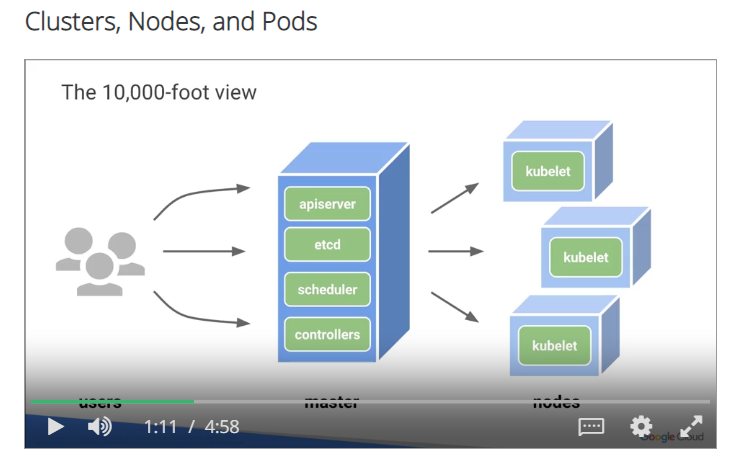
Kubernetes Basics

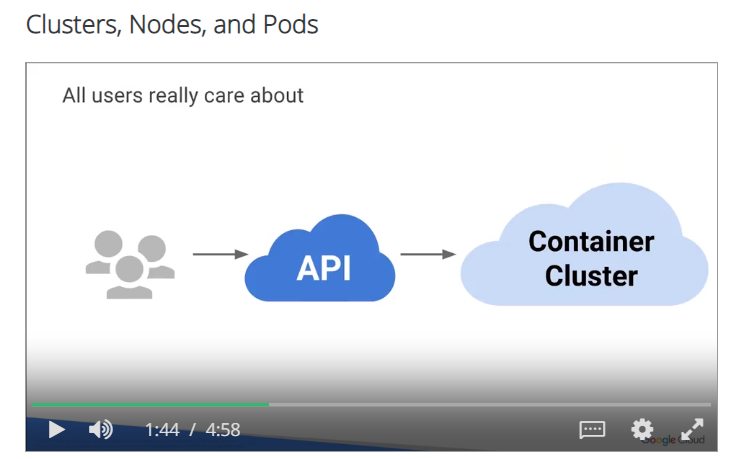
As you saw earlier, containers allow you to break up applications into modules with hardware, software, and operating system requirements. You can run them on the same or even different machines and start and stop them quickly, but you can't specify how many machines or containers to keep running, what to do if they fail, or how to connect them to other containers and persistent storage? For that, you need a container orchestration system like Kubernetes. Kubernetes adds the ability to define how many machines to run, how many containers to deploy, how to scale them, where persistent disks reside, and how to deploy a group of containers as a unit. In this module, you'll learn how to set up and use Kubernetes and container engine to manage a set of machines and containers, and group them into services for hosting real world applications.

# Kubernetes Basics: Clusters, nodes, and pods.

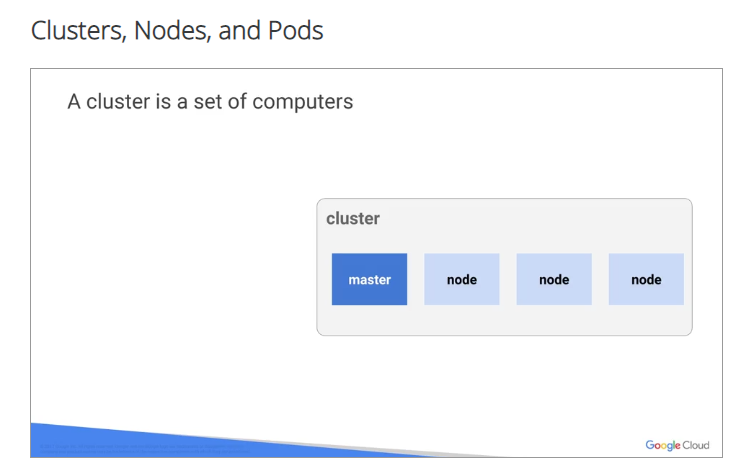
So, let's give you a 10,000-foot view of Kubernetes itself.

You have the users on the left hand side, then you have this master cluster server. Kubernetes and the way it really runs and Dockers and containers, it's a new cluster environment. And then you have all these nodes out there which are running kubelets. The kubelet is a Kubenetes agent and self.

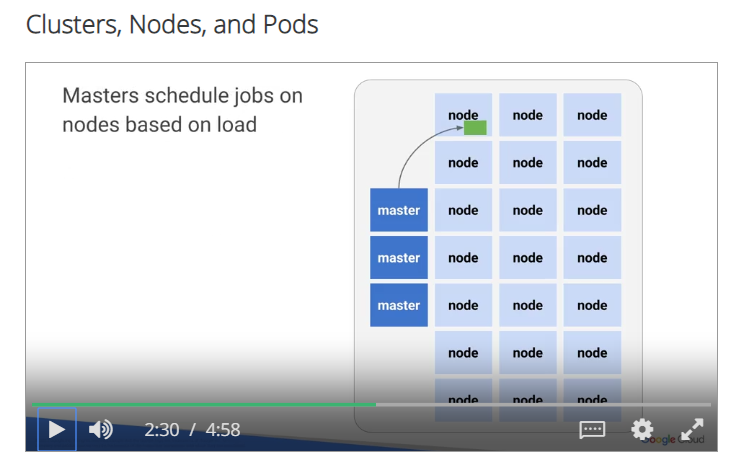
And the master, what it does it controls all the jobs, it controls the scheduling, it controls the etcd, the apiserver itself. Just to be very clear. Kubernetes is an open source orchestrator for a container environment. And what does it really do? What does it provide those developer? So, developers only really care about access to the API. That's all they really care. And Kubernetes and containers provide them that access to that open API so they can program.



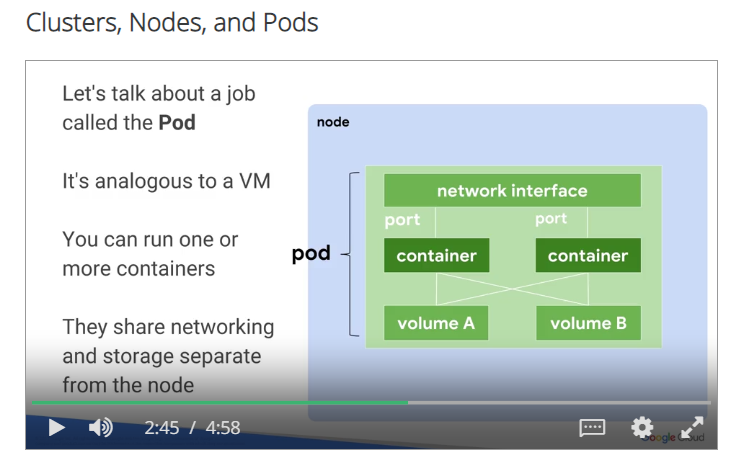
And like I said, a cluster is a set of computers that works is an instance that manages all these nodes.



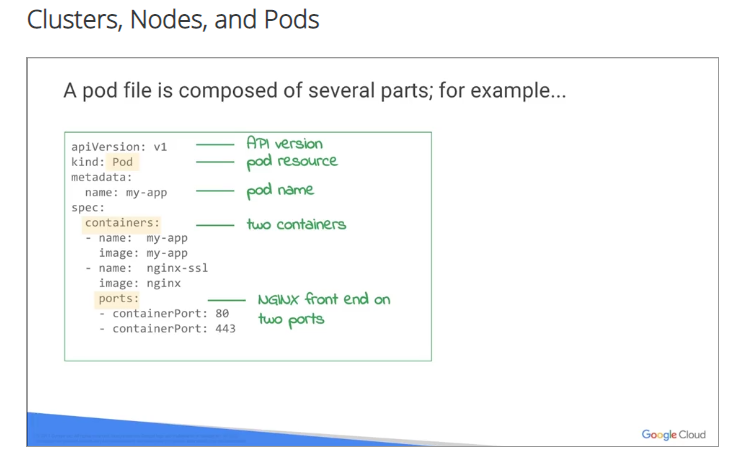
Everything is managed by Kubernetes which is an orchestrator again. And really if you get to the meat of the matter, Kubernetes manages jobs. It knows how to delegate these jobs to these multiple nodes and this is a better example of a real Kubernetes ecosystem. It's not usually one, two, or three nodes, it's thousands of nodes and many multiple masters. But the whole beauty of the system is, it knows how to allocate jobs to these nodes, which are free on resources, or remove the task from nodes, which are high on resources.



