Kubernetes Basics

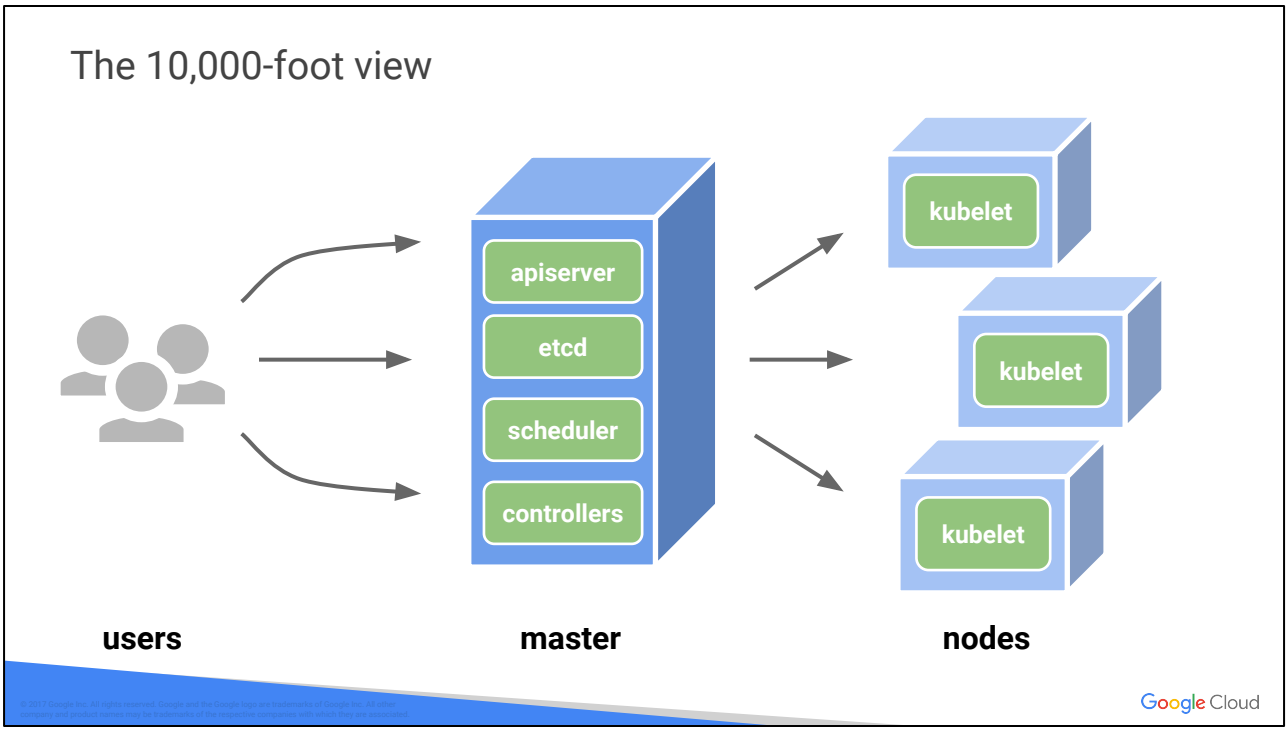
Getting Started With Google Kubernetes Engine

Version 1.5

Google Cloud

[Duration: 30 mins]

[Duration: 5-10 mins]

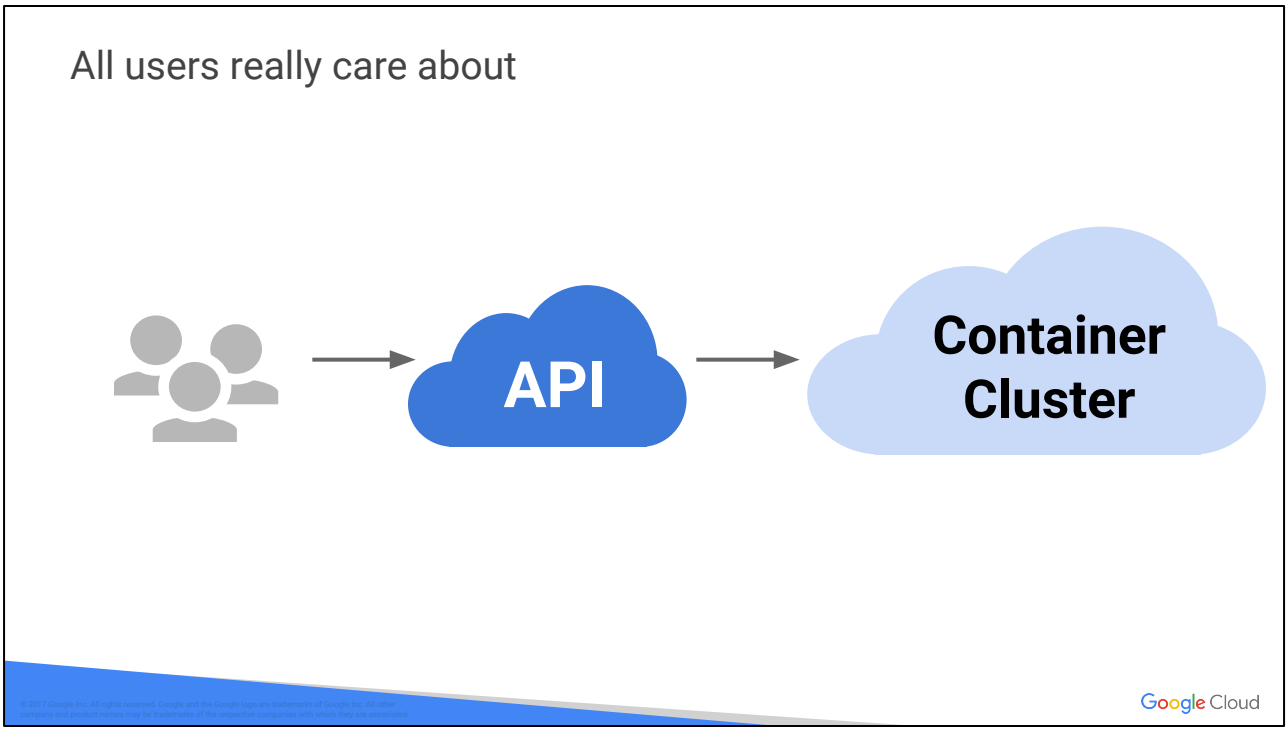


Kubernetes is an open source project (available on kubernetes.io) that can run on many different environments, from laptops to high-availability multi-node clusters, from public clouds to on-premise deployments, from virtual machines to bare metal.

