Creating a Zero-Downtime Deployment

In this lesson, we will create a zero-downtime deployment using the file defined in the previous lesson.

Creating Deployment

Before we explore rolling updates, we should create the Deployment and, with it, the first release of our application.

```
kubectl create \
    -f deploy/go-demo-2-api.yml \
    --record

kubectl get -f deploy/go-demo-2-api.yml
```

We created the Deployment and retrieved the object from the Kubernetes API server.

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

```
NAME          DESIRED  UP-TO-DATE AVAILABLE AGE
go-demo-2-api 3              3          3         1m
```

Please make sure that the number of available Pods is `3`. Wait for a few moments, if that's not the case. Once all the Pods are up-and-running, we'll have a Deployment that created a new ReplicaSet which, in turn, created three Pods based on the latest release of the `vfarcic/go-demo-2` image.

Let's see what happens when we set a new image.

```
kubectl set image \
    -f deploy/go-demo-2-api.yml \
    api=vfarcic/go-demo-2:2.0 \
    --record
```

There are a few ways we can observe what is happening during the update. One of those is through the `kubectl rollout status` command.

```
kubectl rollout status -w \
    -f deploy/go-demo-2-api.yml
```

The **output** is as follows.

```
...
deployment "go-demo-2-api" successfully rolled out
```

From the last entry, we can see that the rollout of the new deployment was successful. Depending on the time that passed between setting the new image and displaying the rollout status, you might have seen other entries marking the progress. However, I think that the events from the `kubectl describe` command are painting a better picture of the process that was executed.

```
kubectl describe \
    -f deploy/go-demo-2-api.yml
```

The last lines of the **output** are as follows.

```
...
Replicas: 3 desired | 3 updated | 3 total | 3 available | 0 unavailable
...
OldReplicaSets:  <none>
NewReplicaSet:   go-demo-2-api-68c75f4f5 (3/3 replicas created)
Events:
  Type   Reason            Age  From                  Message
  ----   ------            ---- ----                  -------
  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68c
  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68c
  Normal ScalingReplicaSet 2m   deployment-controller Scaled down replica set go-demo-2-api-6
  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68c
  Normal ScalingReplicaSet 2m   deployment-controller Scaled down replica set go-demo-2-api-6
  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68c
  Normal ScalingReplicaSet 2m   deployment-controller Scaled down replica set go-demo-2-api-6
```

We can see that the number of desired replicas is `3`. The same number was updated and all are available.

At the bottom of the output are events associated with the Deployment. The process started by increasing the number of replicas of the new ReplicaSet

(`go-demo-2-api-68c75f4f5`) to `1`. Next, it decreased the number of replicas of the old ReplicaSet (`go-demo-2-api-68df567fb5`) to `2`. The same process of increasing replicas of the new, and decreasing replicas of the old ReplicaSet continued until the new one got the desired number (`3`), and the old one dropped to zero.

There was no downtime throughout the process. Users would receive a response from the application no matter whether they sent it before, during, or after the update. The only important thing is that, during the update, a response might have come from the old or the new release. During the update process, both releases were running in parallel.

Let’s take a look at the rollout history.

```
kubectl rollout history \
    -f deploy/go-demo-2-api.yml
```

The **output** is as follows.

```
deployments "go-demo-2-api"
REVISION CHANGE-CAUSE
1        kubectl create --filename=deploy/go-demo-2-api.yml --record=true
2        kubectl set image api=vfarcic/go-demo-2:2.0 --filename=deploy/go-demo-2-api.yml
```

We can see that, so far, there were two revisions of the software. The change cause shows which command created each of those revisions.

