A Quick and Dirty Way to Run Pods

In this lesson, we will create and run a Pod in an imperative way.

WE'LL COVER THE FOLLOWING

Creating a Pod with Mongo

Just as we can execute `docker run` to create containers, `kubectl` allows us to create Pods with a **single command**. For example, if we'd like to create a Pod with a *Mongo database*, the command is as follows.

```
kubectl run db --image mongo \
    --generator "run-pod/v1"
```

You'll notice that the output says that `pod/db` was `created`. We created our first Pod. We can confirm that by listing all the Pods in the cluster.

Verification

We have created a Pod. We can confirm that by listing all the Pods in the cluster.

```
kubectl get pods
```

The **output** is as follows.

```
NAME    READY   STATUS              RESTARTS   AGE
db      0/1     ContainerCreating   0          1m
```

In the **output**, we can see:

- The name of the Pod
- Its readiness
- The status
- The number of times it restarted
- For how long it has existed (its age)

If you were fast enough, or your network is slow, none of the pods might be ready. We expect to have **one** Pod, but there's **zero** running at the moment.

Since the `mongo` image is relatively big, it might take a while until it is pulled from *Docker Hub*. After a while, we can retrieve the Pods one more time to confirm that the Pod with the Mongo database is running.

```
kubectl get pods
```

The **output** is as follows.

```
NAME    READY    STATUS     RESTARTS   AGE
db      1/1      Running    0          6m
```

We can see that, this time, the Pod is ready and we can start using the *Mongo database*.

We can confirm that a container based on the `mongo` image is indeed running inside the cluster.

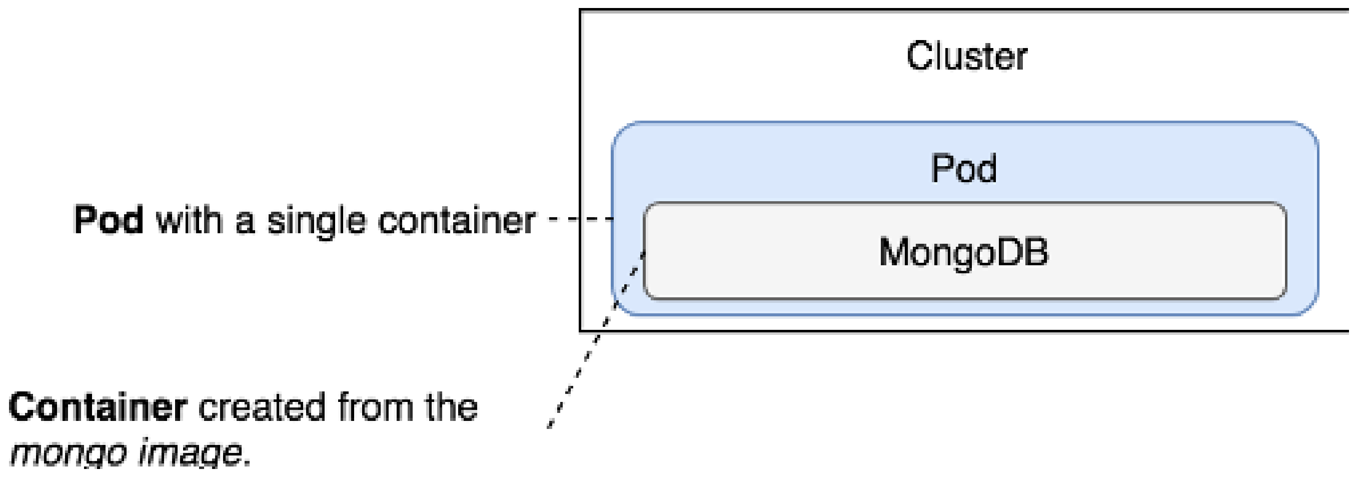
```
eval $(minikube docker-env)
docker container ls -f ancestor=mongo
```

We evaluated `minikube` variables so that our local Docker client is using Docker server running inside the VM. Further on, we listed all the containers based on the `mongo` image.

The **output** is as follows (IDs are removed for brevity).

```
CONTAINER ID IMAGE COMMAND                  CREATED        STATUS        PORTS NAMES
   ...       mongo "docker-entrypoint.s…" 5 minutes ago Up 5 minutes        k8s_db_db-...
```

As you can see, the container defined in the Pod is **running**.


Cluster
Pod
MongoDB
Pod with a single container
Container created from the *mongo image.*

A Pod with a single container

That was not the best way to run Pods so we'll delete it.

```
kubectl delete pod db
```

The **output** is as follows.

```
pod "db" deleted
```

Why This is a Dirty Way?

The above approach used to run Pods is not the best one. We used the imperative way to tell Kubernetes what to do. Even though there are cases when that might be useful, most of the time we want to leverage the declarative approach.

We want to have a way to define what we need in a file and pass that information to Kubernetes. That way, we can have a documented and repeatable process, that can (and should) be version controlled as well.

Moreover, the kubectl run was reasonably simple. In real life, we need to declare much more than the name of the deployment and the image. Commands like `kubectl` can quickly become too long and, in many cases, very complicated. Instead, we'll write specifications in *YAML* format.

In the next lesson, we’ll see how we can accomplish a similar result using the declarative syntax.