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

The system is divided into a set of master components that run as the control plane and a set of nodes that run containers.

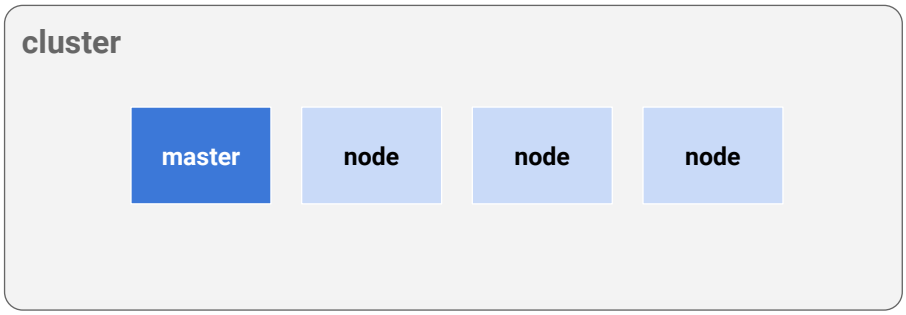
Users access your API via a command-line interface, HTTP, or a user interface.



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 computing instances that Kubernetes manages



Google Cloud

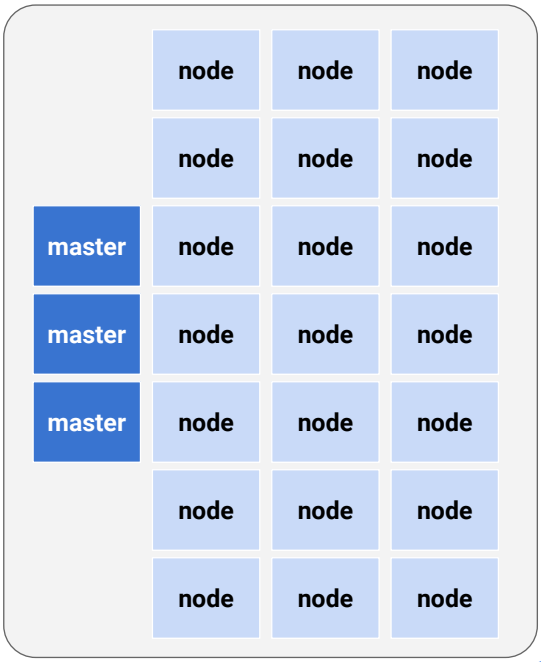
A **cluster** is a set of components (about 5 or 6) running on a set of master nodes and a set of containers running on other nodes. In Kubernetes, a **node** represents a computing instance, like a machine.

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

A cluster can have multiple masters and lots of nodes.

With regional clusters, masters and nodes are spread across three zones in a region for high availability.

By default, three masters are created per zone, but you can control the number.

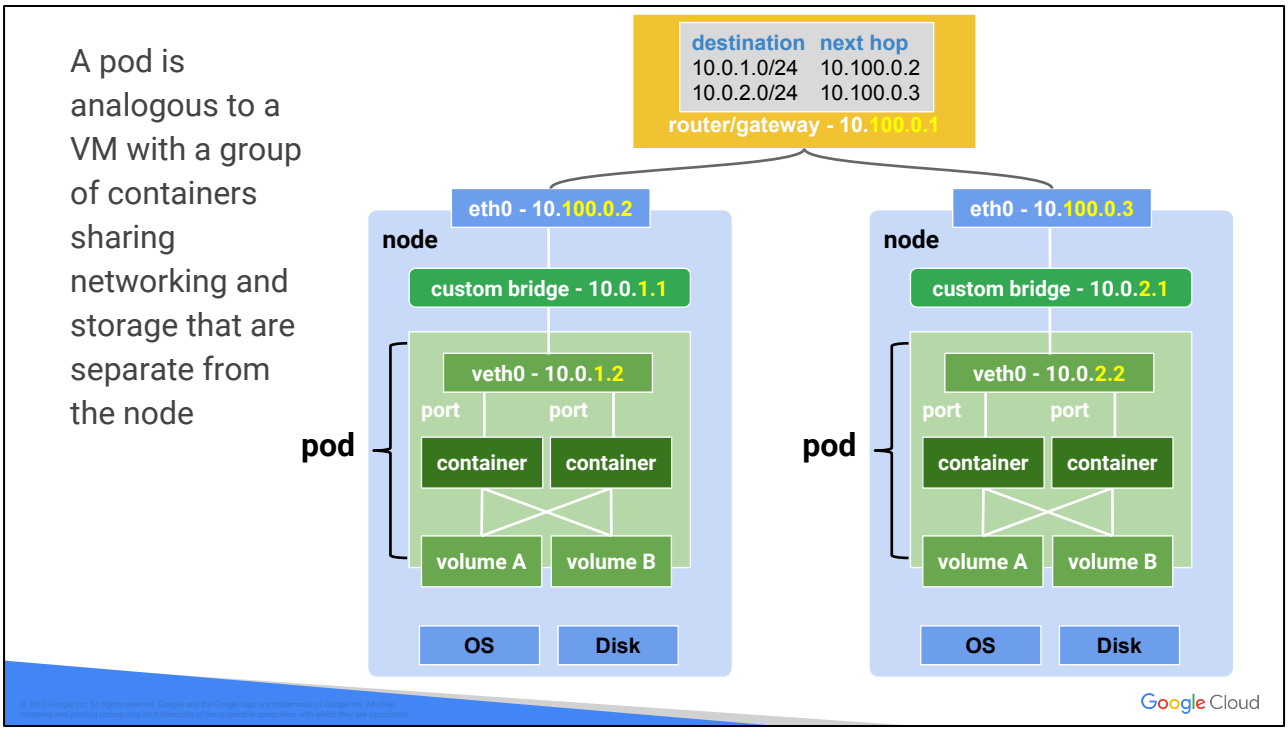


A cluster can have multiple masters so you can have high availability. And you can have up to 5,000 nodes in your cluster for running containers.

