

Kubernetes Basics

Part 1: Clusters, nodes, and pods

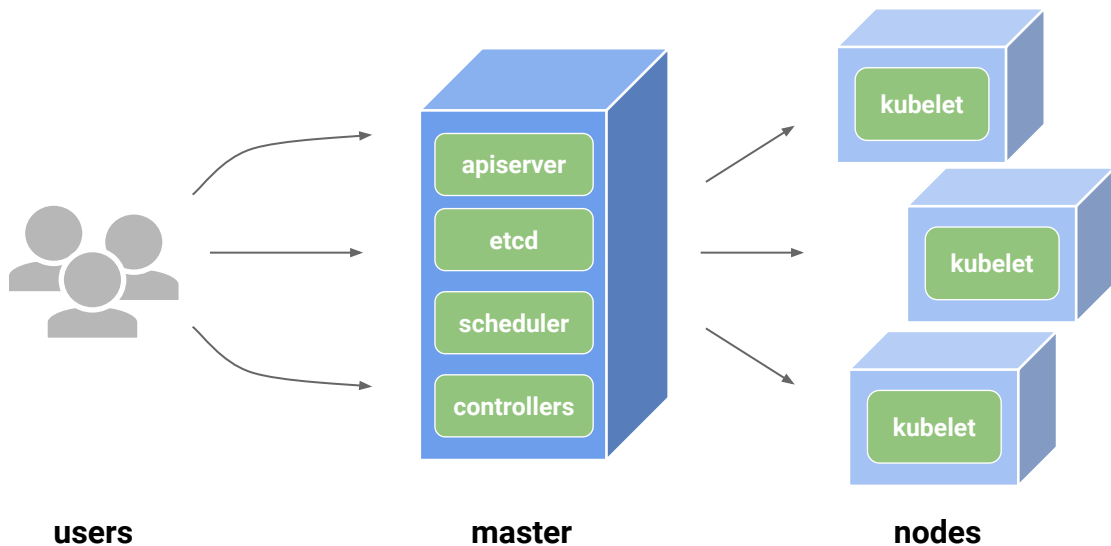
Version 1.6



© 2017 Google Inc. All rights reserved. Google and the Google logo are trademarks of Google Inc. All other company and product names may be trademarks of the respective companies with which they are associated.

[Duration: 30 mins]

The 10,000-foot view



Google Cloud

At the highest level, **Kubernetes** is a set of APIs that you can use to deploy containers on a set of nodes.

On the left you have the user, in the middle you have the master components of Kubernetes and on the right you have the Kubernetes worker nodes.

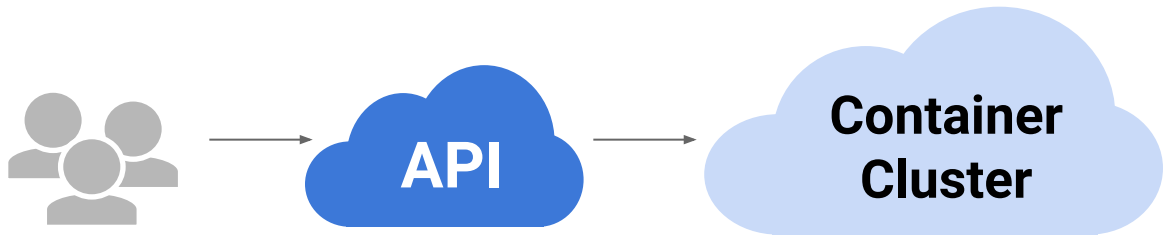
The master controls all the jobs that run on the nodes.

To be clear, Kubernetes is an open source **orchestrator** for running a container environment (see kubernetes.io).

It can run on many different platforms, from laptops to high-availability multi-node clusters, from public clouds to on-premise deployments, from virtual machines to bare metal.

Users access your API via a command-line interface, HTTP, or a user interface.

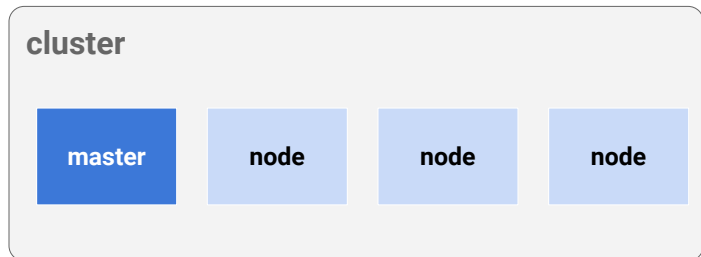
All users really care about



Most users only really care that you provide them with an API.

And most operators only really care that they have a running container **cluster**.

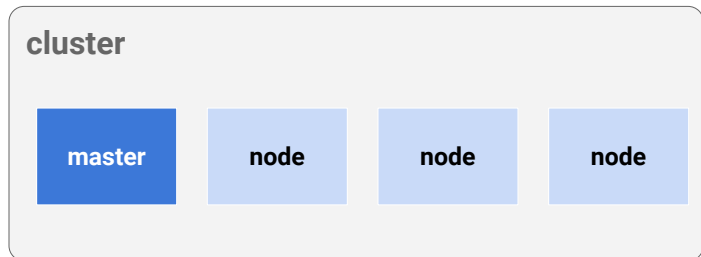
A cluster is a set of computers



A **cluster** is a set of computers working as an instance managed by Kubernetes.

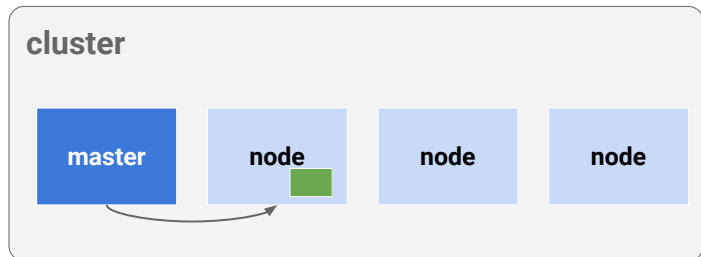
A cluster is a set of computers

Managed by Kubernetes



Everything is managed by Kubernetes which is an orchestrator.

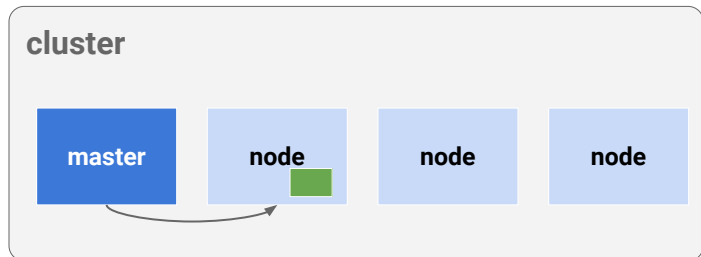
Kubernetes manages jobs



At the heart of the matter, Kubernetes manages jobs.

