**Getting Started with Volumes**

Learn Kubernetes Volumes and create a cluster with volume.

**We'll cover the following**

* [State preservation](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3jP2QljQxZM#State-preservation)
* [The volumes](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3jP2QljQxZM#The-volumes)
* [Troubleshooting tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3jP2QljQxZM#Troubleshooting-tips-for-minikube)

**State preservation**

Having a system without a state is impossible. Even though there is a tendency to develop stateless applications, we still need to deal with the state. There are databases and other stateful third-party applications. No matter what we do, we need to make sure that the state is preserved no matter what happens to containers, Pods, or even whole nodes.

Most of the time, stateful applications store their state on disk. That leaves us with a problem. If a container crashes, kubelet will restart it. The problem is that it will create a new container based on the same image. All data accumulated inside a container that crashed will be lost.

**The volumes**

**Kubernetes Volumes** solve the need to preserve the state across container crashes. In essence, Volumes are references to files and directories made accessible to containers that form a Pod. The significant difference between different types of Kubernetes Volumes is in the way these files and directories are created.

While the primary use-case for Volumes is the preservation of state, there are quite a few others. For example, we might use Volumes to access Docker’s socket running on a host. Or we might use them to access configuration residing in a file on the host file system.

We can describe Volumes as a way to access a file system that might be running on the same host or somewhere else. No matter where that file system is, it is external to the containers that mount volumes. There can be many reasons why someone might mount a Volume, with state preservation being only one of them.

There are over **twenty-five** Volume types supported by Kubernetes. It would take us too much time to go through all of them. Besides, even if we’d like to do that, many Volume types are specific to a hosting vendor. For example, awsElasticBlockStore works only with AWS, azureDisk and azureFile work only with Azure, and so on and so forth.

We’ll limit our exploration to Volume types that can be used within k3d. You should be able to extrapolate that knowledge to Volume types applicable to your hosting vendor of choice.

The following command will create a cluster. It will copy the prometheus.yml from usercode/volume from the user directory to /files in the cluster.

k3d cluster create mycluster --volume "/usercode/volume/prometheus-conf.yml:/files/prometheus-conf.yml"

## Troubleshooting tips for minikube

If you are working with minikube locally, you will also need to copy volume/prometheus-conf.yml file inside the Minikube VM. When it starts, it will copy all the files from ~/.minikube/files on your host, into the /files directory in the VM.

cp volume/prometheus-conf.yml \

~/.minikube/files

⚠️ Depending on your operating system, the ~/.minikube/files directory might be somewhere else. If that’s the case, please modify the above command accordingly.

# Accessing Host’s Resources through hostPath Volumes

Learn about hostPath Volume type and try to access the host's resources through it.

**We'll cover the following**

* [Building docker images](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8w9j7qLwY6#Building-docker-images)
  + [Creating a pod with docker image](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8w9j7qLwY6#Creating-a-pod-with-docker-image)
  + [Creating a Pod with hostPath](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8w9j7qLwY6#Creating-a-Pod-with-hostPath)
    - [Looking into the Definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8w9j7qLwY6#Looking-into-the-Definition)
* [The hostPath volume](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8w9j7qLwY6#The-hostPath-volume)
  + [Types of mounts in hostPath](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8w9j7qLwY6#Types-of-mounts-in-hostPath)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8w9j7qLwY6#Try-it-yourself)

## Building docker images

Sooner or later, we’ll have to build our images. A simple solution would be to execute the docker image build command directly from a server. However, that might cause problems. Building images on a single host means that there is an uneven resource utilization and that there is a single point of failure. Wouldn’t it be better if we could build images anywhere inside a Kubernetes cluster?

Instead of executing the docker image build command, we could create a Pod based on the docker image. Kubernetes will make sure that the Pod is scheduled somewhere inside the cluster, thus distributing resource usage much better.

### Creating a pod with docker image

Let’s start with an elementary example. If we can list the images, we’ll prove that running docker commands inside containers works. Since, from Kubernetes’ point of view, Pods are the smallest entity, that’s what we’ll run.

kubectl run docker \

--image=docker:17.11 --restart=Never \

docker image ls

kubectl get pods

We created a Pod named docker and based it on the official docker image. Since we want to execute a one-shot command, we specified that it should Never restart. Finally, the container command is docker image ls. The second command lists all the Pods in the cluster (including failed ones).

The **output** of the latter command is as follows.

NAME READY STATUS RESTARTS AGE

docker 0/1 Error 0 1m

The output should show that the status is “Error”, thus indicating that there is a problem with the container we’re running. If, in your case, the status is not yet “Error”, Kubernetes is probably still pulling the image. In that case, please wait a few moments, and re-execute the kubectl get pods command.

Let’s take a look at the logs of the container.

kubectl logs -f docker

The **output** is as follows.

Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?

Docker consists of two main pieces. There is a client, and there is a server. When we executed docker image ls, we invoked the client which tried to communicate with the server through its API. The problem is that Docker server is not running in that container. What we should do is tell the client (inside a container) to use Docker server that is already running on the host.

By default, the client sends instructions to the server through the socket located in “/var/run/docker.sock”. We can accomplish our goal if we mount that file from the host into a container.

