KUBERNETES ARCHITECTURE

learnk8s

Table of contents

Chapter 1

Kubernetes architecture

What is a Kubernetes cluster made of?

In this module you will experience first-hand Kubernetes' architectural design choices.

The plan is as follow:

- You will create a three nodes.
- You will bootstrap a cluster with `kubeadm` — a tool designed to create Kubernetes clusters.
- You will deploy a demo application with two replicas.
- One by one, you will take down each Node and inspect the status of the cluster.

Will Kubernetes recover from the failures?

There's only one way to know!

Before you start, please be aware that you will have to SSH into several nodes.

To keep track of what command is sent to a specific node, current host or container, the code snippets follow this convention:

When the command is local, the title of the windows is `bash` or `PowerShell` :

bash

```
$ local command
```

PowerShell — □ X

```
PS> local command
```

When the command is executed against a remote instance, the code snippet is titled `bash@<remote location>`.

bash@node1

```
$ remote command on node1
```

Let's get started!

Chapter 2

Bootstrapping a cluster

In this section, you will create a three-node Kubernetes cluster with two worker nodes.

Instead of using a premade cluster, such as the one you can find on the major cloud providers, you will go through bootstrapping a cluster from scratch.

But before you can create a cluster, you need nodes.

You will use minikube for that:

bash

```
$ minikube start --no-kubernetes --container-runtime=containerd --driver=docker --nodes 3
😄  minikube v1.29.0
✨  Using the docker driver based on user configuration
👍  Starting minikube without Kubernetes in cluster minikube
🚜  Pulling base image ...
🔥  Creating docker container (CPUs=2, Memory=2200MB) ...
📦  Preparing containerd 1.6.15
🏄  Done! minikube is ready without Kubernetes!
```

It may take a moment to create those Ubuntu instances depending on your setup.

You can verify that the nodes are created correctly with:

bash

```
$ minikube node list
minikube      192.168.105.18
minikube-m02  192.168.105.19
minikube-m03  192.168.105.20
```

It's worth noting that those nodes are not vanilla Ubuntu images.

Containerd (the container runtime) is preinstalled.

Apart from that, there's nothing else.

It's time to install Kubernetes!

Bootstrapping the primary node

In this section, you will install Kubernetes on the master Node and bootstrap the control plane.

The control plane is made of the following components:

- **etcd**, a consistent and highly-available key-value store.
- **kube-apiserver**, the API you interact with when you use `kubectl`.
- **kube-scheduler**, used to schedule Pods and assign them to Nodes.
- **kube-controller-manager**, a collection of controllers used to reconcile the state of the cluster.

You can SSH into the primary node with:

```bash
$ minikube ssh --node minikube
```

There are several tools designed to bootstrap clusters from scratch.

However, `kubeadm` is the only official tool and the best supported.
You will use that to create your cluster.
To install `kubeadm` and a few more prerequisites, execute the following script in the primary node:

bash@minikube

```
$ curl -s -o master.sh https://academy.learnk8s.io/master.sh
$ sudo bash master.sh auto
```

In a new terminal session, SSH into the second node with:

bash

```
$ minikube ssh --node minikube-m02
```

And execute the following setup script:

bash@minikube-m02

```
$ curl -s -o worker.sh https://academy.learnk8s.io/worker.sh
$ sudo bash worker.sh auto
```

Repeat the same steps for the last node.
In a new terminal session, SSH into the third node with:

bash

```
$ minikube ssh --node minikube-m03
```

And execute the following setup script:

```
bash@minikube-m03

$ curl -s -o worker.sh https://academy.learnk8s.io/worker.sh
$ sudo bash worker.sh auto
```

Those scripts:

- Downloads `kubeadm`, `kubectl` and the `kubelet`.
- Installs the shared certificates necessary to trust other entities.
- Creates the Systemd unit necessary to launch the `kubelet`.
- Creates the `kubeadm` config necessary to bootstrap the cluster.

Once completed, you can finally switch back to the terminal session for the primary node and bootstrap the cluster with:

```
bash@minikube

$ sudo kubeadm init --config config.yaml
# truncated output
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control plane was initialized successfully!

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

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Then you can join any number of worker nodes by running the following on each as root:

  kubeadm join 192.168.49.2:8443 --token nsyxx6.quc5x0djkjdr564u \
    --discovery-token-ca-cert-hash sha256:3bc3320116914548672323 97bf837dbb73affc96…
```

Please make a note of the join command at the end of `kubeadm init` output. You will need it later to join the workers, and you don't have to run it now.

The command is similar to:

```
kubeadm join :8443 --token [some token] \
  --discovery-token-ca-cert-hash [some hash]
```

The command is necessary to join other nodes in the cluster.

> **Please don't skip the previous step!** Make a note of the command and write it down! You will need it later on.

In the output of the `kubeadm init`, you can notice this part:

```bash
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

The above instructions are necessary to configure `kubectl` to talk to the control plane.

You should go ahead and follow those instructions.

Once you are done, you can verify that `kubectl` is configured correctly with:

```
bash@minikube

$ kubectl cluster-info
Kubernetes control plane is running at https://<master-node-ip>:8443
CoreDNS is running at https://<master-node-ip>:8443/api/v1/namespaces/kube…
```

Let's check the pods running in the control plane with:

```
bash@minikube

$ kubectl get pods --all-namespaces
NAMESPACE     NAME                               READY   STATUS
kube-system   coredns-787d4945fb-knjnx           0/1     ContainerCreating
kube-system   coredns-787d4945fb-vmnzn           0/1     ContainerCreating
kube-system   etcd-minikube                      1/1     Running
kube-system   kube-apiserver-minikube            1/1     Running
kube-system   kube-controller-manager-minikube   1/1     Running
kube-system   kube-proxy-wtdc4                   1/1     Running
kube-system   kube-scheduler-minikube            1/1     Running
```

Why are the CoreDNS pods in the "ContainerCreating" state?

Let's investigate further:

```
bash@minikube

$ kubectl describe pod coredns-787d4945fb-knjnx -n kube-system
Name:                 coredns-787d4945fb-knjnx
Namespace:            kube-system
# truncated output
Events:
Type      Reason                    Message
----      ------                    -------
Warning   FailedCreatePodSandBox    Failed to create pod sandbox: ...(truncated)... network ...
```

The message suggests that the network is not ready!

But how is that possible?

You are using kubectl to send commands to the control plane — it should be ready?

The message is cryptic, but it tells you that you must still configure the network plugin.

In Kubernetes, there is no standard or default network setup.

Instead, you should configure your network and install the appropriate plugin.

You can choose from several network plugins, but for now, you will install Flannel — one of the simplest.

Installing a network plugin

Kubernetes imposes the following networking requirements on the cluster:

1. All Pods can communicate with all other pods.
2. Agents on a node, such as system daemons, kubelet, etc., can communicate with all pods on that node.

Those requirements are generic and can be satisfied in several ways.

That allows you to decide how to design and operate your cluster network.

In this case, each node in the cluster has a fixed IP address, and you only need to assign Pod IP addresses.

Flannel is a network plugin that:

1. Assigns a subnet to every node.
2. Assigns IP addresses to Pods.
3. Maintains a list of Pods and Nodes in the cluster.

In other words, Flannel can route the traffic from any Pod to any Pod — just what we need.

Let's install it in the cluster.

The `master.sh` script you executed earlier also created a `flannel.yaml` in the local directory.

```
bash@minikube

$ ls -1
config.yaml
flannel.yaml
master.sh
traefik.yaml
```

You can submit it to the cluster with:

```
bash@minikube

$ kubectl apply -f flannel.yaml
namespace/kube-flannel
```

```
 created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds created
```

It might take a moment to download the container and create the Pods.

You can check the progress with:

bash@minikube

```
$ kubectl get pods --all-namespaces
```

Once all the Pods are "Ready", the control plane Node should transition to a *"Ready"* state too:

bash@minikube

```
$ kubectl get nodes -o wide
NAME        STATUS   ROLES           VERSION
minikube    Ready    control-plane   v1.26.2
```

Excellent!

The control plane is successfully configured to run Kubernetes.

This time, CoreDNS should be running as well:

bash@minikube

```
$ kubectl get pods --all-namespaces
NAMESPACE     NAME                               READY   STATUS
kube-system   coredns-787d4945fb-knjnx           1/1     Running
kube-system   coredns-787d4945fb-vmnzn           1/1     Running
kube-system   etcd-minikube                      1/1     Running
kube-system   kube-apiserver-minikube            1/1     Running
kube-system   kube-controller-manager-minikube   1/1     Running
kube-system   kube-proxy-wtdc4                   1/1     Running
kube-system   kube-scheduler-minikube            1/1     Running
```

However, there must be more than a control plane to run workloads.

You also need worker nodes.

Connecting worker Nodes

In the control plane node, pay attention to the running nodes:

```
bash@minikube
$ watch kubectl get nodes -o wide
```

The `watch` command executes the `kubectl get nodes` command at a regular interval.

You will use this terminal session to observe nodes as they join the cluster.

In the other terminal, first, list the IP address of the second node.

```
bash
$ minikube node list
minikube        192.168.105.18
minikube-m02    192.168.105.19
minikube-m03    192.168.105.20
```

Then, you should SSH into the first worker node with:

```
bash
$ minikube ssh -n minikube-m02
```

Download and execute the following script to install the prerequisites:
You should now join the worker Node to the cluster with the `kubeadm join` command you saved earlier.
The command should look like this (the only thing we changed is added `sudo` in front):

```
bash@minikube-m02
$ sudo kubeadm join <master-node-ip>:8443 --token [some token] \
  --discovery-token-ca-cert-hash [some hash]
```

Execute the command and pay attention to the terminal window in the control plane Node.

> If you encounter a "Preflight Check Error", append the following flag to the `kubeadm join` command: `--ignore-preflight-errors=SystemVerification`.

The worker node is provisioned and transitions to the "Ready" state.

As soon as the command finishes, execute the following lines to enable kubelet to start after worker node reboot:

```bash
$ sudo systemctl enable kubelet
```

And finally, you should repeat the instructions to join the second worker node.

First, list the IP address of the third node with:

```bash
$ minikube node list
minikube      192.168.105.18
minikube-m02  192.168.105.19
minikube-m03  192.168.105.20
```

Then, SSH into the node with:

```bash
$ minikube ssh -n minikube-m03
```

Download and install the prerequisites (pay attention to the new IP address):
Join the node to the cluster with the same `kubeadm join` command you used earlier:

```
bash@minikube-m03

$ sudo kubeadm join <master-node-ip>:8443 --token [some token] \
  --discovery-token-ca-cert-hash [some hash]
```

> If you encounter a "Preflight Check Error", append the following flag to the `kubeadm join` command: `--ignore-preflight-errors=SystemVerification`.

You should observe even the second worker joining the cluster and transitioning to the "Ready" state.
And finally, complete the kubelet configuration with (enable kubelet autostart):

```
bash@minikube-m03

$ sudo systemctl enable kubelet
```

Excellent, you have a running cluster!
But there needs to be one nicety added to this setup: an Ingress controller.

Installing the Ingress controller

The Ingress controller is necessary to read your Ingress manifests and route traffic inside the cluster.

Kubernetes has no default Ingress controller, so you must install one if you wish to use it.

When you executed the `master.sh` command, it created a `traefik.yaml` file.