Kubernetes manages jobs

Jobs run containers on nodes



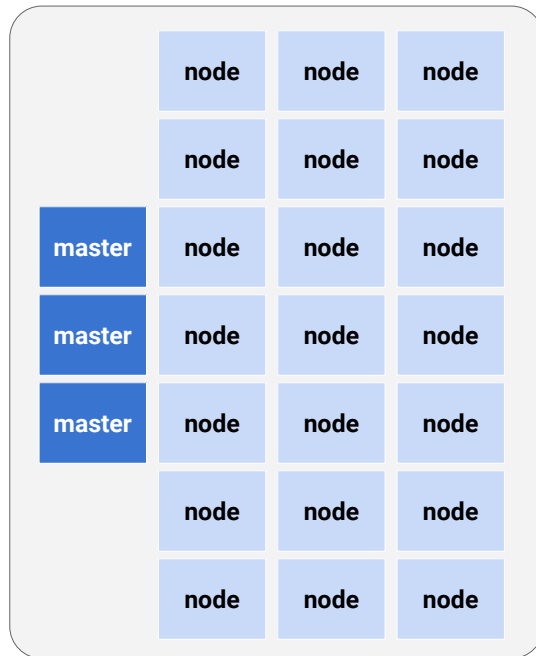
At the heart of the matter, Kubernetes manages jobs.

Jobs run containers on nodes

In Google Cloud, nodes are virtual machines running in Google Compute Engine.

In a real ecosystem, a cluster may have 1000s of nodes and multiple masters.

Regional clusters have masters and nodes spread across 3 zones



Google Cloud

In a real ecosystem, a cluster has may have 1000s of nodes and multiple masters.

You can have up to 5,000 nodes in a cluster.

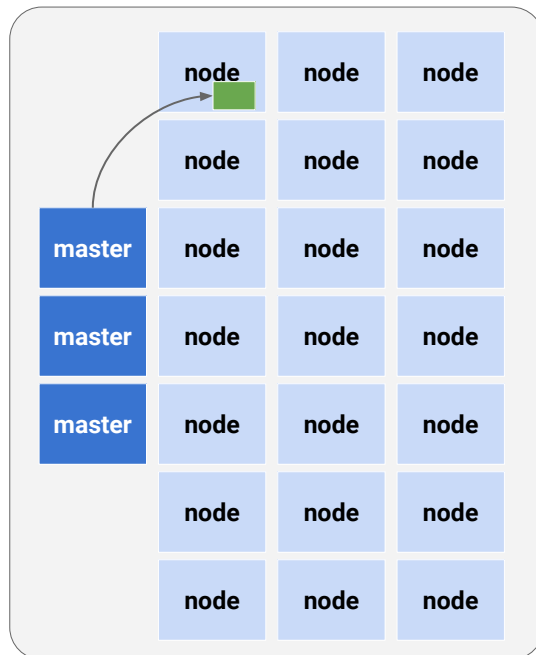
Regional clusters have masters and nodes spread across 3 zones for high availability and resilience from single zone failure and downtime during master upgrades.

By default, 3 masters are created per zone (for a total of 9 masters), but you can control the number with the `--num-nodes` flag..

See the link for more details:

<https://cloud.google.com/blog/products/gcp/with-google-kubernetes-engine-regional-clusters-master-nodes-are-now-highly-available>

Masters schedule jobs on nodes based on load



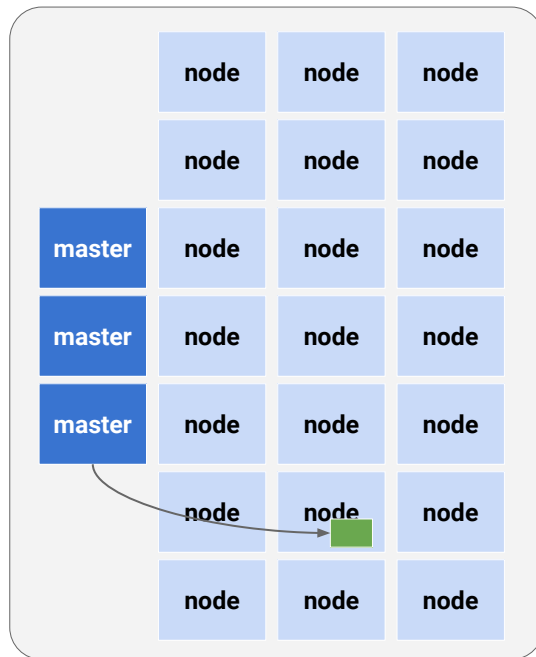
© 2019 Google LLC. All rights reserved. Google and the Google logo are trademarks of Google Inc. All other marks and names are the property of their respective owners. All other marks and names are the property of their respective owners.

Google Cloud

Masters schedule jobs or tasks on nodes based on load.

Can can specify load criteria.

And they can move them as needed to match resources



© 2019 Google LLC. All rights reserved. Google and the Google logo are trademarks of Google Inc. All other marks and names are the property of their respective owners. All other marks and names are the property of their respective owners.

Google Cloud

Masters schedule jobs or tasks on nodes based on load.

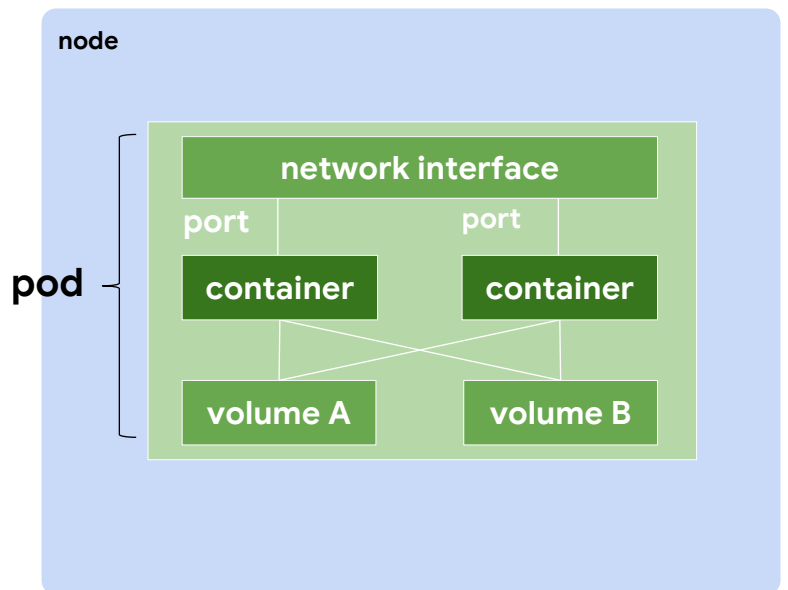
Can can specify load criteria.

A job is called a **Pod**

It's analogous to a VM

It can run multiple containers

It shares networking and storage separate from the node



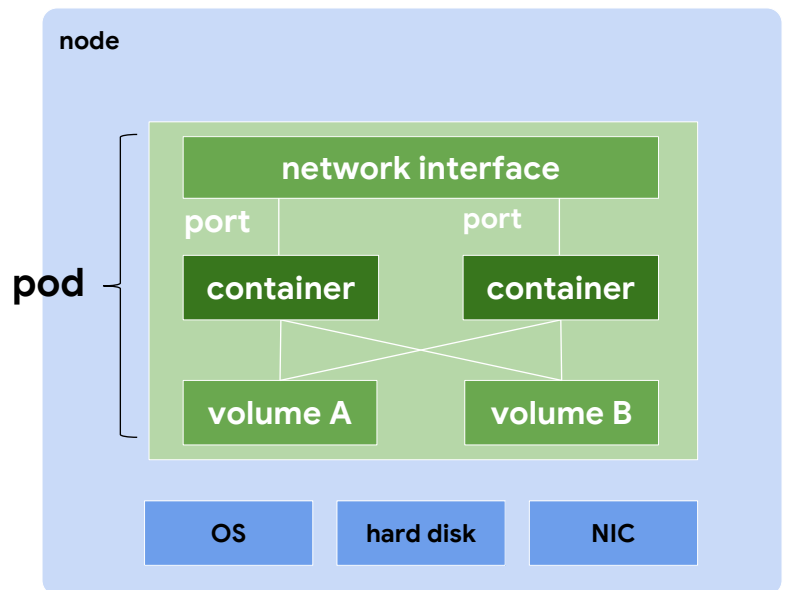
So let's talk about a **pod**.