Before we try to enable communication between a Docker client in a container and Docker server on a host, we’ll delete the Pod we created a few moments ago.

kubectl delete pod docker

### Creating a Pod with hostPath

Let’s mount the file /var/run/docker.sock from the host in our Pod.

#### Looking into the Definition

Let’s take a look at the Pod definition stored in docker.yml.

apiVersion: v1

kind: Pod

metadata:

name: docker

spec:

containers:

- name: docker

image: docker:17.11

command: ["sleep"]

args: ["100000"]

volumeMounts:

- mountPath: /var/run/docker.sock

name: docker-socket

volumes:

- name: docker-socket

hostPath:

path: /var/run/docker.sock

type: Socket

Part of the definition closely mimics the kubectl run command we executed earlier. The only significant difference is in the volumeMounts and volumes sections.

**Line 9-10:** We changed the command and the arguments to sleep 100000. That will give us more freedom since we’ll be able to create the Pod, enter inside its only container, and experiment with different commands.

**Line 11:** The volumeMounts field is relatively straightforward and is the same no matter which type of Volume we’re using. In this section, we’re specifying the mountPath and the name of the volume. The former is the path we expect to mount inside this container. You’ll notice that we are not specifying the type of the volume nor any other specifics inside the VolumeMounts section. Instead, we simply have a reference to a volume called docker-socket.

**Line 14:** The Volume configuration specific to each type is defined in the volumes section. In this case, we’re using the hostPath Volume type.

## The hostPath volume

hostPath allows us to mount a file or a directory from a host to Pods and, through them, to containers. Before we discuss the usefulness of this type, we’ll have a short discussion about use-cases when this is not a good choice.

**Do not** use hostPath to store a state of an application. Since it mounts a file or a directory from a host into a Pod, it is not fault-tolerant. If the server fails, Kubernetes will schedule the Pod to a healthy node, and the state will be lost.

For our use case, hostPath works just fine. We’re not using it to preserve state, but to gain access to Docker server running on the same host as the Pod.

**Line 15-18:** The hostPath type has only **two** fields. The path represents the file or a directory we want to mount from the host. Since we want to mount a socket, we set the type accordingly. There are other types we could use.

### Types of mounts in hostPath

* The Directory type will mount a directory from the host. It must exist on the given path. If it doesn’t, we might switch to DirectoryOrCreate type which serves the same purpose. The difference is that DirectoryOrCreate will create the directory if it does not exist on the host.
* The File and FileOrCreate are similar to their Directory equivalents. The only difference is that this time we’d mount a file, instead of a directory.
* The other supported types are Socket, CharDevice, and BlockDevice. They should be self-explanatory. If you don’t know what character or block devices are, you probably don’t need those types.

These were the types of mounts supported by the hostPath.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl run docker \

--image=docker:17.11 --restart=Never \

docker image ls

kubectl apply -f docker.yaml

kubectl get pods

kubectl logs -f docker

kubectl delete pod docker

**Running the Pod after mounting hostPath**

Learn to create the Pod by mounting a Docker socket and playing around in it.

**We'll cover the following**

* [Creating and testing the pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AXz6zPjQoO#Creating-and-testing-the-pod)
* [Playing around with docker](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AXz6zPjQoO#Playing-around-with-docker)
* [Destroying the Pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AXz6zPjQoO#Destroying-the-Pod)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AXz6zPjQoO#Try-it-yourself)

**Creating and testing the pod**

Let’s create the Pod and check whether, this time, we can execute Docker commands from inside the container it’ll create.

kubectl create -f docker.yml

Since the image is already pulled, starting the Pod should be almost instant.

Let’s see whether we can retrieve the list of Docker images.

kubectl exec -it docker \

-- docker image ls \

--format "{{.Repository}}"

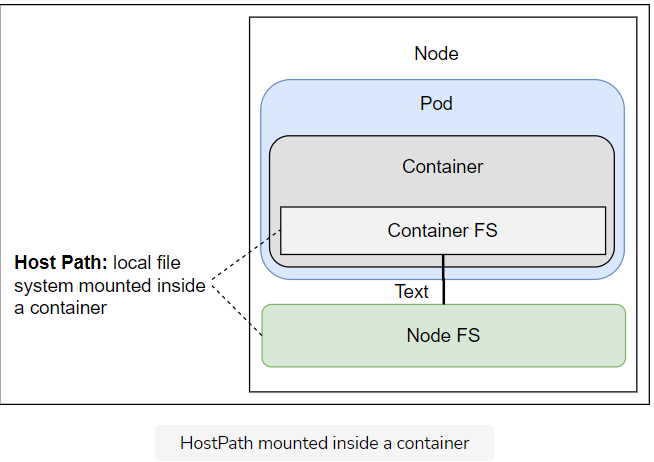
We executed docker image ls command and shortened the output by limiting its formatting only to Repository. The output is as follows. The output of the file may vary because of addons enabled on the cluster.

rancher/k3d-tools

rancher/k3d-proxy

rancher/k3s

Even though we executed the docker command inside a container, the output clearly shows the images from the host. We proved that mounting the Docker socket (/var/run/docker.sock) as a Volume allows communication between Docker client inside the container, and Docker server running on the host.