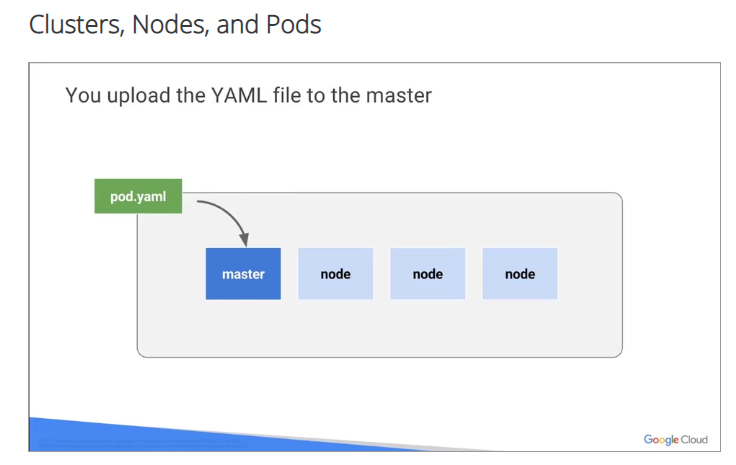
So, let's talk about a pod. A pod is analogous to a VM, sort of, in a group of containers sharing a network and storage that are separated from the nodes themselves. Underneath, you have the OS, you have the hardware, you have the NIC, you have an IP outside, inside you have the pod, and then inside the pod, you have the actual containers, and then the network interfaces for the containers.



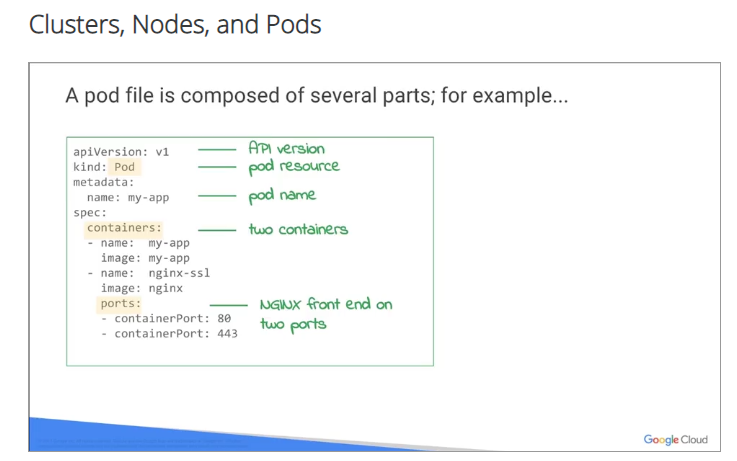
You can define a pod with a YAML file.



Here's an example. It'll give you the version of the YAML file, it gave you the metadata, the name is going to be my-app, the containers, the images, it'll tell you what image it's going to use, and then obviously how you're going to access that containerPort 80 and port 443. You'll upload the YAML file to the master cluster server, and then it will create pods on the nodes that you have dedicated within the YAML file.

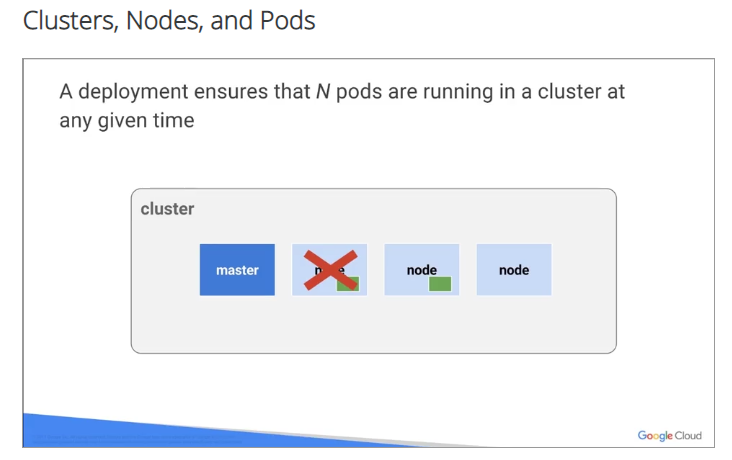


A pod file is composed of several parts. For example, number one, the API version, the pod resource, the pod name, also two containers, and then finally, the front end. What's it going to use? What application will it be running? And then, the ports that application will be running on.

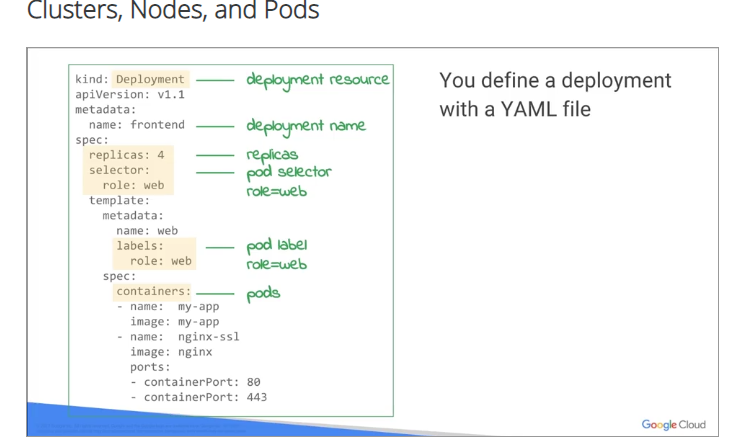


The deployment also ensures that N pods are running in a cluster at any given time.

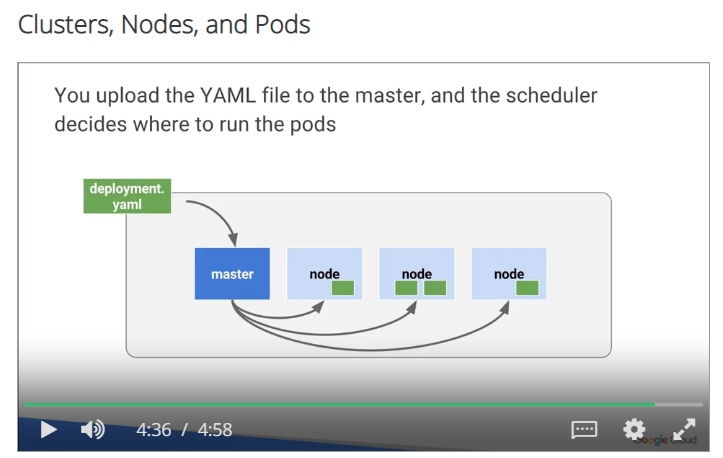
Basically, automatic fail over. If anything happens, if any one of the pod goes down, Kubernetes has the ability to spin up another pod and replace it.



You could define in a YAML file also how many replicas you want. You could identify the pods, the roles, also the labels, and labels are important.



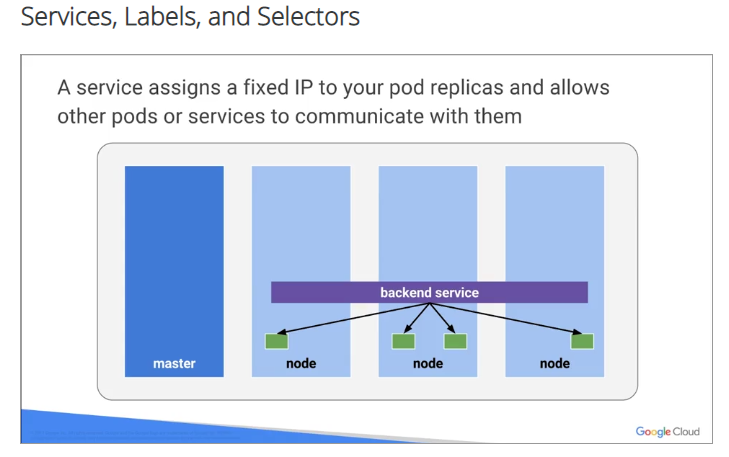
We'll talk about those later on in the course, but also, what is it. Container equals pod. Again, you upload the new YAML file and upload the master, the master then schedules and decides.