In Kubernetes, a job is called a Pod.

It's analogous to a VM and is the basic unit of deployment for containersL.

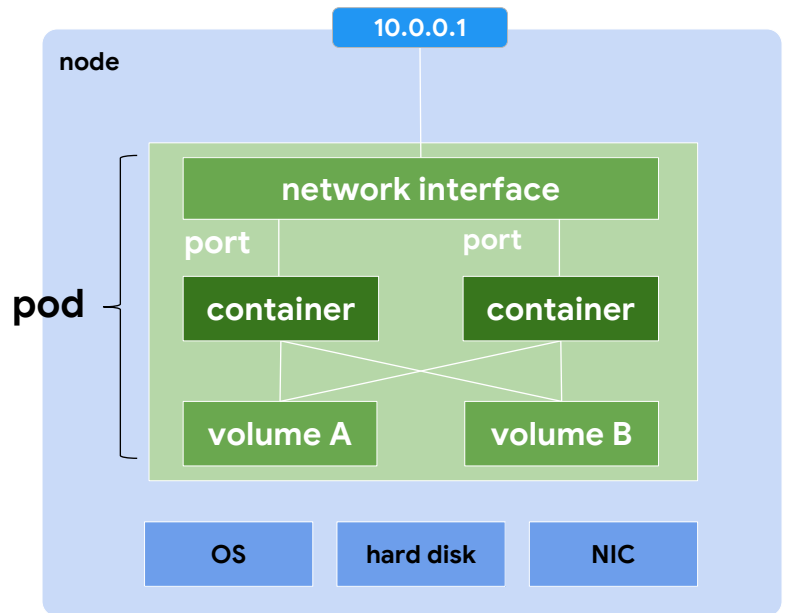
It can run one or more containers sharing networking and storage that are separated from the underlying node.

Underneath the pod
you have the node's
hardware, OS, and
NIC



Underneath, you have the hardware, OS, and NIC, ...

And a node IP

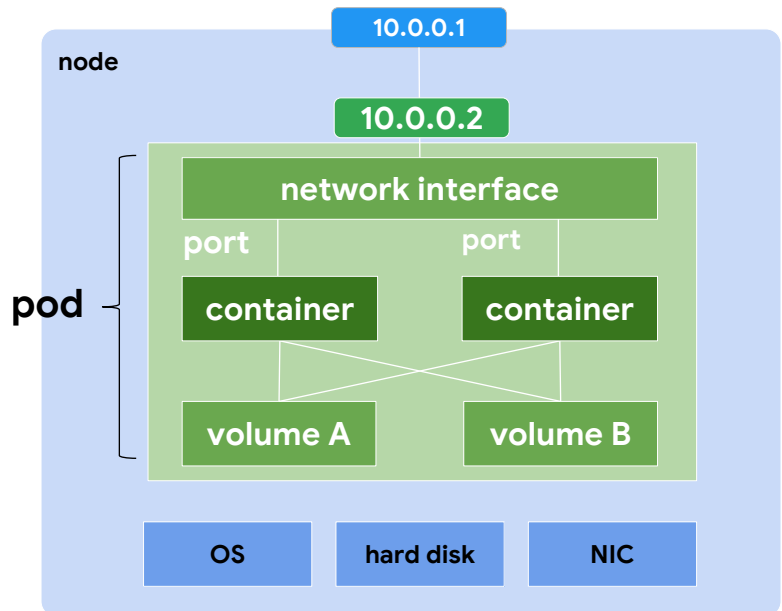


and a node IP outside.

Inside you have the pod with its containers

It gets its own network namespace with unique IP and set of ports

Data is stored in **volumes** in memory or persistent disks



Google Cloud

Inside you have the pod. And then inside the pod, you have the containers and the network interfaces for the containers. It gets its own network namespace with a unique cluster IP and set of ports. Containers in the same Pod can communicate with each other using localhost.

Data is stored in volumes that reside in memory or persistent disks.

This is analogous to a VM and is highly-portable because you can move pods to other nodes without re-configuring or rebuilding anything.

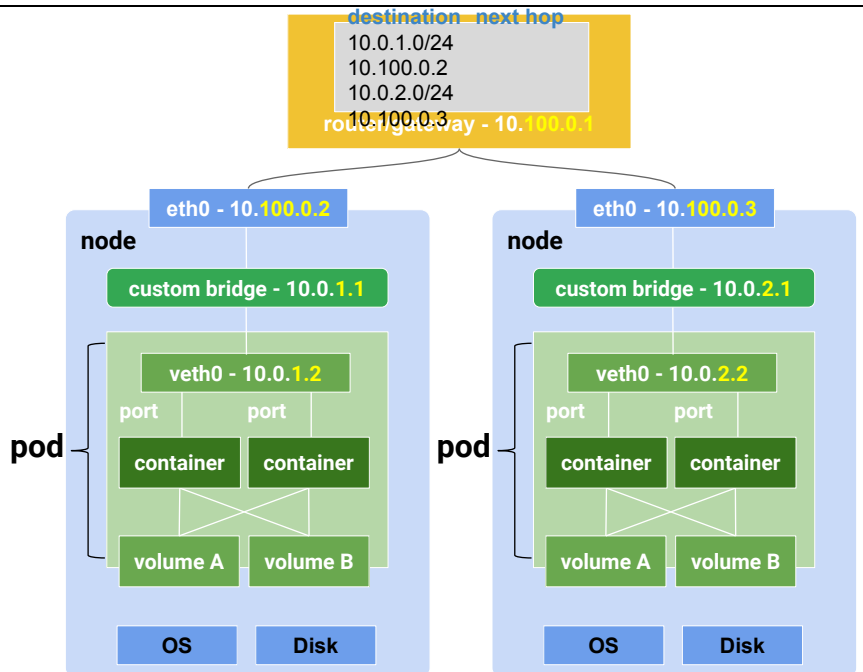
You typically only run 1 container per pod unless they have a hard dependency like a side-car application for logging or file updating.

This also allows for very fine-grained scaling.

Networking between pods in different nodes is configured with a central routing table in masters

GKE pushes routes to node iptables and uses port forwarding for node-to-node communication and fewer failure points

See other GKE courses for more details



Google Cloud

Networking between pods in different nodes is configured with a central routing table in masters.

GKE pushes routes to iptables in nodes and uses port forwarding so nodes can communicate with each other directly and load balancers do not introduce a single point of failure.