## Playing around with docker

Let’s enter the container and see whether we can build a Docker image.

kubectl exec docker -it – sh

To build an image, we need a Dockerfile as well as an application’s source code. We’ll continue using go-demo-2 as the example, so our first action will be to clone the repository.

apk add -U git

git clone \

https://github.com/Faizan-Zia/go-demo-2

cd go-demo-2

We used apk add to install git.On the other hand, docker and many other images use alpine as the base. If you’re not familiar with alpine, it is a very slim and efficient base image, and we strongly recommend that you use it when building your own.

Images like debian, centos, ubuntu, redhat, and similar base images are often a terrible choice made because of a misunderstanding of how containers work.

alpine uses apk package management, so we invoked it to install git. Next, we cloned the vfarcic/go-demo-2 repository, and, finally, we entered into the go-demo-2 directory.

Let’s take a quick look at the Dockerfile.

FROM golang:1.12.6 AS build

ADD . /src

WORKDIR /src

RUN apt update && apt install ca-certificates libgnutls30

RUN go get -d -v

RUN go test --cover -v ./... --run UnitTest

RUN go build -v -o go-demo

FROM alpine:3.4

MAINTAINER Viktor Farcic <viktor@farcic.com>

RUN mkdir /lib64 && ln -s /lib/libc.musl-x86\_64.so.1 /lib64/ld-linux-x86-64.so.2

EXPOSE 8080

ENV DB db

CMD ["go-demo"]

HEALTHCHECK --interval=10s CMD wget -qO- localhost:8080/demo/hello

COPY --from=build /src/go-demo /usr/local/bin/go-demo

RUN chmod +x /usr/local/bin/go-demo

Since this course is dedicated to Kubernetes, we won’t go into details behind this Dockerfile, but only comment that it uses Docker’s multi-stage builds. The first stage downloads the dependencies, it runs unit tests, and it builds the binary. The second stage starts over. It builds a fresh image with the go-demo binary copied from the previous stage.

ℹ️ We hope you’re proficient with Docker and there’s no need to explain image building further.

Let’s test whether building an image indeed works.

docker image build -t vfarcic/go-demo-2:beta .

docker image ls --format "{{.Repository}}"

We executed the docker image build command, followed by docker image ls. The **output** of the latter command is as follows. As mentioned earlier it may vary from system to system.

vfarcic/go-demo-2

<none>

golang

rancher/k3d-tools

rancher/k3d-proxy

rancher/k3s

alpine

If we compare this with the previous docker image ls output, we’ll notice that, this time, a few new images are listed. The “golang” and “alpine” images are used as a basis for each of the build stages. The “vfarcic/go-demo-2” is the result of our build. Finally, “” is only a left-over of the process and it can be safely removed.

docker system prune -f

docker image ls --format "{{.Repository}}"

The docker system prune command removes all unused resources. At least, all those created and unused by Docker. We confirmed that by executing docker image ls again. This time, we can see the <none> image is gone.

## Destroying the Pod

We’ll destroy the docker Pod and explore other usages of the hostPath Volume type.

exit

kubectl delete -f docker.yml

hostPath is a great solution for accessing host resources like /var/run/docker.sock, /dev/cgroups, and others. That is, as long as the resource we’re trying to reach is on the same node as the Pod.

Let’s see whether we can find other use-cases for hostPath.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f docker.yml

kubectl exec -it docker -- docker image ls --format "{{.Repository}}"

kubectl exec docker -it -- sh

apk add -U git

git clone https://github.com/Faizan-Zia/go-demo-2

cd go-demo-2

docker image build -t vfarcic/go-demo-2:beta .

docker image ls --format "{{.Repository}}"

docker system prune -f

docker image ls --format "{{.Repository}}"

exit

kubectl delete -f docker.yml

# Using hostPath Volume Type to Inject Configuration Files

Explore Prometheus and configure it with hostPath Volume.

**We'll cover the following**

* [Using Prometheus](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Using-Prometheus)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Looking-into-the-definition)
  + [Configuring the IP](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Configuring-the-IP)
  + [Testing prometheus](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Testing-prometheus)
  + [Changing Prometheus configuration](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Changing-Prometheus-configuration)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Try-it-yourself)
* [Troubleshooting tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Troubleshooting-tips-for-minikube)
  + [Configuring IP](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYKGEZWD2KD#Configuring-IP)

## Using Prometheus

We are about to deploy [Prometheus](https://prometheus.io/) for the first time. We won’t go into details behind the application except to say that it’s fantastic and that you should consider it for your monitoring and alerting needs. We’re using it only to demonstrate a few Kubernetes concepts. We’re not trying to learn how to operate it.

### Looking into the definition

Let’s take a look at the application’s definition.

