Using RollingUpdate Strategy with Standard Kubernetes Deployments

This lesson discusses the RollingUpdate strategy and how to use it. At the end of lesson we see if the RollingUpdate strategy has fulfilled our needs.

We'll cover the following

We explored one of the only two strategies we can use with Kubernetes Deployment resources. As we saw, the non-default `Recreate` is meant to serve legacy applications that are typically stateful and often do not scale. Next, we'll see what the Kubernetes community thinks is the default way we should deploy our software.

> Please bear in mind that, both in the previous and in this section, we are focused on what Kubernetes Deployments offer. We could have just as well used **StatefulSet** for stateful applications or **DeamonSet** for those that should be running in each node of the cluster. However, even though those behave differently, they are still based on similar principles. We'll ignore those and focus only on Kubernetes Deployment resources, given that I do not want to convert this chapter into a neverending flow of rambling. Later on, we'll go

yet again outside of what Kubernetes offers out-of-the-box.

Now, let's get back to the topic.

Changing deployment strategy to `RollingUpdate` strategy

To make our Deployment use the `RollingUpdate` strategy, we can either remove the whole `strategy` entry given that is the default, or we can change the type. We'll go with the latter since the command to accomplish that is easier.

```
cat charts/jx-progressive/templates/deployment.yaml \
    | sed -e \
    's@type: Recreate@type: RollingUpdate@g' \
    | tee charts/jx-progressive/templates/deployment.yaml
```

All we did was to change the `strategy.type` to `RollingUpdate`. You should see the full definition of the Deployment on the screen.

Changing return message and pushing changes

Next, we'll change the application's return message so that we can track the change easily from one release to the other.

```
cat main.go | sed -e \
    "s@recreate@rolling update@g" \
    | tee main.go

git add .

git commit -m "Recreate strategy"

git push
```

We made the changes and pushed them to the GitHub repository.

Inspecting behavior before and after the new release is deployed

Now, all that's left is to execute another loop. We'll keep sending requests to the application and display the output.

⚠ Please go to the **second terminal** before executing the command that follows.

```
while true
do
    curl "$STAGING_ADDR"
    sleep 0.2
done
```

The output should be a long list of `Hello from: Jenkins X golang http recreate` messages. After a while, when the new release is deployed, it will suddenly switch to `Hello from: Jenkins X golang http rolling update!`. The relevant part of the output should be as follows.

```
...
Hello from:  Jenkins X golang http recreate
Hello from:  Jenkins X golang http recreate
Hello from:  Jenkins X golang http rolling update!
Hello from:  Jenkins X golang http rolling update!
...
```

As you can see, this time, there was no downtime. The application switched from one release to another, or so it seems. But, if that's what happened, we would have seen some downtime, unless that switch happened exactly in those 0.2 seconds between the two requests. To understand better what happened, we'll describe the deployment and explore its events.

⚠ Please stop the loop with *ctrl+c* and return to the **first terminal**.