How about ReplicaSets?

```
kubectl get rs
```

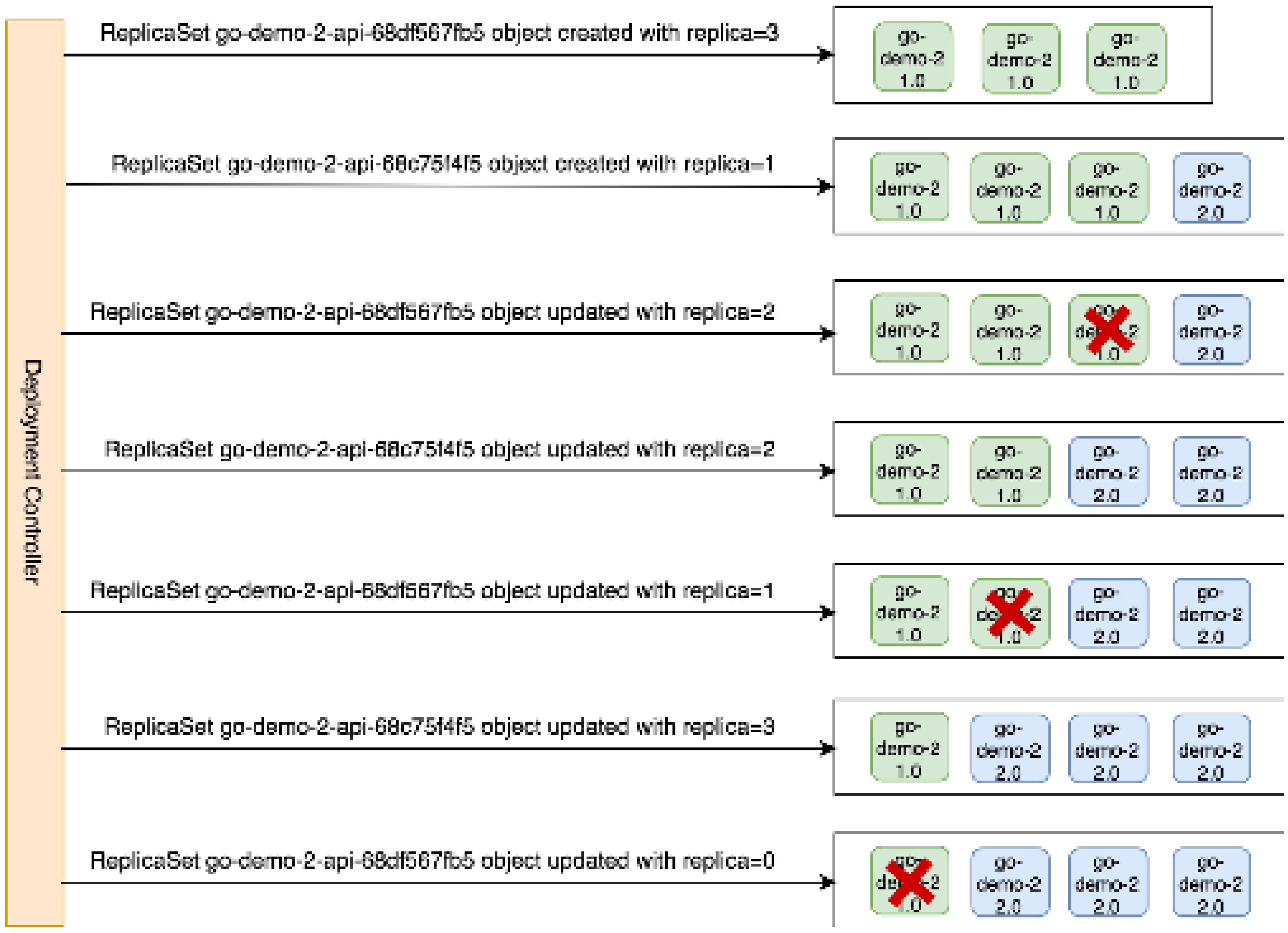
The **output**, limited to `go-demo-2-api`, is as follows.

```
NAME                     DESIRED CURRENT READY AGE
go-demo-2-api-68c75f4f5  3       3       3     4m
go-demo-2-api-68df567fb5 0       0       0     4m
...
```

We can see that the Deployment did not modify the ReplicaSet, but that it

We can see that the Deployment did not modify the ReplicaSet, but that it created a new one and, at the end of the process, the old one was scaled to

zero replicas.


Deployment Controller
ReplicaSet go-demo-2-api-68df567fb5 object created with replica=3
ReplicaSet go-demo-2-api-68c75f4f5 object created with replica=1
ReplicaSet go-demo-2-api-68df567fb5 object updated with replica=2
ReplicaSet go-demo-2-api-68c75f4f5 object updated with replica=2
ReplicaSet go-demo-2-api-68df567fb5 object updated with replica=1
ReplicaSet go-demo-2-api-68c75f4f5 object updated with replica=3
ReplicaSet go-demo-2-api-68df567fb5 object updated with replica=0
go-demo-2 1.0
go-demo-2 2.0

Deployment controller rolling update workflow

The above diagram illustrates the workflow of the Deployment controller rolling update.

In the next lesson, we will explore how to roll back or roll forward the releases.