generally it is non necessary to think about how this network functions. When a pod talks with another pod it most often does so through the abstraction of a service.

Since every pod gets a "real" (not machine-private) IP address, pods can communicate without proxies or translations.

The standard networking scenario in Kubernetes is that each pod has a single network interface (eth0). The pod behaves like a single host, which can communicate with every other host on the same node, and usually with all other nodes on the network, depending on the Container Network Interface(CNI) in use.

We set up the Pods to each have their one network namespace so that they believe they have their own Ethernet device and IP address, and they are connected to the root namespace for the Node. Now, we want the Pods to talk to each other through the root namespace, and for this we use a network *bridge*.

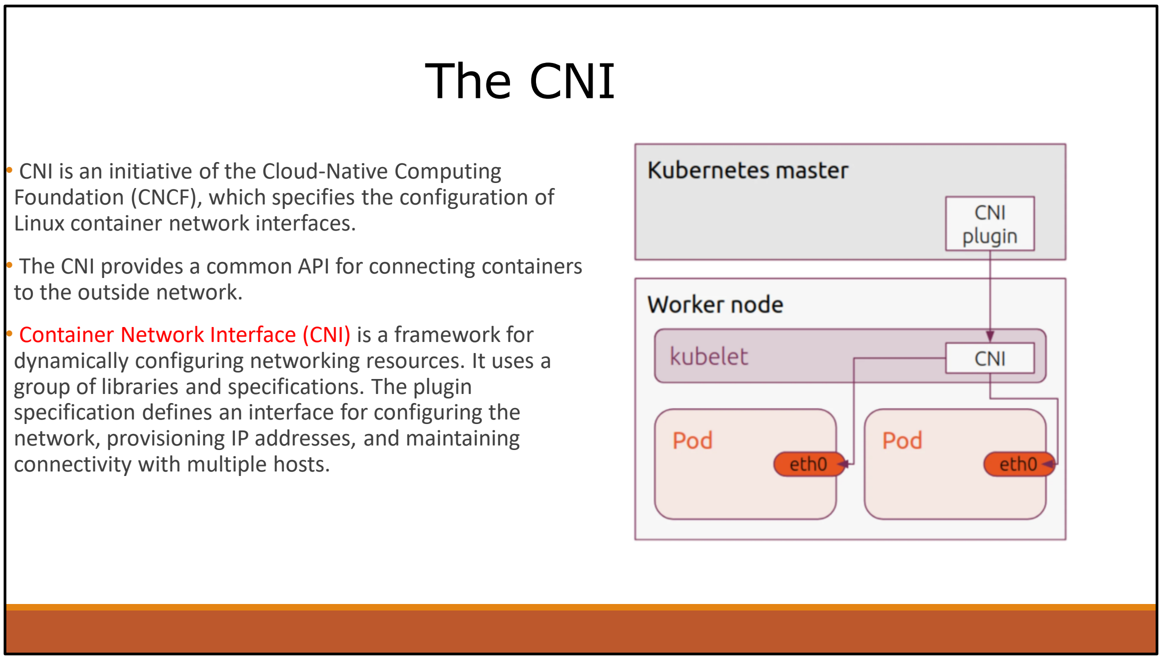
When a pod makes a request to the IP address of another node, it makes that request through its own eth0 interface. This tunnels to the node’s respective virtual vethX interface.

Every Kubernetes Pod includes an empty pause container, which bootstraps the Pod to establish all of the cgroups, reservations, and namespaces before its individual containers are created. The pause container image is always present, so the pod resource allocation happens instantaneously as containers are created.

To display the pause container:
docker ps -a | grep -I pause

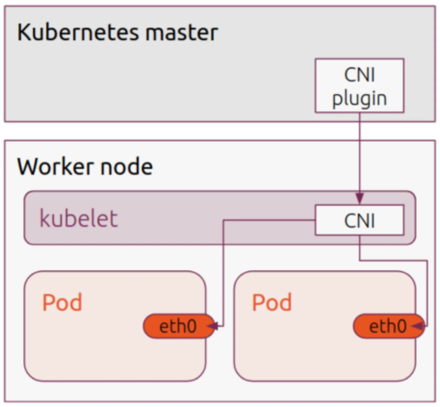
The **veth** devices are virtual Ethernet devices. They can act as tunnels between network namespaces to create a bridge to a physical network device in another namespace, but can also be used as standalone network devices.

From the Pod's perspective, it exists in its own Ethernet namespace that needs to communicate with other network namespaces on the same Node. Thankfully, namespaces can be connected using a Linux Virtual Ethernet Device or *veth pair* consisting of two virtual interfaces that can be spread over multiple namespaces. To connect Pod namespaces, we can assign one side of the veth pair to the root network namespace, and the other side to the Pod's network namespace. Each veth pair works like a patch cable, connecting the two sides and allowing traffic to flow between them. This setup can be replicated for as many Pods as we have on the machine.



The CNI

- CNI is an initiative of the Cloud-Native Computing Foundation (CNCF), which specifies the configuration of Linux container network interfaces.
- The CNI provides a common API for connecting containers to the outside network.
- Container Network Interface (CNI) is a framework for dynamically configuring networking resources. It uses a group of libraries and specifications. The plugin specification defines an interface for configuring the network, provisioning IP addresses, and maintaining connectivity with multiple hosts.



https://kubernetes.io/docs/concepts/cluster-administration/networking/

Since every pod gets a "real" (not machine-private) IP address, pods can communicate without proxies or translations.

The standard networking scenario in Kubernetes is that each pod has a single network interface (eth0). The pod behaves like a single host, which can communicate with every other host on the same node, and usually with all other nodes on the network, depending on the Container Network Interface(CNI) in use.

Cluster networking provides communication between different Pods.

The Kubernetes networking model requires that Pod IPs are reachable across the network, but it doesn’t specify how that must be done. In practice, this is network specific, but some patterns have been established to make this easier.

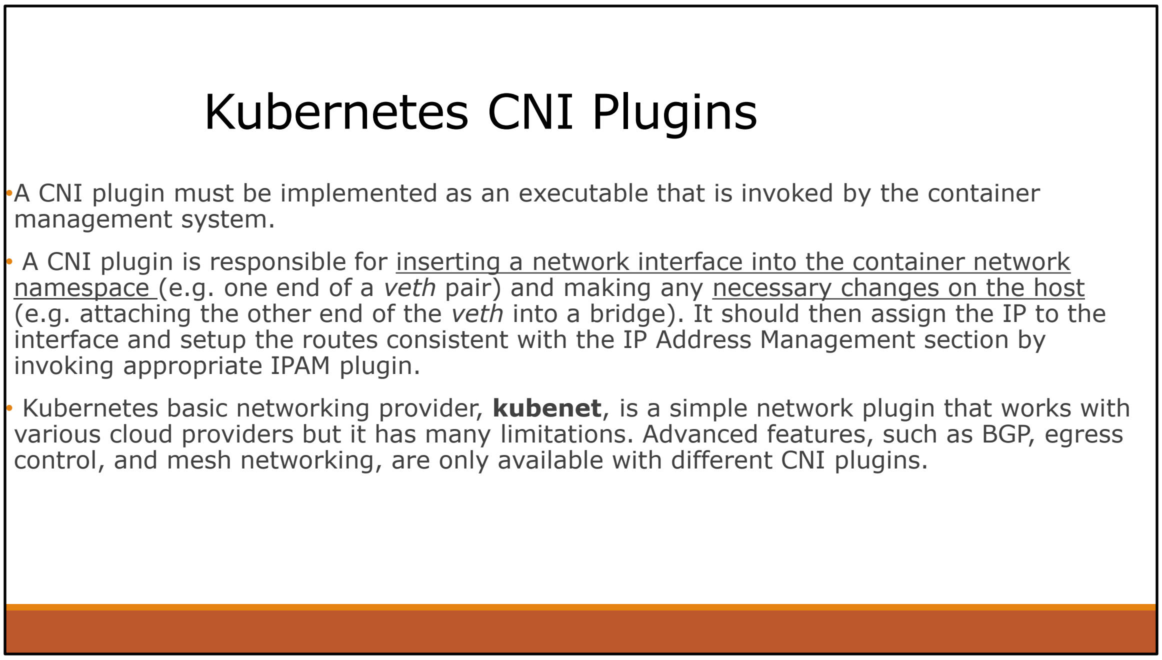
Generally, every Node in your cluster is assigned a CIDR block specifying the IP addresses available to Pods running on that Node. Once traffic destined for the CIDR block reaches the Node it is the Node’s responsibility to forward traffic to the correct

Pod. The figure illustrates the flow of traffic between two Nodes *assuming that the network can route traffic in a CIDR block to the correct Node*.

The CNI provides a common API for connecting containers to the outside network. As developers, we want to know that a Pod can communicate with the network using IP addresses, and we want the mechanism for this action to be transparent. For example, the CNI plugin developed by AWS tries to meet these needs while providing a secure and manageable environment through the existing VPC, IAM, and Security Group functionality provided by AWS. The solution to this is using elastic network interfaces.

You must use a CNI plugin that is compatible with your cluster and that suits your needs. Different plugins are available (both open- and closed- source) in the wider Kubernetes ecosystem.

The Container Runtime Interface (CRI) manages its own CNI plugins. There are two Kubelet command line parameters to keep in mind when using plugins:
•cni-bin-dir: Kubelet probes this directory for plugins on startup
•network-plugin: The network plugin to use from cni-bin-dir. It must match the name reported by a plugin probed from the plugin directory. For CNI plugins, this is cni.

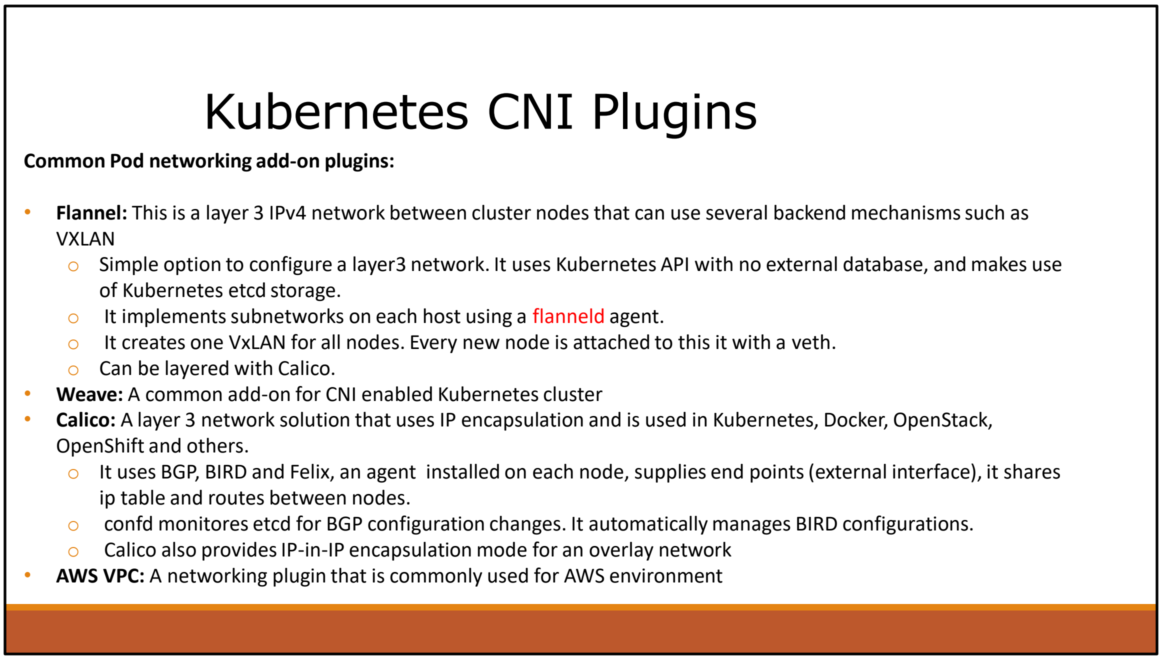


Kubernetes CNI Plugins

- A CNI plugin must be implemented as an executable that is invoked by the container management system.
- A CNI plugin is responsible for inserting a network interface into the container network namespace (e.g. one end of a *veth* pair) and making any necessary changes on the host (e.g. attaching the other end of the *veth* into a bridge). It should then assign the IP to the interface and setup the routes consistent with the IP Address Management section by invoking appropriate IPAM plugin.
- Kubernetes basic networking provider, **kubenet**, is a simple network plugin that works with various cloud providers but it has many limitations. Advanced features, such as BGP, egress control, and mesh networking, are only available with different CNI plugins.

IPAM: IP address management plugin.

Kubenet is a very basic, simple network plugin, on Linux only. It does not, of itself, implement more advanced features like cross-node networking or network policy. It is typically used together with a cloud provider that sets up routing rules for communication between nodes, or in single-node environments.
Kubenet creates a Linux bridge named cbr0 and creates a veth pair for each pod with the host end of each pair connected to cbr0. The pod end of the pair is assigned an IP address allocated from a range assigned to the node either through configuration or by the controller-manager. cbr0 is assigned an MTU matching the smallest MTU of an enabled normal interface on the host.



Kubernetes CNI Plugins

Common Pod networking add-on plugins:

- **Flannel:** This is a layer 3 IPv4 network between cluster nodes that can use several backend mechanisms such as VXLAN
  - Simple option to configure a layer3 network. It uses Kubernetes API with no external database, and makes use of Kubernetes etcd storage.
  - It implements subnetworks on each host using a flanneld agent.
  - It creates one VxLAN for all nodes. Every new node is attached to this it with a veth.
  - Can be layered with Calico.
- **Weave:** A common add-on for CNI enabled Kubernetes cluster
- **Calico:** A layer 3 network solution that uses IP encapsulation and is used in Kubernetes, Docker, OpenStack, OpenShift and others.
  - It uses BGP, BIRD and Felix, an agent installed on each node, supplies end points (external interface), it shares ip table and routes between nodes.
  - confd monitores etcd for BGP configuration changes. It automatically manages BIRD configurations.
  - Calico also provides IP-in-IP encapsulation mode for an overlay network
- **AWS VPC:** A networking plugin that is commonly used for AWS environment

https://www.golinuxcloud.com/calico-kubernetes/

The project Calico attempts to solve the speed and efficiency problems that using virtual LANs, bridging, and tunneling can cause. It achieves this by connecting your containers to a vRouter, which then routes traffic directly over the L3 network. This can give huge advantages when you are sending data between multiple data centers as there is no reliance on NAT and the smaller packet sizes reduce CPU utilization.

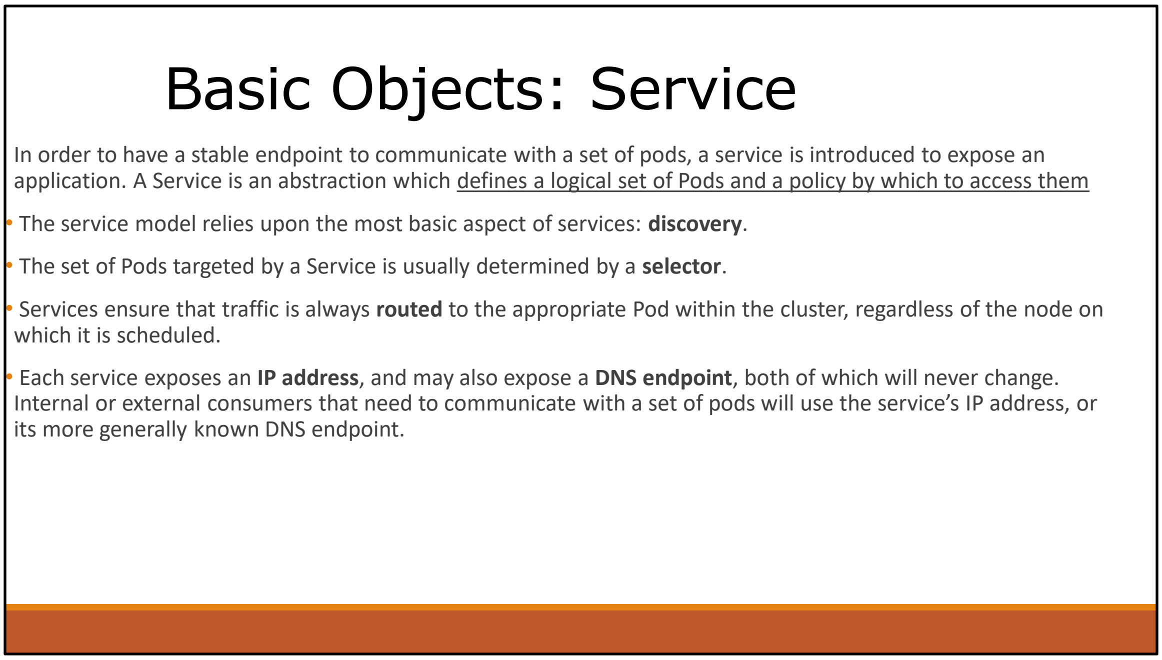
Basic Objects: Service

Pod to Service Networking

- Pod networking in a cluster is neat stuff, but by itself it is insufficient to enable the creation of **durable** systems.
- Unlike in the non-Kubernetes world, where a sysadmin would configure each client app by specifying the exact IP address or hostname of the server providing the service in the client's configuration files, doing the same in Kubernetes wouldn't work, because pods in kubernetes are ephemeral. They may **come and go at any time** - e.g a pod is removed from a node to make room for other pods, because someone scaled down the number of pods, or because a cluster node has failed. if you use a pod IP address as an endpoint there is no guarantee that the address won't change the next time the pod is recreated, which might happen for any number of reasons.
- Kubernetes assigns an **IP address** to a pod after the pod has been scheduled to a node and **before it is started**. Clients thus can't know the IP address of the server pod up front.
- **Horizontal scaling** means multiple pods may provide the same service. Each of those pods has its own IP address. Clients shouldn't care how many pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of pods. Instead, all those pods should be accessible through a single IP address.

