Operating ReplicaSets

In this lesson, we will explore the operating procedure of ReplicaSets and see its self-healing property in action.

Deleting ReplicaSets

What would happen if we delete the ReplicaSet? As you might have guessed, both the ReplicaSet and everything it created (the Pods) would **disappear** with a single `kubectl delete -f rs/go-demo-2.yml` command.

However, since ReplicaSets and Pods are loosely coupled objects with matching labels, we can remove one **without deleting** the other.

We can, for example, remove the ReplicaSet we created while leaving the two Pods intact.

```
kubectl delete -f rs/go-demo-2.yml \
    --cascade=false
```

We used the `--cascade=false` argument to prevent Kubernetes from removing all the downstream objects. As a result, we got the confirmation that `replicaset "go-demo-2"` was `deleted`.

Let's confirm that it is indeed removed from the system.

Let's confirm that it is indeed removed from the system.

```
kubectl get rs
```

As expected, the **output** states that `no resources` were `found`.

If `--cascade=false` indeed prevents Kubernetes from removing the downstream objects, the Pods should continue running in the cluster. Let's confirm the assumption.

```
kubectl get pods
```

The **output** is as follows.

```
NAME             READY STATUS  RESTARTS AGE
go-demo-2-md5xp 2/2   Running 0        9m
go-demo-2-vnmf7 2/2   Running 0        9m
```

The **two Pods** created by the ReplicaSet are indeed still running in the cluster even though we removed the ReplicaSet.

The Pods that are currently running in the cluster do not have any relation with the ReplicaSet we created earlier. We deleted the ReplicaSet, and the Pods are still there.

Knowing that the ReplicaSet uses labels to decide whether the desired number of Pods is already running in the cluster, should lead us to the conclusion that if we create the same ReplicaSet again, it should reuse the two Pods that are running in the cluster. Let's confirm that.

Re-using the Same Pods

In addition to the `kubectl create` command we executed previously, we'll also add the `--save-config` argument. It'll save the configuration of the ReplicaSet thus allowing us to perform a few additional operations later on. We'll get to them shortly. For now, the important thing is that we are about to create the same ReplicaSet we had before.

```
kubectl create -f rs/go-demo-2.yml \
    --save-config
```

The **output** states that the `replicaset "go-demo-2"` was `created`. Let's see what happened with the Pods.

```
kubectl get pods
```

The **output** is as follows.

```
NAME            READY STATUS  RESTARTS AGE
go-demo-2-md5xp 2/2   Running 0        10m
go-demo-2-vnmf7 2/2   Running 0        10m
```

If you compare the names of the Pods, you'll see that they are the **same as before** we created the ReplicaSet. It looked for matching labels, deduced that there are two Pods that match them, and decided that there's no need to create new ones. The matching Pods fulfill the desired number of replicas.

Updating the Definition

Since we saved the configuration, we can `apply` an updated definition of the ReplicaSet. For example, we can use `rs/go-demo-2-scaled.yml` file that differs only in the number of replicas set to `4`.

We could have created the ReplicaSet with `apply` in the first place, but we didn't. The `apply` command automatically saves the configuration so that we can edit it later on. The `create` command does not do such thing by default so we had to save it with `--save-config`.

```
kubectl apply -f rs/go-demo-2-scaled.yml
```

This time, the **output** is slightly different. Instead of saying that the ReplicaSet was created, we can see that it was `configured`.

Let's take a look at the Pods.

```
kubectl get pods
```

The **output** is as follows.

```
NAME            READY STATUS  RESTARTS AGE
go-demo-2-ckmtv 2/2   Running 0        50s
go-demo-2-lt4qm 2/2   Running 0        50s
go-demo-2-md5xp 2/2   Running 0        11m
go-demo-2-vnmf7 2/2   Running 0        11m
```

As expected, now there are **four Pods** in the cluster. If you pay closer attention to the names of the Pods, you'll notice that two of them are the same as before.

When we applied the new configuration with `replicas` set to `4` instead of `2`, Kubernetes updated the ReplicaSet which, in turn, evaluated the current state of the Pods with matching labels. It found two with the same labels and decided to create two more so that the new desired state can match the actual state.

Self-healing in Action

We have already discussed that ReplicsSets have self-healing property. Let's test this property by making a few changes to our system.