```
bash@minikube
$ ls -1
config.yaml
flannel.yaml
master.sh
traefik.yaml
```

Traefik is an ingress controller and can be installed with the following command:

```
bash@minikube
$ kubectl apply -f traefik.yaml
namespace/traefik created
serviceaccount/traefik created
clusterrole.rbac.authorization.k8s.io/traefik created
clusterrolebinding.rbac.authorization.k8s.io/traefik created
daemonset.apps/traefik created
```

The Ingress controller is deployed as a Pod, so you should wait until the image is downloaded.

If the installation was successful, you should be able to return the host and `curl` the first worker Node:

```bash
$ curl <ip minikube-m02>
curl: (7) Failed to connect to 192.168.49.3 port 80: Operation timed out
```

Unfortunately, that IP address lives in the Docker network and is not reachable from your Mac or Windows (it's reachable if you are working on Linux).

But, worry not.

You can launch a jumpbox — a container with a terminal session in the same network:

```bash
$ docker run -ti --rm --network=minikube ghcr.io/learnk8s/netshoot:2023.03
```

From this container, you can reach any node of the cluster — let's retrieve and repeat the experiment:

```bash@netshoot
$ curl <ip minikube-m02>
404 page not found
```

You should see a `404 page not found` message.

> `404 page not found` is not an error. This is a message from the Ingress controller saying no routes are set up for this URL.

Is your cluster ready now?

Yes, there's one minor step needed.

You have to be logged in to the control plane node to issue `kubectl` commands.

Wouldn't it be easier if you could send commands from your computer instead?

Exposing the kubeconfig

The `kubeconfig` file holds the credentials to connect to the cluster.

Currently, the file is saved on the control plane node, but you can copy the content and save it on your computer (outside of the minikube virtual machine).

You can retrieve the content with:

bash

```bash
$ minikube ssh --node minikube cat '$HOME/.kube/config' >kubeconfig
```

Now, the content is saved in your local file, named `kubeconfig`.

If you are on Mac or Windows, you should apply one small change: replace `<master-node-ip>:8443` with localhost and the correct port exposed by Docker.

First, list your nodes with:

```bash
$ docker ps
CONTAINER ID   IMAGE                                PORTS                       NAMES
5717b8d142ac   gcr.io/k8s-minikube/kicbase:v0.0.37  127.0.0.1:53517->8443/tcp   minikube-m03
d5e1dbe9611c   gcr.io/k8s-minikube/kicbase:v0.0.37  127.0.0.1:53503->8443/tcp   minikube-m02
648efe712022   gcr.io/k8s-minikube/kicbase:v0.0.37  127.0.0.1:53486->8443/tcp   minikube
```

Find the port that forwards to 8443 for the control plane (in the above example is `53486`).

And finally, replace `<master-node-ip>:8443` in your kubeconfig:

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CR...
    server: https://127.0.0.1:<insert-port-here>
  name: mk
contexts:
```

Finally, navigate to the directory where the file is located and execute the following line:

```bash
$ export KUBECONFIG="${PWD}/kubeconfig"
```

```bash
$ export KUBECONFIG="${PWD}/kubeconfig"
```

```powershell
PS> $Env:KUBECONFIG="${PWD}\kubeconfig"
```

You can verify that you are connected to the cluster with:

```bash
$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:53486
CoreDNS is running at https://127.0.0.1:53486/api/v1/namespaces/kube…
```

> Please note that you should export the path to the `kubeconfig` whenever you create a new terminal session.

You can also store the credentials alongside the default `kubeconfig` file instead of changing environment variables. However, since you will destroy the cluster at the end of this module, it's better to keep them separated for now.

> If you are still convinced you should merge the details with your kubeconfig, you can find the instructions on how to do so here.

Congratulations!

You just configured a fully functional Kubernetes cluster!

Recap

- You created three virtual machines using minikube.
- You bootstrapped the Kubernetes control plane node using `kubeadm`.
- You installed Flannel as the network plugin.
- You installed an Ingress controller.
- You configured `kubectl` to work from outside the control plane.

The cluster is fully functional, and it's time to deploy an application.

Chapter 3

Deploying an application

Before exploring the different components in your Kubernetes cluster, you should deploy a simple *"Hello World"* application.
The app doesn't do much, but it's an excellent example of what happens in the control plane.

Hello Pod

You will create a Pod that has a single container with the `ghcr.io/learnk8s/podinfo:2023.03` image.
If you wish to test the application locally before you deploy it in the cluster, you can do so with:

```bash
$ docker run -ti -p 8080:9898 ghcr.io/learnk8s/podinfo:2023.03
```

You can inspect the output with:

```bash
$ curl http://localhost:8080
{
  "hostname": "podinfo-787d4945fb-fs9cc",
  "version": "6.3.4",
  "revision": "1abc44f0d8dd6cd9df76090ea4ad694b70e03ee4",
  "color": "#34577c",
  "logo": "https://raw.githubusercontent.com/stefanprodan/podinfo/gh-pages/cuddle_clap.gif",
  "message": "greetings from podinfo v6.2.2",
  "goos": "linux",
  "goarch": "arm64",
```

```
  "runtime": "go1.20.1",
  "num_goroutine": "7",
  "num_cpu": "4"
}%
```

Notice how the container exposes the pod name in the hostname field.

Creating a Deployment

You should create a deployment for your Pod with the following requirements:

1. All the *Pods* in the *Deployment* should have the label `app: hello`.
2. You should use `ghcr.io/learnk8s/podinfo:2023.03` as the container image.
3. The container exposes port 9898.
4. Scale the deployment to two replicas.
5. You should name the deployment `hello-world`.

Please find below an example of a Deployment.
You still have to customize it!

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
```

```yaml
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.23
        ports:
        - containerPort: 80
```

If you want to explore the properties of the Deployment or need a refresher, these links might help:

- [The official documentation for a Deployment.]()
- [The official API specification for a Deployment.]()

Creating a Service

1. The Service should have an `app: hello` selector.
2. The Service should be named `hello`.

Please find below an example of a Service.
You still have to customize it!

```yaml
# service.yaml
apiVersion: v1
kind: Service
```

```yaml
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

If you want to explore the properties of the Service or need a refresher, these links might help:

- The official documentation for a Service.
- The official API specification for a Service.