Basic Objects: Service

- This is a quite old problem, and it has a standard solution: run the traffic through a **reverse-proxy/load balancer**. Clients connect to the proxy and the proxy is responsible for maintaining a list of healthy servers to forward requests to. This implies a few requirements for the proxy: it must itself be durable and resistant to failure; it must have a list of servers it can forward to; and it must have some way of knowing if a particular server is healthy and able to respond to requests.
- The kubernetes designers solved this problem in an elegant way that builds on the basic capabilities of the platform to deliver on all three of those requirements, and it starts with a resource type called a **service**.



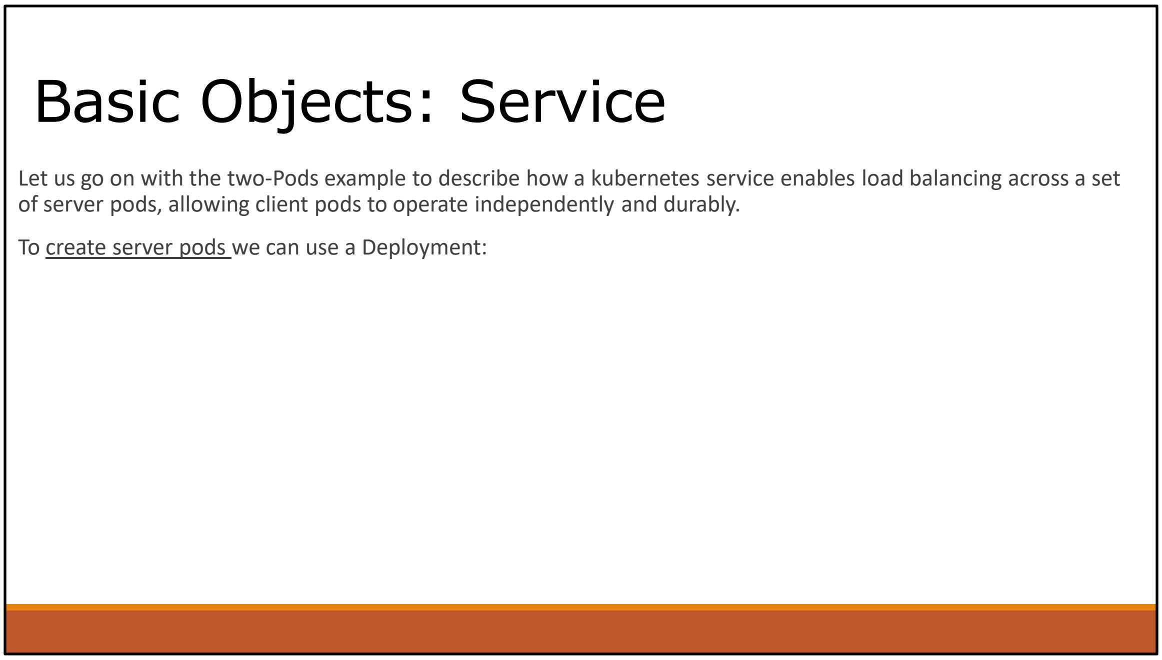
Basic Objects: Service

In order to have a stable endpoint to communicate with a set of pods, a service is introduced to expose an application. A Service is an abstraction which defines a logical set of Pods and a policy by which to access them

- The service model relies upon the most basic aspect of services: **discovery**.
- The set of Pods targeted by a Service is usually determined by a **selector**.
- Services ensure that traffic is always **routed** to the appropriate Pod within the cluster, regardless of the node on which it is scheduled.
- Each service exposes an **IP address**, and may also expose a **DNS endpoint**, both of which will never change. Internal or external consumers that need to communicate with a set of pods will use the service's IP address, or its more generally known DNS endpoint.

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a sevice can stay alive and using the new pod replacing the died one along with the new IP address of the pod.



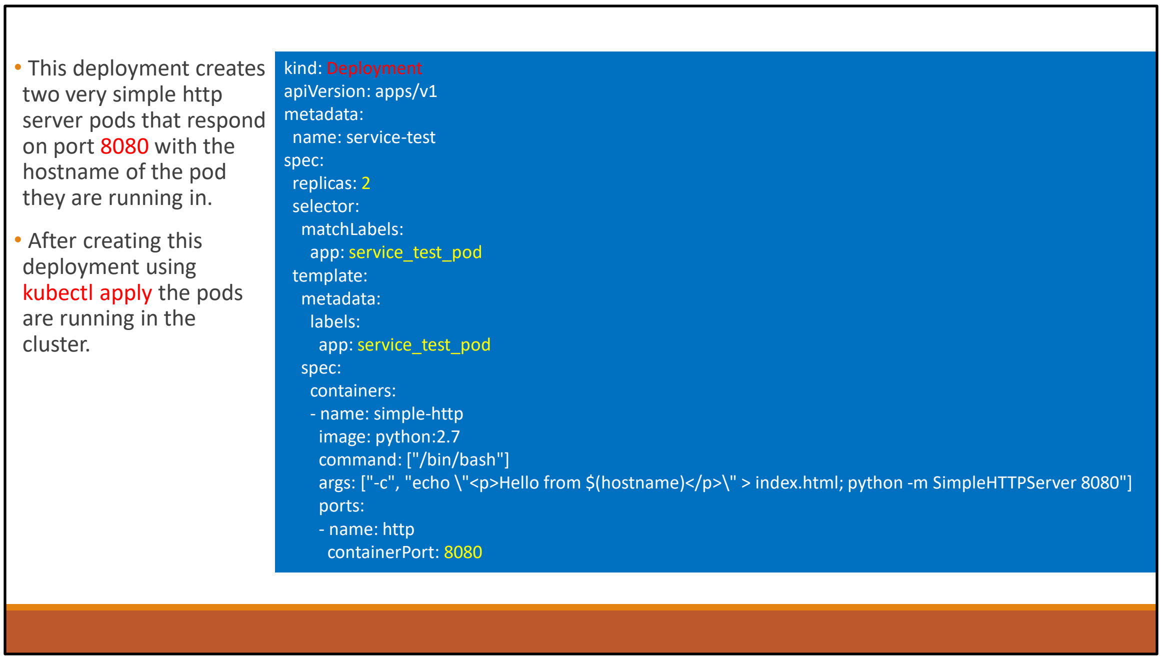
Basic Objects: Service

Let us go on with the two-Pods example to describe how a kubernetes service enables load balancing across a set of server pods, allowing client pods to operate independently and durably.

To create server pods we can use a Deployment:

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a sevice can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

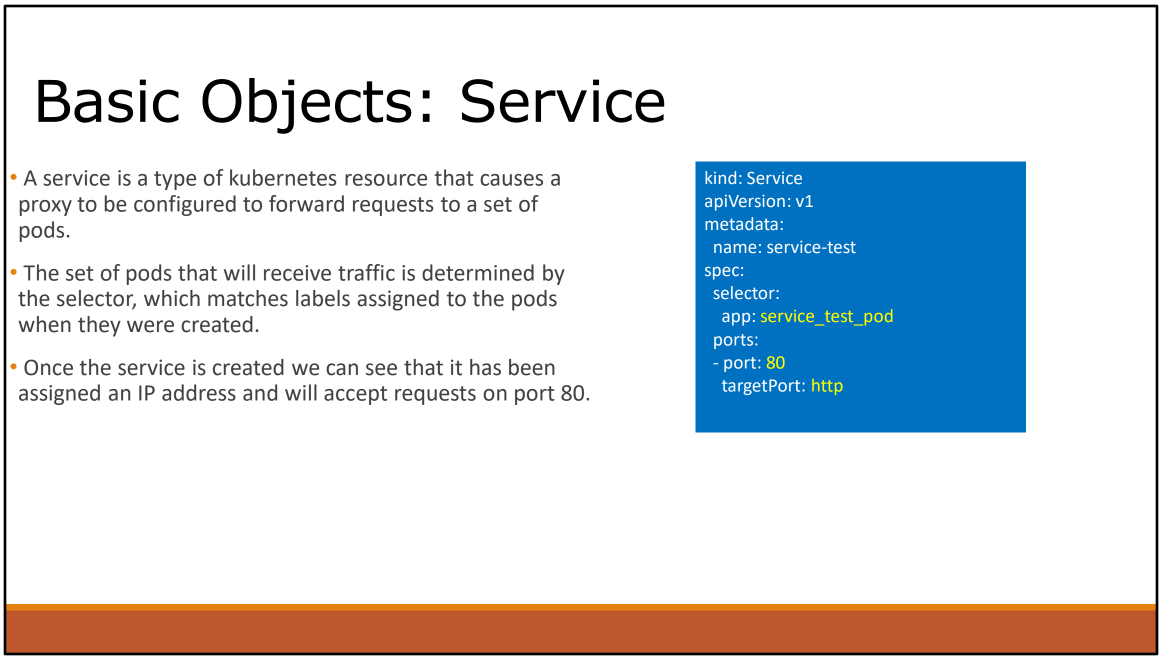


- This deployment creates two very simple http server pods that respond on port 8080 with the hostname of the pod they are running in.
- After creating this deployment using kubectl apply the pods are running in the cluster.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: service-test
spec:
  replicas: 2
  selector:
    matchLabels:
      app: service_test_pod
  template:
    metadata:
      labels:
        app: service_test_pod
    spec:
      containers:
      - name: simple-http
        image: python:2.7
        command: ["/bin/bash"]
        args: ["-c", "echo \"<p>Hello from $(hostname)</p>\" > index.html; python -m SimpleHTTPServer 8080"]
        ports:
        - name: http
          containerPort: 8080
```

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a sevice can stay alive and using the new pod replacing the died one along with the new IP address of the pod.



Basic Objects: Service

- A service is a type of kubernetes resource that causes a proxy to be configured to forward requests to a set of pods.
- The set of pods that will receive traffic is determined by the selector, which matches labels assigned to the pods when they were created.
- Once the service is created we can see that it has been assigned an IP address and will accept requests on port 80.

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  selector:
    app: service_test_pod
  ports:
  - port: 80
    targetPort: http
```

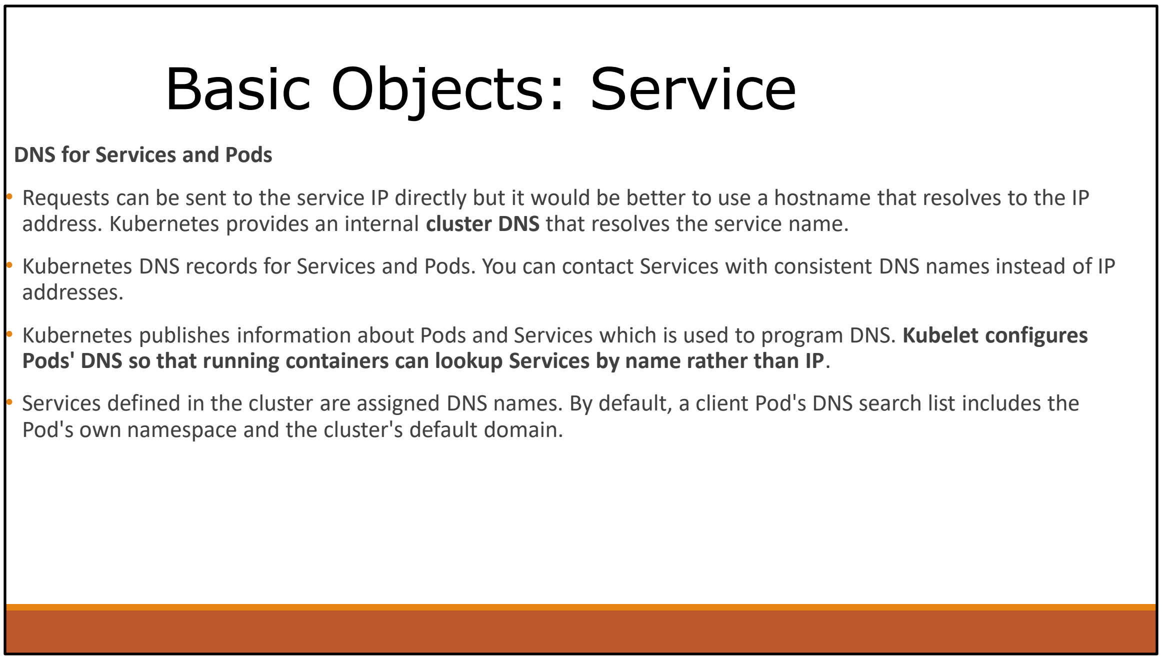
Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a sevice can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

In Kubernetes, when creating a service, you have to set the “port” and “targetPort” properties.
A “port” refers to the container port exposed by a pod or deployment, while a “targetPort” refers to the port on the host machine that traffic is directed to.

1. ClusterIP
•ClusterIP is the default and most common service type.
•Kubernetes will assign a cluster-internal IP address to ClusterIP service. This makes the service only reachable within the cluster.
•You cannot make requests to service (pods) from outside the cluster.

•You can optionally set cluster IP in the service definition file.
Use Cases
•Inter service communication within the cluster. For example, communication between the front-end and back-end components of your app.


Basic Objects: Service

DNS for Services and Pods

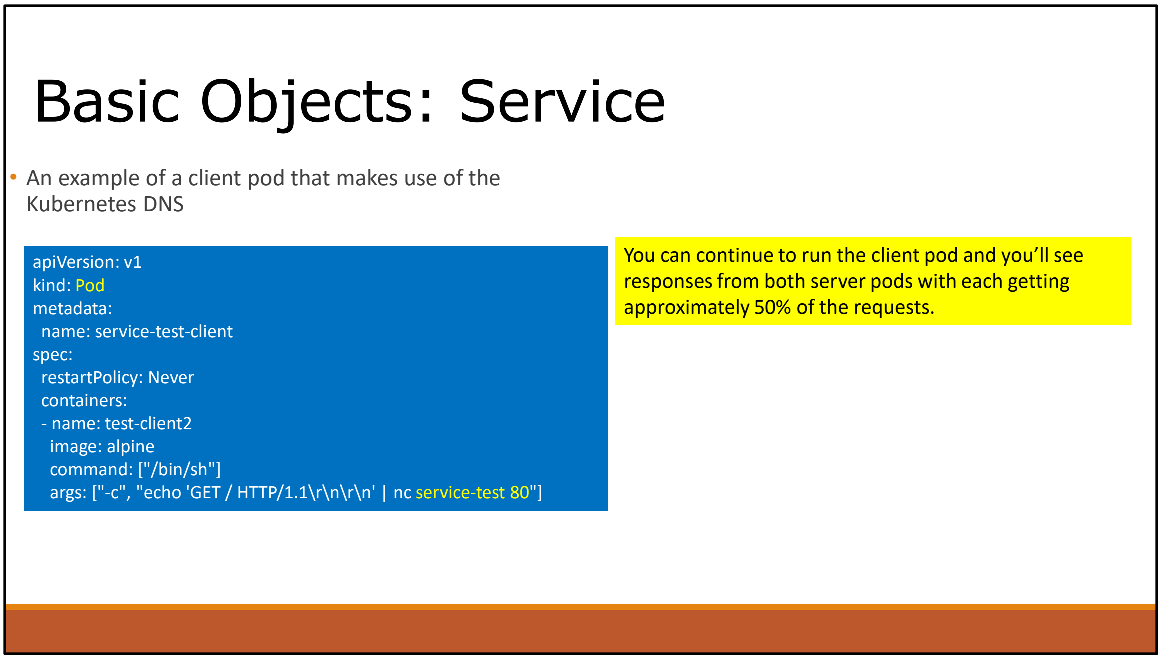
- Requests can be sent to the service IP directly but it would be better to use a hostname that resolves to the IP address. Kubernetes provides an internal **cluster DNS** that resolves the service name.
- Kubernetes DNS records for Services and Pods. You can contact Services with consistent DNS names instead of IP addresses.
- Kubernetes publishes information about Pods and Services which is used to program DNS. **Kubelet configures Pods' DNS so that running containers can lookup Services by name rather than IP**.
- Services defined in the cluster are assigned DNS names. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain.