Destroying a Pod

Let's see what happens when a Pod is destroyed.

```
POD_NAME=$(kubectl get pods -o name \
    | tail -1)
kubectl delete $POD_NAME
```

We retrieved all the Pods and used `-o name` to retrieve only their names. The result was piped to `tail -1` so that only one of the names is output. The result is stored in the environment variable `POD_NAME`. The latter command used that variable to remove the Pod as a simulation of a failure.

Let's take another look at the Pods in the cluster.

```
kubectl get pods
```

The **output** is as follows.

```
NAME                READY   STATUS        RESTARTS   AGE
go-demo-2-ckmtv     2/2     Running       0          10m
go-demo-2-lt4qm     2/2     Running       0          10m
go-demo-2-md5xp     2/2     Running       0          13m
go-demo-2-t8sfs     2/2     Running       0          30s
go-demo-2-vnmf7     0/2     Terminating   0          13m
```

We can see that the Pod we deleted is `terminating`. However, since we have a ReplicaSet with `replicas` set to `4`, as soon as it discovered that the number of Pods dropped to `3`, it created a new one. We just witnessed **self-healing** in action.

> 📝 We get the final output after the system goes through several stages so your output might differ from the above.

As long as there are enough available resources in the cluster, ReplicaSets will make sure that the specified number of Pod replicas are (almost) always up-and-running.

Removing a label

Let's see what happens if we remove one of the Pod labels ReplicaSet uses in its selector.

```
POD_NAME=$(kubectl get pods -o name \
    | tail -1)
kubectl label $POD_NAME service-
kubectl describe $POD_NAME
```

We used the same command to retrieve the name of one of the Pods and executed the command that removed the label `service`.

> 𝐢 Please note `-` at the end of the name of the label. It is the syntax that indicates that a label should be removed.

Finally, we described the Pod.

The **output** of the last command, limited to the labels section, is as follows.

```
...
Labels: db=mongo
        language=go
        type=backend
...
```

As you can see, the label `service` is gone.

Now, let's list the Pods in the cluster and check whether there is any change.

```
kubectl get pods --show-labels
```

The **output** is as follows.

```
NAME            READY STATUS  RESTARTS AGE LABELS
go-demo-2-ckmtv 2/2   Running 0         24m db=mongo,language=go,service=go-demo-2,type=backen
go-demo-2-lt4qm 2/2   Running 0         24m db=mongo,language=go,service=go-demo-2,type=backen
go-demo-2-md5xp 2/2   Running 0         28m db=mongo,language=go,type=backend
go-demo-2-nrnbh 2/2   Running 0          4m db=mongo,language=go,service=go-demo-2,type=backen
go-demo-2-t8sfs 2/2   Running 0         15m db=mongo,language=go,service=go-demo-2,type=backen
```

The total number of Pods increased to **five**. The moment we removed the `service` label from one of the Pods, the ReplicaSet discovered that the number of Pods matching the `selector` labels is three and created a new Pod.

Right now, we have four Pods controlled by the ReplicaSet and one running freely due to non-matching labels.

Re-adding the Label

What would happen if we add the label we removed?

```
kubectl label $POD_NAME service=go-demo-2

kubectl get pods --show-labels
```

We added the `service=go-demo-2` label and listed all the Pods.

The **output** of the latter command is as follows.

```
NAME            READY STATUS      RESTARTS AGE LABELS
go-demo-2-ckmtv 2/2   Running     0         28m db=mongo,language=go,service=go-demo-2,type=ba
go-demo-2-lt4qm 2/2   Running     0         28m db=mongo,language=go,service=go-demo-2,type=ba
go-demo-2-md5xp 2/2   Running     0         31m db=mongo,language=go,service=go-demo-2,type=ba
go-demo-2-nrnbh 0/2   Terminating 0          7m db=mongo,language=go,service=go-demo-2,type=ba
```

```
go-demo-2-t8sfs 2/2    Running    0        18m db=mongo,language=go,service=go-demo-2,type=ba
```

The moment we added the label, the ReplicaSet discovered that there are five Pods with matching selector labels. Since the specification states that there should be four replicas of the Pod, it *removed* one of the Pods so that the desired state matches the actual state.

The previous few examples showed, one more time, that ReplicaSets and Pods are **loosely coupled** through matching labels and that ReplicaSets are using those labels to maintain the parity between the actual and the desired state.

So far, self-healing worked as expected.

In the next lesson, we will go through a quick quiz to test our understanding of ReplicaSets.