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: prometheus

annotations:

kubernetes.io/ingress.class: "nginx"

ingress.kubernetes.io/ssl-redirect: "false"

nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:

rules:

- http:

paths:

- path: /prometheus

pathType: ImplementationSpecific

backend:

service:

name: prometheus

port:

number: 9090

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: prometheus

spec:

selector:

matchLabels:

type: monitor

service: prometheus

strategy:

type: Recreate

template:

metadata:

labels:

type: monitor

service: prometheus

spec:

containers:

- name: prometheus

image: prom/prometheus:v2.0.0

command:

- /bin/prometheus

args:

- "--config.file=/etc/prometheus/prometheus.yml"

- "--storage.tsdb.path=/prometheus"

- "--web.console.libraries=/usr/share"

- "--web.external-url=http://192.168.99.100/prometheus"

---

apiVersion: v1

kind: Service

metadata:

name: prometheus

spec:

ports:

- port: 9090

selector:

type: monitor

service: Prometheus

There’s nothing genuinely new in that YAML file. It defines an Ingress, a Deployment, and a Service. There is, however, one thing we might need to change.

### Configuring the IP

Prometheus needs a full external-url if we want to change the base path. For practicing on our platform the URL is set to Educative at **line 49** in the following definition of prometheus.yml.

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: prometheus

annotations:

kubernetes.io/ingress.class: "nginx"

ingress.kubernetes.io/ssl-redirect: "false"

nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:

rules:

- http:

paths:

- path: /prometheus

pathType: ImplementationSpecific

backend:

service:

name: prometheus

port:

number: 9090

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: prometheus

spec:

selector:

matchLabels:

type: monitor

service: prometheus

strategy:

type: Recreate

template:

metadata:

labels:

type: monitor

service: prometheus

spec:

containers:

- name: prometheus

image: prom/prometheus:v2.0.0

command:

- /bin/prometheus

args:

- "--config.file=/etc/prometheus/prometheus.yml"

- "--storage.tsdb.path=/prometheus"

- "--web.console.libraries=/usr/share"

- "--web.external-url=https://ed-6069882977779712\_dc.educative.run/prometheus"

---

apiVersion: v1

kind: Service

metadata:

name: prometheus

spec:

ports:

- port: 9090

selector:

type: monitor

service: Prometheus

Now that the URL is updated let's create deployments.

kubectl create -f prometheus.yml

kubectl rollout status deploy Prometheus

Once we created the application, we used the kubectl rollout status command to confirm that the deployment finished.

### Testing prometheus

Now we can open Prometheus in a browser.

kubectl port-forward service/prometheus --address 0.0.0.0 3000:9090

#open the link besides the run button

At first glance, the application seems to be running correctly. However, since the targets are the crucial part of the application, we should check them as well. For those not familiar with Prometheus, it pulls data from targets (external data sources) and, by default, comes with only one target pre-configured: Prometheus itself. Prometheus will always pull data from this target unless we configure it otherwise.

Let’s take a look at its targets. While port-forward is still running open the following in the browser or press the link beside run button in the playground and add /targets at the end.

# open the link besides run button and add /targets at the end

# make sure port-forward is running

There’s something wrong. The default target is not reachable. Before we start panicking, we should take a closer look at its configuration.

# open the link besides run button and add /config at the end

# make sure port-forward is running

The problem is with the metrics\_path field. By default, it is set to /metrics. However, since we changed the base path to /prometheus, the field should have /prometheus/metrics as the value.

### Changing Prometheus configuration

Long story short, we must change the Prometheus configuration.

We could, for example, enter the container, update the configuration file, and send the reload request to Prometheus. That would be a terrible solution since it would last only until the next time we update the application, or until the container fails, and Kubernetes decides to reschedule it.

Let’s explore alternative solutions. We could, for example, use hostPath Volume for this as well. If we can guarantee that the correct configuration file is inside the cluster, the Pod could attach it to the prometheus container. Let’s try it out. The output of the prometheus-hostpath with relevant segments is shown below.

apiVersion: apps/v1

kind: Deployment

metadata:

name: prometheus

spec:

selector:

...

spec:

containers:

...

volumeMounts:

- mountPath: /etc/prometheus/prometheus.yml

name: prom-conf

volumes:

- name: prom-conf

hostPath:

path: /files/prometheus-conf.yml

type: File

...

The only significant difference, when compared with the previous definition, is in the added volumeMounts and volumesfields. We’re using the same schema as before, except that, this time, the type is set to File. Once we apply this Deployment, the file /files/prometheus-conf.yml on the host will be available as /etc/prometheus/prometheus.yml inside the container.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f prometheus.yml

kubectl rollout status deploy prometheus

kubectl port-forward service/prometheus --address 0.0.0.0 3000:9090

# click the link beside run button

# open the link beside the run button and add /targets at the end

# open the link beside the run button and add /config at the end

## Troubleshooting tips for minikube

Following are some steps you may want to follow while working with minikube clusters.

### Configuring IP

Prometheus needs a full external-url if we want to change the base path. At the moment, it’s set to the IP of our Minikube VM. In your case, that IP might be different. We’ll fix that by adding a bit of sed “magic” that will make sure the IP matches that of your Minikube VM.

cat volume/prometheus.yml | sed -e \

"s/192.168.99.100/$(minikube ip)/g" \

| kubectl create -f - \

--record --save-config

kubectl rollout status deploy Prometheus

We output the contents of the volume/prometheus.yml file, we used sed to replace the hard-coded IP with the actual value of your Minikube instance, and we passed the result to kubectl create.