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a sevice can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

In Kubernetes, you can set up a DNS system with two well-supported add-ons: CoreDNS and Kube-DNS. CoreDNS is a newer add-on that became a default DNS server as of Kubernetes v1.12. However, Kube-DNS may still be installed as a default DNS system by certain Kubernetes installer tools.
Both add-ons schedule a DNS pod or pods and a service with a static IP on the cluster and both are named kube-dns in the metadata.name field for interoperability. When the cluster is configured by the administrator or installation tools, the kubelet passes DNS functionality to each container with the--cluster-dns=<dns-service-ip> flag.
When configuring the kubelet , the administrator can also specify the name of a local domain using the flag --cluster-domain=<default-local-domain> .
Kubernetes DNS add-ons currently support forward lookups (A records), port lookups

(SRV records), reverse IP address lookups (PTR records), and some other options. In the following sections, we discuss the Kubernetes naming schema for pods and services within these types of records.



Basic Objects: Service

- An example of a client pod that makes use of the Kubernetes DNS

```
apiVersion: v1
kind: Pod
metadata:
  name: service-test-client
spec:
  restartPolicy: Never
  containers:
  - name: test-client2
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "echo 'GET / HTTP/1.1\r\n\r\n' | nc service-test 80"]
```

You can continue to run the client pod and you'll see responses from both server pods with each getting approximately 50% of the requests.

Kubernetes service discovery is **an abstraction that allows an application running on a set of Pods to be exposed as a network service**. This enables a set of Pods to run using a single DNS name, and allows Kubernetes load balancing across them all.

Lifecycle of pods and services are NOT dependent on each other. If a pod dies, a sevice can stay alive and using the new pod replacing the died one along with the new IP address of the pod.

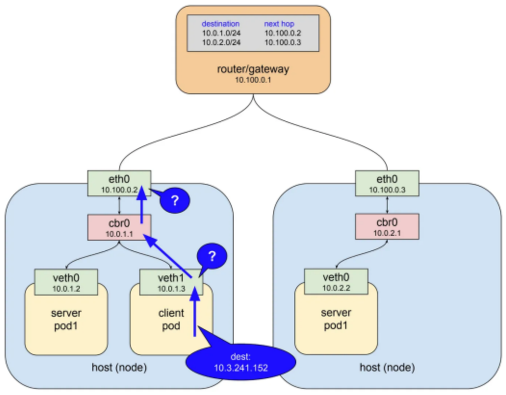
Basic Objects: Service

- The **IP** that the test service was assigned represents an address on a network that **is not the same as the one the pods are on.** The network specified by this address space is called the “service network.”
- Every service that is of type **“ClusterIP” will be assigned an IP address on this network**.
- There are other types of services, but ClusterIP is the default and it means “the service will be assigned an IP address reachable from any pod in the cluster.” You can see the type of a service by running the kubectl describe services command with the service name – e.g. kubectl describe services service-test
- Like the pod network the service network is **virtual**, but it differs from the pod network in some interesting ways. Consider the pod network address range 10.0.0.0/14 in the running example. If you go looking on the hosts that make up the nodes in your cluster, listing bridges and interfaces you’re going to see actual devices configured with addresses on this network. Those are the virtual ethernet interfaces for each pod and the bridges that connect them to each other and the outside world.
- You can examine the routing rules at the gateway that connects all the nodes and you won’t find any routes for this network. The service network **does not exist**, at least not as connected interfaces. However, it work. How did that happen?

Basic Objects: Service

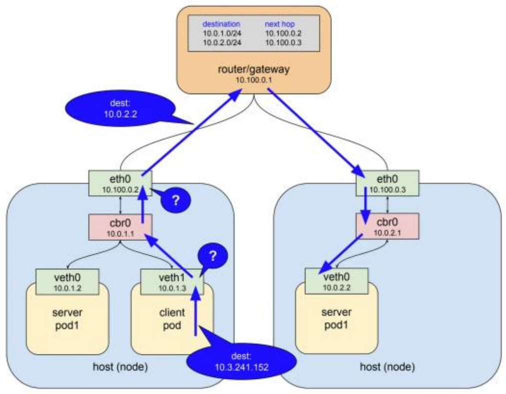
- The IP that the test service was assigned represents an address on a network that is not the same as the one the pods are on. The network specified by this address space is called the “service network.”
- Every service that is of type “ClusterIP” will be assigned an IP address on this network.
- There are other types of services, but ClusterIP is the default and it means “the service will be assigned an IP address reachable from any pod in the cluster.” You can see the type of a service by running the kubectl describe services command with the service name – e.g. kubectl describe services service-test
- Like the pod network the service network is **virtual**, but it differs from the pod network in some interesting ways. Consider the pod network address range 10.0.0.0/14 in the running example. If you go looking on the hosts that make up the nodes in your cluster, listing bridges and interfaces you’re going to see actual devices configured with addresses on this network. Those are the virtual ethernet interfaces for each pod and the bridges that connect them to each other and the outside world.
- You can examine the routing rules at the gateway that connects all the nodes and you won’t find any routes for this network. The service network **does not exist**, at least not as connected interfaces. However, it work. How did that happen?
- **Like everything in kubernetes a service is just a resource, a record in a central database, that describes how to configure some bit of software to do something.**

Basic Objects: Service



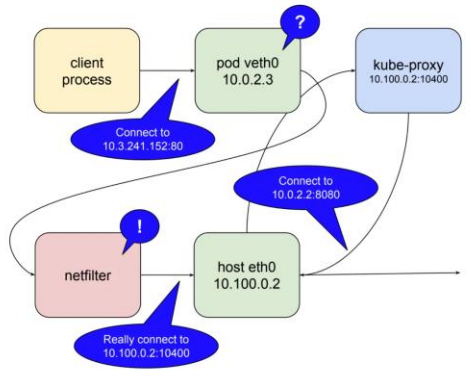
- Assume the client makes an http request to the service using the DNS name service-test. The cluster DNS system resolves that name to the service cluster IP 10.3.241.152, and the client pod ends up creating an http request that results in some packets being sent with that IP in the destination field.
- IP networks are typically configured with routes such that when an interface cannot deliver a packet to its destination because no device with that specified address exists locally it forwards the packet on to its upstream gateway.

Basic Objects: Service

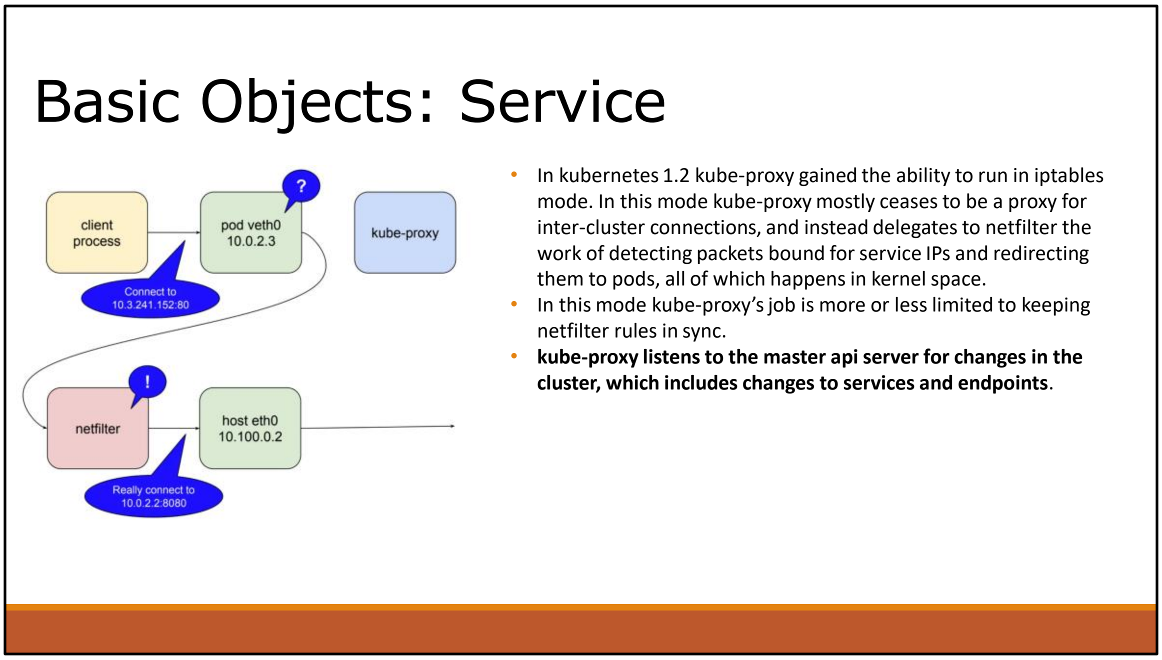


- The host/node ethernet interface in this example is on the network 10.100.0.0/24 and it doesn't know any devices with the address 10.3.241.152 either, so again what would normally happen is the packet would be forwarded out to this interface's gateway, the top level router shown in the drawing. Instead what actually happens is that the packet is snagged in flight and redirected to one of the live server pods.
- The reason of this behavior is on the intervention of the **kube-proxy**.
- Initially kube-proxy was implemented as just such a user-space proxy, but with a twist. A proxy needs an interface, both to listen on for client connections, and to use to connect to back-end servers. Kubernetes made use of a linux kernel called **netfilter**, and a user space interface to it called **iptables**.

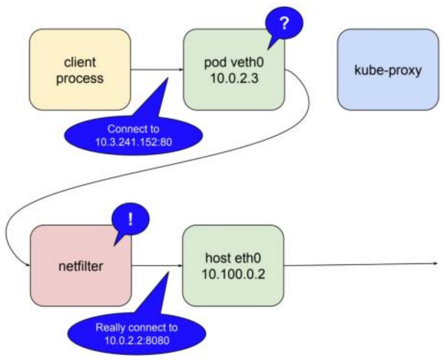
Basic Objects: Service


?
client process
pod veth0 10.0.2.3
kube-proxy 10.100.0.2:10400
Connect to 10.3.241.152:80
Connect to 10.0.2.2:8080
!
netfilter
host eth0 10.100.0.2
Really connect to 10.100.0.2:10400

- In this mode kube-proxy opens a port (10400 in the example figure) on the local host interface to listen for requests to the test-service, inserts netfilter rules to reroute packets destined for the service IP to its own port, and forwards those requests to a pod on port 8080. That is how a request to 10.3.241.152:80 magically becomes a request to 10.0.2.2:8080.
- Given the capabilities of netfilter all that is required to make this all work for any service is for kube-proxy to open a port and insert the correct netfilter rules for that service, which it does in response to notifications from the master api server of changes in the cluster.
- Unfortunately, user space proxying is expensive due to marshaling packets between user space and kernel space.



Basic Objects: Service



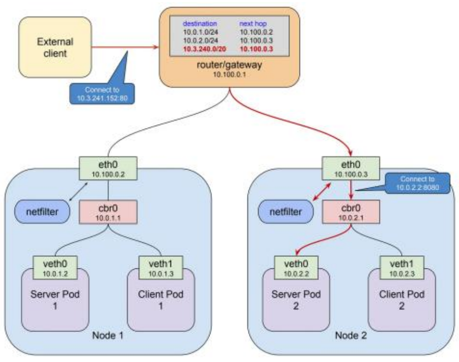
- In kubernetes 1.2 kube-proxy gained the ability to run in iptables mode. In this mode kube-proxy mostly ceases to be a proxy for inter-cluster connections, and instead delegates to netfilter the work of detecting packets bound for service IPs and redirecting them to pods, all of which happens in kernel space.
- In this mode kube-proxy's job is more or less limited to keeping netfilter rules in sync.
- **kube-proxy listens to the master api server for changes in the cluster, which includes changes to services and endpoints**.

Askube-proxy receives updates it uses iptables to keep the netfilter rules in sync. When a new service is created and its endpoints are populated kube-proxy gets the notification and creates the necessary rules. Similarly it removes rules when services are deleted. Health checks against endpoints are performed by the kubelet, another component that runs on every node, and when unhealthy endpoints are found kubelet notifies kube-proxy via the api server and netfilter rules are edited to remove this endpoint until it becomes healthy again.

Basic Objects: Service

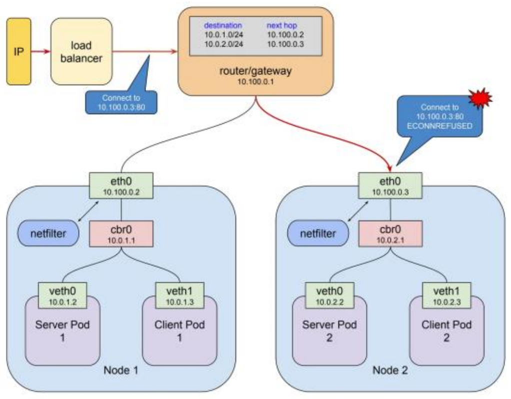
- “ClusterIP” only works for requests that originate inside the cluster, i.e. requests from one pod to another.
- Any mechanism we use to allow external clients to call into our pods has to make use of this same routing infrastructure.
- Those external clients have to end up calling the cluster IP and port, because that is the “front end” for all the machinery that makes it possible not to care where a pod is running at any given time.
- **The cluster IP of a service is only reachable <u>from a node’s ethernet interface</u>. Nothing outside the cluster knows what to do with addresses in that range. How can we forward traffic from a publicly visible IP endpoint to an IP that is only reachable <u>once the packet is already on a node</u>?**

Basic Objects: Service



- We could just give clients the cluster IP, maybe assign it a friendly domain name, and then add a route to get those packets to one of the nodes.
- Clients call the cluster IP, the packets follow a route down to a node, and get forwarded to a pod.
- This solution suffers from some serious problems. The first is simply that nodes are ephemeral. They are not *as* ephemeral as pods, but they can be migrated to a new vm, clusters can scale up and down, etc. Routers do not know healthy services from unhealthy. They expect the next hop in the route to be stable and available. If the node becomes unreachable the route will break and stay broken for a significant time in most cases.
- Even if the route were durable if all external traffic passing through a single node is not optimal.

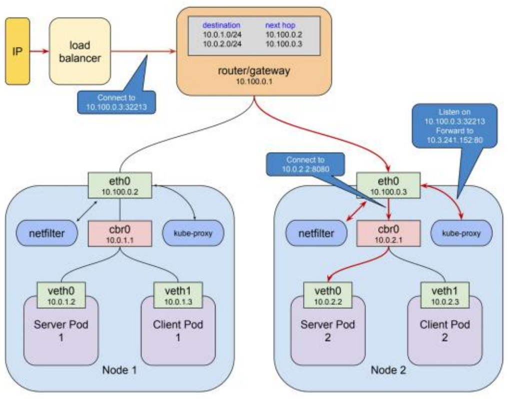
Basic Objects: Service



- To could use a load balancer for distributing client traffic to the nodes in a cluster we need a public IP the clients can connect to, and we need addresses on the nodes themselves to which the load balancer can forward the requests.
- For the reasons discussed we can’t easily create a stable static route between the gateway router and the nodes using the service network (cluster IP). The only other addresses available are on the network the nodes’ ethernet interfaces are connected to, 10.100.0.0/24 in the example.
- The gateway router already knows how to get packets to these interfaces, and connections sent from the load balancer to the router will get to the right place. But if a client wants to connect to our service on port 80 we can’t just send packets to that port on the nodes’ interfaces since there is no process listening on 10.100.0.3:80.