For more information, see our other courses on GKE or "Understanding Kubernetes Networking: Pods"

<https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727>

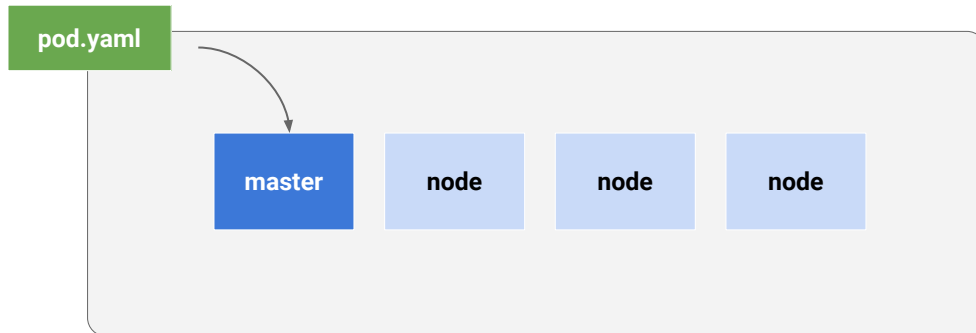
You define a pod with a YAML file

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app
    image: my-app
  - name: nginx-ssl
    image: nginx
    ports:
    - containerPort: 80
    - containerPort: 443
```

So how do you define a pod?

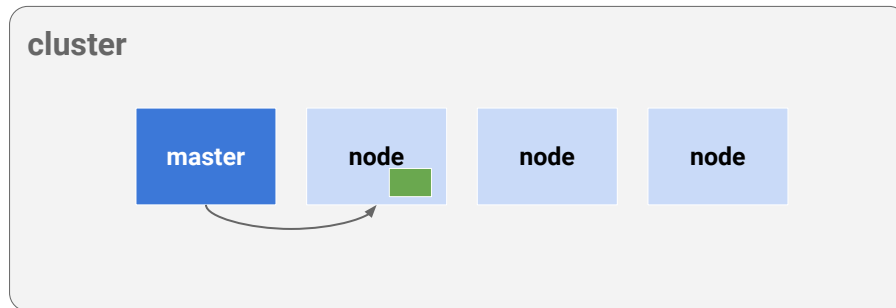
Most things in Kubernetes are defined via the API, and you define a YAML document like this that you upload to the API server, where it runs your containers.

You upload the YAML file to the master



After you create the pod file, you can upload it to the master.

And the master creates a pod on your set of nodes



Then etcd on the master creates a pod on your set of nodes.

A pod file is composed of several parts; for example...

```
apiVersion: v1      — API version
kind: Pod           — pod resource
metadata:
  name: my-app      — pod name
spec:
  containers:       — two containers
  - name: my-app
    image: my-app
  - name: nginx-ssl
    image: nginx
  ports:            — NGINX front end on two ports
  - containerPort: 80
  - containerPort: 443
```

Going back to the definition, you have:

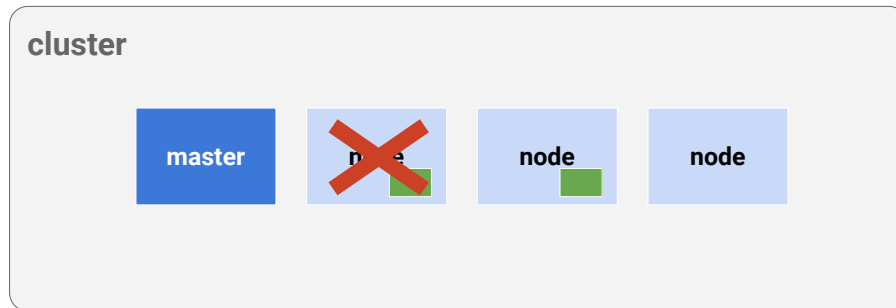
- The API version
- The resource **kind** (in this case a pod)
- Some metadata including the pod name
- A list of containers.

In this case, you have two containers: my-app and nginx. NGINX serves as the app front end on two ports: 80 and 443.

The containers can talk to each other using the pod's localhost network and my-app is the backend that doesn't need to be exposed on the network.

Again, you upload it the file to the master and the master decides which node to run it on.

A deployment ensures that N pods are running in a cluster at any given time



© 2019 Google LLC. All rights reserved. Google and the Google Cloud logo are trademarks of Google Inc. All other marks and names are the property of their respective owners.

Google Cloud

One other powerful concept in Kubernetes is the **deployment**, and this is how you ensure that N number of pods are running at a given time.

If you upload the pod file as shown in the previous example, you will have one pod running in the cluster, which is fine, but if the pod or the node that the pod is running on stops or restarts, the pod will not be restored in the cluster.

In order to restore it, you have to define and run a deployment with a ReplicaSet that defines how many pods you want running.

In the previous example, it would be 1.