Creating an Ingress manifest

1. The ingress should point to the `hello` Service.
2. It's not necessary to set the hostname.
3. the `path` should be `/`.

Please find below an example of an Ingress manifest.
You still have to customize it!

ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
spec:
  rules:
  - http:
```

```yaml
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

If you want to explore the properties of the Ingress or need a refresher, these links might help:

- The official documentation for an Ingress manifest.
- The official API specification for an Ingress manifest.

Verifying the deployment

List the current IP addresses for the cluster from your host machine with:

```bash
$ minikube node list
minikube       192.168.105.18
minikube-m02   192.168.105.19
minikube-m03   192.168.105.20
```

If your app is deployed correctly, you should be able to execute:

- `kubectl get pods -o wide` and see two pods deployed — one for each node.
- `curl <ip minikube-m02>` from the jumpbox and see the pod hostname in the JSON output.
- `curl <ip minikube-m03>` from the jumpbox and see the pod hostname in the JSON output.

If your deployment isn't quite right, try to [debug it using this handy flowchart.]()

Otherwise, congrats for making it this far!

In the next chapter, you will start exploring the Kubernetes control plane.

Chapter 4

Exploring etcd

Let's investigate how Kubernetes uses etcd under the hood.
Since etcd is hosted in the control plane, you will have to SSH into the control plane node with:

```bash
$ minikube ssh
```

You will use `etcdctl` to connect to etcd.
`etcdctl` is a client just like `kubectl`.
The binary is already installed on the virtual machine, and you can verify it with:

```
$ etcdctl version
etcdctl version: 3.5.7
API version: 3.5
```

There's something else that you need to connect to etcd, though.

Connecting to etcd

`kubeadm` deploys etcd with [mutual TLS](#) authentication, so you must provide TLS certificates and keys when you issue requests.

This is a bit tedious, but a Bash alias can help speed things along:

```
ssh@minikube

$ alias etcdctl="sudo etcdctl --cacert /var/lib/minikube/certs/etcd/ca.crt \
  --cert /var/lib/minikube/certs/etcd/healthcheck-client.crt \
  --key /var/lib/minikube/certs/etcd/healthcheck-client.key"
```

> The certificate and keys used here are generated by `kubeadm` during Kubernetes cluster bootstrapping. They're designed for use with etcd health checks, but they also work well for debugging.

Then you can run `etcdctl` commands like this:

```
ssh@minikube

$ etcdctl member list --write-out=table
+------------------+---------+--------+----------------------------+----------------------------+
|        ID        | STATUS  |  NAME  |         PEER ADDRS         |        CLIENT ADDRS        |
+------------------+---------+--------+----------------------------+----------------------------+
| 3b17aaa147134dd  | started | master | https://192.168.56.10:2380 | https://192.168.56.10:2379 |
+------------------+---------+--------+----------------------------+----------------------------+
```

As you can see, the cluster has a single etcd node running.

How does the Kubernetes API store data in etcd?

Let's list all the data:

```
ssh@minikube
```

```
$ etcdctl get --prefix / --keys-only
```

As we can see everything is under `/registry`, and e.g. pods are under the `/registry/pods` prefix:

```
ssh@minikube

$ etcdctl get --prefix /registry/pods --keys-only
```

The naming scheme is `/registry/pods/<namespace>/<pod-name>`.
For instance, here you can see the scheduler pod definition:

```
ssh@minikube

$ etcdctl get --prefix /registry/pods/kube-system/ --keys-only | grep scheduler
/registry/pods/kube-system/kube-scheduler-minikube
```

> This uses the `--keys-only` option, which, as you might expect, returns the keys of the queries.

What does the actual data look like?

Reading, writing and watching keys

Let's investigate:

```
ssh@minikube

$ etcdctl get /registry/pods/kube-system/kube-scheduler-minikube | head -6 ; echo
/registry/pods/kube-system/kube-scheduler-minikube
k8s

v1Pod�
�
kube-scheduler-master�
                        kube-system"*$f8e4441d-fb03-4c98-b48b-61a42643763a2���童Z
```

It looks kind of like junk!

That's because the Kubernetes API stores the actual object definitions in a binary format instead of something human-readable.

To see an object specification in a friendly format like JSON, you must go through the API instead of accessing etcd directly.

Kubernetes' naming scheme for etcd keys should make perfect sense now: it allows the API to query or watch all objects of a particular type in a specific namespace using an etcd prefix query.

This is a widespread pattern in Kubernetes and is how Kubernetes controllers and operators subscribe to changes for objects they're interested in.

Let's try subscribing to pod changes in the `default` namespace to see this in action.

First, use the `watch` command with the appropriate prefix:

ssh@minikube

```
$ etcdctl watch --prefix /registry/pods/default/ --write-out=json
```

Then, in another terminal, scale your deployment to three replicas:

ssh@minikube

```
$ kubectl scale --replicas=3 deployment/hello-world
deployment.apps/first-deployment scaled
```

You should see several JSON messages appear in the etcd watch output, one for each status change in the pod (for instance, going from *"Pending"* to *"Scheduled"* to *"Running"* statuses).

Each message should look something like this:

output.json


```
{
  "Header": {
    "cluster_id": 18038207397139143000,
    "member_id": 12593026477526643000,
    "revision": 935,
    "raft_term": 2
  },
  "Events": [
    {
      "kv": {
        "key": "L3JlZ2lzdHJ5L3BvZHMvZGVmYXVsdC9uZ2lueA==",
        "create_revision": 935,
        "mod_revision": 935,
        "version": 1,
        "value": "azh...ACIA"
```

```
      }
    }
  ],
  "CompactRevision": 0,
  "Canceled": false,
  "Created": false
}
```

Let's try to scale down the deployment to two replicas with:

```
ssh@minikube

$ kubectl scale --replicas=2 deployment/hello-world
deployment.apps/first-deployment scaled
```

You should observe a similar flow of messages.