The reason why this fails is readily apparent. There is no process listening on 10.100.0.3:80 (or if there is it’s the wrong one), and the netfilter rules that we were hoping would intercept our request and direct it to a pod don’t match that destination address. They only match the cluster IP on the service network at 10.3.241.152:80. So those packets can’t be delivered when they arrive on that interface and the kernel responds with ECONNREFUSED.

Basic Objects: Nodeport Service



- The solution is to create something called a **NodePort.**
- A service of type NodePort is a ClusterIP service with an additional capability: it is reachable at the IP address of the node as well as at the assigned cluster IP on the services network.
- The way this is accomplished is pretty straightforward: when kubernetes **creates a NodePort service kube-proxy allocates a port in the range 30000–32767 and opens this port on the eth0 interface of every node** (thus the name "NodePort"). Connections to this port are forwarded to the service's cluster IP.
- If we create the service above and run kubectl get svc service-test we can see the NodePort that has been allocated for it.

2. NodePort
•NodePort service is an extension of ClusterIP service. A ClusterIP Service, to which the NodePort Service routes, is automatically created.
•It exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP.
•NodePort exposes the service on each Node's IP at a static port (the NodePort). Each node proxies that port into your Service. So, external traffic has access to fixed port on each Node. It means any request to your cluster on that port gets forwarded to the service.
•You can contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
•Node port must be in the range of 30000–32767. Manually allocating a port to the service is optional. If it is undefined, Kubernetes will automatically assign one.
•If you are going to choose node port explicitly, ensure that the port was not already used by another service.
Use Cases
•When you want to enable external connectivity to your service.
•Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to

expose one or more nodes' IPs directly.
•Prefer to place a load balancer above your nodes to avoid node failure.

The NodePort service exposes a single port on each node in the Kubernetes cluster. Remember, a cluster is just a group of available nodes, so that means when you create this service, you should be able to ping the IP address of the node. NodePorts can be used to make your application accessible on the internet.

https://www.tkng.io/services/nodeport/

NodePort builds on top of the ClusterIP Service and provides a way to expose a group of Pods to the outside world. At the API level, the only difference from the ClusterIP is the mandatory service type which has to be set to NodePort, the rest of the values can remain the same.
Whenever a new Kubernetes cluster gets built, one of the available configuration parameters is service-node-port-range which defines a range of ports to use for NodePort allocation and usually defaults to 30000-32767. One interesting thing about NodePort allocation is that it is not managed by a controller.

From the networking point of view, NodePort's implementation is very easy to understand:
•For each port in the NodePort Service, API server allocated a unique port from the service-node-port-range.
•This port is programmed in the dataplane of each Node by the kube-proxy (or its equivalent) – the most common implementations with IPTables, IPVS and eBPF are covered in the Lab section below.
•Any incoming packet matching one of the configured NodePorts will get destination NAT'ed to one of the healthy Endpoints and source NAT'ed (via masquerade/overload) to the address of the incoming interface.
•The reply packet coming from the Pod will get reverse NAT'ed using the connection tracking entry set up by the incoming packet.

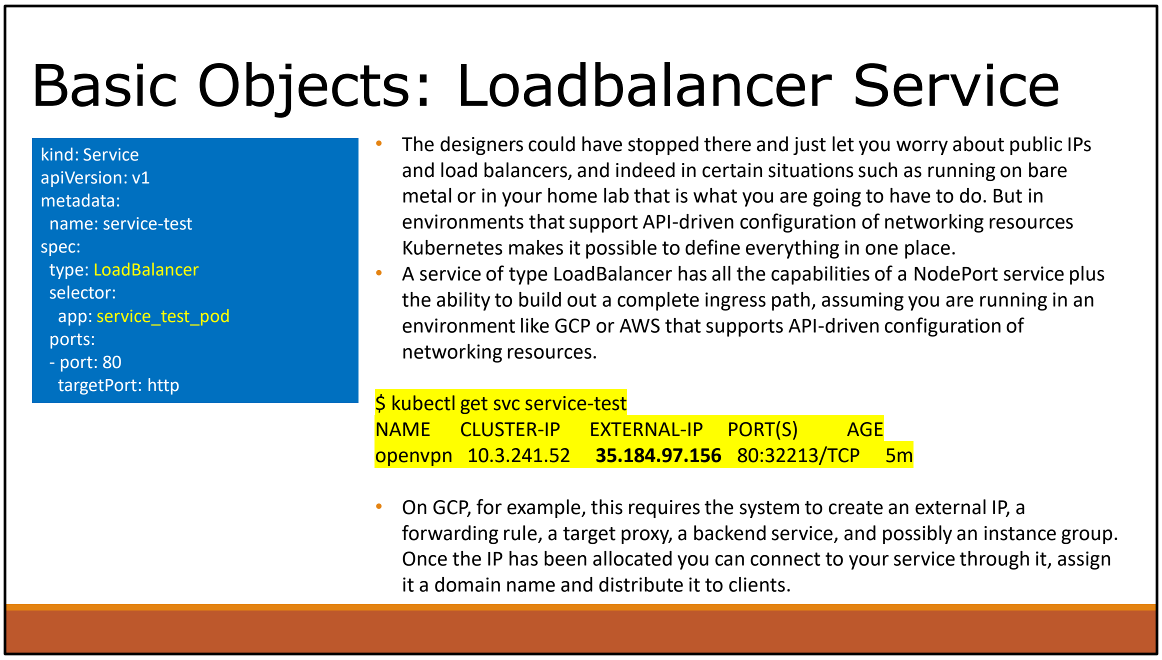
Basic Objects: Nodeport Service

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  type: NodePort
  selector:
    app: service_test_pod
  ports:
  - port: 80
    targetPort: http
```

- If we create the service and run kubectl get svc service-test we can see the NodePort that has been allocated for it.

```
$ kubectl get svc service-test
NAME         CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
service-test   10.3.241.152   <none>        80:32213/TCP     1m
```

- In this case our service was allocated the NodePort 32213. This means that we can now connect to the service on either node in the example cluster, at 10.100.0.2:32213 or 10.100.0.3:32213 and traffic will get forwarded to the service. With this piece in place we now have a complete pipeline for load balancing external client requests to all the nodes in the cluster.



Basic Objects: Loadbalancer Service

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  type: LoadBalancer
  selector:
    app: service_test_pod
  ports:
  - port: 80
    targetPort: http
```

- The designers could have stopped there and just let you worry about public IPs and load balancers, and indeed in certain situations such as running on bare metal or in your home lab that is what you are going to have to do. But in environments that support API-driven configuration of networking resources Kubernetes makes it possible to define everything in one place.
- A service of type LoadBalancer has all the capabilities of a NodePort service plus the ability to build out a complete ingress path, assuming you are running in an environment like GCP or AWS that supports API-driven configuration of networking resources.

```
$ kubectl get svc service-test
NAME      CLUSTER-IP     EXTERNAL-IP    PORT(S)        AGE
openvpn   10.3.241.52    35.184.97.156  80:32213/TCP   5m
```

- On GCP, for example, this requires the system to create an external IP, a forwarding rule, a target proxy, a backend service, and possibly an instance group. Once the IP has been allocated you can connect to your service through it, assign it a domain name and distribute it to clients.

GCP : Google Cloud Platform

3. LoadBalancer
Currently only implemented in public cloud. So if you're on Kubernetes in Azure or AWS, you will find a load balancer.
LoadBalancer service is an extension of NodePort service. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
It integrates NodePort with cloud-based load balancers.
It exposes the Service externally using a cloud provider’s load balancer.
Each cloud provider (AWS, Azure, GCP, etc) has its own native load balancer implementation. The cloud provider will create a load balancer, which then automatically routes requests to your Kubernetes Service.
Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.
The actual creation of the load balancer happens asynchronously.
Every time you want to expose a service to the outside world, you have to create a new LoadBalancer and get an IP address.
Use Cases
When you are using a cloud provider to host your Kubernetes cluster.

The canonical name record, or CNAME record, is a type of DNS record that maps an alias name to a real or canonical domain name. Typically, CNAME records are used to map a www or mail subdomain to the domain hosting its contents. For example, a CNAME record can map the web address www.example.com to the actual website of the domain example.com.

Service without selector: External Service

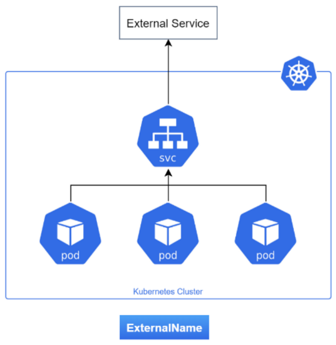
It is used for direct connections based on IP port combinations without an endpoint.

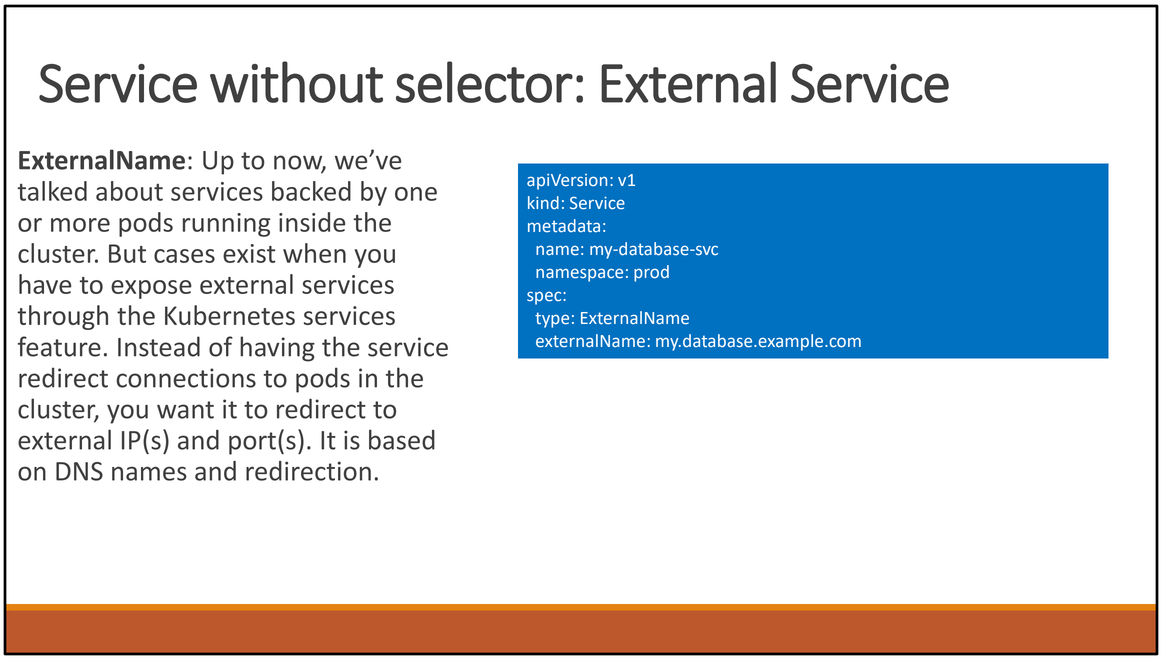
Services most commonly abstract access to Kubernetes Pods thanks to the selector, but when used with a corresponding set of EndpointSlices objects and without a selector, the Service can abstract other kinds of backends, including ones that run outside the cluster.

For example:

- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different Namespace or on another cluster.
- You are migrating a workload to Kubernetes. While evaluating the approach, you run only a portion of your backends in Kubernetes.

In any of these scenarios you can define a Service without specifying a selector to match Pods.





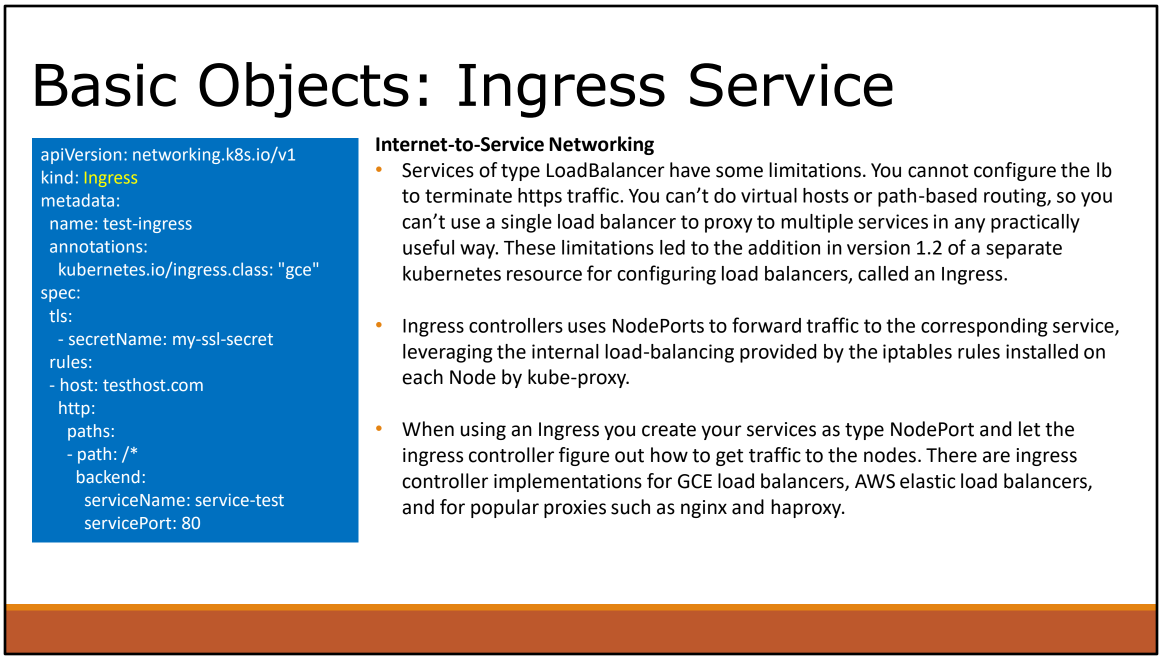
Service without selector: External Service

ExternalName: Up to now, we've talked about services backed by one or more pods running inside the cluster. But cases exist when you have to expose external services through the Kubernetes services feature. Instead of having the service redirect connections to pods in the cluster, you want it to redirect to external IP(s) and port(s). It is based on DNS names and redirection.

```
apiVersion: v1
kind: Service
metadata:
  name: my-database-svc
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

ExternalName: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

The ExternalName service maps a particular service to a DNS name. The ExternalName Service acts as a proxy, allowing a user to redirect requests to a service sitting outside (or inside) the cluster. Essentially it creates a CNAME record that connects the DNS name to some cluster-local name—that way your pods can leverage that service. This all may feel a little vague at first, so let's look at a real-life example.
Let's say you are migrating all of your applications to Kubernetes. However, that is a hard job, and it would be easier to do it piecemeal. So you retain an external database that your new k8s containers can retrieve data from. However, that database resides *outside* the cluster—so your pods have no idea what it is. We get around this problem by using ExternalName Service.



Basic Objects: Ingress Service

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: "gce"
spec:
  tls:
    - secretName: my-ssl-secret
  rules:
  - host: testhost.com
    http:
      paths:
      - path: /*
        backend:
          serviceName: service-test
          servicePort: 80
```

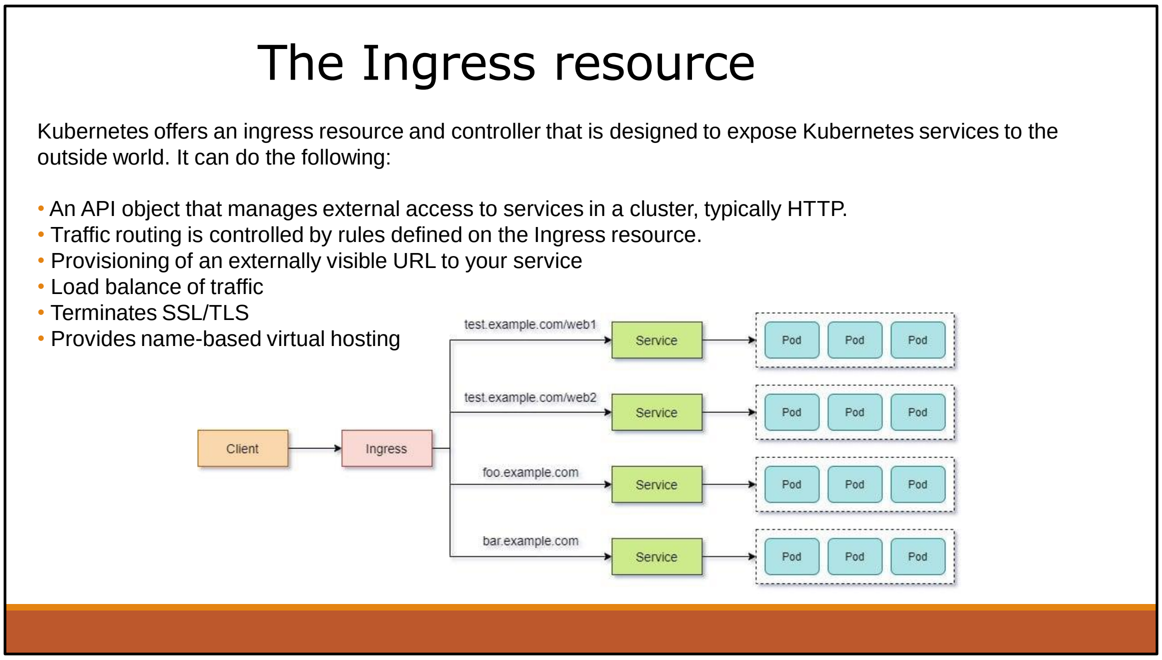
Internet-to-Service Networking

- Services of type LoadBalancer have some limitations. You cannot configure the lb to terminate https traffic. You can't do virtual hosts or path-based routing, so you can't use a single load balancer to proxy to multiple services in any practically useful way. These limitations led to the addition in version 1.2 of a separate kubernetes resource for configuring load balancers, called an Ingress.