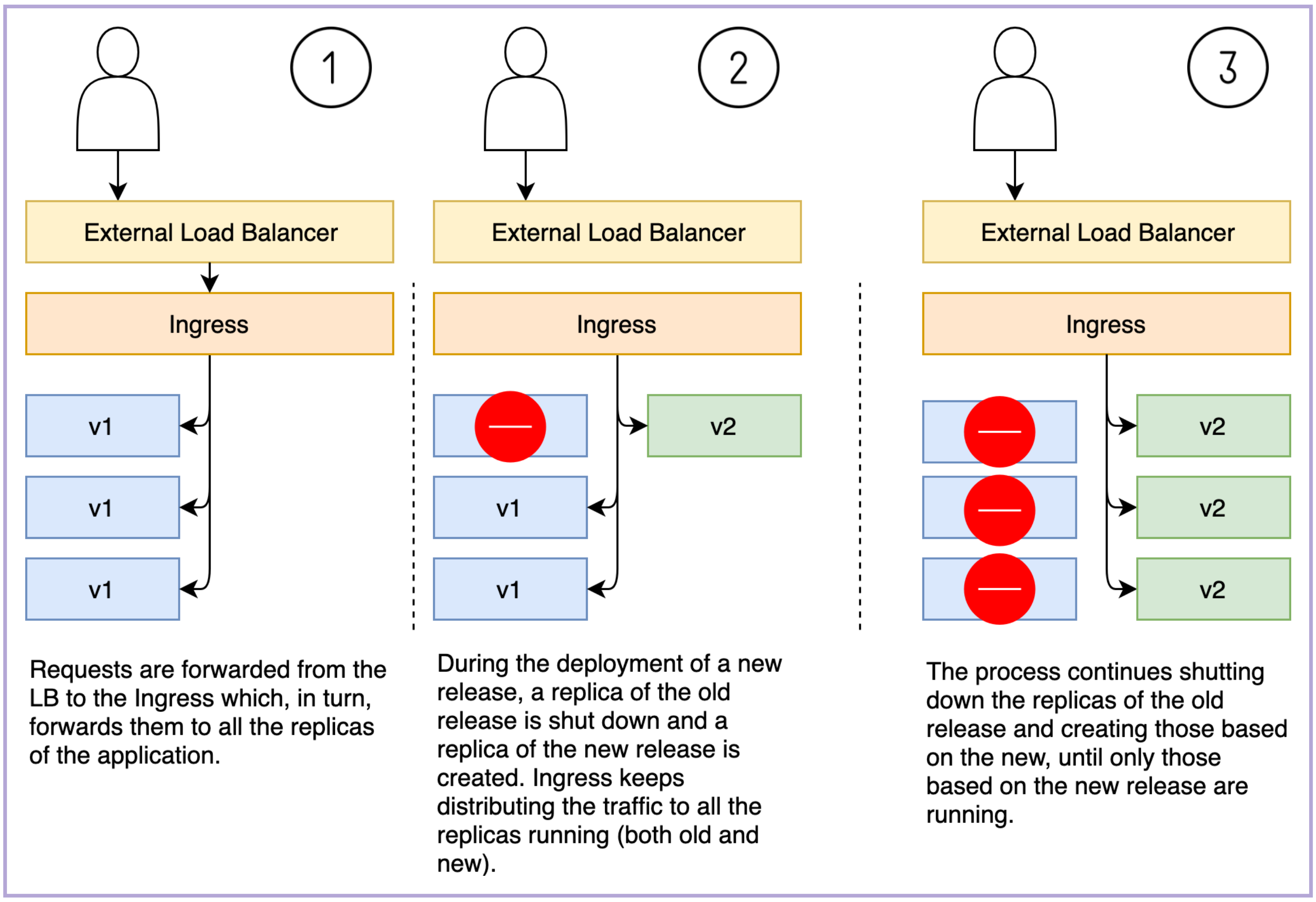
```
kubectl --namespace jx-staging \
    describe deployment jx-jx-progressive
```

The output, limited to the events section, is as follows.

```
...
Events:
  Type    Reason             Age   From                  Message
  ----    ------             ----  ----                  -------
...
  Normal  ScalingReplicaSet  6m24s deployment-controller Scaled down replica set jx-progressive-8b
  Normal  ScalingReplicaSet  6m17s deployment-controller Scaled up replica set jx-progressive-77b64
  Normal  ScalingReplicaSet  80s   deployment-controller Scaled up replica set jx-progressive-658f8
  Normal  ScalingReplicaSet  80s   deployment-controller Scaled down replica set jx-progressive-77b
  Normal  ScalingReplicaSet  80s   deployment-controller Scaled up replica set jx-progressive-658f8
  Normal  ScalingReplicaSet  72s   deployment-controller Scaled down replica set jx-progressive-77b
  Normal  ScalingReplicaSet  70s   deployment-controller Scaled up replica set jx-progressive-658f8
  Normal   ScalingReplicaSet   69s    deployment-controller Scaled down replica set jx-progressive-77
```

From those events, we can see what happened to the Deployment so far. The first entry in my output (the one that happened over 6 minutes ago) we can see that it scaled one replica set to `0` and the other to `3`. That was the rollout of the new release we created when we used the `Recreate` strategy. Everything was shut down before the new release was put in its place. That was the cause of downtime.

Now, with the `RollingUpdate` strategy, we can see that the system was gradually increasing replicas of one ReplicaSet (`jx-progressive-658f88478b`) and decreasing the other (`jx-progressive-77b6455c87`). As a result, instead of having "big bang" deployment, the system was gradually replacing the old release with the new one, one replica at a time. That means that there was not a single moment without one or the other release available and, during a brief period, both were running in parallel.