This time the status of the Pod goes from *"Running"* to "*Terminating*", though.

In real-world usage, you would rarely interact with etcd directly in this way and would instead subscribe to changes through the Kubernetes API.

But it's easy to imagine how the API interacts with etcd using precisely these watch queries.

Summary

Peeking under the covers of Kubernetes is always interesting, and etcd is one of the most central pieces of the Kubernetes puzzle.

In this chapter, you learned:

- How to connect to etcd.
- How to read and watch for value changes.
- How Kubernetes stores the actual resource definition.

It's important to remember that etcd is where Kubernetes stores all of the information about a cluster's state; in fact, it's the only stateful part of the entire Kubernetes control plane.

Chapter 5

Exploring the API server

Users can access the API server using `kubectl`, client libraries, or by making REST requests.
While it's convenient to use `kubectl`, let's skip that for now and focus on issuing `curl` commands.

Creating a proxy

The API server comprises several components, including authentication, authorization, admission controllers, etc.
To avoid authenticating and authorizing every single request, you can create a proxy with:

```bash
$ kubectl proxy --port=8888 &
[1] 16699
Starting to serve on 127.0.0.1:8888
```

Kubectl proxy creates a tunnel from your local machine to the remote API server.
It also uses your credentials stored in the 'kubeconfig' file to authenticate.
From now on, when you send requests to `127.0.0.1:8888`, kubectl forwards them to the API server in your cluster.
You can verify it by issuing a request in another terminal:

```bash
$ curl localhost:8888
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/apis/admissionregistration.k8s.io",
    "/apis/admissionregistration.k8s.io/v1",
    // more APIs ...
  ]
}
```

What are all those APIs, though?

Kubernetes API

Every request to the API server follows a RESTful API pattern:

- `/api/` (the core APIs) or
- `/apis/` (APIs grouped by API group).

There's one thing you should remember when navigating resources via the API: namespaces.

Namespaced resources can only be created within a namespace, and the name of that namespace is included in the HTTP path.

If the resource is global, like in the case of a Node, the namespace is not present in the HTTP path.

Let's explore the differences with an example.

You can retrieve all Pods with:

```bash
$ curl http://localhost:8888/api/v1/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "5155"
  },
  "items": [
    {
      "metadata": {
        "name": "hello-world-56686f7466-db9g5",
# truncated output
```

And you can retrieve a single Pod with:

```bash
$ curl http://localhost:8888/api/v1/namespaces/default/pods/hello-world-56686f7466-jbwvt
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "hello-world-56686f7466-jbwvt",
    "generateName": "hello-world-56686f7466-",
    "namespace": "default",
    "uid": "6f0c094a-1d4a-4e3a-9050-001104e61f5d",
# truncated output
```

Please note the structure of the HTTP path:

```
/api/v1/namespaces/[namespace-name]/[resource-type-name]/[resource-name]
```

Now let's have a look at a global resource such as a Node:

```bash
$ curl http://localhost:8888/api/v1/nodes
{
  "kind": "NodeList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "5765"
  },
# truncated output
```

You can retrieve a single Node with:

```bash
$ curl http://localhost:8888/api/v1/nodes/minikube
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "minikube",
    "uid": "b6b5aa96-3bf6-43a6-aa69-183e16a4397b",
# truncated output
```

In this case, the HTTP path is:

```
/api/v1/[resource-type-name]/[resource-name]
```

OpenAPI spec

It's improbable that you will have to issue HTTP requests directly to the API server.

However, there are times when it's convenient to do so.

For example:

- You might want to create your scripts to orchestrate Kubernetes.
- You might want to build a Kubernetes operator.
- You enjoy the freedom of issuing `curl` requests and mangling the responses with `jq`.

In all of those cases, you should not memorize the API endpoints.

Instead, you should learn how to navigate the OpenAPI spec for the Kubernetes API.

The OpenAPI lists all the resources and endpoints available in the cluster.

If you open the page and search for *"Create Connect Proxy"*, you might notice that there's a section on how to create a proxy and the HTTP path to issue the command.

Let's see if you can complete a quick challenge.

Can you find the API endpoint to display the logs for one of the running Pods?

Once you're done, remember to stop the proxy with:

```bash
$ killall kubectl
```

Summary

In this section, you learned how to connect to the Kubernetes API and issue requests.

You also learned:

- The structure of the HTTP requests in Kubernetes.
- How namespaced resources are handled in the API server.
- How to issue commands to the API server with `curl` and `kubectl proxy`.
- How to read the OpenAPI spec for all the APIs available in the cluster.

Chapter 6

Exploring the kubelet

The `kubelet` is the Kubernetes agent that runs on the worker nodes.

Its job is straightforward and crucial: keeping the node's state updated with the control plane.

The kubelet is the only component of Kubernetes that is often deployed as a binary and **not** as a container.

There's something else that you should remember.

The kubelet doesn't create the containers.

Instead, the kubelet delegates creating the container to the Container Runtime.

Historically, Docker used to be the Container Runtime of choice, but lately, this has been superseded by tools such as:

- containerd
- CRI-O

The cluster that you are using has containerd installed as the Container Runtime.

Exploring containerd

Containerd can run every container image that you build with Docker because there is a standard for building and running containers (the Open Container Initiative).

However, you can't use the Docker CLI to inspect the containers as it works only with the Docker daemon.

Fortunately, `crictl` [crictl is a command-line interface for CRI-compatible container runtimes.](#)

You can use it to inspect and debug container runtimes and applications on a Kubernetes node.

Let's test that.

Access worker1 with:

```bash
$ minikube ssh
```

The nerdctl CLI is already installed, and you can verify it with:

```bash@minikube
$ sudo crictl version
Version:  0.1.0
RuntimeName:  containerd
RuntimeVersion:  v1.6.15
RuntimeApiVersion:  v1alpha2
```

Let's list all the running containers.