- Ingress controllers uses NodePorts to forward traffic to the corresponding service, leveraging the internal load-balancing provided by the iptables rules installed on each Node by kube-proxy.

- When using an Ingress you create your services as type NodePort and let the ingress controller figure out how to get traffic to the nodes. There are ingress controller implementations for GCE load balancers, AWS elastic load balancers, and for popular proxies such as nginx and haproxy.

Google Compute Engine (GCE) is **an infrastructure as a service (IaaS) offering that allows clients to run workloads on Google's physical hardware**. Google Compute Engine provides a scalable number of virtual machines (VMs) to serve as large compute clusters for that purpose.



The Ingress resource

Kubernetes offers an ingress resource and controller that is designed to expose Kubernetes services to the outside world. It can do the following:

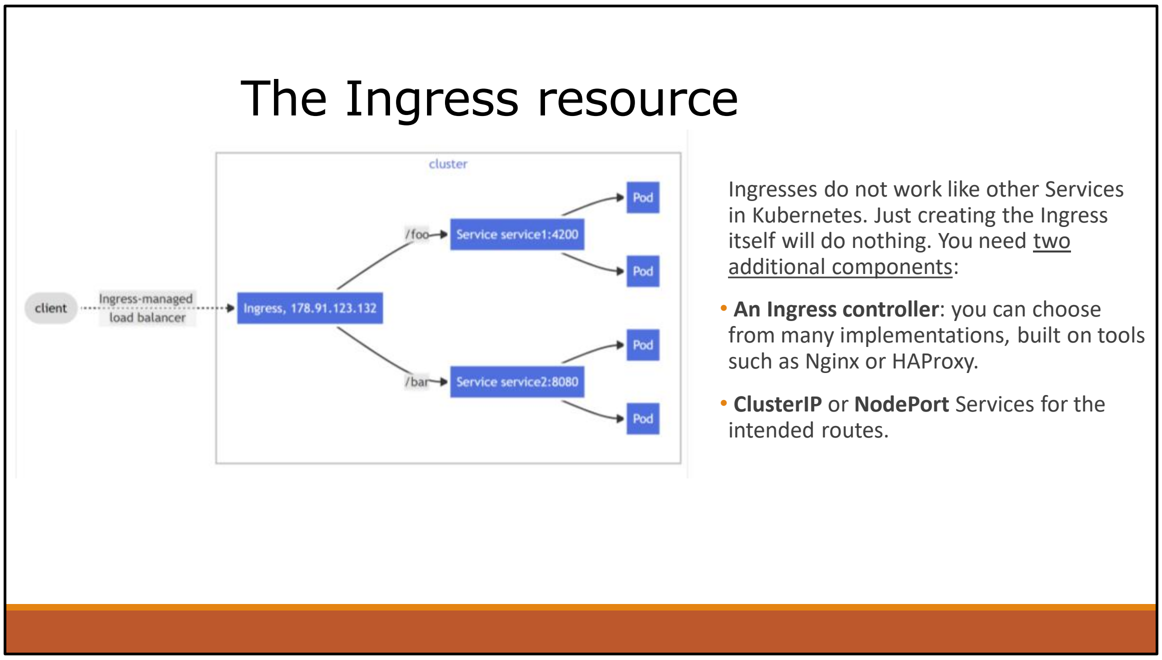
- An API object that manages external access to services in a cluster, typically HTTP.
- Traffic routing is controlled by rules defined on the Ingress resource.
- Provisioning of an externally visible URL to your service
- Load balance of traffic
- Terminates SSL/TLS
- Provides name-based virtual hosting

As a customer I just want the hostname/IP address on which I can access the nginx web server and I don't want to remember all these additional Port no. *So we use Kubernetes ingress.*

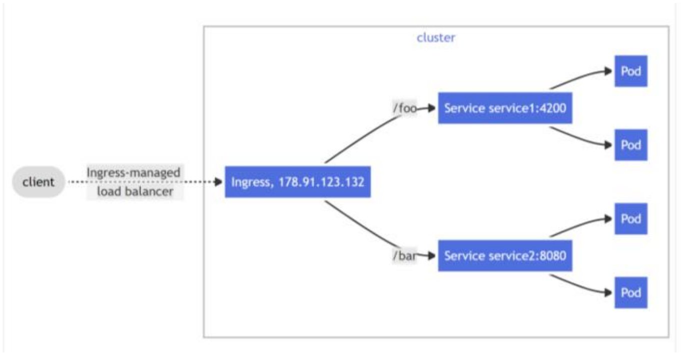
The life of a packet flowing through an Ingress is very similar to that of a LoadBalancer. The key differences are that an Ingress is aware of the URL's path (allowing and can route traffic to services based on their path), and that the initial connection between the Ingress and the Node is through the port exposed on the Node for each service.

With the name based virtual hosting you can host several **domains/websites** on a single machine with a single **IP**. All domains on that server will be sharing a single IP. It's easier to configure than IP based virtual hosting, you only need to configure **DNS** of the domain to map it with its correct IP address and then configure the Ingress to recognize it with the domain names.

A **TLS termination proxy** (or **SSL termination proxy,**[1] or **SSL offloading**[2]) is a proxy server that acts as an intermediary point between client and server applications, and is used to terminate and/or establish TLS (or DTLS) tunnels by decrypting and/or encrypting communications. This is different to **TLS pass-through proxies** that forward encrypted (D)TLS traffic between clients and servers without terminating the tunnel.



The Ingress resource



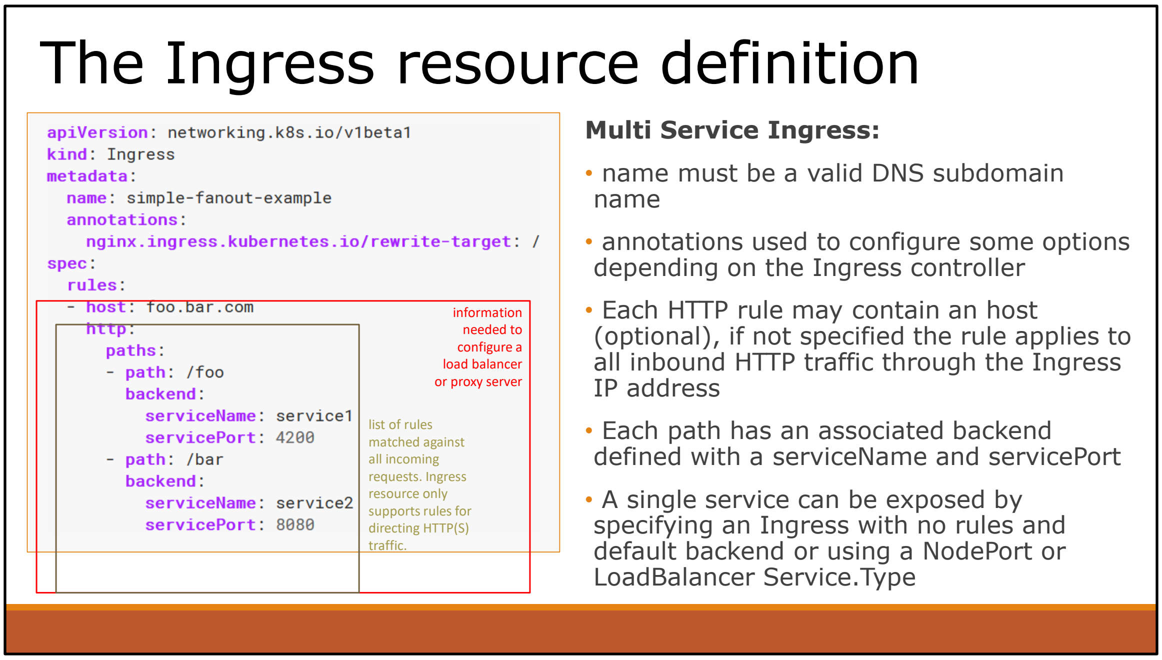
Ingresses do not work like other Services in Kubernetes. Just creating the Ingress itself will do nothing. You need two additional components:

- **An Ingress controller**: you can choose from many implementations, built on tools such as Nginx or HAProxy.
- **ClusterIP** or **NodePort** Services for the intended routes.

By using an Ingress it possible to contact a service by using a URL instead of an IP address.

If you set the Service's type field to NodePort, the Kubernetes master will allocate a port from a range you specify, and each Node will proxy that port (the same port number on every Node) into your Service. That is, any traffic directed to the Node's port will be forwarded on to the service using iptables rules. This Service to Pod routing follows the same internal cluster load-balancing pattern we've already discussed when routing traffic from Services to Pods.
To expose a Node's port to the Internet you use an Ingress object. An Ingress is a higher-level HTTP load balancer that maps HTTP requests to Kubernetes Services. The Ingress method will be different depending on how it is implemented by the Kubernetes cloud provider controller. HTTP load balancers, like Layer 4 network load balancers, only understand Node IPs (not Pod IPs) so traffic routing similarly leverages the internal load-balancing provided by the iptables rules installed on each Node by kube-proxy.

Generally, clusters will not come configured with any pre-existing Ingress controllers. **You must have an Ingress controller to satisfy an Ingress**. Only creating an Ingress resource has no effect. You'll need to select and deploy one to your cluster. ingress-nginx is likely the most popular choice, but there are several others, you can get the complete list on Kubernetes official page



The Ingress resource definition

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

information needed to configure a load balancer or proxy server
list of rules matched against all incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic.

Multi Service Ingress:

- name must be a valid DNS subdomain name
- annotations used to configure some options depending on the Ingress controller
- Each HTTP rule may contain an host (optional), if not specified the rule applies to all inbound HTTP traffic through the Ingress IP address
- Each path has an associated backend defined with a serviceName and servicePort
- A single service can be exposed by specifying an Ingress with no rules and default backend or using a NodePort or LoadBalancer Service.Type

An Ingress needs apiVersion, kind, metadata and spec fields. The name of an Ingress object must be a valid DNS subdomain name. Ingress frequently uses annotations to configure some options depending on the Ingress controller, an example of which is the rewrite-target annotation. Different Ingress controllers support different annotations. Review the documentation for your choice of Ingress controller to learn which annotations are supported.

The Ingress spec has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic.

If the ingressClassName is omitted, a default Ingress class should be defined.

Ingress rules

Each HTTP rule contains the following information:

•An optional host. In this example, no host is specified, so the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, foo.bar.com), the rules apply to that host.

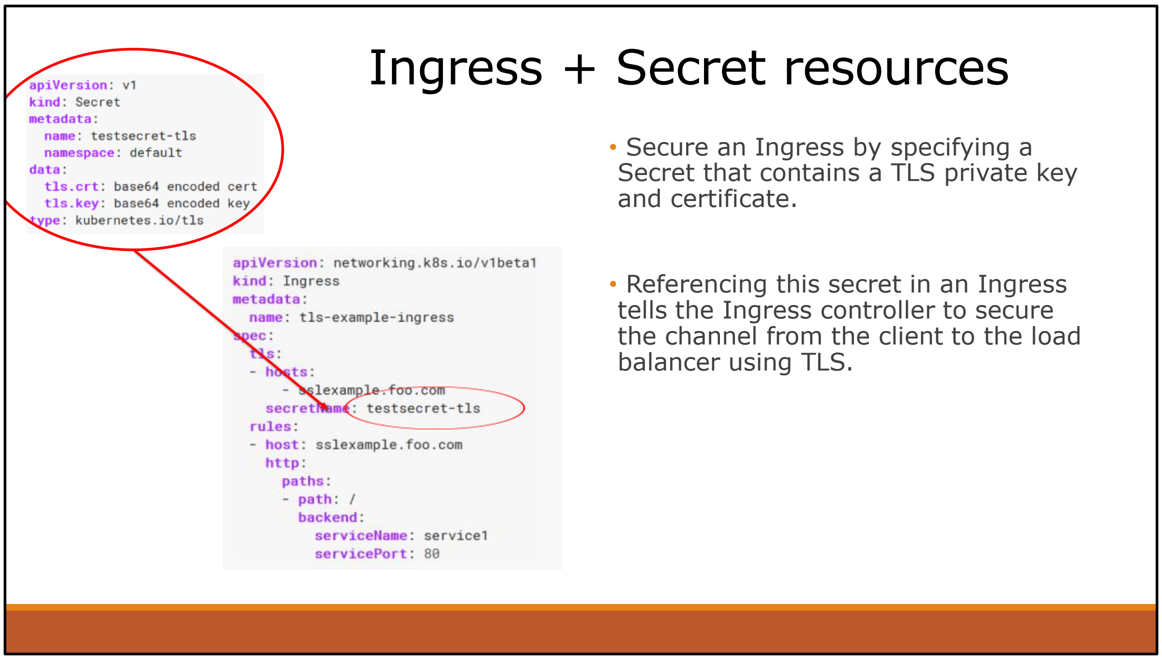
•A list of paths (for example, /testpath), each of which has an associated backend defined with a service.name and a service.port.name or service.port.number. Both the host and path must match the content of an incoming request before the load balancer directs traffic to the referenced Service.

•A backend is a combination of Service and port names as described in the Service doc or a custom resource backend by way of a CRD. HTTP (and HTTPS) requests to the Ingress that matches the host and path of the rule are sent to the listed backend.
A defaultBackend is often configured in an Ingress controller to service any requests that do not match a path in the spec.

Each path in an Ingress is required to have a corresponding path type. Paths that do not include an explicit pathType will fail validation. There are three supported path types:
•ImplementationSpecific: With this path type, matching is up to the IngressClass. Implementations can treat this as a separate pathType or treat it identically to Prefix or Exact path types.
•Exact: Matches the URL path exactly and with case sensitivity.
•Prefix: Matches based on a URL path prefix split by /. Matching is case sensitive and done on a path element by element basis. A path element refers to the list of labels in the path split by the / separator. A request is a match for path *p* if every *p* is an element-wise prefix of *p* of the request path.

Hosts can be precise matches (for example “foo.bar.com”) or a wildcard (for example “*.foo.com”). Precise matches require that the HTTP host header matches the host field. Wildcard matches require the HTTP host header is equal to the suffix of the wildcard rule.

