# Working with the New Prometheus Configuration

In this lesson, we will test the modified configuration of Prometheus and look into the dos and don'ts of hostPath.

WE'LL COVER THE FOLLOWING



## Testing the New Configuration #

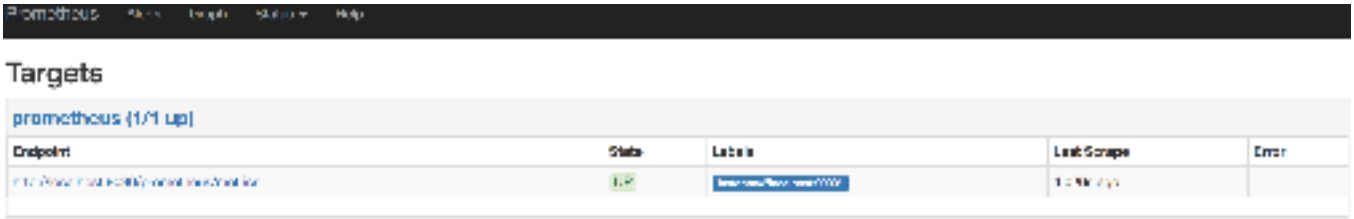
Let's see whether Prometheus with the new configuration works as expected.

```
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.




Prometheus targets screen

## 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 Minikube 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`?

`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 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
```