Using Recreate Strategy with Standard Kubernetes Deployments

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

We'll cover the following

- Changing deployment to Recreate strategy
- Pushing the changes and observing the outcome
- Checking the Pods
- Inspecting the description of the deployment
- Why is Ingress missing and how to fix it?
- Inspecting behavior before and after a new release is deployed
- What did we observe?
- The recreate deployments strategy
- Does recreate strategy fulfill our needs?
  - High Availability
    - Why can static Jenkins not scale?
  - Responsiveness
  - Cost-effectiveness
  - Progressive rollout
  - Rollback
- Conclusion

Previously, most applications were deployed with what today we call the *recreate* strategy which we will discuss shortly. For now, we'll focus on implementing it and observing the outcome.

Changing deployment to `Recreate` strategy

By default, Kubernetes Deployments use the `RollingUpdate` strategy. If we do not specify any, that's the one that is implied. We'll get to that one later. For now, what we need to do is add the `strategy` into the `deployment.yaml` file that defines the deployment.

```
cd jx-progressive

cat charts/jx-progressive/values.yaml \
    | sed -e \
    's@replicaCount: 1@replicaCount: 3@g' \
    | tee charts/jx-progressive/values.yaml

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

We entered the local copy of the *jx-progressive* repository, and we used a bit of `sed` magic to increase the number of replicas in `values.yaml` and to add the `strategy` entry just above `replicas` in `deployment.yaml`. If you are not a `sed` ninja, that command might have been confusing, so let's output the file and see what we got.

```
cat charts/jx-progressive/templates/deployment.yaml
```

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

```
...
spec:
  strategy:
    type: Recreate
...
```

Pushing the changes and observing the outcome

Now that we changed our deployment strategy to `recreate`, all that's left is to push it to GitHub, wait until it is deployed, and observe the outcome. Right?

```
git add .

git commit -m "Recreate strategy"

git push --set-upstream origin master

jx get activities \
    --filter jx-progressive/master \
    --watch
```

We pushed the changes, and we started watching the activities. Please press *ctrl+c* to cancel the watcher once you confirm that the newly launched build is finished.

If you're using serverless Jenkins X, the build of an application does not wait for the activity associated with automatic promotion to finish. So, we'll confirm whether that is done as well.

> ⚠ Please execute the command that follows only if you are using **serverless Jenkins X**.

```
jx get activities \
    --filter environment-jx-rocks-staging/master \
    --watch
```

You know what needs to be done. Press *ctrl+c* when the build is finished.

Checking the Pods

Let's take a look at the Pods we got.

```
kubectl --namespace jx-staging \
    get pods
```

The output is as follows

```
NAME                      READY STATUS  RESTARTS AGE
jx-jx-progressive-... 1/1   Running 0        2m
jx-jx-progressive-... 1/1   Running 0        2m
jx-jx-progressive-... 1/1   Running 0        2m
```

There's nothing new here. Judging by the look of the Pods, if we did not change the strategy to `Recreate`, you would probably think that it is still the default one. The only difference we can notice is in the description of the deployment, so let's output it.

Inspecting the description of the deployment

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

The output, limited to the relevant part, is as follows.

```
...
StrategyType:       Recreate
...
Events:
  Type    Reason             Age  From                   Message
  ----    ------             ---- ----                   -------
...
  Normal ScalingReplicaSet 20s  deployment-controller Scaled up replica set jx-progressive-589c478
```

Judging by the output, we can confirm that the `StrategyType` is now `Recreate`. That's not a surprise. What is more interesting is the last entry in the `Events` section. It scaled replicas of the new release to three. *Why is that a surprise? Isn't that the logical action when deploying the first release with the new strategy?* Well... It is indeed logical for the first release so we'll have to create and deploy another to see what's really going on.

If you had **Knative** deployment running before, there is a small nuisance we need to fix. **Ingress** is missing, and I can prove that.

```
kubectl --namespace jx-staging \
    get ing
```

The output claims that `no resources` were `found`.

> ⚠ Non-**Knative** users will have **Ingress** running and will not have to execute the workaround we are about to do. Feel free to skip the few commands that follow. Alternatively, you can run them as well. No harm will be done. Just remember that their purpose is to create the missing **Ingress** that is already running and that there will be no visible effect.

Why is Ingress missing and how to fix it?

What happened? Why isn't there ***Ingress*** *when we saw it countless times before in previous exercises?*

Jenkins X creates **Ingress** resources automatically unless we tell it otherwise. You know that already. What you might not know is that there is a bug (undocumented feature) that prevents **Ingress** from being created the first time we change the deployment type from **Knative** to plain-old Kubernetes Deployments. That happens only when we switch and not in consecutive deployments of new releases.

So, all we have to do is deploy a new release, and Jenkins X will pick it up correctly and create the missing **Ingress** resource the second time. Without it we won't be able to access the application from outside the cluster. So, all we have to do is make a trivial change and push it to GitHub. That will trigger yet another pipeline activity that will result in the creation of a new release and its deployment to the staging environment.