```

kind: Deployment — deployment resource
apiVersion: v1.1
metadata:
  name: frontend — deployment name
spec:
  replicas: 4 — replicas
  selector: — pod selector
    role: web — role=web
  template:
    metadata:
      name: web
      labels: — pod label
        role: web — role=web
    spec:
      containers: — containers
      - name: my-app
        image: my-app
      - name: nginx-ssl
        image: nginx
      ports:
      - containerPort: 80
      - containerPort: 443

```

You define a deployment with a YAML file

Here is a deployment YAML file which is similar to a pod file.

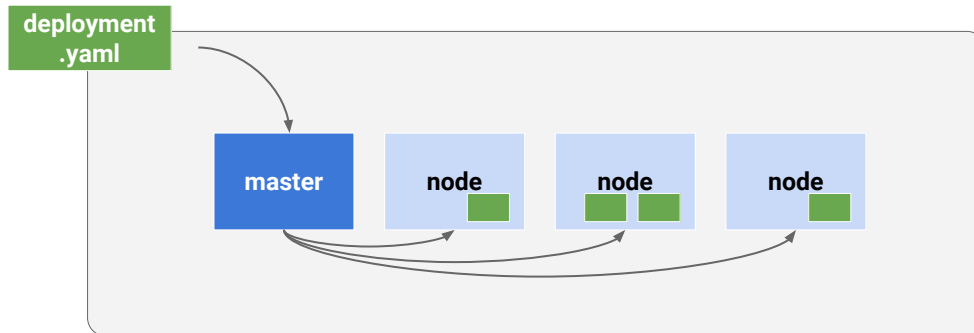
In it, you have:

- The resource **kind** (in this case deployment)
- Metadata with the deployment name
- A **ReplicaSet** with the number of pods to run
- A **selector** label for choosing which pods to include in the deployment
- And pods with their **labels** and containers in a **template**.

You identify and group pods with labels and select them in the deployment with a selector.

In this example, you are going to run 4 pod replicas with role set to web.

You upload the YAML file to the master, and the scheduler decides where to run the pods



Then you upload the file to the master.

When you do this, the master scheduler decides where to run the pods.

In this case, you want 4 replicas, but only have 3 nodes so the scheduler doubles up pods on one of the nodes, perhaps the node with the least CPU utilization.

Kubernetes Basics

Part 2: Services, labels, selectors

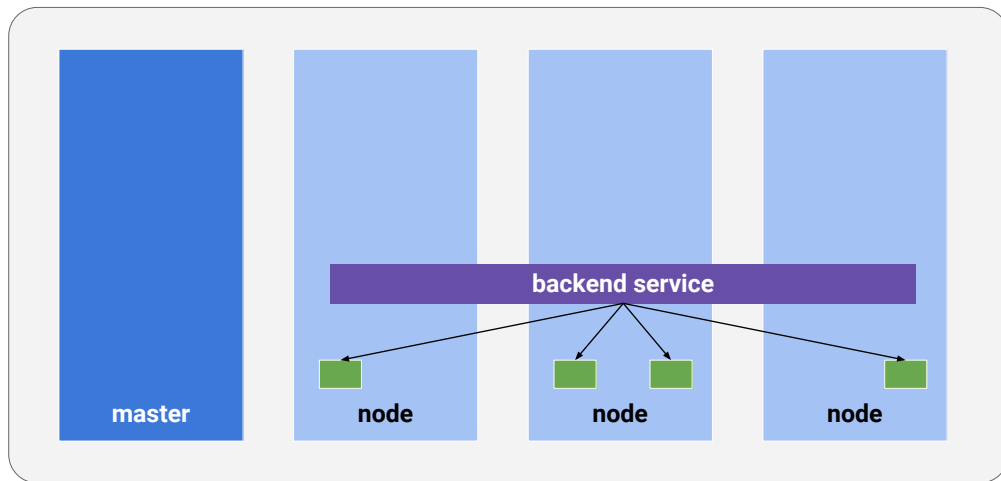
Version 1.6



© 2017 Google Inc. All rights reserved. Google and the Google logo are trademarks of Google Inc. All other company and product names may be trademarks of the respective companies with which they are associated.

Now that you have a deployment of replica pods running across your cluster, how do you get other pods or deployments to talk to them? They're not going to run in isolation. They probably need to communicate with other pieces of the cluster and other applications running in the cluster.

A service assigns a fixed IP to your pod replicas and allows other pods or services to communicate with them



© 2019 Google LLC. All rights reserved. Google and the Google logo are trademarks of Google Inc. All other marks and names are the property of their respective owners.

Google Cloud

Pods aren't meant to be persistent. They can be stopped or started for many reasons (like failed liveness or readiness checks). When they get restarted, they might have a different IP address, and this leads to a problem.

That's where [Services](#) come in.

A **service** is a network abstraction in Kubernetes that provides a stable endpoint (or fixed IP) for a group of pods. This allows other pods or deployments to reference them as a unit and communicate with them over time.

Services (like deployments) use selectors to determine what pods they operate on. If pods have the correct labels, they are automatically picked up and exposed by a service.

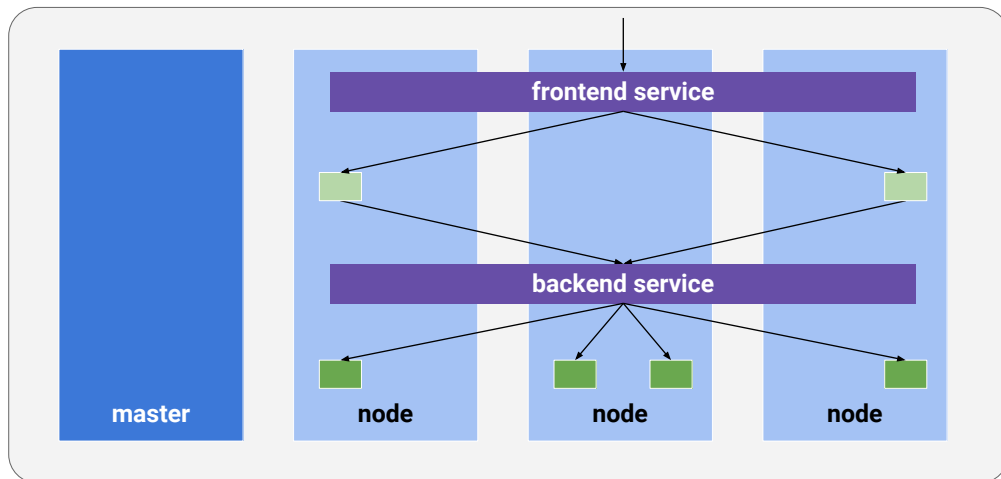
The level of access a service provides to a set of pods depends on the service type. Currently, there are three types:

1. **ClusterIP** (internal): The default type means that this Service is only visible inside the cluster.
2. **NodePort** gives each node in the cluster an externally accessible IP.
3. **LoadBalancer** adds a load balancer from the cloud provider that forwards traffic from the service to Nodes within it.

Here is an example where you have a back-end service. You want other pods, say in the front-end, to talk to these pods. So you create a service that groups your pods in the back-end service and you tell your front-end pods to communicate through that service.

The service creates a DNS entry with a fixed IP address that other pods and services can use to reach the service.

You can have multiple services with different configurations and features



© 2019 Google LLC. All rights reserved. Google and the Google logo are trademarks of Google Inc. All other marks and names are the property of their respective owners. All other marks and names are the property of their respective owners.

Google Cloud

In this example, you want the front end publicly exposed, so you create another service.

This time you want to receive traffic from outside your cluster. When you set the service type to **load balancer**, you get a Google Cloud Load Balancer and you can set details using Cloud Console or the GCP API.

You define a service with a YAML file