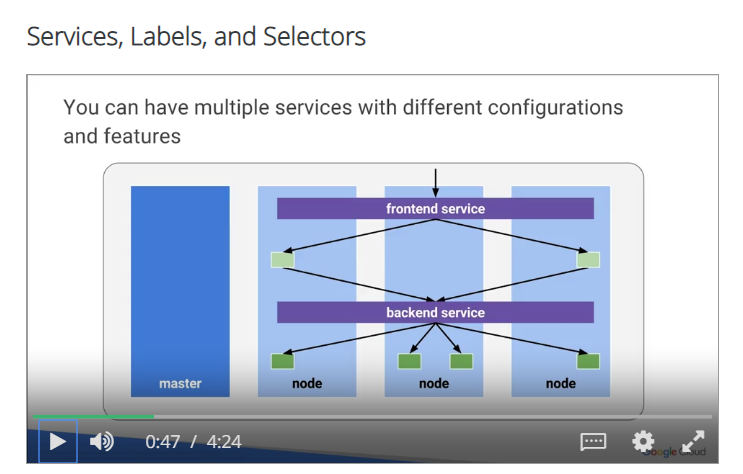
Remember, the master is running Kubernetes. It is deciding these jobs and these tasks when they're going to be run. So, scheduling it depending on the master and depending on the workload and everything else happen, including CPU utilization.

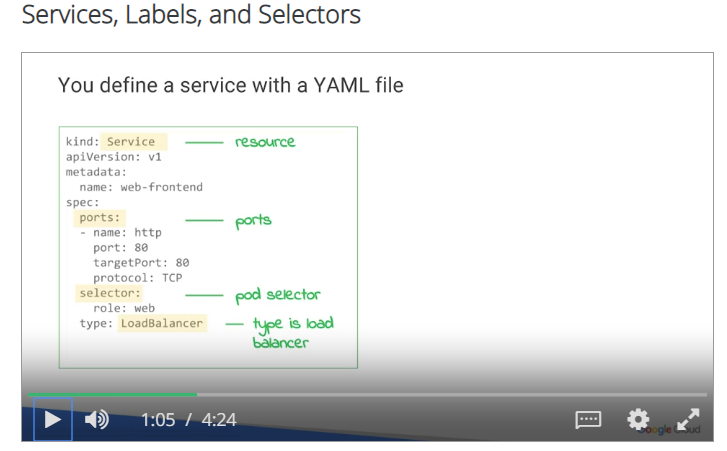
# Kubernetes Basics: Services, Labels, and Selectors.

All right, so a service is assigned to a fixed IP to your pod and then it replicates and allows other pods and other services to communicate with them.

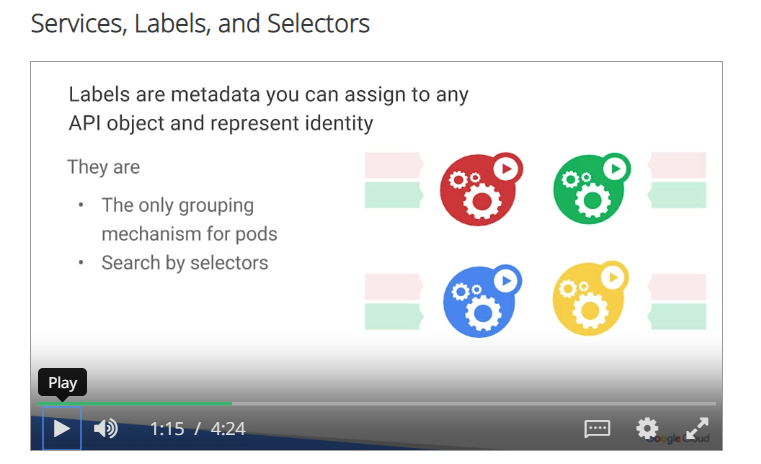


Services really act like a way of communication between the pods, okay? Number one, the cluster has an IP, an internal IP, and the port nodes also have awareness of that access to that IP. It also uses a load balancer as a bound for that browsing that traffic that comes into the nodes themselves. You're going to have multiple services with different configurations and features running.

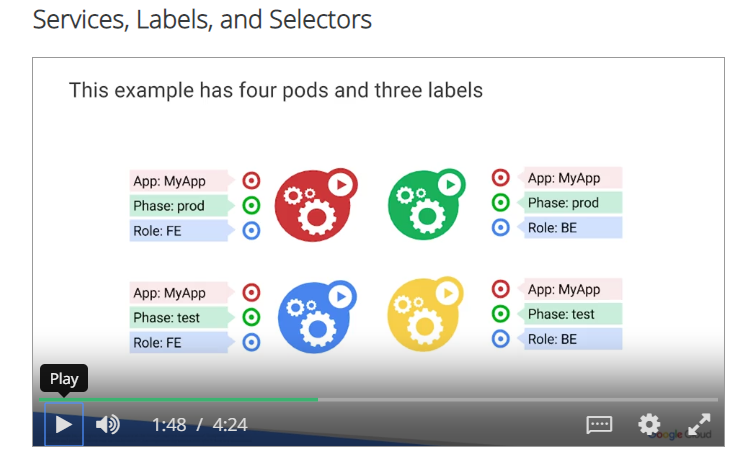
 Usually, the front end is a web server, application server, some kind of server, back ends are usually the nodes and maybe a database server. They run with usually a load balancer, Google Cloud load balancer, which will let you detailed what GCP API you use.

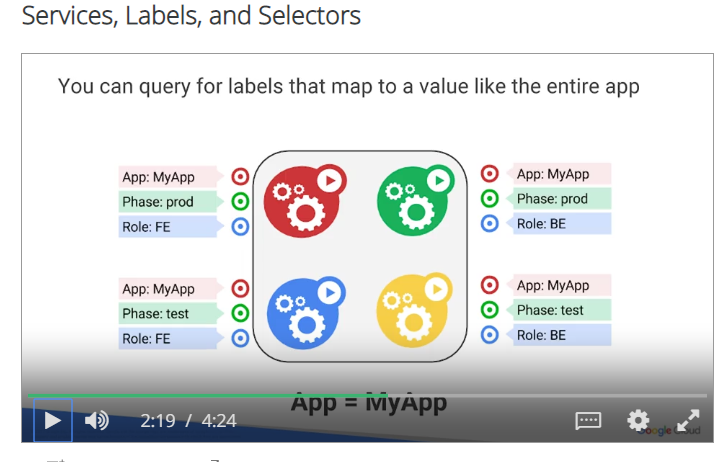


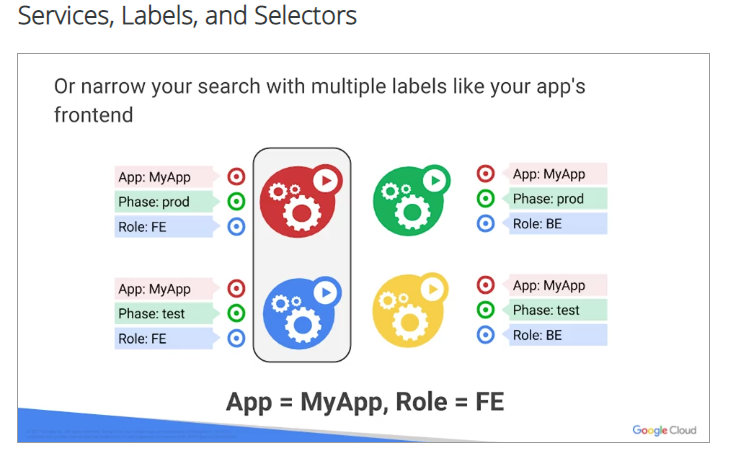
Again, a definition in YAML file will tell you the resources, the pods you'll be using, and then the **pod selector**, and then the type of load balancer you'll be applying to this.

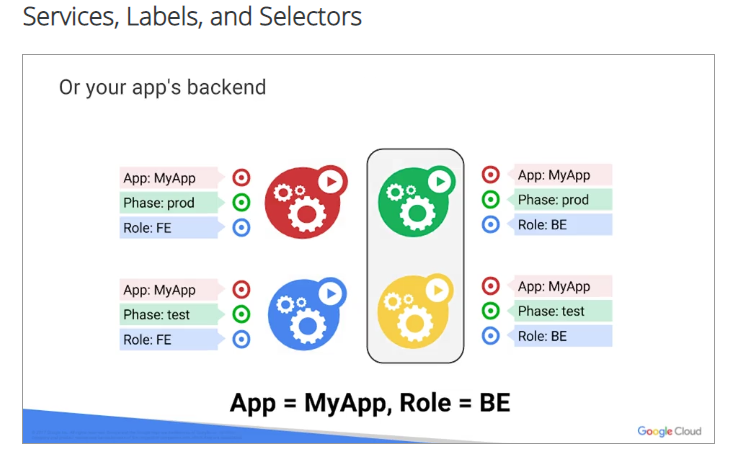


Labels are metadata that you can assign to any API object and represent an identity. They are used for grouping, mechanisms, searching, but not only that, they're using pre-filtering, finding something. Pods could contain a lot of entities and containers that could be in many, many containers in your environment. By labeling them logically, you'll be able to quickly find them and not only that, per mechanisms inside the pods to quickly identify them.