DNS Subdomain Names
Most resource types require a name that can be used as a DNS subdomain name as defined in RFC 1123. This means the name must:
•contain no more than 253 characters
•contain only lowercase alphanumeric characters, '-' or '.'
•start with an alphanumeric character
•end with an alphanumeric character
RFC 1123 Label Names



Ingress + Secret resources

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
  - hosts:
      - sslexample.foo.com
    secretName: testsecret-tls
  rules:
  - host: sslexample.foo.com
    http:
      paths:
      - path: /
        backend:
          serviceName: service1
          servicePort: 80
```

- Secure an Ingress by specifying a Secret that contains a TLS private key and certificate.
- Referencing this secret in an Ingress tells the Ingress controller to secure the channel from the client to the load balancer using TLS.

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code.

Example: To create base64 encoded certificate file:

apiVersion: v1
data:
  tls.crt:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURKakNDQWc2Z0F3SUJBZ0lKQUw2Y
3R2bk9zMzlUTUEwR0NTcUdTSWIzRFFFQkJRVUFNQll4RkRBU0JnTlYKQkFNVEMyWnZi
eTVpWVhJdVkyOXRNQjRYRFRFNE1USXhOREUxTWpJeU1Gb1hEVEU1TVRJeE5ERTFNa
kl5TUZvdwpGakVVTUJJR0ExVUVBeE1MWm05dkxtSmhjaTVqYjIwd2dnRWlNQTBHQ1N
xR1NJYjNEUUVCQVFVQUE0SUJEd0F3Cm dnRUtBb0lCQVFDbWVsQTNqVy9NZ2REejJNa
zMwbXZ4K2VOSHJkQlIwMEJ4ZUR1VjBjYWVFUGNFa2RmSnk5V28KaTFpSXV1V04vZGV
6UEhyTWMxenBPNGtzbWU5NThRZVFCWjNmVThWeGpRYktmb1JzNnhQQUlNKZVVSck
VCcWE4SQpUSXpEVVdaUTAwQ2xsa1dOejE4dDYvVjJycWxJd1VvaTVZWHloOVJsaWR4
MjZRaXJBcFFFaXZDY2QzdUExc3AwCkUxRXdIVGxVdzFqSE9Eb3BLZGxaRndmcWhFSHN

```
mYjZvLzJFb1A1MXMwY2JuTld6MHNsUjhhejdzOExVYnhBWnkKQkNQdDY1Z2VhT3hYW
WUxaWhLYzN4SE4wYSsxMXpBYUdDMnpTemdOcEVWeFFJQ3lZdVZld3dNb0FrcHNkdG
EybwpnMnFTaDZQZzRHeFFabzRwejIwN0c2SkFUaFIyNENNiTEFnTUJBQUdqZHpCMU1C
MEdBMVVkRGdRV0JCU3NBcUZoCkpPS0xZaXNHTkNVRGU4N1VWRkp0UERCR0JnTlZI
U01FUHpBOWdCU3NBcUZoSk9LTFlpc0dOQ1VEZTg3VVZGSnQKUEtFYXBCZ3dGakVVT
UJJR0ExVUVBeE1MWm05dkxtSmhjaTVqYjIyQ0NRQytuTGI1enJOL1V6QU1CZ05WSFJN
RQpCVEFEQVFIL01BMEdDU3FHU0liM0RRRUJCUVVBQTRJQkFRQU1wcDRLSEtPM2k1N
zR3dzZ3eU1pTExHanpKYXI4Cm8xbHBBa3BJR3FMOHVnQWg5d2ZNQWhsYnhJcWZJRH
lqNWQ3QlZIQlc1UHZweHpKV3pWbmhPOXMrdzdWRTlNVHVKWlJHSXVRMjdEeExxueS
9DVjVQdmJUSTBrcjcwYU9FcGlvTWYyUVUvaTBiN1B2ajJoeEJEMVZTVkd0bHFTSVpqUA
o0VXZQYk1yTWZUWmJka1pIbG1SUjJmbW4zK3NTVndrZTRhWXlENVVHNnpBVitjd3BB
bkZWS25VR0d3TkpVMjA4CmQrd3J2UUZ5bi9kcVBKTEdlNTkvODY4WjFCcFIxRmJYMit
UVW4yWTExZ0dkL0J4VmlzeGJ0b29GQkhlVDFLbnlKTTZCVUhEeFNvWVF0VnJWSDRJM
Wh5UGRkdmhPczgwQkQ2K01Dd203OXE2UExaclVKOURGbFl2VTAKLS0tLS1FTkQgQ0V
SVElGSUNBVEUtLS0tLQo=
  tls.key:
LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFb3dJQkFBS0NBUUVBcG5wU
U40MXZ6SUhRODlqSk45SnI4Zm5qUjYzUVVkTkFjWGc3bGRIR25oRDNCSkhYCnljdlZxSX
RZaUxybGpmM1hzeng2ekhOYzZUdUpMSm52ZWZFSGtBV2QzMVBGY1kwR3luNkViT3
NUMFVpWGxFYXgKQWFtdkNFeU13MUZtVU5OQXBaWkZqYzlmTGV2MWRxNnBTTUZL
SXVXRjhvZlVaWW5jZHVrSXF3S1VCSXJ3bkhkNwpnTmJLZEJOUk1CMDVWTU5ZeHpnNk
tTblpXUmNINm9SQjdIMitxUDloS0QrZGJOSEc1elZzOUxKVVZHcys3UEMxCkc4UUdjZ1
FqN2V1WUhtanNWMkh0WW9Tbk44UnpkR3Z0ZGN3R2hndHMwczREYVJGY1VDQXNt
TGxYc01ES0FKS2lKSGJXdHFJTnFrb2VqNE9Cc1VHYU9LYzl0T3h1aVFFNFVkdUFteXdJREF
RQUJBb0lCQUMvSitzOEhwZWxCOXJhWgpLNkgvb0ljVTRiNkkwYjA3ZEV0ZVpWUnJwS1
ZwWDArTGdqTm1kUTN0K2xzOXMzbmdQWlF4TDFzVFhyK0JISzZWCi9kMjhQQ0pheW1
mNmh6cENib21nYWVsT1RpRU13cDZJOEhUMnZjMFhGRzFaSjVMYUlidW0rSTV0MGZl
L3ZYWDEKUzVrY0Mya2JGQ2w3L21lcmZJTVNBQy8vREhpRTUyV1QydEIrQk01U2FMV3
p4cDhFa3NwNkxWN3ZwYmR4dGtrTwpkZ1A4QjkwWlByck5SdUN5ekRwRUkvMnhBY2
4yVzNidlBqRGpoTjBXdlhTbTErVk9DcXNqOEkrRkxoUzZJemVuCm1MUkFZNnpWVGpZV
05TU2J3dTRkbnNmNElIOEdiQkZJajcrdlN5YVNVTEZiVGJzY3ZzQ3I1MUszbWt2bEVMVjg
KaWsvMlJoa0NnWUVBMFpmV2xUTjR2alh2T0FjU1RUU3MwMFhIRWh6QXFjOFpUTE
w2S1d4YkxQVFJNaXBEYklEbQp6b3BiMGNTemxlTCtNMVJCY3dqMk5HcUNodXcyczBaN
TQyQVhSZXdteG1EcWJaWkFQY0UzbERQNW5wNGRpTFRCClZaMFY4UExSYjMrd2tUdE
83VThJZlY1alNNdmRDTWtneI4dU1yQ1VMYnhxMXlVUGtLdGpJdThDZ1lFQXkxYWMK
WjEyZC9HWWFpQjJDWpuN0NXZE5YdGhFS2dOYUFob21nNlFMZmlKakVLajk3SExKalF
abFZ0b3kra1RrdTJjZAp0Wm1zUi9IU042YmZLbEpxckpUWWkzY2E1TGY4a3NxR0Z5Y0x
1MXo3cmN6K1lUaEVWSFIyOVkrVHVWYXRDTnkzCklCOGNUQW1ORWlVMlVHR2VKeU
lME44Z1VZRXRCYzFaMEg2QWllVUNnWUFETDIrUGJPelUxelQvZ1B3Q09GNjQKQjBOel
B3U2VrQXN1WXpueUR6ZURnMlQ2Z2pIc0lEbGh3akNMQzVZL0hPZ0lGNnUyOTlmbUR
BaFh6SmM0T2tYMwo4cW5uNGlMa3VPeFhKZ1ZyNnRmUlpNalNaRXpHbXhpbEdISVE2
MS9MZGdGVTg3WExYWHdmaTZPdW80cUVhNm9YCjhCRmZxOWRVcXB4bEVLY2Y1N3
```