```
kind: Service      — resource
apiVersion: v1
metadata:
  name: web-frontend
spec:
  ports:           — ports
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:        — pod selector
  role: web
  type: LoadBalancer — type is load balancer
```

So once again, you create a service with a YAML file.

It contains:

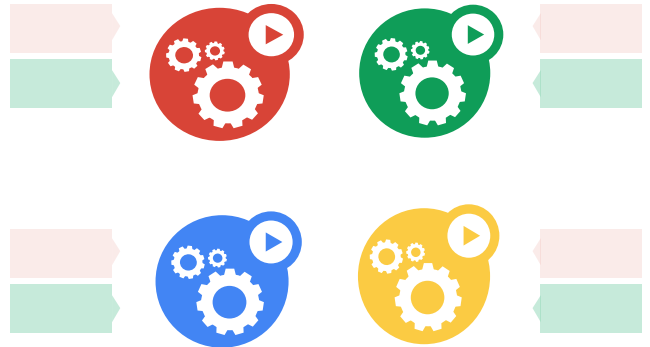
- The **kind** of resource (it's a service)
- A service name
- The source and target **ports**
- A pod **selector**
- And the **type** (in this case, load balancer).

In this case, the service is the front end. You set its port number (80) and its type (load balancer). The default type is **Cluster IP** which means that only services inside your cluster can access that service. Load balancer means it's external and can be reached from anywhere that knows the DNS name or IP address. The service also has selector set to role=web. So only pods with labels set to role=web will be included in the service.

Labels are metadata you can assign to any API object and represent identity

They are

- The only grouping mechanism for pods
- Search by selectors



Labels are arbitrary metadata you can attach to any object in the Kubernetes API. Labels tell you how to group these things to get an identity. This is the only way you can group things in Kubernetes.

So let's see an example.

This example has four pods and three labels



Here we have 4 pods running. They each have 3 labels.

One is the name of the app; another is the phase (whether is running in test mode or production); and the third is the role (whether it's part of the front end or back end).

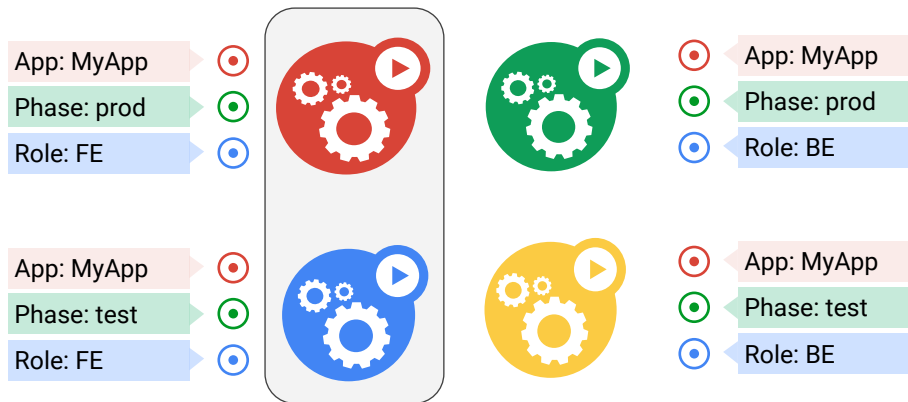
You can query for labels that map to a value like the entire app



App = MyApp

If you want to group these things and we wanted to get all the pods that map to MyApp—if for example we wanted to see how many pods are running as part of my entire application—we could filter requests with the App = MyApp label.

Or narrow your search with multiple labels like your app's frontend



App = MyApp, Role = FE

You could also narrow it even further by adding two labels to my query and say "I only want to see pods that are App, MyApp and Role front end."

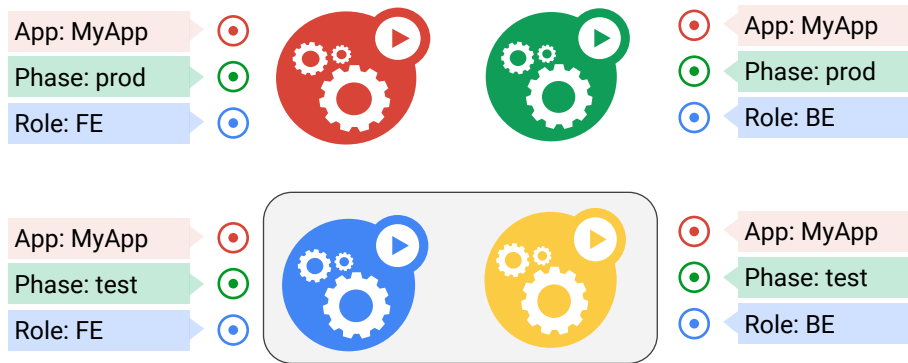
Or your app's backend



App = MyApp, Role = BE

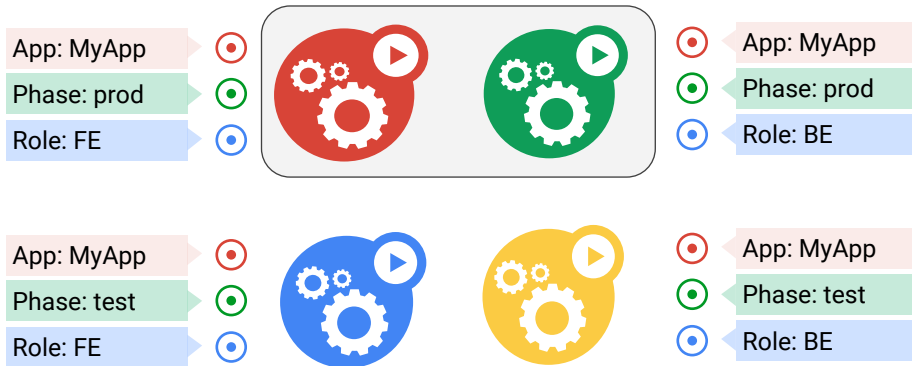
Similarly for the back end.

Or your app's test phase



Operations folks may only want to look at your test pods.

Or your app's production release

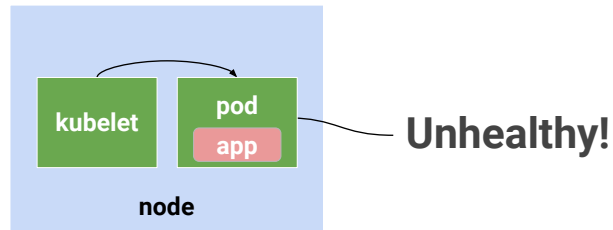


Or your production pods.

So you can see how labels and selectors can group pods or other items into sets with different semantic meanings for different users in Kubernetes.

An example you've already seen is where services map pods into groups so you can route traffic to the right pods in your cluster.

Kubelet checks whether the pod is alive and healthy; if it gets a negative response or no reply...



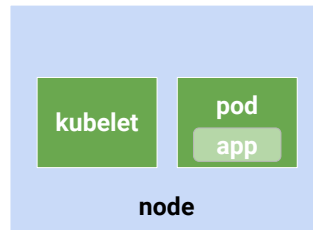
So we've talked about labeling. Let's talk about monitoring and health checks so nodes know they're running healthy pods.

Each pod that you run can be monitored with a health check. The health check is run by the node and checks whether that pod is alive and healthy.

After you define a health check, the Kubernetes component on the node, called the **kubelet**, issues a health check to the pod that asks "are you alive and healthy?"

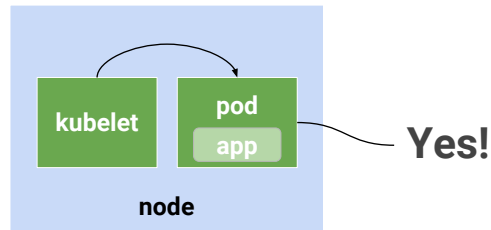
And if not, ...

Kubelet restarts the pod



... the kubelet restarts that pod on the same node.

And continues until it gets a healthy reply



And the process continues until the kubelet gets a healthy reply.

To define the health check, you specify an API request, an HTTP request, or a TCP port to check on the pod running on the node. And the kubelet continuously checks that request or port to make sure it's successful.

So, you can get some resiliency out of your app by adding health checks to ensure that your application is healthy and ready to receive traffic.

Kubernetes Basics

Part 3: Volumes

Version 1.6



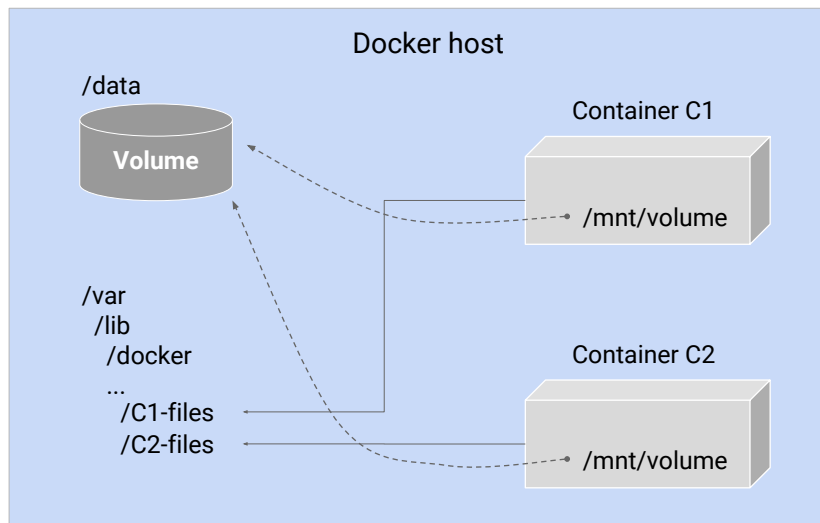
© 2017 Google Inc. All rights reserved. Google and the Google logo are trademarks of Google Inc. All other company and product names may be trademarks of the respective companies with which they are associated.

Now let's look at how to get data into your pod and store data persistently using Kubernetes volumes.

Even though a pod isn't meant to be persistent, its data may be.

Docker also has a concept of **volumes** though it is somewhat looser and less managed than a Kubernetes volume.

Docker provides data storage for containers, but volumes do not provide sharing between containers or lifecycle management



[a lot of the following is paraphrased from: kubernetes.io and docker.com]

Docker allows containers to store data to on-disk files in their local storage area (/var/lib/docker/...), but they are ephemeral. When a container is deleted, any data written to this area is deleted.

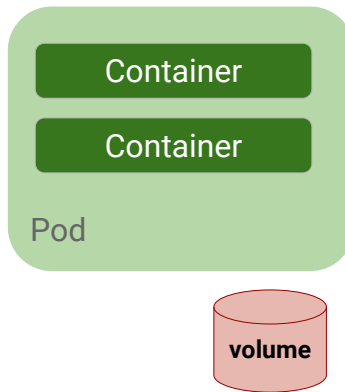
There is also a single shared data volume located at /data on the Docker host. This is mounted directly into host containers. When a container is deleted, any data stored in data volumes persists on the Docker host as long as the host continues to run.

But lifetimes of Docker volumes are not managed, and until very recently there were only local disk-backed volumes.

Docker now provides volume drivers (not shown) so volumes can reside off the host and remain persistent, but the functionality is limited for now.

Also, when running multiple containers, it may be necessary to share files between containers, but this is not available with Docker drivers yet.

Kubernetes volumes allow containers in pods to share data and be stateful



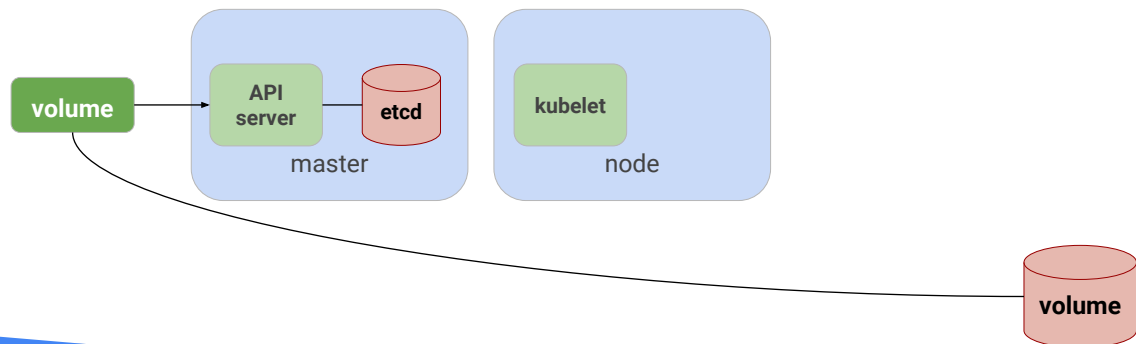
Kubernetes volumes solve both of these problems.

Volumes are a way for containers within a pod to share data, and they allow for Pods to be stateful. These are two very important concerns for production applications.

There are many different types of volumes in Kubernetes. Some of the volume types include long-lived persistent volumes, temporary, short-lived emptyDir volumes, and networked nfs volumes.

A volume is just a directory, and how it gets created depends on its type