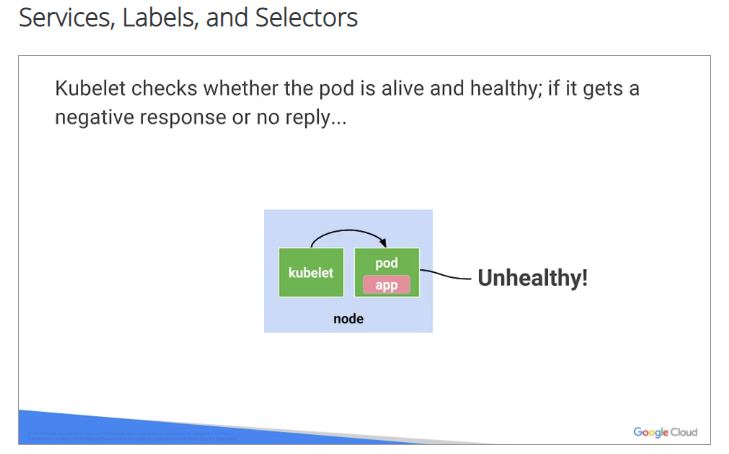




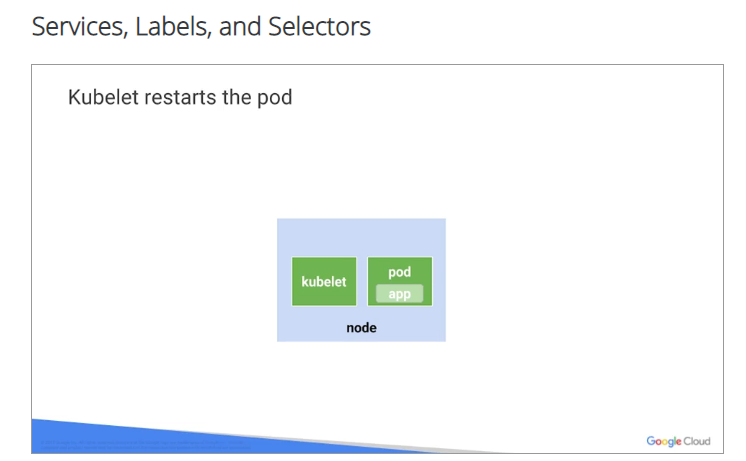




In this example, you have four pods and three labels, so my application, my application, my application, my application, but notice the labels, one is prod, one is test, there's another for tests, there's another prod, and then you have different roles underneath each. That way, not only is it a way of segregating for reporting, but it's also a way of mechanizing and orchestrating the container environment itself. You could query, you could find labels, like I said, when you need to find something in 9,000 containers, it's going to be very handy to have a label. And you could also use it to map the entire application to that label itself. So, app label equals all my applications. So an easy way to again group all your applications. Or narrow your search, like I said. Trying to find a needle in the haystack of a thousand containers could be very challenging. With tags and labels, it could be made very, very easy. You'll be able to similarly apply labels for your back end services also. This is my BE server, this is my back end server, my database, whatever. Or, your applications test phase. Call it test. Here you go. Both of them are test, you could group them together, and you could have a red and blue or green, yellow application. We'll talk about those later on. But it's a way again of testing your application in an environment without disrupting anything in production again. And then on your production release, like I said, make the same changes. You could label these, and understand that labeling is just an example right here that we're showing you, but it's really an example and it's really a logical way of managing Kubernetes.



Kubernetes also checks whether your pod is alive or healthy. If it gets a negative response, or no reply, it's unhealthy. In other words, if I'm knocking on your door and you don't open that door, I'm going to call the police, have not come on in and probably maybe rescue, because I know there's something wrong. Kubernetes has that unhealthy environment already built in. I call it, a.k.a. Doctor in a can. Kubernetes then automatically restarts the pod if it needs restarts to see that, if everything comes up bright.

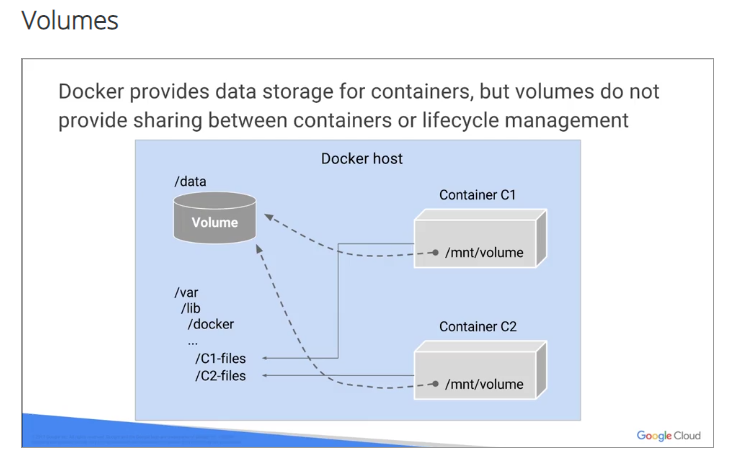




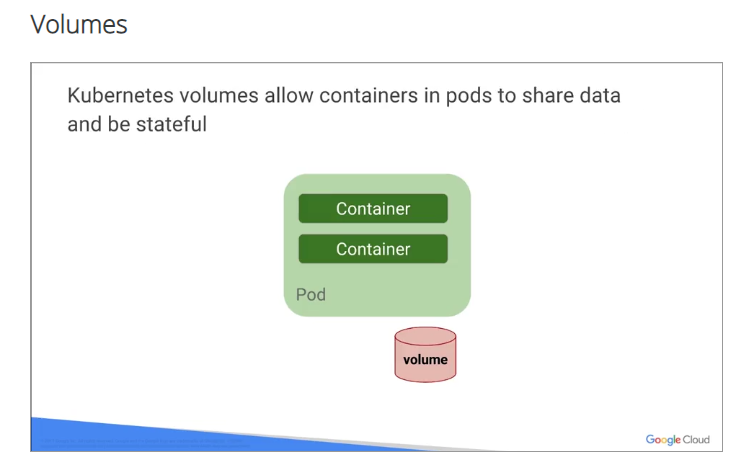
And if everything is good, it goes back into a healthy state. Doctors not needed any longer, pod is up and running.

# Kubernetes basics, part3: volumes

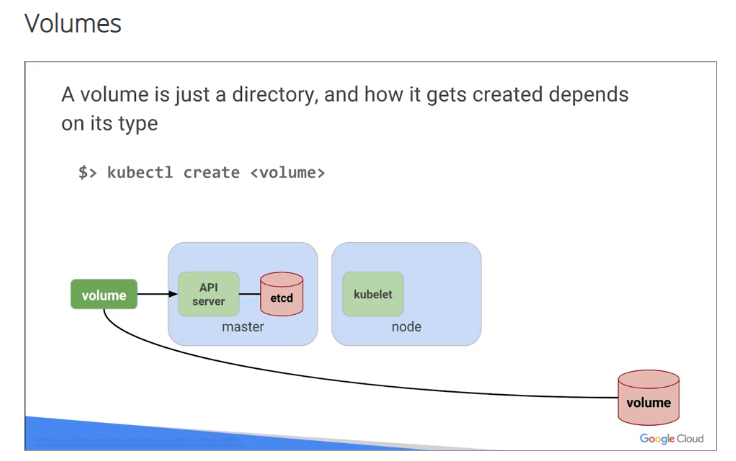
So Docker provides data storage for containers, right?



