Accessing Host’s Resources through hostPath Volumes

In this lesson, we will go through the hostPath Volume type and try to access the host's resources through it.

WE'LL COVER THE FOLLOWING

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 \
    --generator "run-pod/v1" \
    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 docker
```

The **output** is as follows.

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

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 (Minikube VM).

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 `volume/docker.yml`.

```
cat volume/docker.yml
```

The **output** is as follows.

```
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

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.

In the next lesson, we will run the Pod with socket type hostPath volume mounted in it.