Rolling Back or Rolling Forward?

In this lesson, we will discuss different scenarios to help us decide whether to roll back the Deployment or roll forward.

Understanding the Scenarios

At this point, we are, more or less, capable of deploying new releases to production as soon as they are ready. However, there will be problems. Something unexpected will happen. A bug will sneak in and put our production cluster at risk. *What should we do in such a case?* The answer to that question largely depends on the size of the changes and the frequency of deployments.

If we are using continuous deployment process, we are deploying new releases to production fairly often. Instead of waiting until features accumulate, we are deploying small chunks. In such cases, fixing a problem might be just as fast as rolling back.

After all, how much time would it take you to fix a problem caused by only a few hours of work (maybe a day) and that was discovered minutes after you committed? Probably not much. The problem was introduced by a very recent change that is still in the engineer's head. Fixing it should not take long, and we should be able to deploy a new release soon.

You might not have frequent releases, or the amount of changes included is more than a couple of hundreds of lines of code. In such a case, rolling forward might not be as fast as it should be. Still, rolling back might not even be possible.

We might not be able to revert the deployment if database schema changed, and it is not compatible with the previous versions of the back-end that uses it. The moment the first transaction enters, we might lose the option to roll-back. At least, not without losing the data generated since the new release.

> **i** Rolling back a release that introduced database changes is often not possible. Even when it is, rolling forward is usually a better option when practicing continuous deployment with high-frequency releases limited to a small scope of changes.

We did our best to discourage you from rolling back. Still, in some cases that is a better option. In others, that might be the only option. Luckily, rolling back is reasonably straightforward with Kubernetes.

Rolling Back with Kubernetes

We'll imagine that we just discovered that the latest release of the `vfarcic/go-demo-2` image is faulty and that we should roll back to the previous release. The command that will do just that is as follows.

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

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

The **output** of the latter command, limited to the last lines, is as follows.

```
OldReplicaSets:  <none>
NewReplicaSet:   go-demo-2-api-68df567fb5 (3/3 replicas created)
Events:
  Type   Reason             Age          From                  Message
  ----   ------             ----         ----                  -------
  Normal ScalingReplicaSet  6m           deployment-controller Scaled up replica set go-de
  Normal ScalingReplicaSet  6m           deployment-controller Scaled down replica set go-
  Normal ScalingReplicaSet  6m           deployment-controller Scaled up replica set go-de
  Normal ScalingReplicaSet  6m           deployment-controller Scaled down replica set go-
  Normal ScalingReplicaSet  6m           deployment-controller Scaled up replica set go-de
  Normal ScalingReplicaSet  6m           deployment-controller Scaled down replica set go-
  Normal DeploymentRollback 1m           deployment-controller Rolled back deployment "go-
  Normal ScalingReplicaSet  1m           deployment-controller Scaled up replica set go-de
  Normal ScalingReplicaSet  1m           deployment-controller Scaled down replica set go-
  Normal ScalingReplicaSet  1m (x2 over 6m) deployment-controller Scaled up replica set go-de
  Normal ScalingReplicaSet  1m (x3 over 1m) deployment-controller (combined from similar ever
```

We can see from the events section that the Deployment initiated rollback and, from there on, the process we experienced before was reversed. It started increasing the replicas of the older ReplicaSet, and decreasing those from the latest one. Once the process is finished, the older ReplicaSet became active with all the replicas, and the newer one was scaled down to zero.

The end result might be easier to see from the `NewReplicaSet` entry located just above `Events`. Before we undid the rollout, the value was `go-demo-2-api-68c75f4f5`, and now it's `go-demo-2-api-68df567fb5`.

Getting the Rollout History

Knowing only the current state of the latest Deployment is often insufficient, and we might need a list of the past rollouts. We can get it with the `kubectl rollout history` command.

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

The **output** is as follows.

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

If you look at the third revision, you'll notice that the change cause is the same command we used to create the Deployment the first time. Before we executed `kubectl rollout undo`, we had two revisions; `1` and `2`. The `undo` command checked the second-to-last revision (`1`).

Since new deployments do no destroy ReplicaSets but scale them to `0`, all it had to do to undo the last change was to scale it back to the desired number of replicas and, at the same time, scale the current one to zero.

In the next lesson, we will try playing around further with the existing Deployment in our cluster.