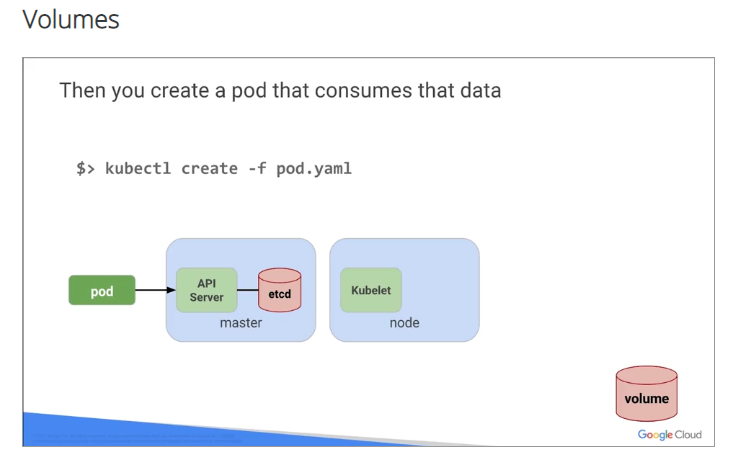
You've got to put your data somewhere, you have to have storage. But volumes do not provide shared between all containers, or a lifecycle management either. It's very important, but lifecycle management in data is an important topic that we all talk about these days. Anyway so, dockers now provides volume driven, okay? Not shown in the slide here but volume-driven host and remaining persistent volumes. And we'll show you those in the coming slides.

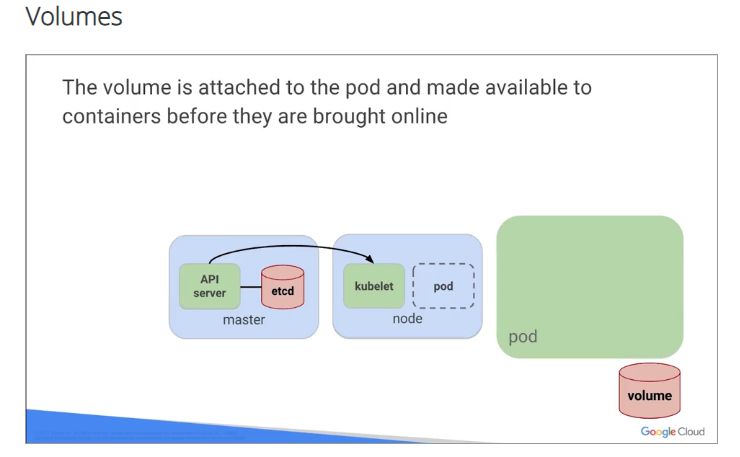


Kubernetes, volumes allow containers in part to share data, and to be stateful. A volume is just a directory. And how it gets created depends on the type.

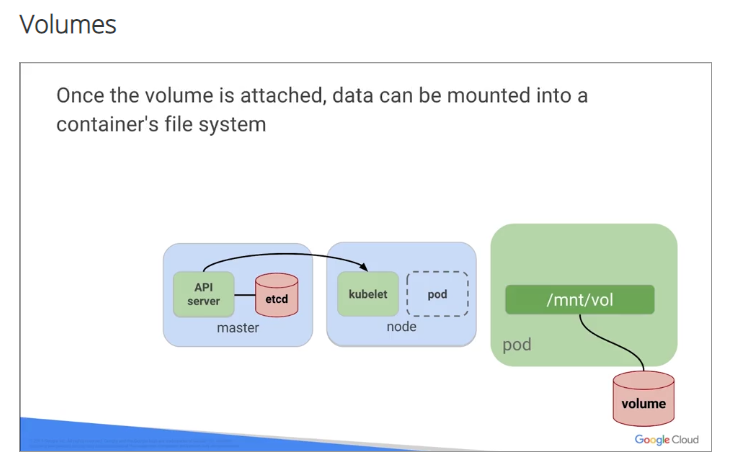


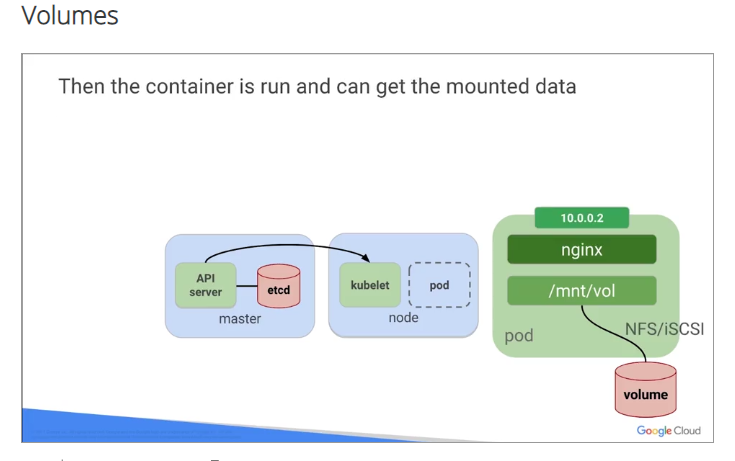
Here is an example, kubectl Create volume, and that's all, you run that out of Kubernetes line itself command line, and it'll create a volume.



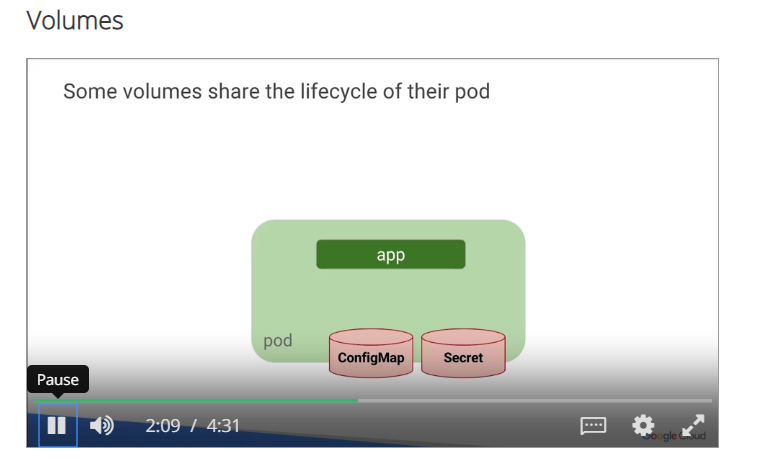


Here's another one that will create a volume that consumes that data, so again kubectl create -f pod, which is the YAML file created depending on the YAML description. The volume is attached to the pod, and made available to the containers before they are brought online. So you have that storage there before the containers are brought online. We want to be able to have storage available in case it already has to purchase something, or has to relocate something, or put something in that container as it comes up, or during the process of it coming up maybe logs.

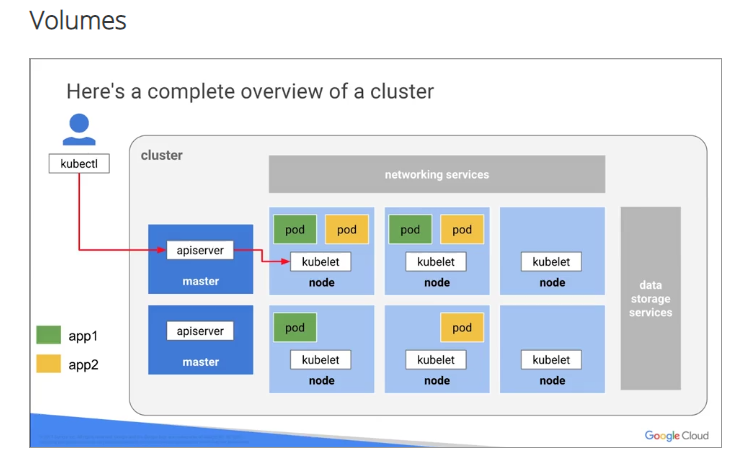




Once the volume is attach, and you could see here mnt/vol, which is the Unix equivalent mounting a volume, the data can be mounted into that container file system. Then the container is run, and it can get the mounted data from that volume itself. And some volumes can be shared. The lifecycle of the pod itself, in other words they'll stay around as long as that pod is around. And see you have here also on the bottom the configuration map and the secret map.



Let's talk a little bit about the secret stores and all the sensitive information, the encryption information, the passwords, the keys, stores a lot of information in there, and the config maps are basically stubby differential maps, they are used for sharing string data, strong configuration, making sure everything and all the pieces are put together and are working basically at the ConfigMap.



Here's a complete overview of a cluster. So you have kubectl, that's a guy sitting in the blue guy at the left-hand side, there's the Blue Man Group. Okay? Have a couple applications, application one and application two. Then you have the app server, it could be anything, okay? It could be a web server. Okay? And then you have the networking services on the top, and as you go towards the right, you'll see the different nodes with the pods inside. So you'll have a pod with a kubelet in it, running a green and yellow, which are two different applications inside one node. Okay? This is what we're talking about, these are the advantages Kubernetes. Again same thing over and over again. You could also have nodes that are running no pods at all, being a standby or just hanging there if you need them. Okay? As a backup. And you could have pods like on the bottom, which are running on a single node. So it could be multiple or it could be one for that matter. At the end, far right, you finally have your data storage services where everything goes. All your data is stored.