With regional clusters, master nodes are highly available. Regional clusters allow you to create a Kubernetes Engine cluster with a multi-master, highly available control plane that helps ensure higher cluster uptime. With regional clusters, you gain resilience from single zone failure and no downtime during master upgrades.

When you create a regional cluster, your masters and nodes are spread across three zones in a region. By default, three nodes are created in each zone (giving you nine total nodes), but you can change the number of nodes in your cluster 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



One of the most critical parts of Kubernetes, and an important concept to understand, is the notion of a pod.

A **pod** is a group of containers deployed as a unit on a node that share networking, namespaces, and storage.

Pods represent a logical piece of an application. Generally, you only have one container per pod, but if you have multiple containers with a hard dependency, you can package them into a single pod.

Networking is handled through a **central routing table** in cluster master nodes and Docker bridges in nodes and pods so containers in one node and pod can find containers in others.

Pods also share a **network namespace** with a single IP address across the pod and a port per container.

Docker and Kubernetes networking are explained very well in the following articles:

- Understanding Kubernetes Networking: Pods
  https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727
- How Docker containers get IP addresses (not in Kubernetes)
  https://stackoverflow.com/questions/43803849/my-docker-container-does-not-have-ip-address-why

Data is stored in **volumes** that reside in memory or disks. Volumes can live as long as pods (in memory) or survive beyond them (in persistent disks) that are then mounted into the pods (not shown). Pods share a **content namespace** that allows them to communicate with each other and share attached volumes.

This is very analogous to the world of VMs and is a very powerful concept for running your containers in production.

You define a pod with a YAML file

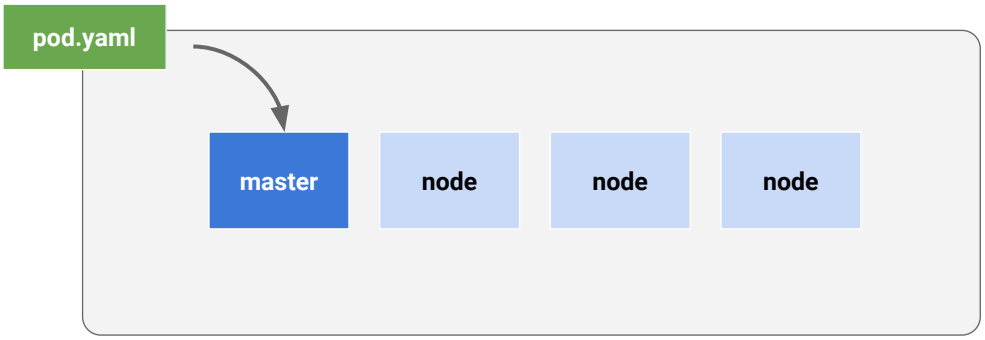
```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name:  my-app
    image: my-app
  - name:  nginx-ssl
    image: nginx
    ports:
    - containerPort: 80
    - containerPort: 443
```


Google Cloud

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.

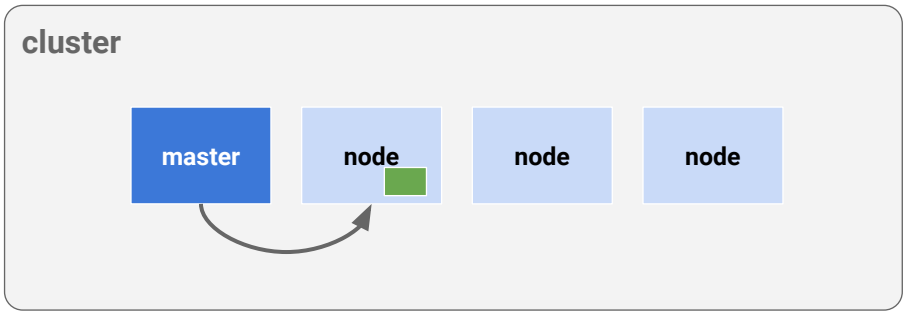
You upload the YAML file to the master



Google Cloud

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

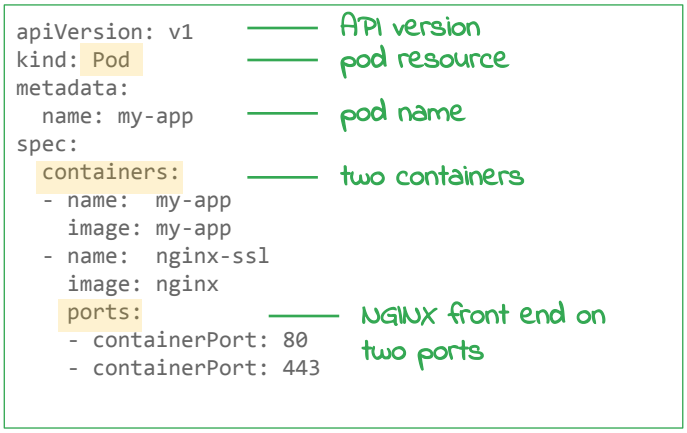
And the master creates a pod on your set of nodes



Google Cloud

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

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



Google Cloud

Going back to the definition, you have:

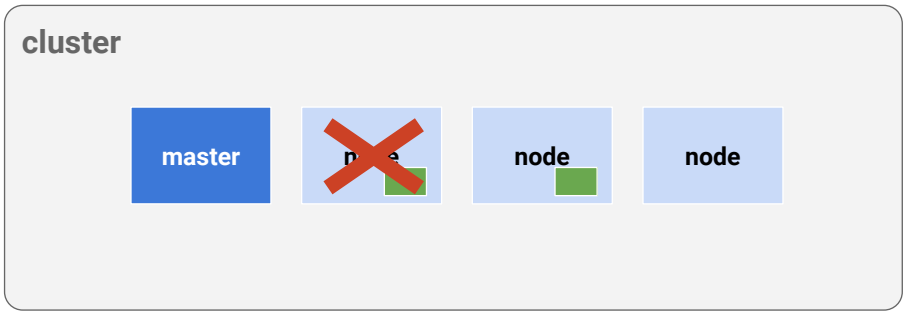
- The API version
- The resource **kind** (in this case a pod)
- Some metadata including the pod name
- A list of containers.