```
JsK1FLQmdGbjVSaFc2NS9oU0diVlhFWVZQU0pSOW9FK3lkRjcrd3FvaGRoOVQKekQ0U
TZ6THBCQXJITkFYeDZZK0gxM3pFVlUzVEwrTTJUM1E2UGFHZ2Rpa2M5TlRPdkE3aU1n
VTRvRXMzMENPWQpoR2x3bUhEc1B6YzNsWXlsU0NvYVVPeDJ2UFFwN2VJSndoU25P
VVBwTVdKWi80Z2pZZTFjZmNseTFrQTJBR0x3ClJ1STlBb0dCQU14aGFJSUdwTGdmcHk0
K24rai9BSWhJUUhLZFRCNVBqaGx0WWhqZittK011UURwK21OeTVMbzEKT0FRc0Q0en
Z1b3VxeHlmQlFQZlllYThvcm4vTDE3WlJyc3lSNHlhS1M3cDVQYmJKQlNlcTc5Z0g5ZUNI
QkxMbQo0aThCUFh0K0NmWktMQzg3NTNHSHVpOG91V25scUZ0NGxMQUlWaGJZQ
mtUbURZSWo4Q0NaCi0tLS0tRU5EIFJTQSBQUklWQVRFIEtFWS0tLS0tCg==
kind: Secret
metadata:
  name: test-tls
  namespace: default
type: kubernetes.io/tls
```

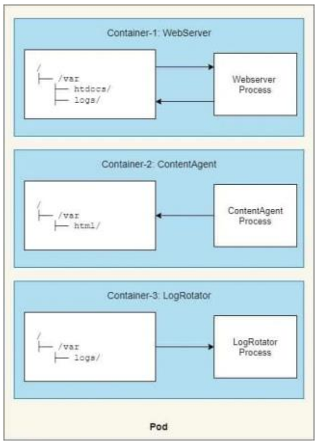
Kubernetes Volumes

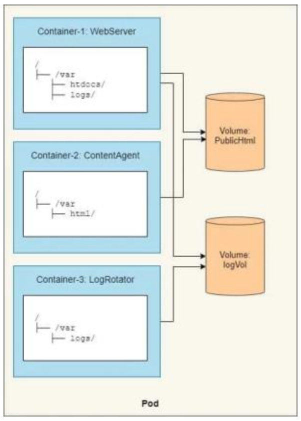
- Assume that a container within a Pod gets restarted (either because the process died or because the liveness probe signaled to Kubernetes that the container wasn't healthy anymore). You'll realize that the new container **will not see anything** that was written to the filesystem by the previous container, even though the newly started container runs in the same pod.

- In certain scenarios you want the new container to continue where the last one finished, such as when restarting a process on a physical machine. You may not need (or want) the whole filesystem to be persisted, but you do want to **preserve certain directories** that hold actual data.

- Kubernetes provides this by defining **storage volumes**. They are not top-level resources like pods, but are instead defined as a part of a pod and share the same lifecycle as the pod. This means a volume is created when the pod is started and is destroyed when the pod is deleted. Because of this, a volume's contents will persist across container restarts. After a container is restarted, the new container can see all the files that were written to the volume by the previous container. Also, if a pod contains multiple containers, the volume can be used by all of them at once.

Kubernetes Volumes





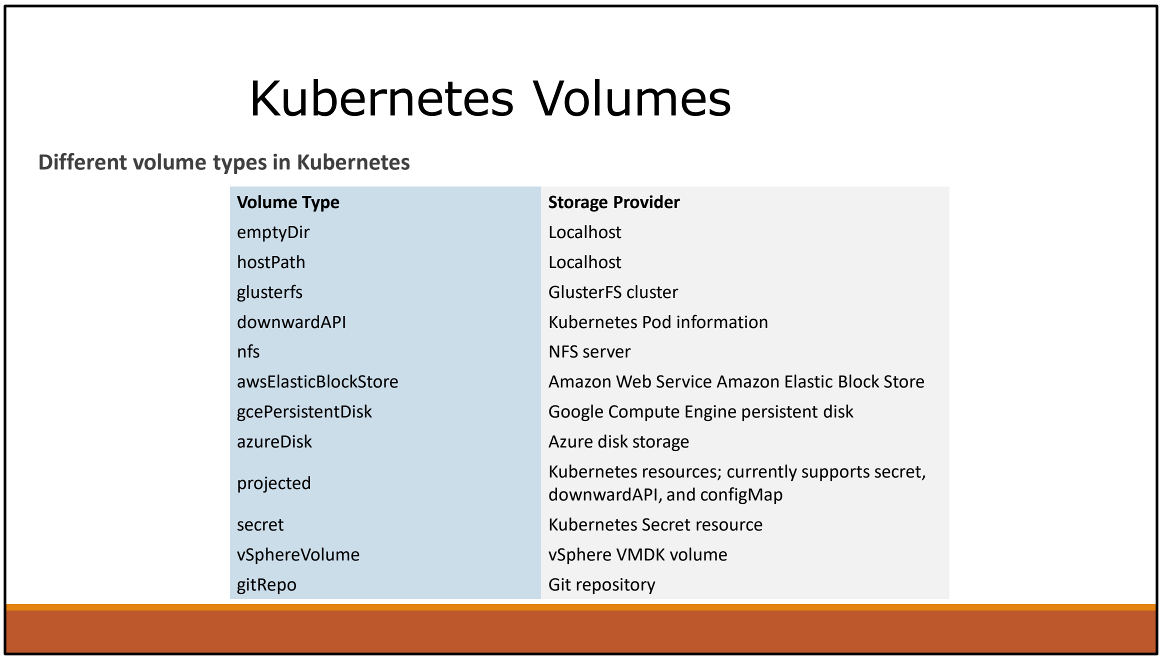
- Volumes are not a standalone Kubernetes object and cannot be created or deleted on their own.
- A volume is available to all containers in the pod, but it must be mounted in each container that needs to access it.
- In each container, you can mount the volume in any location of its filesystem

Let us take this example to get a more clear understanding:
One container runs a web server that serves HTML pages from the /var/htdocs directory and stores the access log to /var/logs. The second container runs an agent that creates HTML files and stores them in /var/html. The third container processes the logs it finds in the /var/logs directory (rotates them, compresses them, analyzes them, or whatever).

These 3 containers within the Pod are working just fine but they all are performing read and write operation to their own file system, even though all of them are using /var directory. So it doesn't make sense to have such architecture, we need our logRotator container to process the logs from other containers and similarly contentAgent will write new content inside /var/html.

But by mounting the same volume into two containers, they can operate on the same files. In your case, you're mounting two volumes in three containers. By doing this, your three containers can work together and do something useful.

First, the pod has a volume called publicHtml. This volume is mounted in the WebServer container at /var/htdocs, because that's the directory the web server serves files from. The same volume is also mounted in the ContentAgent container, but at /var/html, because that’s where the agent writes the files to. By mounting this single volume like that, the web server will now serve the content generated by the content agent. Similarly, the pod also has a volume called logVol for storing logs. This volume is mounted at /var/logs in both the WebServer and the LogRotator containers. Note that it isn’t mounted in the ContentAgent container. The container cannot access its files, even though the container and the volume are part of the same pod.



Kubernetes Volumes

Different volume types in Kubernetes

Volume Type	**Storage Provider**
emptyDir	Localhost
hostPath	Localhost
glusterfs	GlusterFS cluster
downwardAPI	Kubernetes Pod information
nfs	NFS server
awsElasticBlockStore	Amazon Web Service Amazon Elastic Block Store
gcePersistentDisk	Google Compute Engine persistent disk
azureDisk	Azure disk storage
projected	Kubernetes resources; currently supports secret, downwardAPI, and configMap
secret	Kubernetes Secret resource
vSphereVolume	vSphere VMDK volume
gitRepo	Git repository

Now we know that Kubernetes introduces volume, **which lives with a Pod across a container life cycle**. It supports various types of volumes, including popular network disk solutions and storage services in different public clouds.

Storage providers are required when you start to use volume in Kubernetes, except for emptyDir, which will be erased when the Pod is removed. For other storage providers, folders, servers, or clusters have to be built before using them in the Pod definition.

Volumes are defined in the volumes section of the pod definition with unique names. Each type of volume has a different configuration to be set. Once you define the volumes, you can mount them in the volumeMounts section in the container specs.volumeMounts.name and volumeMounts.mountPath are required, which indicate the name of the volumes you defined and the mount path inside the container, respectively.

Kubernetes Volumes

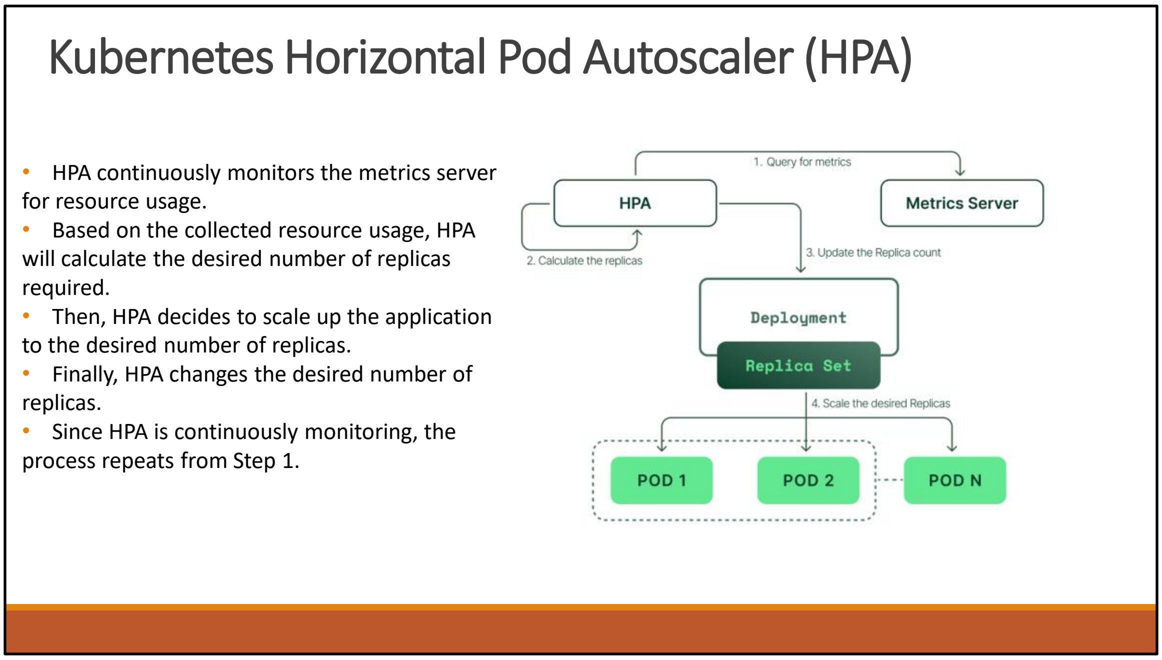
```
[root@controller ~]# cat shared-volume-emptyDir.yml
apiVersion: v1
kind: Pod
metadata:
  name: shared-volume-emptyDir
spec:
  containers:
  - name: alpine1
    image: alpine
    command: ["/bin/sleep", "999999"]
    volumeMounts:
    - mountPath: /alpine1
      name: data
  - name: alpine2
    image: alpine
    command: ["/bin/sleep", "999999"]
    volumeMounts:
    - mountPath: /alpine2
      name: data
  volumes:
  - name: data
    emptyDir: {}
```

- By this configuration file, we can a Pod running alpine with commands to sleep for 999999 seconds. As you can see, one volume is defined in the volumes section with name data, and the volumes will be mounted under the /alpine1 path in the alpine1 container and /alpine2 in the alpine2 container respectively:

Kubernetes Horizontal Pod Autoscaler (HPA)

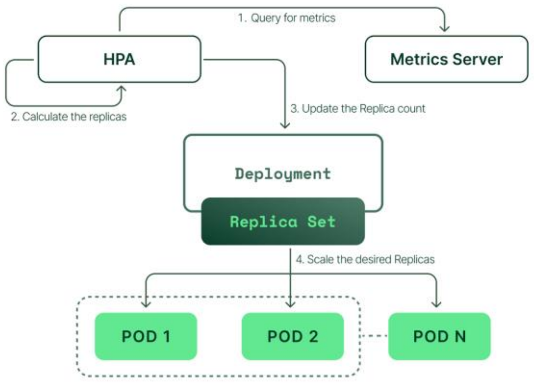
- An HPA object watches the resource consumption of pods that are managed by a controller (Deployment, ReplicaSet, or StatefulSet) at a given interval and controls the replicas by **comparing the desired target of certain metrics with their real usage**.

- For instance, suppose that we have a Deployment controller with two pods initially, and they are currently using 1,000 m of CPU on average while we want the CPU percentage to be 200 m per pod. The associated HPA would calculate how many pods are needed for the desired target with 2*(1000 m/200 m) = 10, so it will adjust the replicas of the controller to 10 pods accordingly. Kubernetes will take care of the rest to schedule the eight new pods.



Kubernetes Horizontal Pod Autoscaler (HPA)

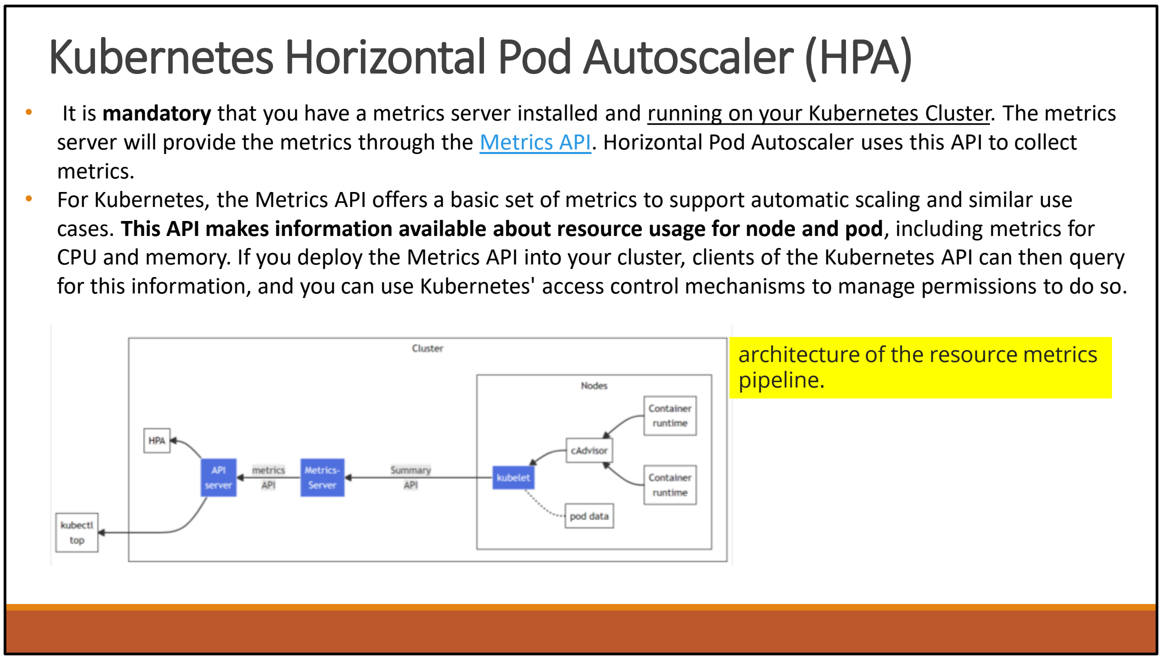
- HPA continuously monitors the metrics server for resource usage.
- Based on the collected resource usage, HPA will calculate the desired number of replicas required.
- Then, HPA decides to scale up the application to the desired number of replicas.
- Finally, HPA changes the desired number of replicas.
- Since HPA is continuously monitoring, the process repeats from Step 1.


1. Query for metrics
HPA
Metrics Server
2. Calculate the replicas
3. Update the Replica count
Deployment
Replica Set
4. Scale the desired Replicas
POD 1
POD 2
POD N

In Kubernetes, a *HorizontalPodAutoscaler* automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.
Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.
If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

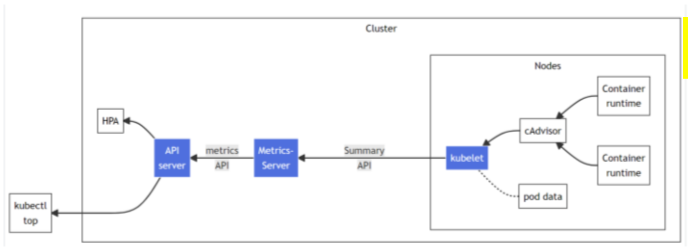
Horizontal pod autoscaling does not apply to objects that can't be scaled (for example: a DaemonSet.)

The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric you specify.



Kubernetes Horizontal Pod Autoscaler (HPA)

- It is **mandatory** that you have a metrics server installed and running on your Kubernetes Cluster. The metrics server will provide the metrics through the Metrics API. Horizontal Pod Autoscaler uses this API to collect metrics.
- For Kubernetes, the Metrics API offers a basic set of metrics to support automatic scaling and similar use cases. **This API makes information available about resource usage for node and pod**, including metrics for CPU and memory. If you deploy the Metrics API into your cluster, clients of the Kubernetes API can then query for this information, and you can use Kubernetes' access control mechanisms to manage permissions to do so.



architecture of the resource metrics pipeline.

The architecture components, from right to left in the figure, consist of the following:

- cAdvisor: Daemon for collecting, aggregating and exposing container metrics included in Kubelet.
- kubelet: Node agent for managing container resources. Resource metrics are accessible using the /metrics/resource and /stats kubelet API endpoints.
- Summary API: API provided by the kubelet for discovering and retrieving per-node summarized stats available through the /stats endpoint.
- metrics-server: Cluster addon component that collects and aggregates resource metrics pulled from each kubelet. The API server serves Metrics API for use by HPA, VPA, and by the kubectl top command. Metrics Server is a reference implementation of the Metrics API.
- Metrics API: Kubernetes API supporting access to CPU and memory used for workload autoscaling. To make this work in your cluster, you need an API extension server that provides the Metrics API.

Kubernetes Horizontal Pod Autoscaler (HPA)

- Metrics Server can be installed either directly from YAML manifest or via the official Helm chart. To install the latest Metrics Server release from the components.yaml manifest, run the following command.

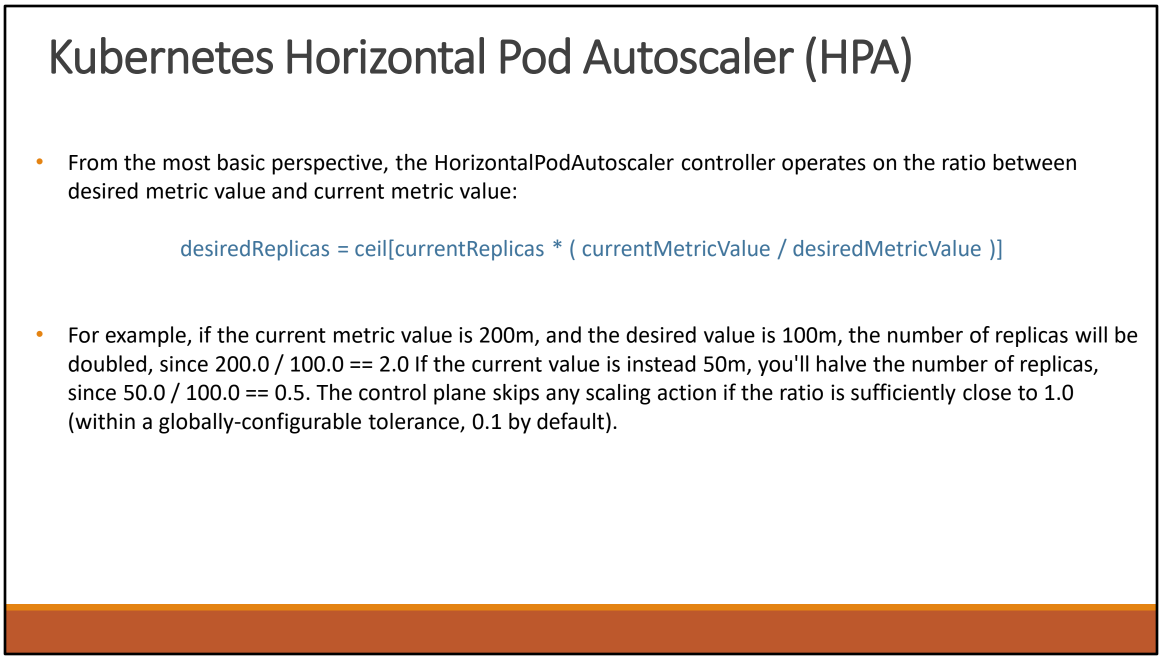
```
# kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

- You could also download the file, change it and deploy through the kubectl apply command.

Kubernetes Horizontal Pod Autoscaler (HPA)

- HPA runs intermittently (it is not a continuous process). The interval is set by the --horizontal-pod-autoscaler-sync-period parameter to the kube-controller-manager (and the default interval is 15 seconds).
- Once during each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition. The controller manager finds the target resource defined by the scaleTargetRef, then selects the pods based on the target resource's .spec.selector labels, and obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).
  - For per-pod resource metrics (like CPU), **the controller fetches the metrics from the resource metrics API** for each Pod targeted by the HorizontalPodAutoscaler. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each Pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted Pods, and produces a ratio used to scale the number of desired replicas.

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



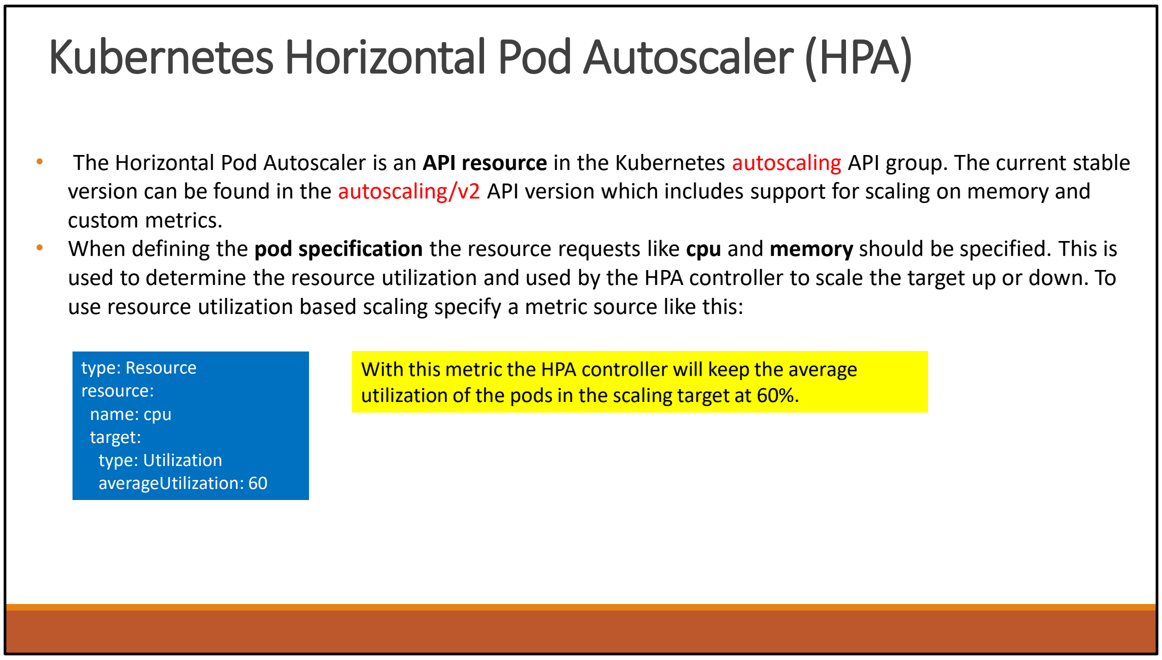
Kubernetes Horizontal Pod Autoscaler (HPA)

- From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:

  desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

- For example, if the current metric value is 200m, and the desired value is 100m, the number of replicas will be doubled, since 200.0 / 100.0 == 2.0 If the current value is instead 50m, you'll halve the number of replicas, since 50.0 / 100.0 == 0.5. The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a globally-configurable tolerance, 0.1 by default).