## Lab

Now it's time to use Google container engine and Kubernetes to deploy, manage, and update a sample application. It's a modern web-based application called a 12-Factor app. That's well-suited for being built using containers. It includes modules for user authentication, back-end data retrieval, and a web front-end. It's based on several docker images, one that includes authentication and greeting services, two micro services and an Nginx web front-end. You'll provision it's resources such as pods, services, and volumes, and split them into micro services, and scale and test them.

# Kubernetes Basics v1.6

3 hours1 Credit

Rate Lab

## Overview

In this lab, you learn how to:

* Provision a [Kubernetes](http://kubernetes.io/) cluster using [Google Kubernetes Engine.](https://cloud.google.com/container-engine)
* Deploy and manage Docker containers using kubectl.
* Split an application into microservices using Kubernetes' Deployments and Services.

You use Kubernetes Engine and its Kubernetes API to deploy, manage, and upgrade applications. You use an example application called "app" to complete the labs.

[App](https://github.com/kelseyhightower/app) is hosted on GitHub. It's a 12-Factor application with the following Docker images:

* **Monolith**: includes auth and hello services.
* **Auth** microservice: generates JWT tokens for authenticated users.
* **Hello** microservice: greets authenticated users.
* [nginx](https://hub.docker.com/_/nginx): frontend to the auth and hello services.

## Setup

#### ****Step 1****

#### What you'll need

To complete this lab, you’ll need:

* Access to a standard internet browser (Chrome browser recommended).
* Time. Note the lab’s **Completion** time in Qwiklabs. This is an estimate of the time it should take to complete all steps. Plan your schedule so you have time to complete the lab. Once you start the lab, you will not be able to pause and return later (you begin at step 1 every time you start a lab).
* The lab's **Access** time is how long your lab resources will be available. If you finish your lab with access time still available, you will be able to explore the Google Cloud Platform or work on any section of the lab that was marked "if you have time". Once the Access time runs out, your lab will end and all resources will terminate.
* You **DO NOT** need a Google Cloud Platform account or project. An account, project and associated resources are provided to you as part of this lab.
* If you already have your own GCP account, make sure you do not use it for this lab.
* If your lab prompts you to log into the console, **use only the student account provided to you by the lab**. This prevents you from incurring charges for lab activities in your personal GCP account.

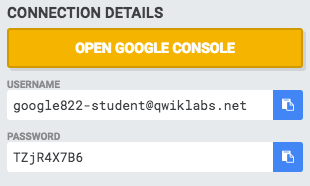
#### Start your lab

When you are ready, click **Start Lab**. You can track your lab’s progress with the status bar at the top of your screen.

**Important** What is happening during this time? Your lab is spinning up GCP resources for you behind the scenes, including an account, a project, resources within the project, and permission for you to control the resources needed to run the lab. This means that instead of spending time manually setting up a project and building resources from scratch as part of your lab, you can begin learning more quickly.

#### Find Your Lab’s GCP Username and Password

To access the resources and console for this lab, locate the Connection Details panel in Qwiklabs. Here you will find the account ID and password for the account you will use to log in to the Google Cloud Platform:



If your lab provides other resource identifiers or connection-related information, it will appear on this panel as well.

#### ****Step 2****

Make sure the following APIs are enabled in Cloud Platform Console:

* Kubernetes Engine API
* Container Registry API

On the **Navigation menu** (Navigation menu), click **APIs & services**.

Scroll down and confirm that your APIs are enabled.

If an API is missing, click **ENABLE APIS AND SERVICES** at the top, search for the API by name, and enable it for your project.

#### ****Step 3****

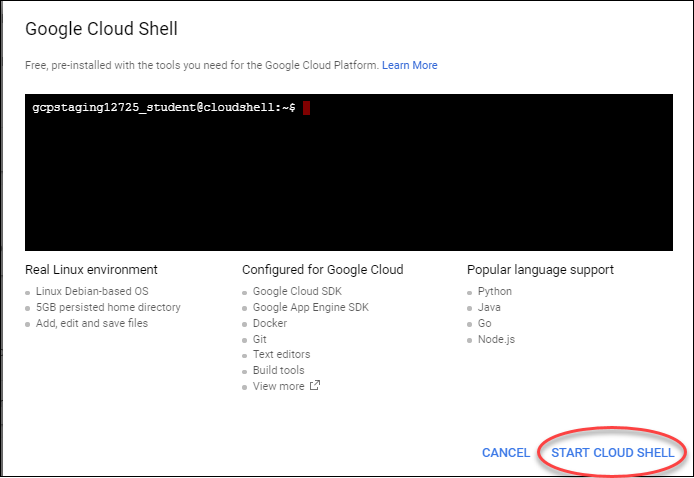
### Activate Google Cloud Shell

Google Cloud Shell provides command-line access to your GCP resources.

From the GCP Console click the **Cloud Shell** icon on the top right toolbar:

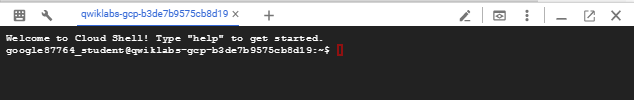


Then click **START CLOUD SHELL**:



You can click **START CLOUD SHELL** immediately when the dialog comes up instead of waiting in the dialog until the Cloud Shell provisions.

It takes a few moments to provision and connects to the environment:



The Cloud Shell is a virtual machine loaded with all the development tools you’ll need. It offers a persistent 5GB home directory, and runs on the Google Cloud, greatly enhancing network performance and authentication.

Once connected to the cloud shell, you'll see that you are already authenticated and the project is set to your PROJECT\_ID:

gcloud auth list

Output:

Credentialed accounts:

- <myaccount>@<mydomain>.com (active)

**Note:** gcloud is the powerful and unified command-line tool for Google Cloud Platform. Full documentation is available on [Google Cloud gcloud Overview](https://cloud.google.com/sdk/gcloud). It comes pre-installed on Cloud Shell and supports tab-completion.

gcloud config list project

Output:

[core]

project = <PROJECT\_ID>

#### ****Step 4****

Get the sample code from the Git repository.

git clone https://github.com/googlecodelabs/orchestrate-with-kubernetes.git

#### ****Step 5****

Review the app layout.

cd orchestrate-with-kubernetes/kubernetes

ls

You'll see the following structure.

|  |  |
| --- | --- |
| **deployments/** | **Deployment manifests** |
| **nginx/** | **nginx config files** |
| **pods/** | **pod manifests** |
| **services/** | **Services manifests** |
| **tls/** | **TLS certificates** |
| **cleanup.sh** | **Cleanup script** |

Now that you have the code, it's time to try Kubernetes.

## A quick demo of Kubernetes

### ****Start a Kubernetes cluster****

#### ****Step 1****

Define your zone as a project default zone. This way you do not need to specify --zone parameter in gcloud commands.

gcloud config set compute/zone us-central1-a

In Cloud Shell, run the following command to start a Kubernetes cluster called bootcamp that runs 5 nodes.

gcloud container clusters create bootcamp --num-nodes 5 --scopes "https://www.googleapis.com/auth/projecthosting,storage-rw"

The scopes argument provides access to project hosting and Google Cloud Storage APIs that you'll use later.

It takes several minutes to create a cluster as Kubernetes Engine provisions virtual machines for you. It spins up one or more master nodes and multiple configured worker nodes. This is one of the advantages of a managed service.

Click Check my progress to verify the objective.

Create a Kubernetes cluster

Check my progress

#### ****Step 2****

After the cluster is created, check your installed version of Kubernetes using the kubectl version command.

kubectl version

The gcloud container clusters create command automatically authenticated kubectl for you.

#### ****Step 3****

Use kubectl cluster-info to find out more about the cluster.

kubectl cluster-info

|  |
| --- |
| google3570462\_student@cloudshell:**~/orchestrate-with-kubernetes/kubernetes (qwiklabs-gcp-4d10735f143f84d4)**$ kubectl cluster-info  Kubernetes master is running at [https://35.202.122.195](https://35.202.122.195/) GLBCDefaultBackend is running at <https://35.202.122.195/api/v1/namespaces/kube-system/services/default-http-backend:http/proxy>  Heapster is running at <https://35.202.122.195/api/v1/namespaces/kube-system/services/heapster/proxy>  KubeDNS is running at <https://35.202.122.195/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy>  Metrics-server is running at <https://35.202.122.195/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy> |
|  |

#### ****Step 4****

View your running nodes in Cloud Platform Console.