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



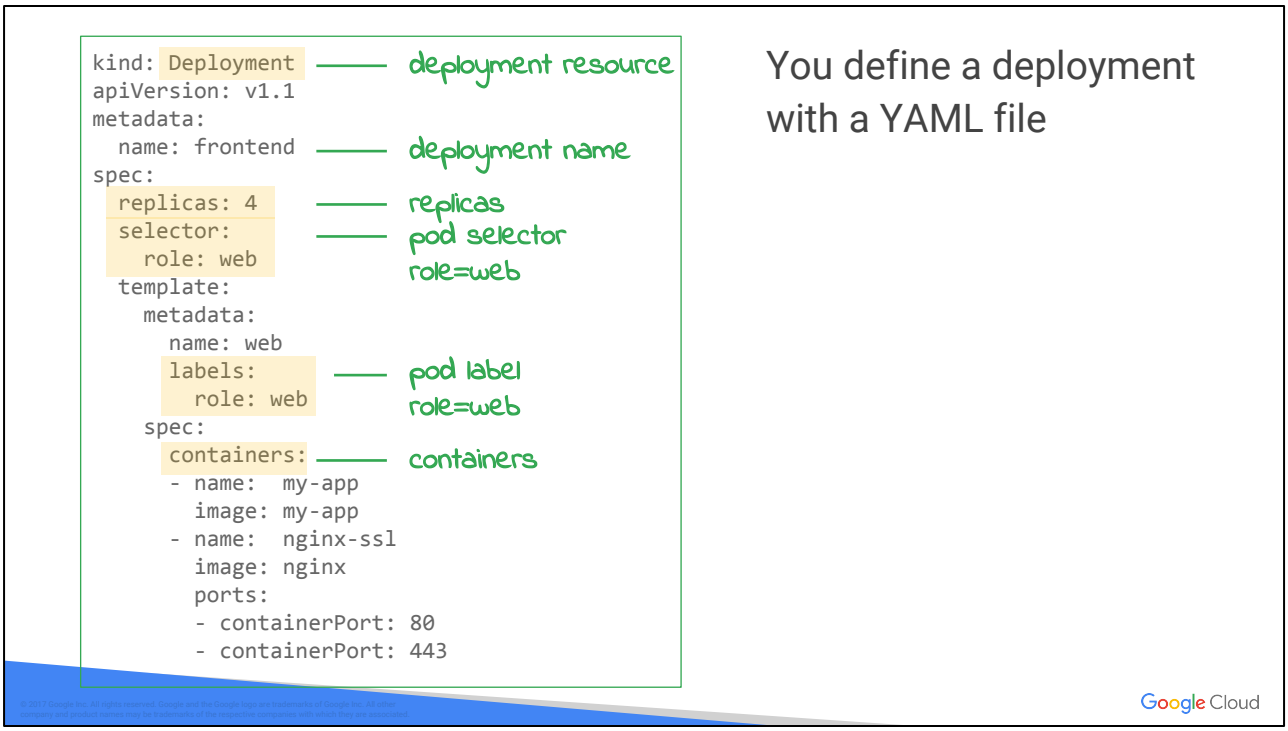
Google Cloud

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.



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

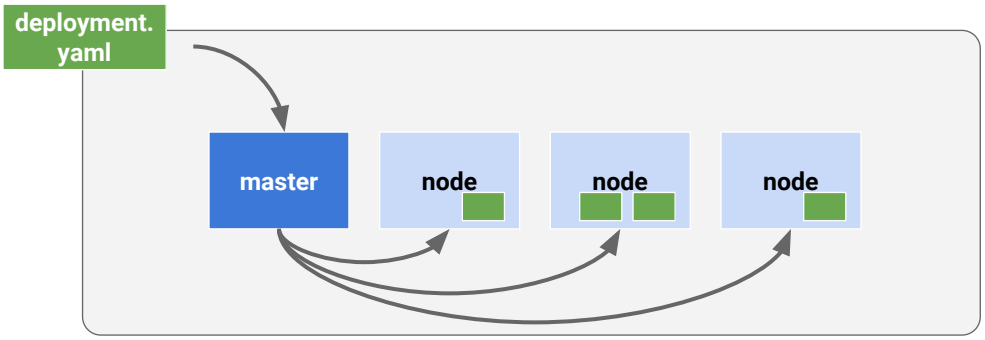
In it, you have:

- 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



Google Cloud

Then you upload the file to the master.

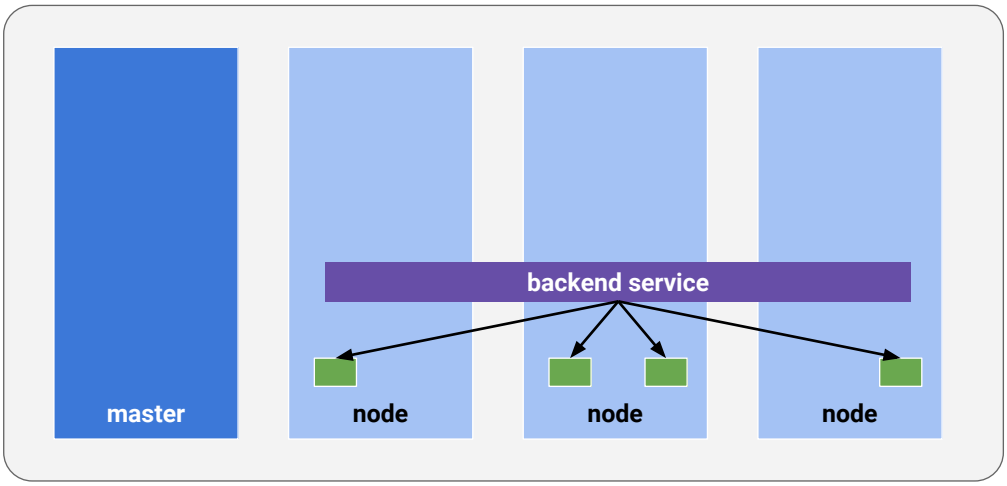
When you do this, the master scheduler decides where to run the pods.