📝 Please note that, this time, the create command has dash (-) instead of the path to the file. That’s an indication that stdin should be used instead.

If you recall, we copied one file to the ~/.minikube/files directory, and Minikube copied it to the /files directory inside the VM.

In some cases, files might end up being copied to the VM’s root (/), instead of to /files. If this has happened to you, please enter the VM (minikube ssh), and move the files to /files, by executing the commands that follow (**only if the /files directory does not exist or is empty**).

We output the contents of the volume/prometheus.yml file, we used sed to replace the hard-coded IP with the actual value of your Minikube instance, and we passed the result to kubectl create.

#Run the folloiwng commands one by one

minikube ssh

sudo mkdir /files

sudo mv /prometheus-conf.yml /files/

exit

The time has come to take a look at the content of the file.

minikube ssh

sudo mkdir /files

sudo mv /prometheus-conf.yml /files/

minikube ssh sudo chmod +rw \

/files/prometheus-conf.yml

minikube ssh cat \

/files/prometheus-conf.yml

Exit

We changed the permissions of the file and displayed its content.

The time has come to take a look at the content of the file. The output is as follows.

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: prometheus

metrics\_path: /prometheus/metrics

static\_configs:

- targets:

- localhost:9090

This configuration is almost identical to what Prometheus uses by default. The only difference is in the metrics\_path, which is now pointing to /prometheus/metrics.

# Working with the New Prometheus Configuration

Explore the modified configuration of Prometheus and look into the dos and don'ts of hostPath.

**We'll cover the following**

* [Testing the new configuration](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY1mmlMA3Og#Testing-the-new-configuration)
* [Where to use hostPath](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY1mmlMA3Og#Where-to-use-hostPath)
* [Exploring the solutions](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY1mmlMA3Og#Exploring-the-solutions)
* [Destroying the pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY1mmlMA3Og#Destroying-the-pod)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY1mmlMA3Og#Try-it-yourself)
* [Troubleshooting tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY1mmlMA3Og#Troubleshooting-tips-for-minikube)

## Testing the new configuration

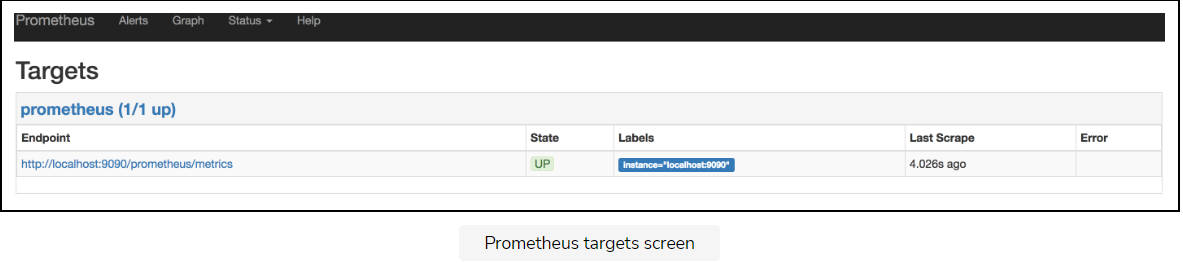
Let’s see whether Prometheus with the new configuration works as expected. As described earlier the external-url is also changed for prometheus-host-path.yml just like prometheus. You can confirm it in the playground at the end of this lesson at **line 49** in the prometheus-host-path.yml file.

kubectl apply -f volume/prometheus-host-path.yml

kubectl rollout status deploy prometheus

kubectl port-forward service/prometheus --address 0.0.0.0 3000:9090

<http://0.0.0.0:3000/prometheus/targets>



## Where to use hostPath

The next logical step would be to configure Prometheus with additional targets. Specifically, you may want to configure it to fetch metrics that are already made available through the Kubernetes API. We, however, will NOT be doing this. First of all, this chapter is not about monitoring and alerting. The second, and the more important reason, is that using the hostPath Volume type to provide configuration is NOT a good idea.

A hostPath Volume maps a directory from a host to where the Pod is running. Using it to “inject” configuration files into containers would mean that we’d have to make sure that the file is present on every node of the cluster.

Working with a single-node cluster can be potentially misleading. The fact that we’re running a single-node cluster means that every Pod we run will be scheduled on one node. Copying a configuration file to that single node, as we did in our example, ensures that it can be mounted in any Pod. However, the moment we add more nodes to the cluster, we’d experience side effects. We’d need to make sure that each node in our cluster has the same file we wish to mount, as we would not be able to predict where individual Pods would be scheduled. This would introduce far too much unnecessary work and added complexity.

## Exploring the solutions

An alternative solution would be to mount an NFS drive to all the nodes and store the file there. That would provide the guarantee that the file will be available on all the nodes, as long as we do NOT forget to mount NFS on each.

Another solution could be to create a custom Prometheus image. It could be based on the official image, with a single COPY instruction that would add the configuration. The advantage of that solution is that the image would be entirely immutable. Its state would not be polluted with unnecessary Volume mounts. Anyone could run that image and expect the same result. That is my preferred solution. However, in some cases, you might want to deploy the same application with a slightly different configuration. Should we, in those cases, fall back to mounting an NFS drive on each node and continue using hostPath?

