Running the Pod after mounting hostPath

In this lesson, we will create the Pod by mounting Docker socket and play around in it.

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 volume/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.

```
k8s.gcr.io/kube-proxy
k8s.gcr.io/kube-controller-manager
k8s.gcr.io/kube-scheduler
k8s.gcr.io/kube-apiserver
quay.io/kubernetes-ingress-controller/nginx-ingress-controller
k8s.gcr.io/kube-addon-manager
k8s.gcr.io/coredns
k8s.gcr.io/kubernetes-dashboard_amd64
```

```
k8s.gcr.io/etcd
k8s.gcr.io/k8s-dns-sidecar-amd64
k8s.gcr.io/k8s-dns-kube-dns-amd64
k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64
k8s.gcr.io/pause
docker
gcr.io/k8s-minikube/storage-provisioner
gcr.io/google_containers/defaultbackend
```

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.

HostPath mounted inside a container
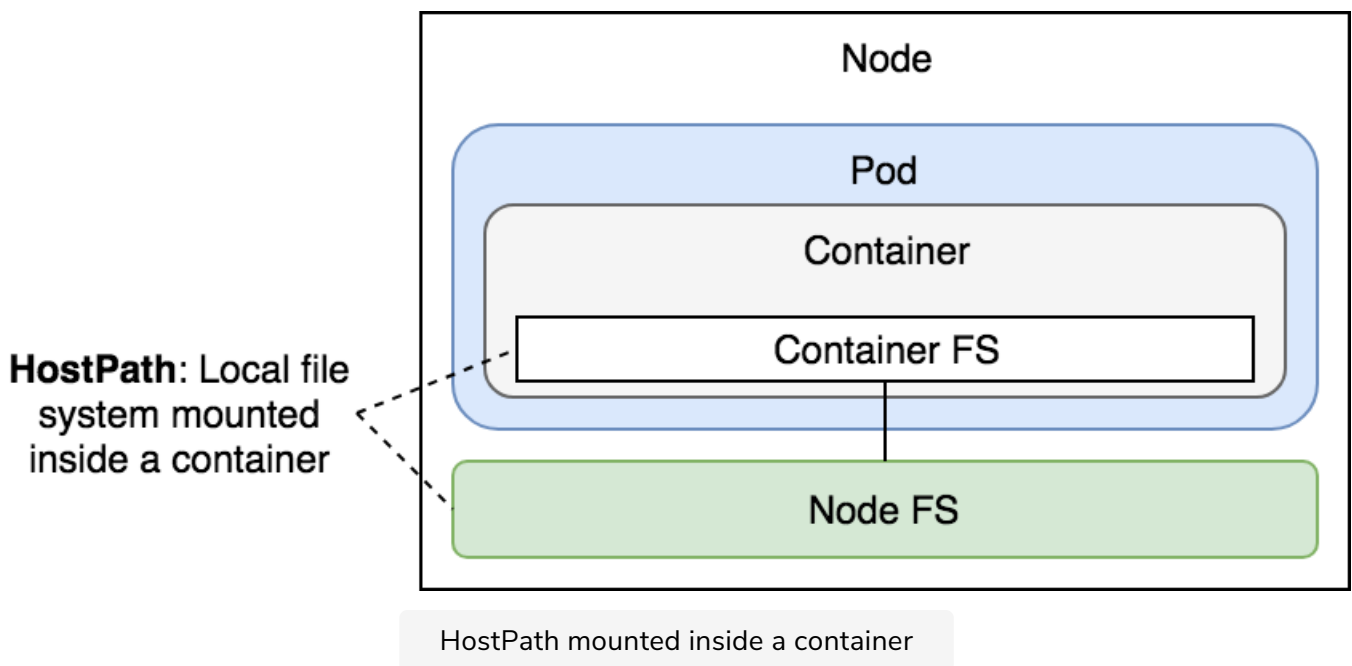
Playing Around with Docker

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

```
kubectl exec -it docker 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/vfarcic/go-demo-2.git
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`.

```
cat Dockerfile
```

The **output** is as follows.

```
FROM golang:1.9 AS build
ADD . /src
WORKDIR /src
RUN go get -d -v -t
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.

> **i** 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.

```
vfarcic/go-demo-2
<none>
golang
docker
alpine
gcr.io/google_containers/nginx-ingress-controller
gcr.io/google_containers/k8s-dns-sidecar-amd64
gcr.io/google_containers/k8s-dns-kube-dns-amd64
gcr.io/google_containers/k8s-dns-dnsmasq-nanny-amd64
gcr.io/google_containers/kubernetes-dashboard-amd64
gcr.io/google_containers/kubernetes-dashboard-amd64
gcr.io/google-containers/kube-addon-manager
gcr.io/google_containers/defaultbackend
gcr.io/google_containers/pause-amd64
```

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, `<none>` 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 volume/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`.

In the next lesson, we learn to use hostPath Volume type to inject configuration files.