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.

[Duration: 5 mins]

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



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.

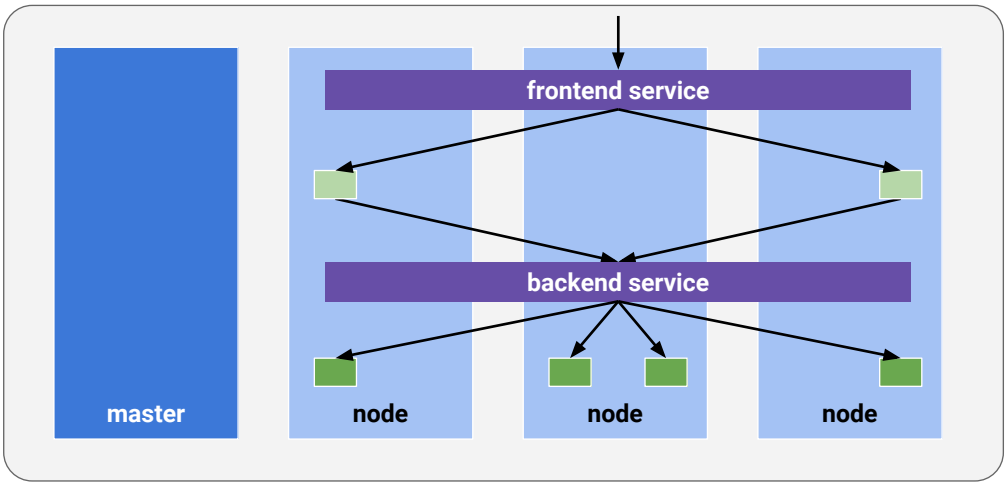
The level of access a service provides to a set of pods depends on the service type. Currently, there are three types:

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

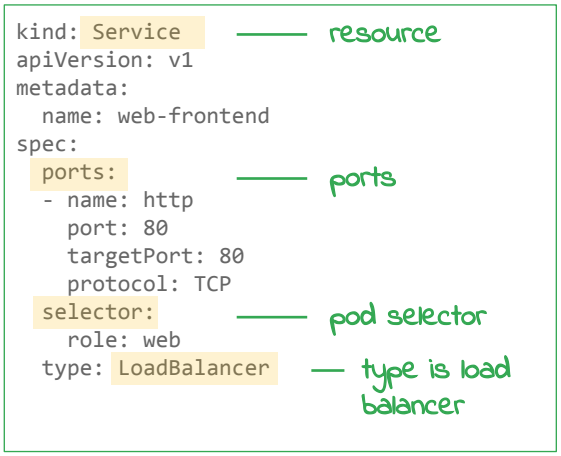


Google Cloud

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



Google Cloud

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

Google Cloud

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

Google Cloud

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).

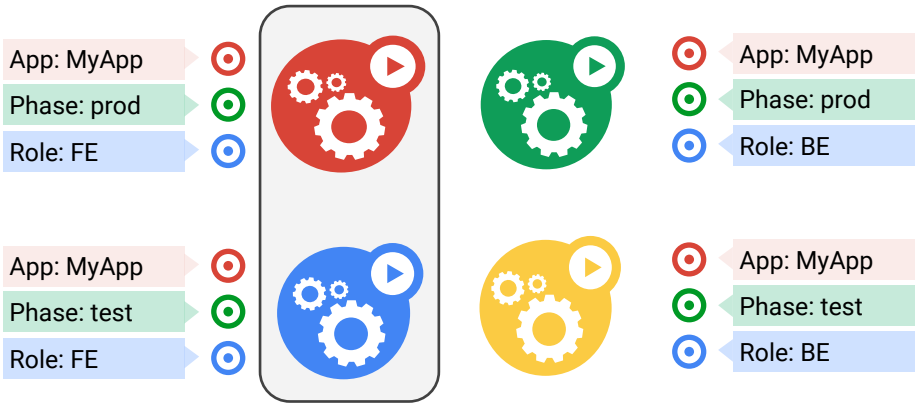
You can query for labels that map to a value like the entire app

App = MyApp

Google Cloud

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

Google Cloud

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

Google Cloud

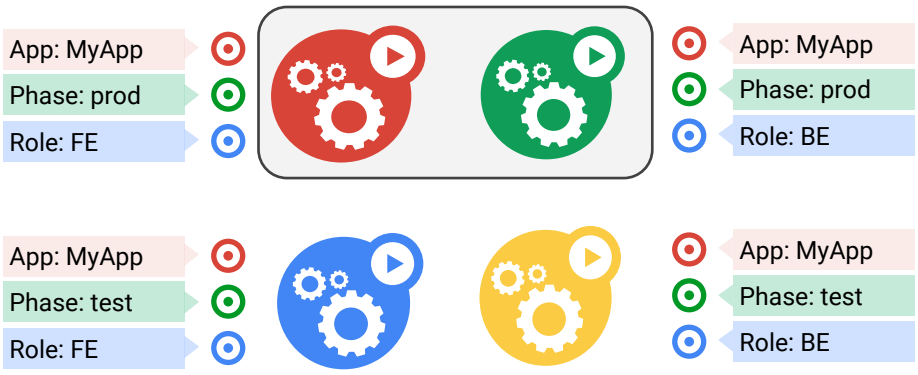
Similarly for the back end.