Open the **Navigation menu** and go to **Compute Engine > VM Instances**.

Congratulations! Your Kubernetes cluster is now ready for use!

### ****Bash Completion (Optional)****

Kubernetes comes with auto-completion. You can use the [kubectl completion](https://kubernetes.io/docs/user-guide/kubectl/kubectl_completion/) command and the built-in source command to set this up.

#### ****Step 1****

Run this command.

source <(kubectl completion bash)

#### ****Step 2****

Press **Tab** to display a list of available commands.

Try the following examples:

kubectl <TAB><TAB>

35350b93c86e21bc.png

You can also complete a partial command.

kubectl co<TAB><TAB>

439b36f031f3e4ed.png

This feature makes using kubectl even easier.

### ****Run and deploy a container****

The easiest way to get started with Kubernetes is to use the kubectl run command.

#### ****Step 1****

Use kubectl run to launch a single instance of the nginx container.

kubectl run nginx --image=nginx:1.10.0

In Kubernetes, all containers run in pods. And in this command, Kubernetes created what is called a deployment behind the scenes, and runs a single pod with the nginx container in it. A deployment keeps a given number of pods up and running even when the nodes they run on fail. In this case, you run the default number of pods, which is 1.

You'll learn more about deployments later.

#### ****Step 2****

Use the kubectl get pods command to view the pod running the nginx container.

kubectl get pods

#### ****Step 3****

Use the kubectl expose command to expose the nginx container outside Kubernetes.

kubectl expose deployment nginx --port 80 --type LoadBalancer

Kubernetes created a service and an external load balancer with a public IP address attached to it (you will learn about services later). The IP address remains the same for the life of the service. Any client who hits that public IP address (for example an end user or another container) is routed to pods behind the service. In this case, that would be the nginx pod.

#### ****Step 4****

Use the kubectl get command to view the new service.

kubectl get services

You'll see an external IP that you can use to test and contact the nginx container remotely.

It may take a few seconds before the ExternalIP field is populated for your service. This is normal—just re-run the kubectl get services command every few seconds until the field is populated.

#### ****Step 5****

Use the kubectl scale command to scale up the number of backend applications (pods) running on your service using.

kubectl scale deployment nginx --replicas 3

This is useful when you want to increase workload for a web application that is becoming more popular.

#### ****Step 6****

Get the pods one more time to confirm that Kubernetes has updated the number of pods.

kubectl get pods

#### ****Step 7****

Use the kubectl get services command again to confirm that your external IP address has not changed.

kubectl get services

#### ****Step 8****

Use the external IP address with the curl command to test your demo application.

curl http://<External IP>:80

Kubernetes supports an easy-to-use workflow out of the box using the kubectl run, expose, and scale commands.

### ****Clean Up****

Clean up nginx by running the following commands.

kubectl delete deployment nginx

kubectl delete service nginx

Now that you've seen a quick tour of Kubernetes, it's time to dive into each of the components and abstractions.

You covered a lot of information. The rest of this lab goes over these concepts in depth. You can always come back to this demo if you need to see it again.

## Pods

Investigate pods in more detail.

### ****Creating Pods****

Pods can be created using pod configuration files.

#### ****Step 1****

Explore the built-in pod documentation using the kubectl explaincommand.

kubectl explain pods

While you explore the Kubernetes API, kubectl explain will be one of the most common commands you use. Note how you used it above to investigate an API object and how you will use it below to check on various properties of API objects.

#### ****Step 2****

Explore the monolith pod's configuration file.

cat pods/monolith.yaml

The pod is made up of one container (called monolith). You pass a few arguments to the container when it starts up and open port 80 for HTTP traffic.

#### ****Step 3****

Use the kubectl explain command with the .spec option to view more information about API objects. This example inspects containers.

kubectl explain pods.spec.containers

Explore the rest of the API before you continue.

#### ****Step 4****

Create the monolith pod using kubectl create.

kubectl create -f pods/monolith.yaml

Click Check my progress to verify the objective.

Create the monolith pod

Check my progress

#### ****Step 5****

Use the kubectl get pods command to list all pods running in the default namespace.

kubectl get pods

It may take a few seconds before the monolith pod is up and running, because the monolith container image must be pulled from the Docker Hub before you can run it.

#### ****Step 6****

When the pod is running, use the kubectl describe command to get more information about the monolith pod.

kubectl describe pods monolith

You'll see a lot of the information about the monolith pod, including the pod IP address and the event log. This information will be useful when troubleshooting.

It's time for a quick knowledge check. Answer the following questions about the monolith pod.

* What is the pod IP address?
* Which node is the pod running on?
* Which containers are running in the pod?
* Which labels are attached to the pod?
* Which arguments are set on the container?

As you can see, Kubernetes makes it easy to create pods by describing them in configuration files and to view information about them when they are running. At this point, you can create all the pods your deployment requires!

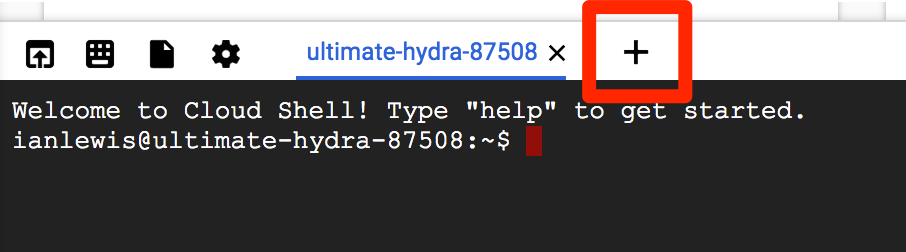
### ****Interacting with pods****

Pods are allocated a private IP address by default that cannot be reached outside of the cluster. Use the kubectl port-forwardcommand to map a local port to a port inside the monolith pod.

Use two terminals: one to run the kubectl port-forward command, and the other to issue curl commands.

#### ****Step 1****

Click the + button in Cloud Shell to open a new terminal.



#### ****Step 2****

Run the following command to set up port-forwarding from a local port, 10080, to a pod port, 80 (where your container is listening).

kubectl port-forward monolith 10080:80

#### ****Step 3****

To access your pod, return to the first terminal window and run the following curl command.

curl http://127.0.0.1:10080

You get a friendly "Hello" back from the container.

#### ****Step 4****

See what happens when you hit a secure endpoint.

curl http://127.0.0.1:10080/secure

You should get an error.

You get an error because you need to include an auth token in your request.

#### ****Step 5****

Log in to get an auth token from monolith.

curl -u user http://127.0.0.1:10080/login

At the login prompt, enter the password as password to sign in.

Logging in causes a JWT token to be printed out. You'll use it to test your secure endpoint with curl.

#### ****Step 6****

Cloud Shell doesn't handle copying long strings well, so copy the token into an environment variable.

TOKEN=$(curl http://127.0.0.1:10080/login -u user|jq -r '.token')

At the login prompt, enter the password as password to sign in.

#### ****Step 7****

Access the secure endpoint again, and this time include the auth token.

curl -H "Authorization: Bearer $TOKEN" http://127.0.0.1:10080/secure

You should get a response back from your application letting you know it works again!

#### ****Step 8****

Use the kubectl logs command to view logs for the monolith pod.

kubectl logs monolith

#### ****Step 9****

Open another terminal and use the -f flag to get a stream of logs in real-time!

To create the third terminal, click the + button in Cloud Shell and run the following command.

kubectl logs -f monolith

#### ****Step 10****

Use curl in terminal 1 to interact with monolith. And you see logs update in terminal 3.

curl http://127.0.0.1:10080

You can see the logs updating back in terminal 3.

#### ****Step 11****

Use the kubectl exec command to run an interactive shell inside the monolith pod. This can be useful when you want to troubleshoot from within a container.

kubectl exec monolith --stdin --tty -c monolith /bin/sh

#### ****Step 12****

Optional: In the shell, you can test external (outward facing) connectivity using the ping command.

ping -c 3 google.com

#### ****Step 13****

Sign out of the shell.

exit

As you can see, interacting with pods is as easy as using the kubectlcommand. If you need to test a container remotely or get a login shell, Kubernetes provides everything you need to start.

#### ****Step 14****

To quit kubectl port-forward and kubectl logs in terminal 2 and 3, press Ctrl+C.

