Playing Around with the Running Pod

In this lesson, we will play around with the Pod running in our Cluster.

WE'LL COVER THE FOLLOWING

Describing the Resources

In many cases, it is more useful to describe resources by referencing the file that defines them. That way there is no confusion nor need to remember the names of resources. Instead of using `kubectl describe pod db` we could have executed the command that follows:

```
kubectl describe -f pod/db.yml
```

The output should be the same as the previous lesson since, in both cases, `kubectl` sent a request to Kubernetes API requesting information about the Pod named `db`.

Executing a New Process

Just as with Docker, we can execute a new process inside a running container inside a Pod.

```
kubectl exec db ps aux
```

The **output** will be similar as follows.

```
USER PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root   1  0.5  2.9 967452 59692 ?   Ssl  21:47 0:03 mongod --rest --httpinterface
root  31  0.0  0.0  17504  1980 ?   Rs   21:58 0:00 ps aux
```

We told Kubernetes that we'd like to execute a process inside the first container of the Pod `db`. Since our Pod defines only one container, this container and the first container are one and the same. The `--container` (or `-c`) argument can be set to specify which container should be used. That is particularly useful when running multiple containers in a Pod.

Apart from using Pods as the reference, `kubectl exec` is almost the same as the `docker container exec` command. The significant difference is that `kubectl` allows us to execute a process in a container running in any node inside a cluster, while `docker container exec` is limited to containers running on a specific node.

Instead of executing a new short-lived process inside a running container, we can enter into it. For example, we can make the execution interactive with `-i` (`stdin`) and `-t` (terminal) arguments and run `shell` inside a container.

```
kubectl exec -it db sh
```

We're inside the `sh` process inside the container. Since the container hosts a Mongo database, we can, for example, execute `db.stats()` to confirm that the database is indeed running.

```
echo 'db.stats()' | mongo localhost:27017/test
```

We used `mongo` client to execute `db.stats()` for the database `test` running on `localhost:27017`. Since we're not trying to learn Mongo, the only purpose of this exercise was to prove that the database is up-and-running. Let's get out of the container.

```
exit
```

Getting Logs

Logs should be shipped from containers to a central location. However, since we did not yet explore that subject, it would be useful to be able to see logs of a container in a Pod.

The command that outputs logs of the only container in the `db` Pod is as follows:

```
kubectl logs db
```

The **output** is too big and not that important in its entirety. One of the last line is as follows.

```
...
2017-11-10T22:06:20.039+0000 I NETWORK  [thread1] waiting for connections on port 27017
...
```

With the `-f` (or `--follow`) we can follow the logs in real-time. Just as with the `exec` sub-command, if a Pod defines multiple containers, we can specify which one to use with the `-c` argument.

Exploring the Failure

What happens when a container inside a Pod dies?

Killing the Container

Let's simulate a failure and observe what happens.

```
kubectl exec -it db pkill mongod
kubectl get pods
```

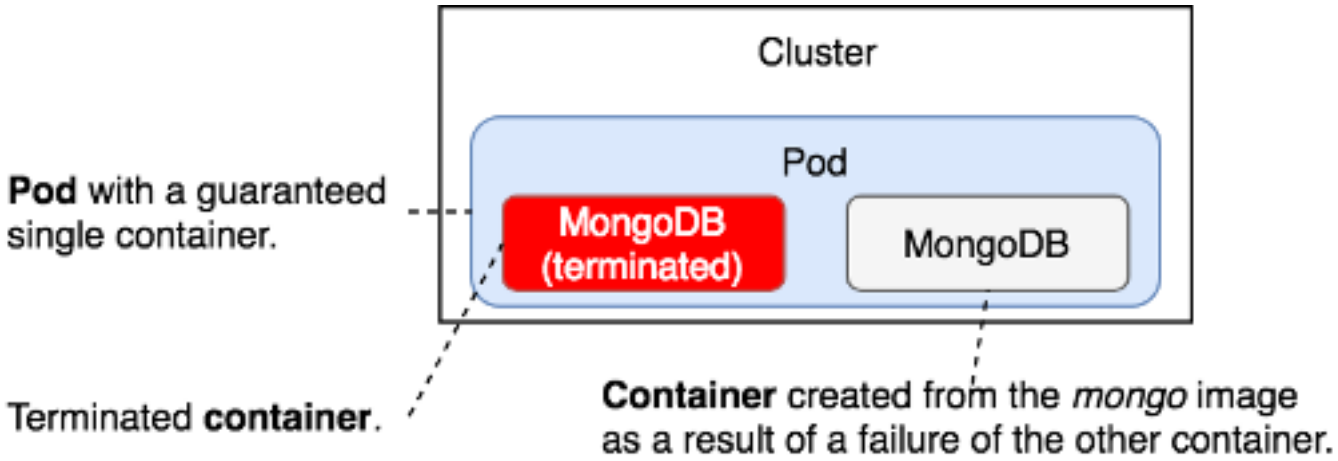
We killed the main process of the container and listed all the Pods. The **output** is as follows.

```
NAME READY STATUS  RESTARTS AGE
db   1/1   Running 1        13m
```

The container is running (`1/1`). Kubernetes guarantees that the containers

inside a Pod are (almost) always running. Please note that the `RESTARTS` field

now has the value of `1`. Every time a container fails, Kubernetes will restart it.



Pod with a failed container

Deleting the Pod

Finally, we can delete a Pod if we don't need it anymore.

```
kubectl delete -f pod/db.yml

kubectl get pods
```

We removed the Pods defined in `db.yml` and retrieved the list of all the Pods in the cluster. The **output** of the latter command is as follows.

```
NAME READY STATUS      RESTARTS AGE
db   0/1   Terminating 1        3h
```

The number of ready containers dropped to `0`, and the status of the `db` Pod is `terminating`.

When we sent the instruction to delete a Pod, Kubernetes tried to terminate it gracefully.

- The first thing it did was to send the `TERM` (terminate) signal to all the main processes inside the containers that form the Pod.
- From there on, Kubernetes gives each container a period of thirty seconds so that the processes in those containers can shut down

gracefully.

- Once the grace period expires, the `KILL` signal is sent to terminate all the main processes forcefully and, with them, all the containers. The default grace period can be changed through the `gracePeriodSeconds` value in YAML definition or `--grace-period` argument of the `kubectl delete` command.

If we repeat the `get pods` command thirty seconds after we issued the `delete` instruction, the Pod should be removed from the system.

```
kubectl get pods
```

This time, the **output** is different.

```
No resources found.
```

The only Pod we had in the system is no more!

In the next lesson, we will learn how to run multiple containers inside a Pod.