Or your app's test phase

Google Cloud

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

Or your app's production release



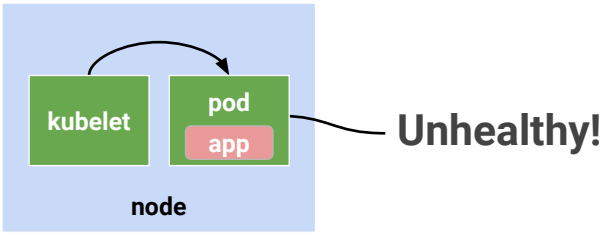
Google Cloud

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...



Google Cloud

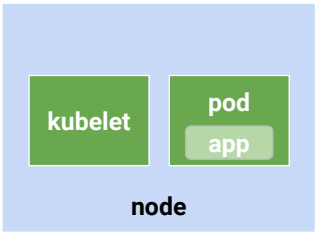
So we've talked about labeling. Let's talk about monitoring and health checks so nodes know they're running healthy pods.

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, ...

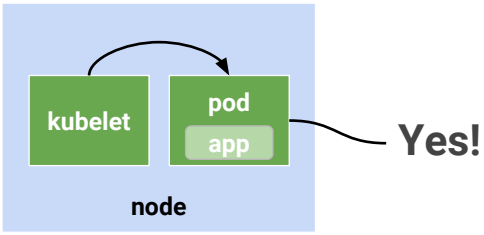
Kubelet restarts the pod



Google Cloud

… the kubelet restarts that pod on the same node.

And continues until it gets a healthy reply



Google Cloud

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.

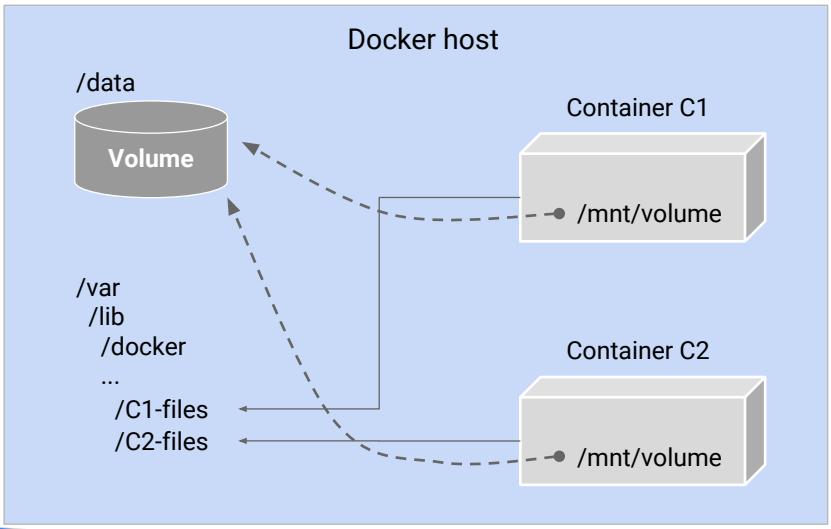
[Duration: 5-6 mins]

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.

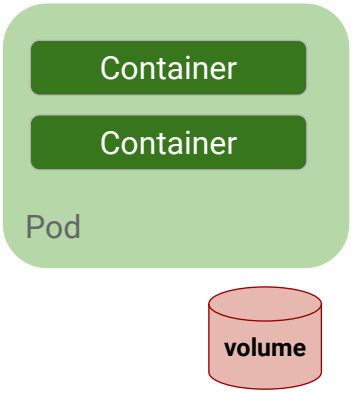
There is also a single shared data volume located at /data on the Docker host. This is mounted directly into host containers. When a container is deleted, any data stored in data volumes persists on the Docker host as long as the host continues to run.

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



Google Cloud

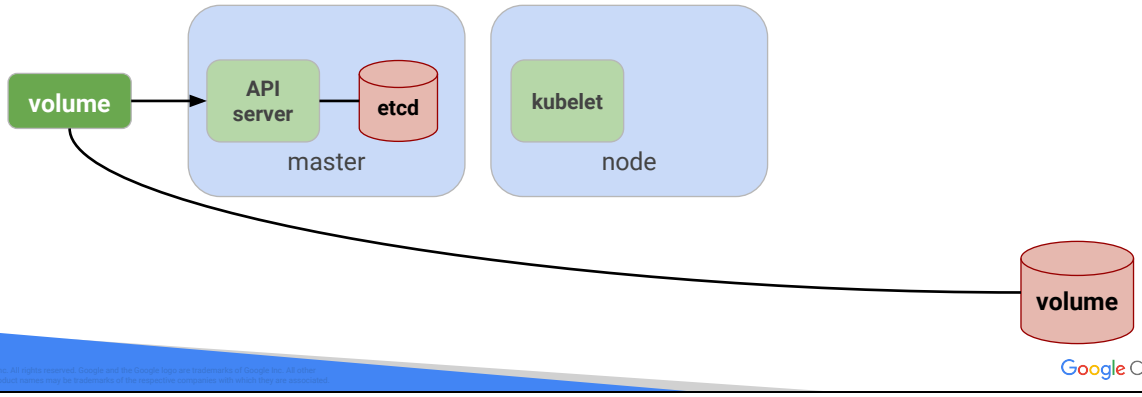
Kubernetes volumes solve both of these problems.

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

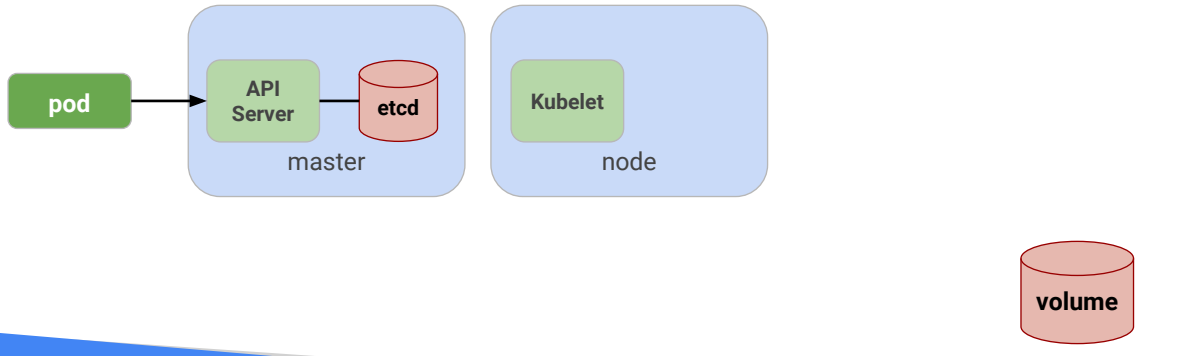
At its core, a volume is just a directory that is accessible to the containers in a pod. How that directory is created, the medium that backs it, and its contents are determined by the particular volume type used.