Even though mounting an NFS drive would solve some of the problems, it is still not a great solution. In order to mount a file from NFS, we need to use the [nfs](https://kubernetes.io/docs/concepts/storage/volumes/" \l "nfs" \t "_blank) Volume type instead of hostPath. Even then it would be a sub-optimal solution. A much better approach would be to use configMap. We’ll explore it in the next chapter.

**Do** use hostPath to mount host resources like /var/run/docker.sock and /dev/cgroups. **Do not** use it to inject configuration files or store the state of an application.

## Destroying the pod

We’ll move onto a more exotic Volume type. But, before that, we’ll remove the Pod we’re currently running.

kubectl delete \

-f volume/prometheus-host-path.yml

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl apply -f volume/prometheus-host-path.yml

kubectl rollout status deploy prometheus

kubectl port-forward service/prometheus --address 0.0.0.0 3000:9090

kubectl delete -f volume/prometheus-host-path.yml

**Your app can be found at:**<https://ed-069882977779712_dc.educative.run/prometheus/targets>

## Troubleshooting tips for minikube

If you are working with minikube locally, please follow these steps to update the **IP**.

cat volume/prometheus-host-path.yml | sed -e "s/192.168.99.100/$(minikube ip)/g" | kubectl apply -f -

kubectl rollout status deploy prometheus

open http://$(minikube ip)/prometheus/targets

We applied the new definition (after the sed “magic”), we waited until the rollout finished, and we then opened the Prometheus targets in a browser. This time, with the updated configuration, Prometheus is successfully pulling data from the only target currently configured.

# Non-Persisting State

Learn how to deploy Jenkins and analyze its state

**We'll cover the following**

* [Deploying Jenkins](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R1pvENn2r3w#Deploying-Jenkins)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R1pvENn2r3w#Looking-into-the-definition)
  + [Creating the objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R1pvENn2r3w#Creating-the-objects)
* [Killing the Pod and Analysing the State](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R1pvENn2r3w#Killing-the-Pod-and-Analysing-the-State)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/R1pvENn2r3w#Try-it-yourself)

## Deploying Jenkins

This time we’ll deploy Jenkins and see what challenges we will face.

### Looking into the definition

Let’s take a look at the jenkins.yml definition.

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: jenkins

annotations:

kubernetes.io/ingress.class: "nginx"

ingress.kubernetes.io/ssl-redirect: "false"

nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:

rules:

- http:

paths:

- path: /jenkins

pathType: ImplementationSpecific

backend:

service:

name: jenkins

port:

number: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: jenkins

spec:

selector:

matchLabels:

type: master

service: jenkins

strategy:

type: Recreate

template:

metadata:

labels:

type: master

service: jenkins

spec:

containers:

- name: jenkins

image: vfarcic/jenkins

env:

- name: JENKINS\_OPTS

value: --prefix=/jenkins

---

apiVersion: v1

kind: Service

metadata:

name: jenkins

spec:

ports:

- port: 8080

selector:

type: master

service: Jenkins

There’s nothing special in that YAML file. It defines an Ingress with /jenkins path, a Deployment, and a Service. We won’t waste time with it. Instead, we’ll move on and create the objects.

### Creating the objects

Its time to create the deployments.

kubectl create -f jenkins.yml --record --save-config

kubectl rollout status deploy Jenkins

We created the objects and waited until the processes finished. Now we can open Jenkins in our browser of choice.

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

# click the link besides run button

Jenkins UI opened, thus confirming that the application is deployed correctly. Jenkins’ primary function is to execute jobs, so it’s only fair to create one.

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#click link besides run button and add /newjob at the end

Please type test in the item name field, select Pipeline as the type, and click the OK button.

There’s no need to make the Pipeline do any specific set of tasks. For now, you should be fine if you just Save the job.

## Killing the Pod and Analysing the State

Let’s explore what happens if the main process inside the Jenkins container dies.

POD\_NAME=$(kubectl get pods -l service=jenkins,type=master -o jsonpath="{.items[\*].metadata.name}")

kubectl exec $POD\_NAME -it -- kill 1

We retrieved the name of the Pod, and we used it to execute kill 1 inside its only container. The result is a simulation of a failure. Soon afterward, Kubernetes detected the failure and recreated the container. Let’s double-check all that.

kubectl get pods

The **output** is as follows.

NAME READY STATUS RESTARTS AGE

jenkins-76d59945d8-zcz8m 1/1 Running 1 12m

We can see that a container is running. Since we killed the main process and, with it, the first container, the number of restarts was increased to one.

Let’s go back to Jenkins UI and check what happened to the job. I’m sure you already know the answer, but we’ll double-check it anyways.

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

As expected, the job we created is gone. When Kubernetes recreated the failed container, it created a new one from the same image. Everything we generated inside the running container is no more. We reset to the initial state.

## Try it yourself

For your ease, all the commands used in this lesson are given below.

kubectl create -f jenkins.yml --record --save-config

kubectl rollout status deploy jenkins

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

# click the link besides run button

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#click link besides run button and add /newJob at the end

POD\_NAME=$(kubectl get pods -l service=jenkins,type=master

-o jsonpath="{.items[\*].metadata.name}")

kubectl exec $POD\_NAME -it -- kill 1

kubectl get pods

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#click the link besides run button

# Persisting State through the emptyDir Volume Type

Analyze the state of an updated Jenkins Deployment and discuss emptyDir Volume type.