If you recall, the command for listing running containers in Docker is `docker ps`.

With crictl, the command is similar:

bash@minikube

```
$ sudo crictl ps
CONTAINER       IMAGE           STATE     NAME                      POD ID
58cfd54d948ef   b6cb6292a4954   Running   traefik                   0c995ff1892c0
38efab4272865   b19406328e70d   Running   coredns                   16eec5b244e46
a1e73f7fca20a   b19406328e70d   Running   coredns                   8e5f4bbe36d72
714baac1290ad   02de1966ebfd3   Running   kube-flannel              a13cddbee7230
c331fef9d40d8   3bb41bb94b1dd   Running   kube-proxy                cf1e9267a3b8b
20b4870309227   c07d98db81fde   Running   kube-controller-manager   c52f9c34ee021
298d6af09b5d1   21fe7c4e54aea   Running   kube-apiserver            e0e13cecb0c8e
f4b549dfd51a7   93331be9505c4   Running   kube-scheduler            aec2ca6d4025f
6f67c6c43096d   ef24580282403   Running   etcd                      13ff9aec1e011
```

If you want to inspect the details of a running container, you can do so with:

bash@minikube

```
$ sudo crictl inspect <container-id>
{
  "status": {
    "id": "58cfd54d948ef0e0663318539e5ec3d377132fe7651b987d1f2c0c3437e4dc62",
    "metadata": {
      "attempt": 0,
      "name": "traefik"
    },
    "state": "CONTAINER_RUNNING",
    "createdAt": "2023-03-10T03:35:25.698980467Z",
    "startedAt": "2023-03-10T03:35:25.745019126Z",
    "finishedAt": "0001-01-01T00:00:00Z",
# truncated output
```

Which is the same as the Docker equivalent `docker inspect <container-id>`.

You can often replace the word `docker` with `crictl`; the command should work!

Simulating failures

The `kubelet` always keeps the local state synced with the control plane.

Let's check what happens when you forcefully remove a Pod from the running node.

Connect to the control plane node with:

```bash
$ minikube ssh
```

And monitor the running Pods with:

```bash@minikube
$ watch kubectl get pods -A
NAMESPACE     NAME                              READY   STATUS    RESTARTS
default       hello-world-5d6cfd9db8-jjrn5      1/1     Running   0
default       hello-world-5d6cfd9db8-4fncr      1/1     Running   0
# truncated output
```

Pay attention to the *RESTARTS* column.

In the other terminal session, connect to the worker1:

```bash
$ minikube ssh --node minikube-m02
```

List all the running containers with:

```
bash@minikube-m02

$ sudo crictl ps
CONTAINER       IMAGE          STATE     NAME          POD ID
331bc172b5342   da23b04751258  Running   app           10301445cf6cd
277b37dc8a3a5   b6cb6292a4954  Running   traefik       29c2cbf8e5478
8747eb5c41638   02de1966ebfd3  Running   kube-flannel  03b7501b43535
60c2595281d2d   3bb41bb94b1dd  Running   kube-proxy    c4685de0da2e4
```

Let's stop the app container:

```
bash@minikube-m02

$ sudo crictl stop 331bc172b5342
331bc172b5342
```

In the previous terminal session, you can observe the Pod being restarted.

```
bash@minikube

$ watch kubectl get pods
NAMESPACE   NAME                             READY   STATUS    RESTARTS
default     hello-world-5d6cfd9db8-jjrn5     1/1     Running   1
default     hello-world-5d6cfd9db8-4fncr     1/1     Running   0
```

Getting closer to the containers

Now let's get back to the terminal session in the worker node.

Instead of `crictl` to list the container, let's use `ctr` — a command-line client shipped as part of the containerd project.

While `crictl` provides a consistent interface to access all container runtimes, `ctr` is specifically designed to interact with containerd.

Let's list all containers:

```
bash@minikube-m02

$ sudo ctr --namespace k8s.io container list
CONTAINER    IMAGE
03b7501b…    registry.k8s.io/pause:3.9
10301445…    registry.k8s.io/pause:3.9
277b37dc…    ghcr.io/learnk8s/traefik:2023.03
29c2cbf8…    registry.k8s.io/pause:3.9
331bc172…    ghcr.io/learnk8s/podinfo:2023.3
60c25952…    registry.k8s.io/kube-proxy:v1.26.2
8747eb5c…    ghcr.io/learnk8s/flannel:2023.3
9792c5c2…    ghcr.io/learnk8s/podinfo:2023.3
98d2c9fd…    ghcr.io/learnk8s/flannel-cni-plugin:2023.3
9e660b31…    ghcr.io/learnk8s/flannel:2023.3
c4685de0…    registry.k8s.io/pause:3.9
```

> The `--namespace` flag is necessary to select all containers running in Kubernetes, and has nothing to do with Kubernetes or Linux namespaces.

Why are there way more containers?

What is this pause container that keeps reappearing?

The pause container is the container that initiates the network namespace for the Pod.
All other containers will share the network namespace with it.
So you should expect to see one pause container for each pod in the node.
Since it's paramount to the functioning of the pod, let's forcefully stop it with:

bash@minikube-m02

```
$ sudo ctr --namespace k8s.io task kill <pause-container-id>
```

In the previous terminal session, you can observe the Pod being restarted.

bash

```
$ watch kubectl get pods
NAMESPACE      NAME                              READY    STATUS     RESTARTS
default        hello-world-5d6cfd9db8-jjrn5      1/1      Running    2
default        hello-world-5d6cfd9db8-4fncr      1/1      Running    0
```

But what really happened?
Let's describe the Pod:

bash

```
$ kubectl describe pod hello-world-5d6cfd9db8-jjrn5
```

```
Name:         hello-world-5d6cfd9db8-jjrn5
Namespace:    default
Priority:     0
# truncated output
Events:
  Type     Reason          Age                 From            Message
  ----     ------          ----                ----            -------
  Normal   SandboxChanged  29s                 kubelet         Pod sandbox changed, it will be killed and re-created.
  Normal   Killing         29s                 kubelet         Stopping container app
  Normal   Started         28s                 kubelet         Started container app
```

There was an error with the Pod sandbox (this is when you forcefully deleted the Pod).