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.

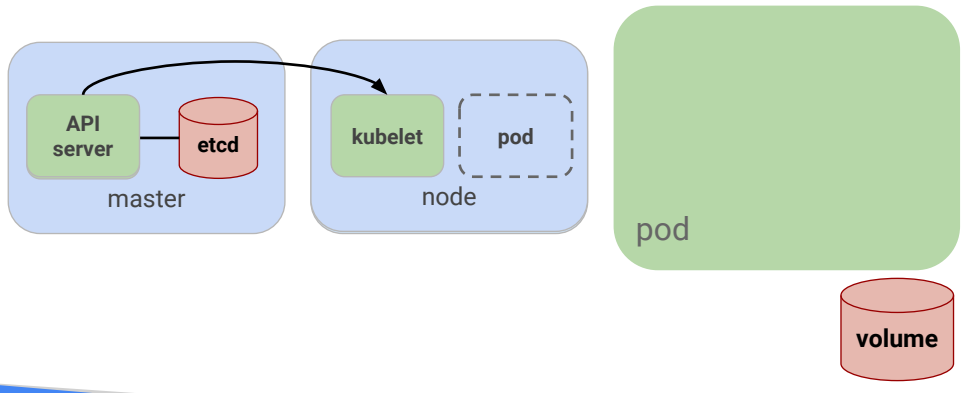
Then you create a pod that consumes that data

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



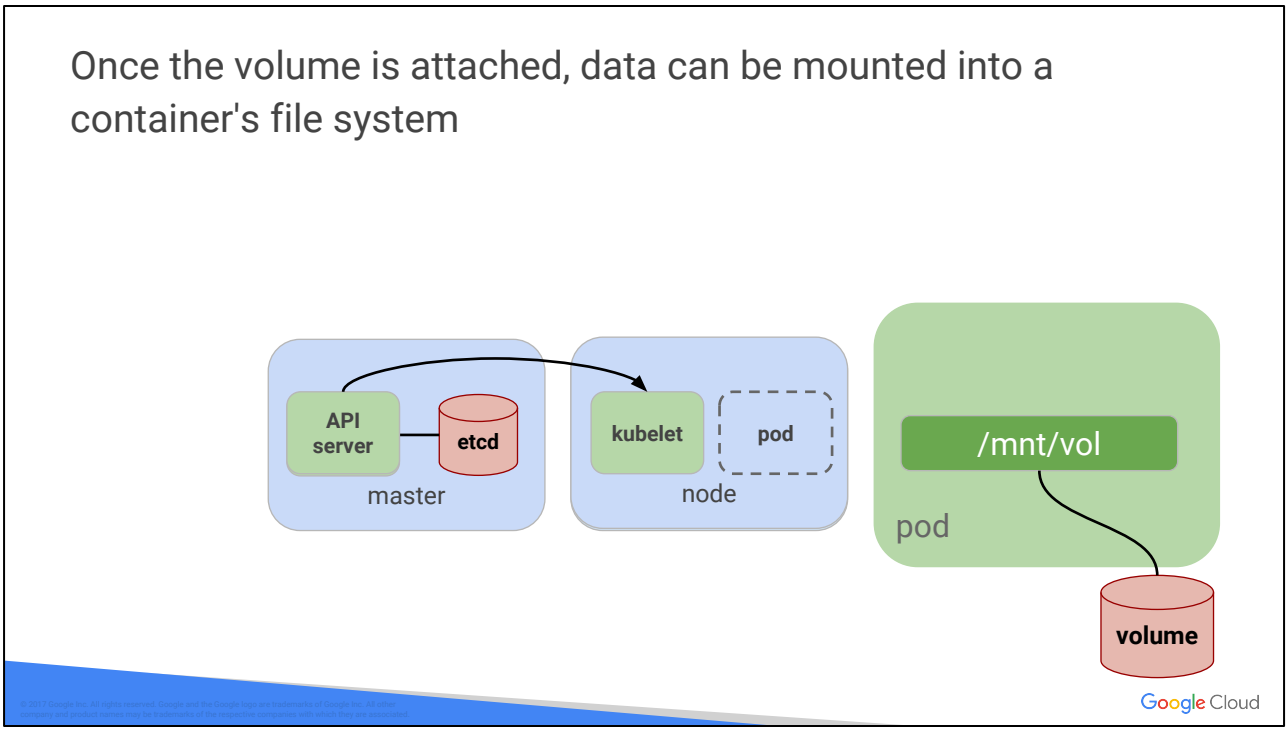
After you have a volume, you create a pod that consumes that data. In this example, that is done with the `kubectl create` command.