```
$> kubectl create <volume>
```



Google Cloud

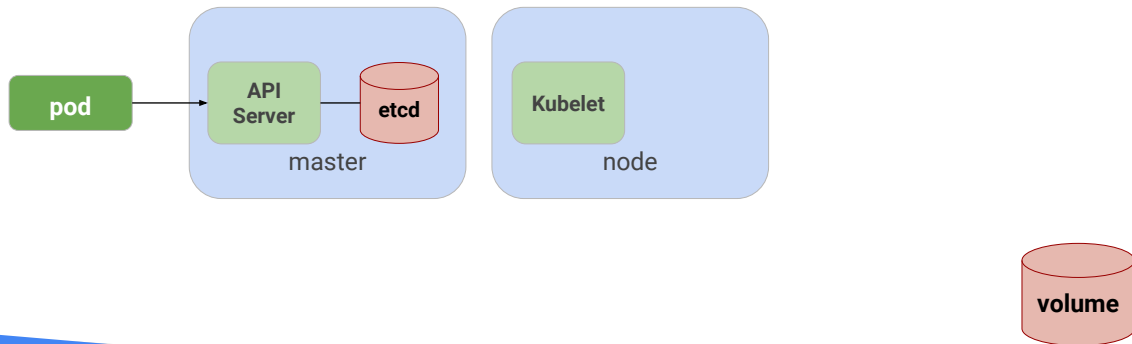
At its core, a volume is just a directory that is accessible to the containers in a pod. How that directory is created, the medium that backs it, and its contents are determined by the particular volume type used.

The slide images follow the lifecycle of a typical volume. In this case, it's an NFS volume.

You begin by creating the volume in your cluster.

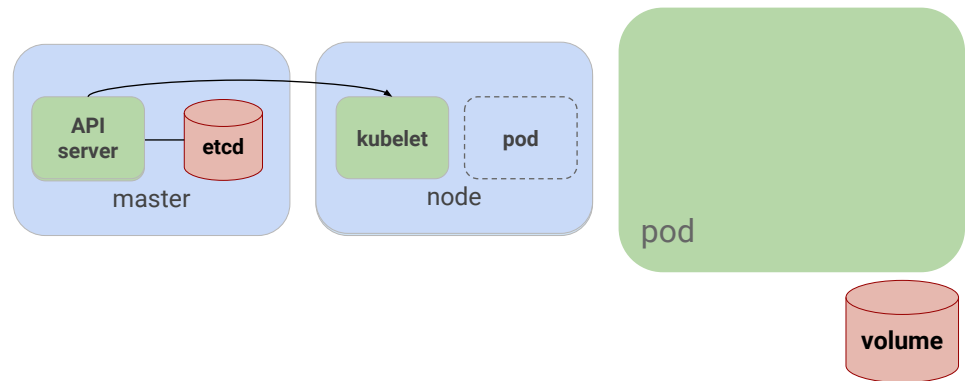
Then you create a pod that consumes that data