## Monitoring and Health Checks

Kubernetes supports monitoring applications in the form of readiness and liveness probes. Health checks can be performed on each container in a pod. Readiness probes indicate when a pod is "ready" to serve traffic. Liveness probes indicate whether a container is "alive." If a liveness probe fails multiple times, the container is restarted. Liveness probes that continue to fail cause a pod to enter a crash loop. If a readiness check fails, the container is marked as not ready and is removed from any load balancers.

In this lab, you deploy a new pod named healthy-monolith, which is largely based on the monolith pod with the addition of readiness and liveness probes.

In this lab, you learn how to:

* Create pods with readiness and liveness probes.
* Troubleshoot failing readiness and liveness probes.

### ****Creating Pods with Liveness and Readiness Probes****

#### ****Step 1****

Explore the healthy-monolith pod configuration file.

cat pods/healthy-monolith.yaml

#### ****Step 2****

Create the healthy-monolith pod using kubectl.

kubectl create -f pods/healthy-monolith.yaml

Click Check my progress to verify the objective.

Create the healthy-monolith pod

Check my progress

#### ****Step 3****

Pods are not marked ready until the readiness probe returns an HTTP 200 response. Use the kubectl describe command to view details for the healthy-monolith pod.

kubectl describe pod healthy-monolith

### ****Readiness Probes****

See how Kubernetes responds to failed readiness probes. The monolith container supports the ability to force failures of its readiness and liveness probes.

This enables you to simulate failures for the healthy-monolith pod.

#### ****Step 1****

Use the kubectl port-forward command in terminal 2 to forward a local port to the health port of the healthy-monolith pod.

kubectl port-forward healthy-monolith 10081:81

#### ****Step 2****

Force the monolith container readiness probe to fail. Use the curlcommand in terminal 1 to toggle the readiness probe status. Note that this command does not show any output.

curl http://127.0.0.1:10081/readiness/status

#### ****Step 3****

Get the status of the healthy-monolith pod using the kubectl getpods -w command.

kubectl get pods healthy-monolith -w

#### ****Step 4****

Press Ctrl+C when there are 0/1 ready containers. Use the kubectl describe command to get more details about the failing readiness probe.

kubectl describe pods healthy-monolith

#### ****Step 5****

Notice the events for the healthy-monolith pod report details about failing readiness probes.

To force the monolith container readiness probe to pass, toggle the readiness probe status by using the curl command.

curl http://127.0.0.1:10081/readiness/status

#### ****Step 6****

Wait about 15 seconds and get the status of the healthy-monolithpod using the kubectl get pods command.

kubectl get pods healthy-monolith

#### ****Step 7****

Press Ctrl+C in terminal 2 to close the kubectl proxy (i.e port-forward) command.

### ****Liveness Probes****

Building on what you learned in the previous tutorial, use the kubectl port-forward and curl commands to force the monolith container liveness probe to fail. Observe how Kubernetes responds to failing liveness probes.

#### ****Step 1****

Use the kubectl port-forward command to forward a local port to the health port of the healthy-monolith pod in terminal 2.

kubectl port-forward healthy-monolith 10081:81

#### ****Step 2****

To force the monolith container readiness probe to pass, toggle the readiness probe status by using the curl command in another terminal.

curl http://127.0.0.1:10081/healthz/status

#### ****Step 3****

Get the status of the healthy-monolith pod using the kubectl get pods -w command.

kubectl get pods healthy-monolith -w

#### ****Step 4****

When a liveness probe fails, the container is restarted. Once restarted, the healthy-monolith pod should return to a healthy state. Press Ctrl+C to exit that command when the pod restarts. Note the restart count.

#### ****Step 5****

Use the kubectl describe command to get more details about the failing liveness probe. You can see the related events for when the liveness probe failed and the pod was restarted.

kubectl describe pods healthy-monolith

#### ****Step 6****

When you are finished, press Ctrl+C in terminal 2 to close the kubectl proxy command.

**Congratulations!**

You learned about Kubernetes pods and Kubernetes support for application monitoring using liveness and readiness probes. You also learned how to add readiness and liveness probes to pods and what happens when probes fail.

## Services

Next steps:

* Create a service.
* Use label selectors to expose a limited set of pods externally.

### ****Creating a Service****

Before creating your services, create a secure pod with an nginx server called secure-monolith that can handle HTTPS traffic.

#### ****Step 1****

Create two volumes that the secure pod will use to bring in (or consume) data.

The first volume of type secret stores TLS cert files for your nginx server.

Return to terminal 1 and create the first volume using the following command:

kubectl create secret generic tls-certs --from-file tls/

This uploads cert files from the local directory tls/ and stores them in a secret called tls-certs.

Create the second volume of type ConfigMap to hold nginx's configuration file.

kubectl create configmap nginx-proxy-conf --from-file nginx/proxy.conf

This uploads the proxy.conf file to the cluster and calls the ConfigMap nginx-proxy-conf.

#### ****Step 2****

Explore the proxy.conf file that nginx will use.

cat nginx/proxy.conf

The file specifies that SSL is ON and specifies the location of cert files in the container file system.

The files really exist in the secret volume, so you need to mount the volume to the container's file system.

#### ****Step 3****

Explore the secure-monolith pod configuration file.

cat pods/secure-monolith.yaml

Under volumes, the pod attaches the two volumes you created. And under volumeMounts, it mounts the tls-certs volume to the container's file system so nginx can consume the data.

#### ****Step 4****

Run the following command to create the secure-monolith pod with its configuration data.

kubectl create -f pods/secure-monolith.yaml

Now that you have a secure pod, expose the secure-monolith pod externally using a Kubernetes service.

#### ****Step 5****

Explore the monolith service configuration file.

cat services/monolith.yaml

The file contains:

* The selector that finds and exposes pods with labels app=monolith and secure=enabled
* targetPort and nodePort that forward external traffic from port 31000 to nginx on port 443.

#### ****Step 6****

Use the kubectl create command to create the monolith service from the monolith service configuration file.

kubectl create -f services/monolith.yaml

The type: NodePort in the Service's yaml file means that it uses a port on each cluster node to expose the service. This means that it's possible to have port collisions if another app tries to bind to port 31000 on one of your servers.

Normally, Kubernetes handles this port assignment for you. In this lab, you chose one so that it's easier to configure health checks later.

#### ****Step 7****

Use the gcloud compute firewall-rules command to allow traffic to the monolith service on the exposed nodeport.

gcloud compute firewall-rules create allow-monolith-nodeport --allow=tcp:31000

Now that everything is set up, you should be able to test the secure-monolith service from outside the cluster without using port forwarding.

#### ****Step 8****

Get an IP address for one of your nodes.

gcloud compute instances list

#### ****Step 9****

Try to open the URL in your browser.

https://<EXTERNAL\_IP>:31000

That timed out or refused to connect. What's going wrong?

It's time for a quick knowledge check. Use the following commands to answer the questions below.

kubectl get services monolith

kubectl describe services monolith

**Questions:**

* Why can't you get a response from the monolith service?
* How many endpoints does the monolith service have?
* What labels must a pod have to be picked up by the monolith service?

### ****Adding Labels to Pods****

Currently the monolith service does not have any endpoints. One way to troubleshoot an issue like this is to use the kubectl get podscommand with a label query.

#### ****Step 1****

Determine that there are several pods running with the monolith label.

kubectl get pods -l "app=monolith"

#### ****Step 2****

But what about app=monolith and secure=enabled?

kubectl get pods -l "app=monolith,secure=enabled"

Notice that this label query does not print any results. You need to add the "secure=enabled" label to them.

#### ****Step 3****

Use the kubectl label command to add the missing secure=enabled label to the secure-monolith pod.

kubectl label pods secure-monolith 'secure=enabled'

Click Check my progress to verify the objective.

Create a secret, service, firewall rule and pod with label

Check my progress

#### ****Step 4****

Check to see that your labels are updated.

kubectl get pods secure-monolith --show-labels

#### ****Step 5****

View the list of endpoints on the monolith service.

kubectl get endpoints monolith

And you have one!

#### ****Step 6****

Test this by testing one of your nodes again.

gcloud compute instances list | grep gke-

Open the following URL in your browser. You will need to click through the SSL warning because secure-monolith is using a self-signed certificate.

https://<EXTERNAL\_IP>:31000

# Quiz:

Why use Kubernetes?

It provides a set of APIs that you can use to deploy containers on a set of nodes.

Which component do you use to send requests to API servers on masters to configure the cluster?- kubectl

Summary:

In this lesson, you learned the concepts of how Kubernetes and Container Engine run and scale applications on a number of machines. You can also run a number of applications on the same set of machines.