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



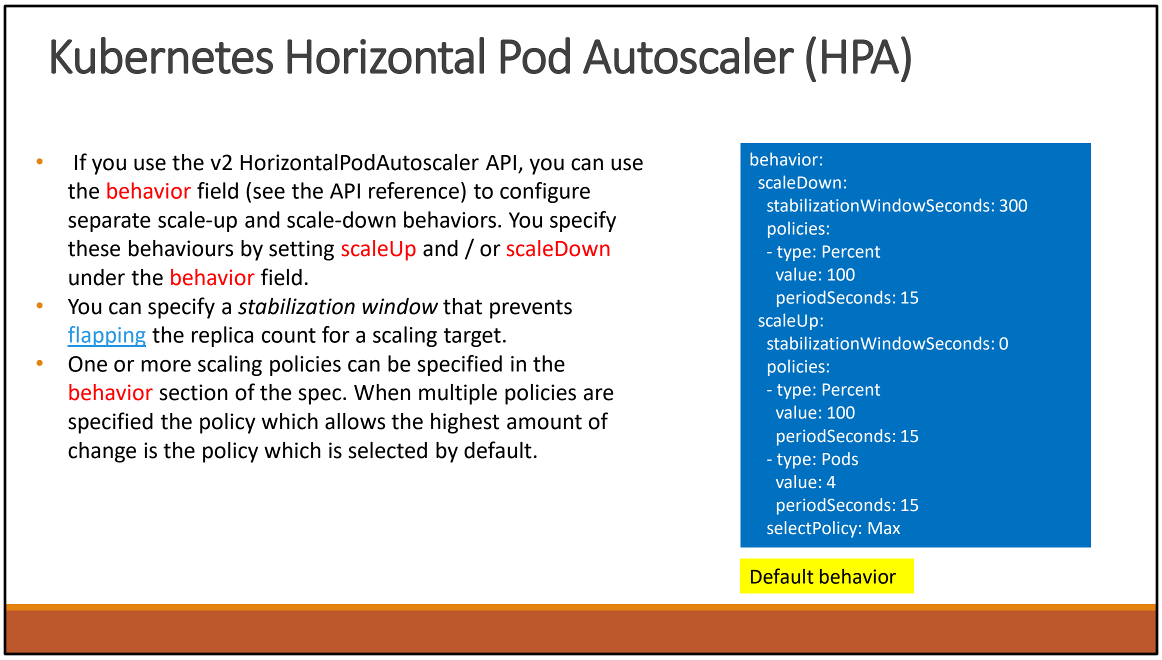
Kubernetes Horizontal Pod Autoscaler (HPA)

- The Horizontal Pod Autoscaler is an **API resource** in the Kubernetes autoscaling API group. The current stable version can be found in the autoscaling/v2 API version which includes support for scaling on memory and custom metrics.
- When defining the **pod specification** the resource requests like **cpu** and **memory** should be specified. This is used to determine the resource utilization and used by the HPA controller to scale the target up or down. To use resource utilization based scaling specify a metric source like this:

```
type: Resource
resource:
  name: cpu
  target:
    type: Utilization
    averageUtilization: 60
```

With this metric the HPA controller will keep the average utilization of the pods in the scaling target at 60%.

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



Kubernetes Horizontal Pod Autoscaler (HPA)

- If you use the v2 HorizontalPodAutoscaler API, you can use the behavior field (see the API reference) to configure separate scale-up and scale-down behaviors. You specify these behaviours by setting scaleUp and / or scaleDown under the behavior field.
- You can specify a *stabilization window* that prevents flapping the replica count for a scaling target.
- One or more scaling policies can be specified in the behavior section of the spec. When multiple policies are specified the policy which allows the highest amount of change is the policy which is selected by default.

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
    selectPolicy: Max
```

Default behavior

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

For scaling down the stabilization window is *300* seconds (or the value of the --horizontal-pod-autoscaler-downscale-stabilization flag if provided). There is only a single policy for scaling down which allows a 100% of the currently running replicas to be removed which means the scaling target can be scaled down to the minimum allowed replicas. For scaling up there is no stabilization window. When the metrics indicate that the target should be scaled up the target is scaled up immediately. There are 2 policies where 4 pods or a 100% of the currently running replicas will be added every 15 seconds till the HPA reaches its steady state.

Kubernetes Horizontal Pod Autoscaler (HPA)

- HorizontalPodAutoscaler, like every API resource, is supported in a standard way by kubectl. You can create a new autoscaler using kubectl create command. You can list autoscalers by kubectl get hpa or get detailed description by kubectl describe hpa. Finally, you can delete an autoscaler using kubectl delete hpa.
- In addition, there is a special kubectl autoscale command for creating a HorizontalPodAutoscaler object. For instance, executing **kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80** will create an autoscaler for ReplicaSet foo, with target CPU utilization set to 80% and the number of replicas between 2 and 5.

Example: Autoscaling applications using HPA for CPU Usage

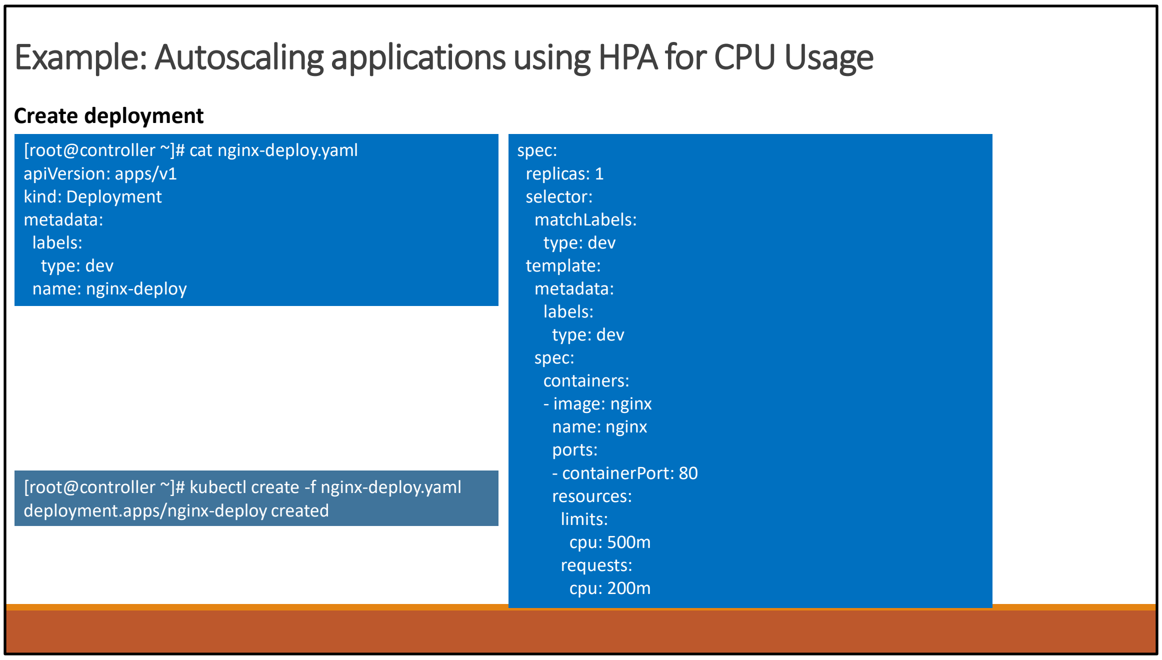
- We could some more configuration to this components.yaml file of the Metric server, as the new entries as highlighted:

```
...
 template:
  metadata:
   labels:
    k8s-app: metrics-server
  spec:
   hostNetwork: true   ## add this line
   containers:
   - args:
    - --cert-dir=/tmp
    - --secure-port=4443
    - --kubelet-insecure-tls   ## add this line
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --kubelet-use-node-status-port
    image: k8s.gcr.io/metrics-server/metrics-server:v0.4.2
...
```

--kubelet-preferred-address-types	- The priority of node address types used when determining an address for connecting to a particular node
--kubelet-insecure-tls	- Do not verify the CA of serving certificates presented by Kubelets.
hostNetwork	- Enable hostNetwork mode

Next we will deploy this manifest using kubectl apply command:

```
[root@controller ~]# kubectl apply –f  components.yaml
```



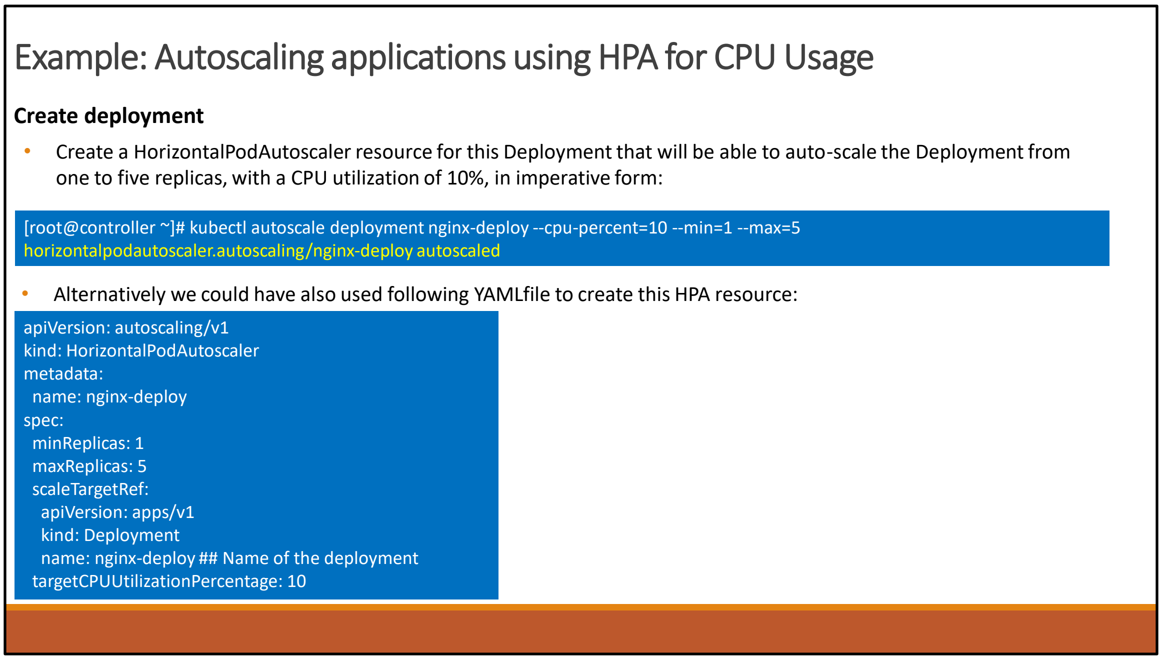
Example: Autoscaling applications using HPA for CPU Usage

Create deployment

```
[root@controller ~]# cat nginx-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    type: dev
  name: nginx-deploy
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      type: dev
  template:
    metadata:
      labels:
        type: dev
    spec:
      containers:
      - image: nginx
        name: nginx
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
```

```
[root@controller ~]# kubectl create -f nginx-deploy.yaml
deployment.apps/nginx-deploy created
```



Example: Autoscaling applications using HPA for CPU Usage

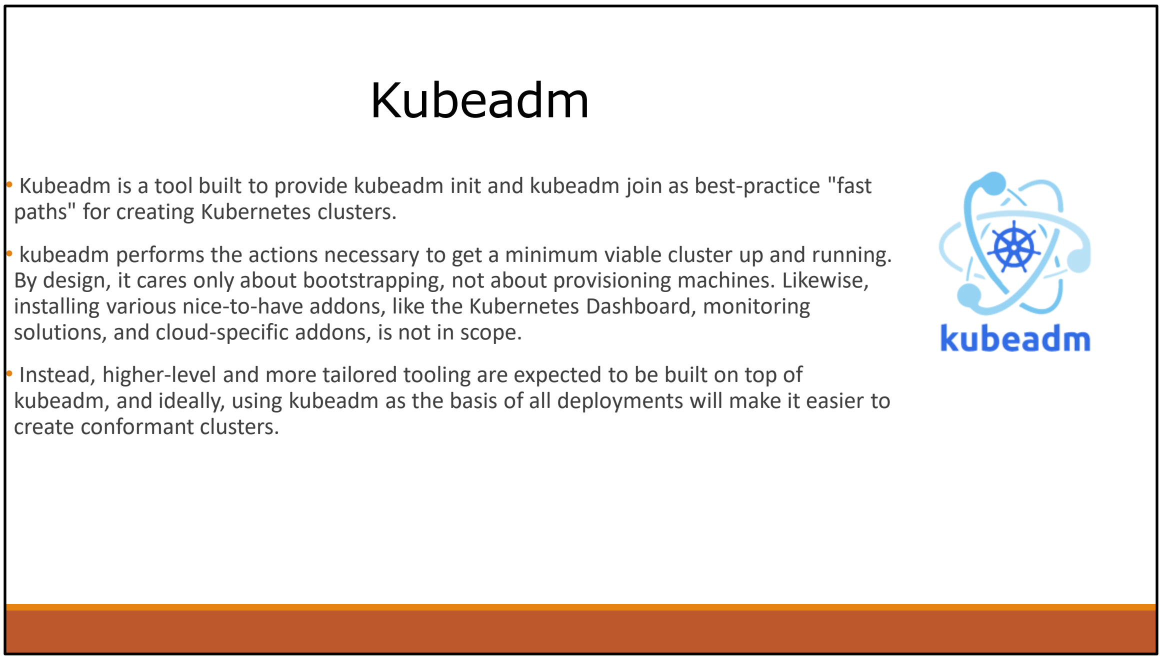
Create deployment

- Create a HorizontalPodAutoscaler resource for this Deployment that will be able to auto-scale the Deployment from one to five replicas, with a CPU utilization of 10%, in imperative form:

```
[root@controller ~]# kubectl autoscale deployment nginx-deploy --cpu-percent=10 --min=1 --max=5
horizontalpodautoscaler.autoscaling/nginx-deploy autoscaled
```

- Alternatively we could have also used following YAMLfile to create this HPA resource:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-deploy
spec:
  minReplicas: 1
  maxReplicas: 5
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deploy ## Name of the deployment
  targetCPUUtilizationPercentage: 10
```



Kubeadm

- Kubeadm is a tool built to provide kubeadm init and kubeadm join as best-practice "fast paths" for creating Kubernetes clusters.
- kubeadm performs the actions necessary to get a minimum viable cluster up and running. By design, it cares only about bootstrapping, not about provisioning machines. Likewise, installing various nice-to-have addons, like the Kubernetes Dashboard, monitoring solutions, and cloud-specific addons, is not in scope.
- Instead, higher-level and more tailored tooling are expected to be built on top of kubeadm, and ideally, using kubeadm as the basis of all deployments will make it easier to create conformant clusters.

kubeadm init
Run this command in order to set up the Kubernetes control plane

Sources

- https://kubernetes.io/docs/home/
- https://www.golinuxcloud.com/kubernetes-tutorial/
- https://linuxacademy.com/site-content/uploads/2019/04/Kubernetes-CheatSheet_07182019.pdf •
- https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
- https://vimeo.com/245778144/4d1d597c5e
- https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/
- https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727
- https://medium.com/google-cloud/understanding-kubernetes-networking-services-f0cb48e4cc82
- https://github.com/containernetworking/cni
- https://github.com/containernetworking/cni/blob/master/SPEC.md
- https://www.objectif-libre.com/en/blog/2018/07/05/k8s-network-solutions-comparison/
- https://chrislovecnm.com/kubernetes/cni/choosing-a-cni-provider/
- https://www.contino.io/insights/kubernetes-is-hard-why-eks-makes-it-easier-for-network-and-security-architects
- https://medium.com/flant-com/calico-for-kubernetes-networking-792b41e19d69
- https://newrelic.com/blog/how-to-relic/how-to-use-kubernetes-volumes