1
External Load Balancer
Ingress
v1
v1
v1
Requests are forwarded from the LB to the Ingress which, in turn, forwards them to all the replicas of the application.
2
External Load Balancer
Ingress
v2
v1
v1
During the deployment of a new release, a replica of the old release is shut down and a replica of the new release is created. Ingress keeps distributing the traffic to all the replicas running (both old and new).
3
External Load Balancer
Ingress
v2
v2
v2
The process continues shutting down the replicas of the old release and creating those based on the new, until only those based on the new release are running.

The RollingUpdate deployment strategy

You saw from the output of the loop that the messages switched from the old to the new release. In "real world" scenarios, you are likely going to have mixed outputs from both releases. For that reason, it is paramount that releases are backward compatible.

Let's take a database as an example. If we updated schema before initiating the deployment of the application, we could assume that for some time both releases would use the new schema. If the change is not backward compatible, we could end up in a situation where some requests fail because the old release running in parallel with the new is incapable of operating with the new schema. If that were to happen, the result would be similar to if we used the `Recreate` strategy. Some requests would fail. Or, even worse, everything might seem to be working correctly from the end-user point of view, but we would end up with inconsistent data. That could be even worse than downtime.

There are quite a few other things that could go wrong with `RollingUpdates`, but most of them can be resolved by answering positively to two crucial questions. Is our application scalable? Are our releases backward compatible? Without scaling (multiple replicas), `RollingUpdate` is impossible, and without backward compatibility, we can expect errors caused by serving requests through multiple versions of our software.

Does `RollingUpdate` strategy fulfill our needs?

So, what did we learn so far? Which requirements did we fulfill with the `RollingUpdate` strategy?

High-availability

Our application was highly available at all times. By running multiple replicas, we are safe from downtime that could be caused by one or more of them failing. Similarly, by gradually rolling out new releases, we are avoiding downtime that we experienced with the `Recreate` strategy.

Responsiveness

Even though we did not use **HorizontalPodAutoscaler (HPA)** in our example, we should add it to our solution. With it, we can make our application scale up and down to meet the changes in traffic. The effect would be similar as if we'd use serverless deployments (e.g., with Knative). Still, since HPA does not scale to zero replicas, it would be even more responsive given that there would be no response delay while the system is going from nothing to something (from zero replicas to whatever is needed).

Cost-effectiveness

On the other hand, this approach comes at a higher cost. We’d have to run at least one replica even if our application is receiving no traffic. Also, some might argue that setting up HPA might be more complicated given that Knative comes with some sensible scaling defaults. That might or might not be an issue, depending on the knowledge one has with deployments and Kubernetes in general. While with Knative HPA and quite a few other resources are implied, with Deployments and the `RollingUpdate` strategy, we do need to define it ourselves. We can say that Knative is more developer-friendly given its simpler syntax and that there is less need to change the defaults.

The only two requirements left to explore are progressive rollout and rollback.

Progressive rollout

Just as with serverless deployments, `RollingUpdate` kind of works. As you already saw, it does roll out replicas of the new release progressively, one or more at the time. However, the best we can do is make it stop the progress based on very limiting health checks. We can do much better on this front and later we’ll see how.

Rollback

Rollback feature does not exist with the `RollingUpdate` strategy. It can, however, stop rolling forward and that, in some cases, we might end up with only one non-functional replica of the new release. From the user’s perspective, that might seem like only the old release is running. But there is no guarantee for such behavior given that on many occasions a problem might be detected after the second, third, or some other replica is rolled out. Automated rollbacks are the only requirement that wasn’t fulfilled by any of the deployment strategies we employed so far. Bear in mind that, just as before, by automated rollback, I’m referring to what deployments offer us. I’m excluding situations in which you would do them inside your Jenkins X pipelines. Anything can be rolled back with a few tests and scripts executed if they fail, but that’s not our current goal.

Conclusion

So, what did we conclude? Do rolling updates fulfill all our requirements? Just as with other deployment strategies, the answer is still “no”. Still, `RollingUpdate` is much better than what we experienced with the `Recreate` strategy. Rolling updates provide **high-availability** and **responsiveness**. They are getting us **half-way towards progressive rollouts**, and they are **more or less cost-effective**. The major drawback is the **lack of automated rollbacks**.

The summary of the fulfillment of our requirements for the `RollingUpdate` deployment strategy is as follows.

Requirement	Fulfilled
High-availability	Fully
Responsiveness	Fully
Progressive rollout	Partly
Rollback	Not
Cost-effectiveness	Partly

The next in line is the blue-green deployment.