```
$> kubectl create -f pod.yaml
```



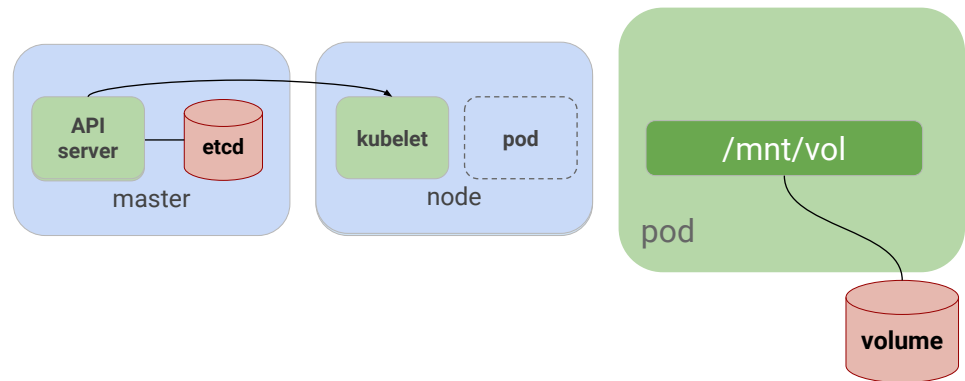
After you have a volume, you create a pod that consumes that data. In this example, that is done with the `kubectl create` command.

The volume is attached to the pod and made available to containers before they are brought online



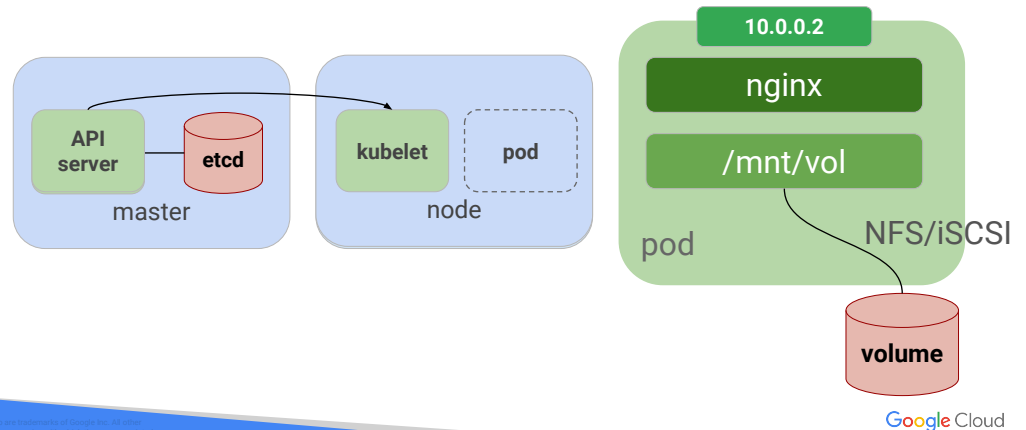
After the pod is created, the volume is attached to the pod. This volume is made available to any container in the pod before the containers are brought online.

Once the volume is attached, data can be mounted into a container's file system



Once the volume is attached, the data in it is mounted into the container's file system.

Then the container is run and can get the mounted data



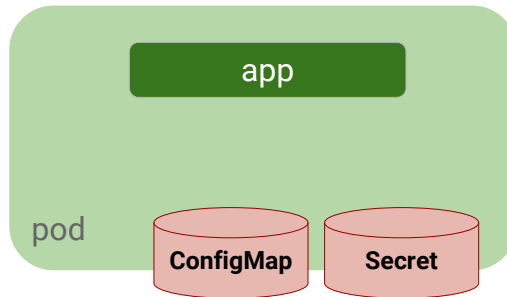
After the data in volumes is mounted, the containers in the pod are brought online and the rest of pod initialization happens as before.

In this example, you have an NGINX pod running an NFS volume. So you attach the volume at the mount volume directory when the pod gets run, and then the NGINX container can see that /mnt/vol directory and it can get the data from it.

Volumes, like those backed by block stores, may outlive the pod. They will be unmounted when a pod goes away and get remounted on new machines if needed.

Some volumes, like ConfigMaps, cease to exist when the pod ceases to exist.

Some volumes share the lifecycle of their pod



Google Cloud

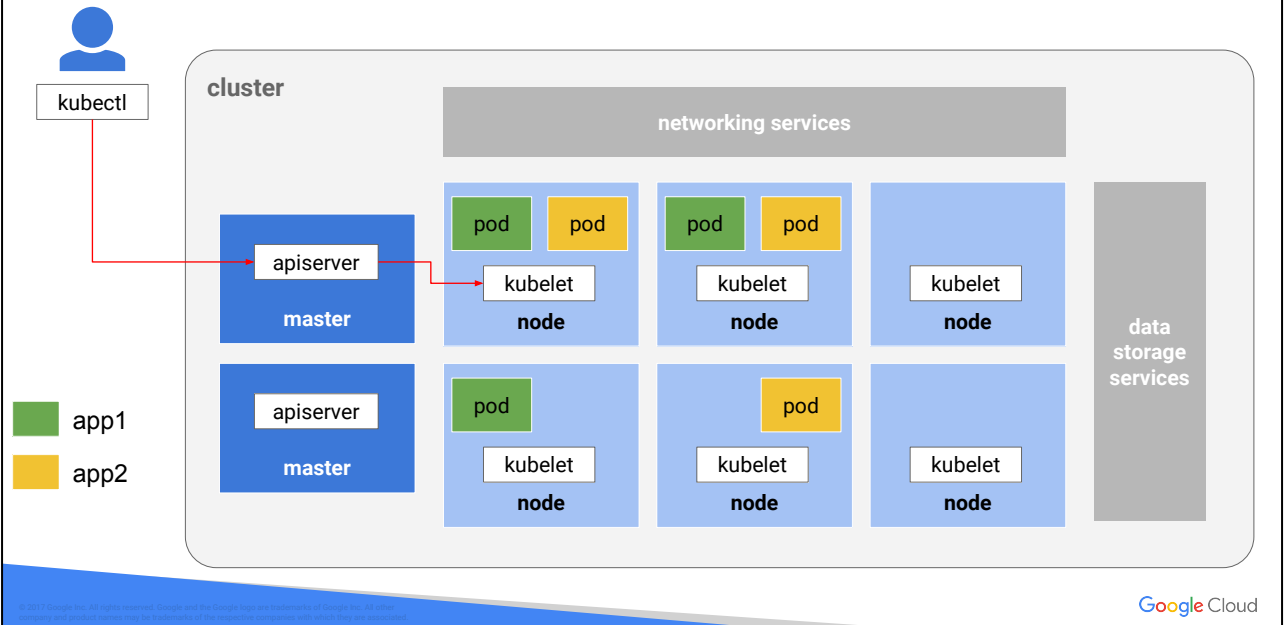
Some volumes, like ConfigMaps and Secrets, are coupled to the life of the pod and cease to exist when the pod ceases to exist. One class of volume is a persistent volume that has a life cycle of its own, independent of the pod.

Secrets are a way to store sensitive, encrypted data, such as passwords or keys. These are used to keep developers from having to bake sensitive information into their pods and containers. Secrets are stored in a temporary filesystem so that they're never written into non-volatile storage.

ConfigMaps are subtly different. They are used for non-sensitive string data. Storing configuration, setting command line variables, and storing environment variables are natural use cases for ConfigMaps.

Currently, Secrets and ConfigMaps are stored in etcd.

Here's a complete overview of a cluster



Here's a complete overview of a cluster with its key components.

You have a set of master servers and worker nodes. The masters provide the control plane for the cluster. Worker nodes run pods with containers in them.

Cluster administrators configure the cluster by sending requests to **apiservers** on masters using a command-line tool called **kubectl**. Kubectl can be installed and run anywhere.

From there, the apiserver communicates with the cluster in two primary ways:

- To the **kubelet** process that runs on each node
- To any node, pod, or service through the apiserver's proxy functionality (not shown).

Then pods are started on various nodes. In this example, there are two types of pods running (shown in yellow and green).

There is also a process on each node called **kube-proxy** (not shown) that sets up networking rules and connection forwarding for services and pods on the host.

Although networking and data storage services are shown outside nodes, most functionality resides on nodes.

You can also access the apiserver using a web interface called the dashboard via

kubectl proxy (not shown).



Lab

The image features a rectangular frame containing an abstract composition of three green geometric shapes. A large, medium-green trapezoid occupies the left and center. To its right is a darker green trapezoid, and below the medium-green shape is a lighter green trapezoid. The word 'Lab' is written in white, sans-serif font within the medium-green area.