**We'll cover the following**

* [Updating the Jenkins deployment definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/39ML7WLOL0O#Updating-the-Jenkins-deployment-definition)
* [Persisting state](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/39ML7WLOL0O#Persisting-state)
* [The emptyDir Volume](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/39ML7WLOL0O#The-emptyDir-Volume)
* [Destroying everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/39ML7WLOL0O#Destroying-everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/39ML7WLOL0O#Try-it-yourself)

## Updating the Jenkins deployment definition

Let’s take a look at a slightly updated YAML definition of jenkins-empty-dir.yml.

...

kind: Deployment

...

spec:

...

template:

...

spec:

containers:

...

volumeMounts:

- mountPath: /var/jenkins\_home

name: jenkins-home

volumes:

- emptyDir: {}

name: jenkins-home

...

We added a mount that references the jenkins-home Volume. The Volume type is, this time, emptyDir. We’ll discuss the new Volume type soon. But, before we dive into explanations, we’ll try to experience its effects.

kubectl apply \

-f jenkins-empty-dir.yml

kubectl rollout status deploy Jenkins

We applied the new definition and waited until the rollout finished.

Now we can open the New Job Jenkins screen and repeat the same process we followed before.

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#Open the link beside run button

Please type test in the item name field, select Pipeline as the type, click the OK button, and finish by clicking the Save button.

Now we’ll kill the container and see what happens.

POD\_NAME=$(kubectl get pods \

-l service=jenkins,type=master \

-o jsonpath="{.items[\*].metadata.name}")

kubectl exec -it $POD\_NAME kill 1

kubectl get pods

The **output** should show that there is a container running or, in other words, that Kubernetes detected the failure and created a new container.

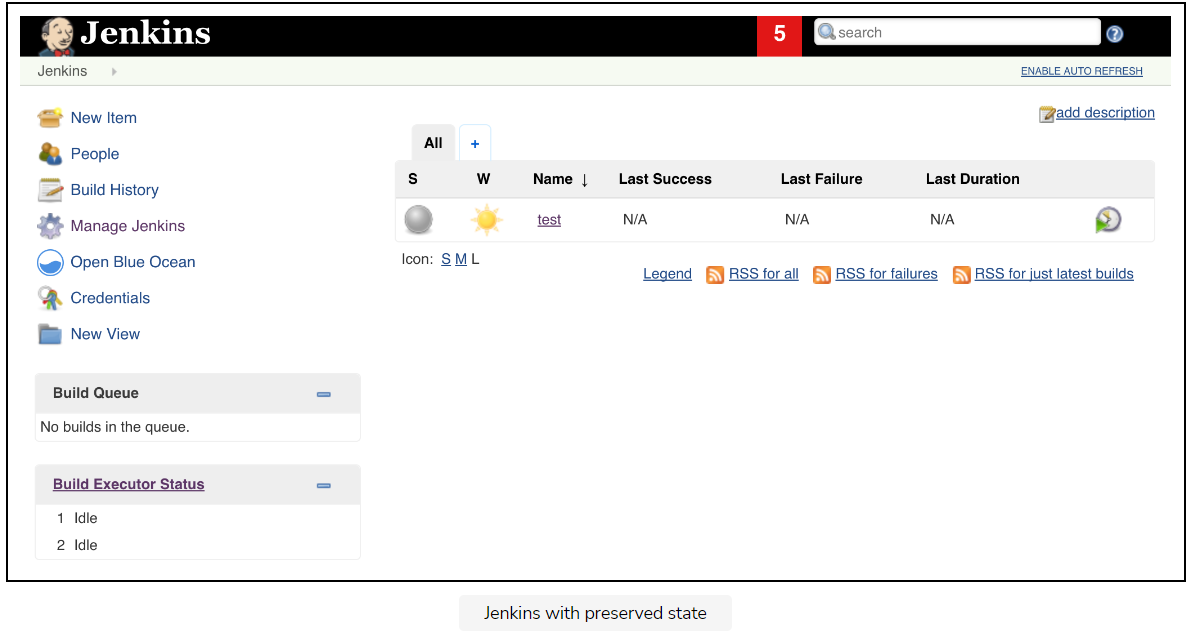
## Persisting state

Finally, let’s open Jenkins’ Home screen one more time.

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#Open the link beside run button without the /newJob

This time, the test job is there. The state of the application was preserved even when the container failed, and Kubernetes created a new one.



## The emptyDir Volume

Now let’s talk about the emptyDir Volume. It is considerably different from those we explored thus far.

An emptyDir Volume is created when a Pod is assigned to a node. It will exist for as long as the Pod continues running on that server.

What that means is that emptyDir can survive container failures. When a container crashes, a Pod is not removed from the node. Instead, Kubernetes will recreate the failed container inside the same Pod and, thus, preserve the emptyDir Volume. All in all, this Volume type is only partially fault-tolerant.