You can inspect the kubelet logs to investigate the issue further:

bash@minikube-m02

```
$ sudo journalctl -u kubelet
# truncated output
worker1 kubelet[648]: "Container not found in pod's containers" containerID="a85…"
worker1 kubelet[648]: "RemoveContainer" containerID="27e…"
```

The kubelet keeps the local state of the node up to date, but it can't find the container.

So it removes all the containers in the Pod and recreates them.

Summary

In this chapter, you learned:

- The kubelet is an agent that lives in every cluster node.

- The kubelet is in charge of keeping the local state of the node up to date with the control plane.
- There is no default Container Runtime in Kubernetes. You need to install one.
- The kublet delegates creating the containers to the Container Runtime.
- How to inspect the logs of the kubelet.

Well done!

Chapter 7

Testing resiliency

Kubernetes is engineered to keep running even if some components are unavailable.

So you could have a temporary failure to one the scheduler, but the cluster will still keep operating as usual.

The same is true for all other components.

The best way to validate this statement is to break the cluster.

What happens when a Node becomes unavailable?

Can Kubernetes gracefully recover?

And what if the primary Node is unavailable?

Let's find out.

Making the nodes unavailable

Observe the nodes and pods in the cluster with:

```bash
$ watch kubectl get nodes,pods -o wide
NAME                  STATUS   ROLES                    INTERNAL-IP
node/minikube         Ready    control-plane            192.168.105.18
node/minikube-m02     Ready    <none>                   192.168.105.19
node/minikube-m03     Ready    <none>                   192.168.105.20

NAME                                 READY   STATUS    NODE
pod/hello-world-5d6cfd9db8-nn256     1/1     Running   minikube-m02
pod/hello-world-5d6cfd9db8-dvnmf     1/1     Running   minikube-m03
```

Observe how Pods and Nodes are in the "Running" and "Ready" states.

Let's break a worker node and observe what happens.

In another terminal session, shut down the second worker node with:

```bash
$ minikube node stop minikube-m03
✋  Stopping node "minikube-m03"  ...
🛑  Successfully stopped node minikube-m03
```

Please note the current time and set the alarm for 5 minutes — (you will understand why soon).

Observe the node almost immediately transitioning to a "Not Ready" state:

```bash
$ kubectl get nodes -o wide
NAME                 STATUS   ROLES                   INTERNAL-IP
node/minikube        Ready    control-plane           192.168.105.18
node/minikube-m02    Ready    <none>                  192.168.105.19
node/minikube-m03    NotReady <none>                  192.168.105.20
```

The application should still serve traffic as usual.

Try to issue a request from the jumpbox with:

```bash@netshoot
$ curl <minikube-m02 IP address>
```

```
Hello, hello-world-5d6cfd9db8-nn256
```

However, there is something odd with the Pods.
Have you noticed?

bash

```bash
$ watch kubectl get pods -o wide
NAME                                 READY    STATUS    NODE
pod/hello-world-5d6cfd9db8-nn256    1/1      Running   minikube-m02
pod/hello-world-5d6cfd9db8-dvnmf    1/1      Running   minikube-m03
```

Why is the Pod on the second worker node still in the "Running" state?
And, even more puzzling, Kubernetes knows the Node is not available (e.g. it's "NotReady"), why isn't rescheduling the Pod?
Kubernetes will wait about 5 minutes before marking the Pods as unavailable and rescheduling it elsewhere.
You should wait for this interval to elapse and observe the change.
The pod state should transition to "Terminating", and a new Pod is created:

bash

```bash
$ watch kubectl get pods -o wide
NAME                                 READY   STATUS        NODE
pod/hello-world-5d6cfd9db8-nn256    1/1     Running       minikube-m02
pod/hello-world-5d6cfd9db8-dvnmf    1/1     Terminating
```

```
 minikube-m03
pod/hello-world-5d6cfd9db8-rjd54   1/1     Running      minikube-m02
```

Why 5 minutes?

The kubelet reports the status of the Pods at regular intervals.

If the primary node doesn't receive updates from the kubelet during those 5 minutes, it assumes that the Pods are gone.

If there's an update, it just resets the timeout.

This is convenient if there are network glitches and the kubelet cannot update the control plane on time.

As long as there's an update within 5 minutes, the cluster works as usual.

Are you happy to wait for 5 minutes before detecting lost Pods?

It depends.

This is not ideal if your applications are optimised for latency and throughput.

If your cluster runs mostly batch jobs, 5 minutes is fine.

You can tweak the timeout with the `--pod-eviction-timeout` flag in the kube-controller-manager component.

> Please note that you have to have access to the control plane to make this change. If you use a managed service such as EKS, AKS, or GKE, you might be unable to configure this value.

But why is the node marked as "NotReady" almost immediately?

Each Kubernetes node sends heartbeats to the control plane.

The cluster determines the availability of each node and takes action when failures are detected.

The default heartbeat pulses every 10s so the cluster can detect a node's availability much quicker.

Excellent, let's move on and do more damage.

Making the control plane unavailable

You should stop the control plane:

```bash
$ minikube node stop minikube
```

> Please note that, from this point onwards, you won't be able to observe the state of the cluster with kubectl.