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

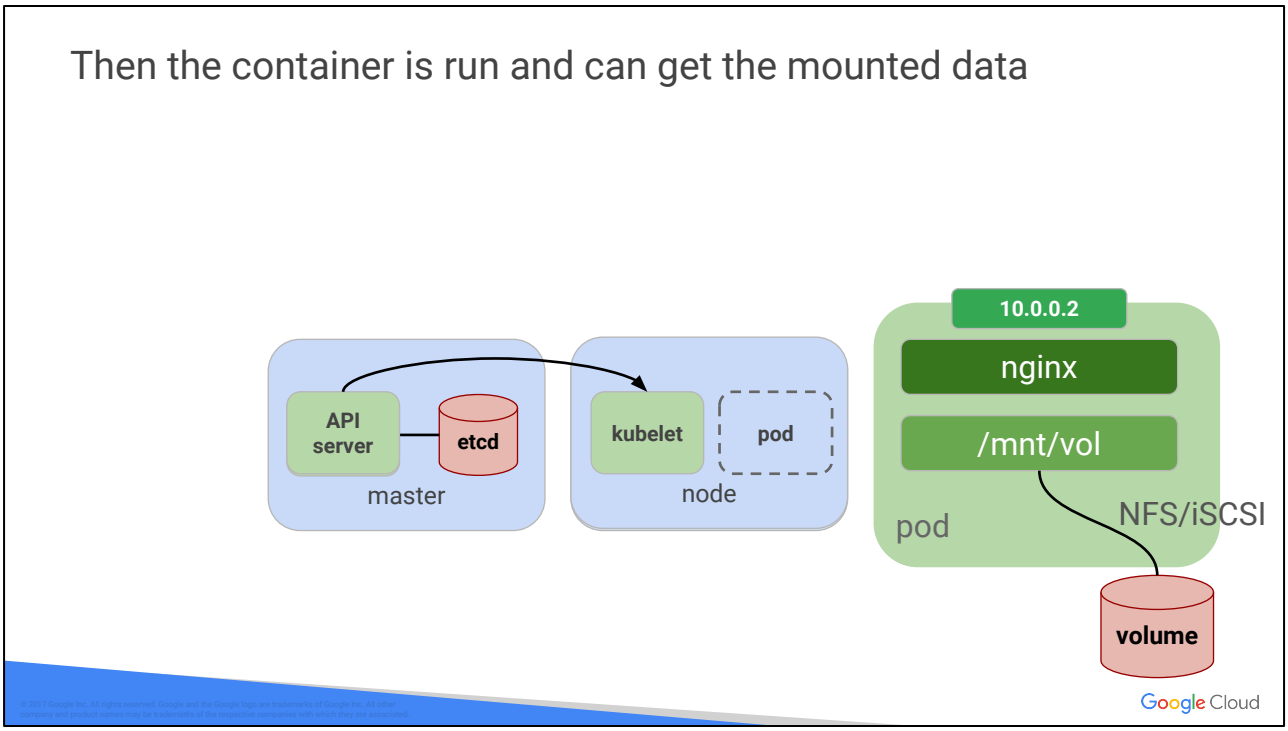


Google Cloud

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, the data in it is mounted into the container’s file system.



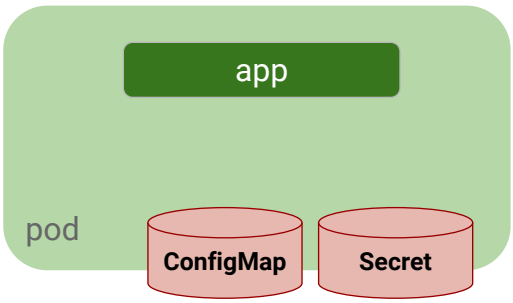
After the data in volumes is mounted, the containers in the pod are brought online and the rest of pod initialization happens as before.

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



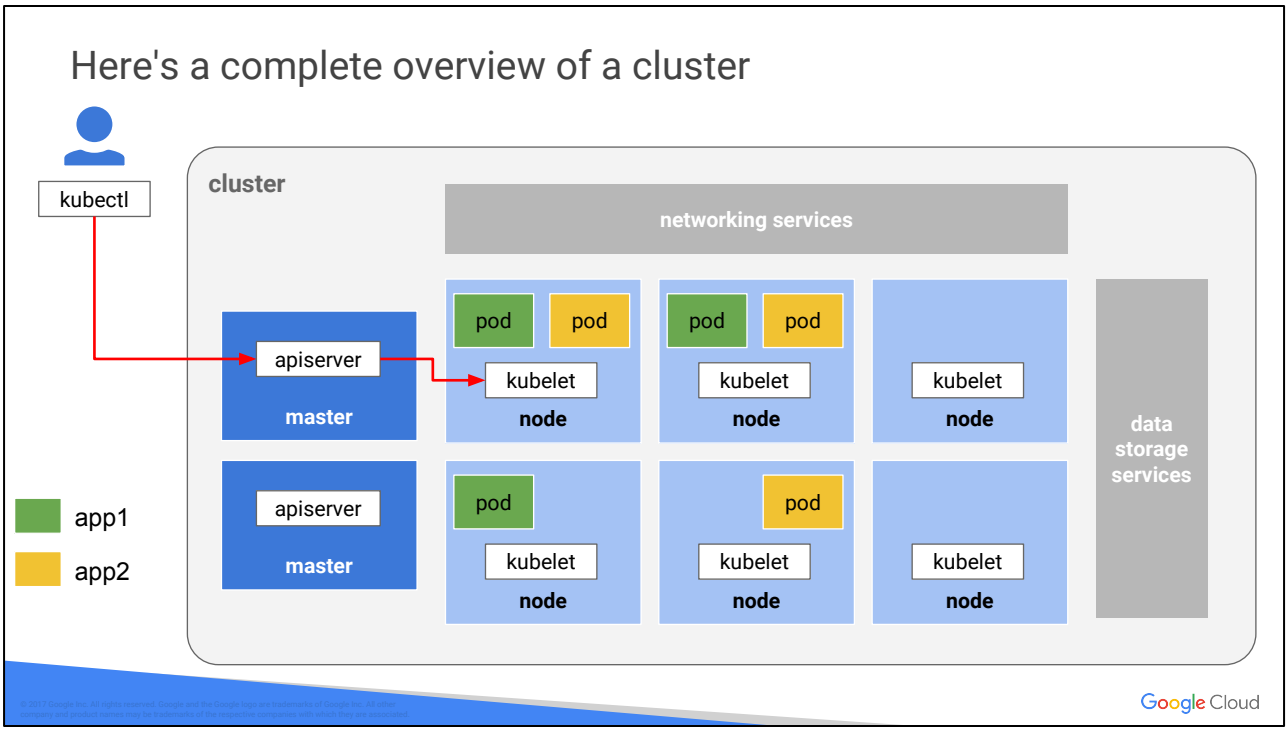
Google Cloud

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 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