If emptyDir is not entirely fault-tolerant, you might be wondering why we are discussing it in the first place.

The emptyDir Volume type is the closest we can get to fault-tolerant volumes without using a network drive. Since we do not have any, we had to resort to emptyDir as the closest-we-can-get-to-fault-tolerant-persistence type of Volume.

As you start deploying third-party applications, you’ll discover that many of them come with the recommended YAML definition. If you pay closer attention, you’ll notice that many are using the emptyDir Volume type. It’s not that emptyDir is the best choice, but that it all depends on your needs, your hosting provider, your infrastructure, and quite a few other things.

There is no one-size-fits-all type of persistent and fault-tolerant Volume type. On the other hand, emptyDir always works. Since it has no external dependencies, it is safe to put it as an example, with the assumption that people will change to whichever type fits them better.

There is an unwritten assumption that emptyDir is used for testing purposes, and will be changed to something else before it reaches production.

As long as we’re using Minikube or k3d to create a Kubernetes cluster, we’ll use emptyDir as a solution for persistent volumes. Do not despair. Later on, once we move into a “more serious” cluster setup, we’ll explore better options for persisting state.

## Destroying everything

For now, we’ll just destroy the cluster and take a break.

k3d cluster delete mycluster –all

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl apply \

-f jenkins-empty-dir.yml

kubectl rollout status deploy jenkins

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#Open the link beside run button

#add a new job

POD\_NAME=$(kubectl get pods \

-l service=jenkins,type=master \

-o jsonpath="{.items[\*].metadata.name}")

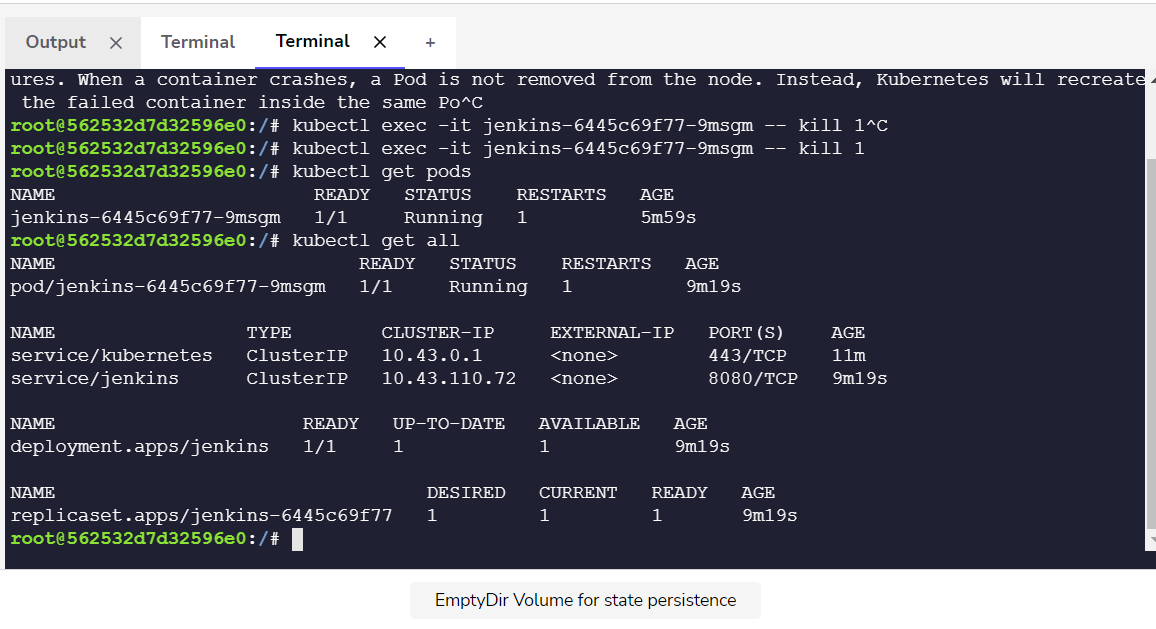
kubectl exec $POD\_NAME -- kill 1

kubectl get pods

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#Open the link beside run button and remove /newJob at the end

k3d cluster delete mycluster –all



# What's Next?

Recap what we have learned so far and what we are going to learn next.

**We'll cover the following**

* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLVEpjmKvDR#Summary)

## Summary

With the exception of emptyDir, our choice of Volume type demonstrated in this chapter was not simply based on the ability to use them in a Minikube cluster. Each of these three Volume types will be an essential piece in the chapters that follow.

* We’ll use hostPath to access Docker server from inside containers.
* The gitRepo Volume type will be very significant once we start designing a continuous deployment pipeline.
* The emptyDir type will be required as long as we’re using Minikube. Until we have a better solution for creating a Kubernetes cluster, emptyDir will continue to be used in our Minikube examples.

We have only scratched the surface with Volumes. There are at least two more that we should explore inside Minikube, and one when we change to a different solution for creating a cluster.

The Volumes that we’ll explore throughout the rest of the course are long enough subjects to deserve a separate chapter or, as we already mentioned, require that we get rid of Minikube.

ℹ️ If you’d like to know more about Volumes, please explore [Volume v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#volume-v1-core) API documentation