Notice how the application still serves traffic as usual.
Execute the following command from the jumpbox:

bash@netshoot

```
$ curl <minikube-m02 IP address>
Hello, hello-world-5d6cfd9db8-nn256
```

In other words, the cluster can still operate even if the control plane is unavailable.
You won't be able to schedule or update workloads, though.
You should restart the control plane with:

bash

```
$ minikube node start minikube
```

Please notice that minikube may assign a different forwarding port to this container, and you might need to fix your kubeconfig file.
You can easily verify if the port has changed with:

bash

```
$ CONTAINER ID   IMAGE                                  PORTS                       NAMES
5717b8d142ac   gcr.io/k8s-minikube/kicbase:v0.0.37   127.0.0.1:53517->8443/tcp   minikube-m03
d5e1dbe9611c   gcr.io/k8s-minikube/kicbase:v0.0.37   127.0.0.1:53503->8443/tcp   minikube-m02
648efe712022   gcr.io/k8s-minikube/kicbase:v0.0.37   127.0.0.1:65375->8443/tcp   minikube
```

In this case, the port used to be `53486` and now is `65375`. You should amend your kubeconfig file accordingly:

kubeconfig

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CR…
    server: https://127.0.0.1:<insert-new-port-here>
  name: mk
contexts:
```

With this minor obstacle out of your way, halt the remaining worker Node with:

bash

```
$ minikube node stop minikube-m02
✋  Stopping node "minikube-m02"  ...
🛑  Successfully stopped node minikube-m02
```

You should verify that both Nodes are in the *NotReady* state:

bash

```
$ kubectl get nodes -o wide
NAME                   STATUS   ROLES
```

```
 INTERNAL-IP
node/minikube       Ready    control-plane         192.168.105.18
node/minikube-m02   NotReady <none>                192.168.105.19
node/minikube-m03   NotReady <none>                192.168.105.20
```

And the application is finally unreachable.

Neither `curl <minikube-m02 IP address>` nor `curl <minikube-m03 IP address>` from the jumpbox will work now.

Scheduling workloads with no worker node

Despite not having any worker nodes, you can still scale the application to 5 instances:

```bash
$ kubectl edit deployment hello-world
deployment.apps/hello-world edited
```

And change the replicas to `replicas: 5`.

Monitor the pods with:

```bash
```

```
$ watch kubectl get pods -o wide
NAME                              READY    STATUS
hello-world-5d6cfd9db8-2k7f9      0/1      Pending
hello-world-5d6cfd9db8-8dpgd      0/1      Pending
hello-world-5d6cfd9db8-cwwr2      0/1      Pending
hello-world-5d6cfd9db8-dvnmf      1/1      Terminating
hello-world-5d6cfd9db8-nn256      1/1      Running
hello-world-5d6cfd9db8-rjd54      1/1      Running
```

The Pods stay pending because no worker node is available to run them.

In another terminal session, start both nodes with:

bash

```
$ minikube node start minikube-m02
$ minikube node start minikube-m03
```

It might take a while for the two virtual machines to start, but, in the end, the Deployment should have five replicas "Running".

You can test that the application is available from the jumpbox with:

bash@netshoot

```
$ curl <minikube-m02 IP address>
Hello, hello-world-5d6cfd9db8-8dpgd
```

But, again, there's something odd.

Have you noticed how the Pods are distributed in the *cluster?*

Let's pay attention to the Pod distribution in the cluster:

```bash
$ kubectl get pods -o wide
NAME                              READY    STATUS     NODE
hello-world-5d6cfd9db8-2k7f9     1/1      Running    minikube-m02
hello-world-5d6cfd9db8-8dpgd     1/1      Running    minikube-m02
hello-world-5d6cfd9db8-cwwr2     1/1      Running    minikube-m02
hello-world-5d6cfd9db8-nn256     1/1      Running    minikube-m02
hello-world-5d6cfd9db8-rjd54     1/1      Running    minikube-m02
```

In this case, five Pods run on worker1 and none on worker2. However, you might experience a slightly different distribution.

You could have any of the following:

- 5 Pods on worker1, 0 on worker2
- 0 Pods on worker1, 5 on worker2
- 3 Pods on worker1, 2 on worker2
- 2 Pods on worker1, 3 on worker2

And if you are lucky, you could also have:

- 4 Pods on worker1, 1 on worker2
- 1 Pods on worker1, 4 on worker2

The final configuration depends on the order in which the worker nodes rejoined the cluster.

Why isn't Kubernetes rebalancing the Pods?

There's the Replication Controller in the controller manager, which is some code that matches the number of Pods to the number of replicas in your ReplicaSet.

The ReplicaSet counts the number of "Running" pods and takes no further action.

You have 5 Pods you requested — the ReplicaSet doesn't care about placement.

Kubernetes does not rebalance workloads automatically.

Wait, I did not create any ReplicaSet?

Does this apply to Deployments too?

You created a Deployment that, in turn, created a ReplicaSet.

You can verify it with:

```bash
$ kubectl get deployment,replicaset
NAME                          READY   UP-TO-DATE   AVAILABLE
deployment.apps/hello-world   5/5     5            5

NAME                                     DESIRED   CURRENT   READY
replicaset.apps/hello-world-5d6cfd9db8   5         5         5
```

How can I make sure that the Pods are rebalanced?

There are a few options in Kubernetes to make sure that the Pods are spread in as many worker nodes as possible:

- You can use Pod Topology Spread Constraints to ask the scheduler consider allocating pods in different nodes.

- You can use Pod anti-affinity to discourage running Pods of the same type on the same Node.
- You can install [the Descheduler — a tool designed to detect over-utilised nodes and trigger Pod deletions.]

Summary

There was a lot to take away from this chapter, so let's do a recap of what you learned:

- You can take down a Node in Kubernetes. The application still works as expected if you have more than a single Pod.
- You can take down the control plane. The cluster is still functioning, albeit there could be some disruptions to the API server.
- The kubelet reports to the control plane:
  1. The state of the Nodes at regular intervals.
  2. The node heartbeat.
- The controller manager reschedules unavailable pods only after 5 minutes. You can change that with the `--pod-eviction-timeout` flag.
- The controller manager detects a node is unavailable in less than 10 seconds.

- Kubernetes does not rebalance Pods, but you can use Pod anti-affinity or the Descheduler project to work around it.

Congratulations, you completed the Kubernetes architecture section!