```
echo "something" | tee README.md

git add .

git commit -m "Recreate strategy"

git push
```

We made a silly change, we pushed it to GitHub, and that triggered yet another build. All we have to do is wait. Or, even better, we can watch the activities of *jx-progressive* and the staging environment pipelines to confirm that everything was executed correctly. I'll skip showing you the `jx get activities` commands given that I'm sure you already know them by heart.

Assuming that you were patient enough and waited until the new release is deployed, now we can confirm that the **Ingress** was indeed created.

```
kubectl --namespace jx-staging \
    get ing
```

The output is as follows.

```
NAME           HOSTS                                             ADDRESS        PORTS AGE
jx-progressive jx-progressive.jx-staging.35.196.143.33.nip.io 35.196.143.33 80    3m56s
```

That's the same output that those that did not run **Knative** before saw after the first release. Now we are all on the same page.

All in all, the application is now running in staging, and it was deployed using the `recreate` strategy.

Inspecting behavior before and after a new release is deployed

Next, we'll make yet another simple change to the code. This time we'll change the output message of the application. That will allow us to easily see how it behaves before and after the new release is deployed.

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

git add .

git commit -m "Recreate strategy"

git push
```

We changed the message. As a result, our current release is outputting `Hello from: Jenkins X golang http example`, while the new release, once it's deployed, will return `Hello from: Jenkins X golang http recreate`.

Now we need to be **very fast** and start sending requests to our application before the new release is rolled out. If you're unsure why we need to do that, it will become evident in a few moments.

Please open a **second terminal** window.

Given that **EKS** requires access key ID and secret access key as authentication, we'll need to declare a few environment variables in the new terminal session. Those are the same ones we used to create the cluster, so you shouldn't have any trouble recreating them.

> ⚠ Please execute the commands that follow **only** if your cluster is running in **EKS**. You'll have to replace the first `[...]` with your access key ID, and the second with the secret access key.

```
export AWS_ACCESS_KEY_ID=[...]

export AWS_SECRET_ACCESS_KEY=[...]

export AWS_DEFAULT_REGION=us-east-1
```

Let's find out the address of our application running in staging.

```
jx get applications --env staging
```

The output should be similar to the one that follows.

```
APPLICATION    STAGING PODS URL
jx-progressive 0.0.4   3/3  http://jx-progressive.jx-staging.35.196.143.33.nip.io
```

Copy the `jx-progressive` URL and paste it instead of `[...]` in the command that follows.

```
STAGING_ADDR=[...]
```

That's it. Now we can start bombing our application with requests.

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

We created an infinite loop inside which we're sending requests to the application running in staging. To avoid burning your laptop, we also added a short delay of `0.2` seconds.

If you were fast enough, the output should consist of an endless list of `Hello from: Jenkins X golang http example` messages. If that's what you're getting, it means that the deployment of the new release did not yet start. In such a case, all we have to do is wait.

At one moment, `Hello from: Jenkins X golang http example` messages will turn into 5xx responses. Our application is down. If this were a "real world" situation, our users would experience an outage. Some of them might even be so disappointed that they would choose not to stick around to see whether we'll recuperate and instead switch to a competing product. I know that I, at least, have a very low tolerance threshold. If something does not work and I do not have a strong dependency on it, I move somewhere else almost instantly. If I'm committed to a service or an application, my tolerance might be a bit more forgiving, but it is not indefinite. I might forgive you one outage. I might even forgive two. But, the third time I cannot consume something, I will start considering an alternative. Then again, that's me, and your users might be more forgiving. Still, even if you do have loyal customers, downtime is not a good thing, and we should avoid it.

While you were reading the previous paragraph, the message probably changed again. Now it should be an endless loop of `Hello from: Jenkins X golang http recreate`. Our application recuperated and is now operational again. It's showing us the output from the new release. If we could erase from our memory the 5xx messages, that would be awesome.

All in all, the output, limited to the relevant parts, should be as follows.

```
...
Hello from:  Jenkins X golang http example
Hello from:  Jenkins X golang http example
...
<html>
<head><title>502 Bad Gateway</title></head>
<body>
<center><h1>502 Bad Gateway</h1></center>
<hr><center>openresty/1.15.8.1</center>
</body>
</html>
<html>
<head><title>503 Service Temporarily Unavailable</title></head>
<body>
<center><h1>503 Service Temporarily Unavailable</h1></center>
<hr><center>openresty/1.15.8.1</center>
</body>
</html>
...
Hello from:  Jenkins X golang http recreate
Hello from:  Jenkins X golang http recreate
...
```

If all you ever saw was only the loop of `Hello from: Jenkins X golang http recreate`, all I can say is that you were too slow. If that's the case, you'll have to trust me that there were some nasty messages in between the old and the new release.

That was enough looping for now. Please press *ctrl+c* to stop it and give your laptop a rest. Leave the second terminal open and go **back to the first one**.

What did we observe?

What happened was neither pretty nor desirable. Even if you are not familiar with the `RollingUpdate` strategy (the default one for Kubernetes Deployments), you already experienced it countless times before. You probably did not see those 5xx messages in the previous exercises, and that might make you wonder why did we switch to `Recreate`. *Why would anyone want it?* The answer to that question is that switch to `Recreate`. *Why would anyone want it?* The answer to that question is that no one desires such outcomes, but many are having them anyway. I'll explain soon

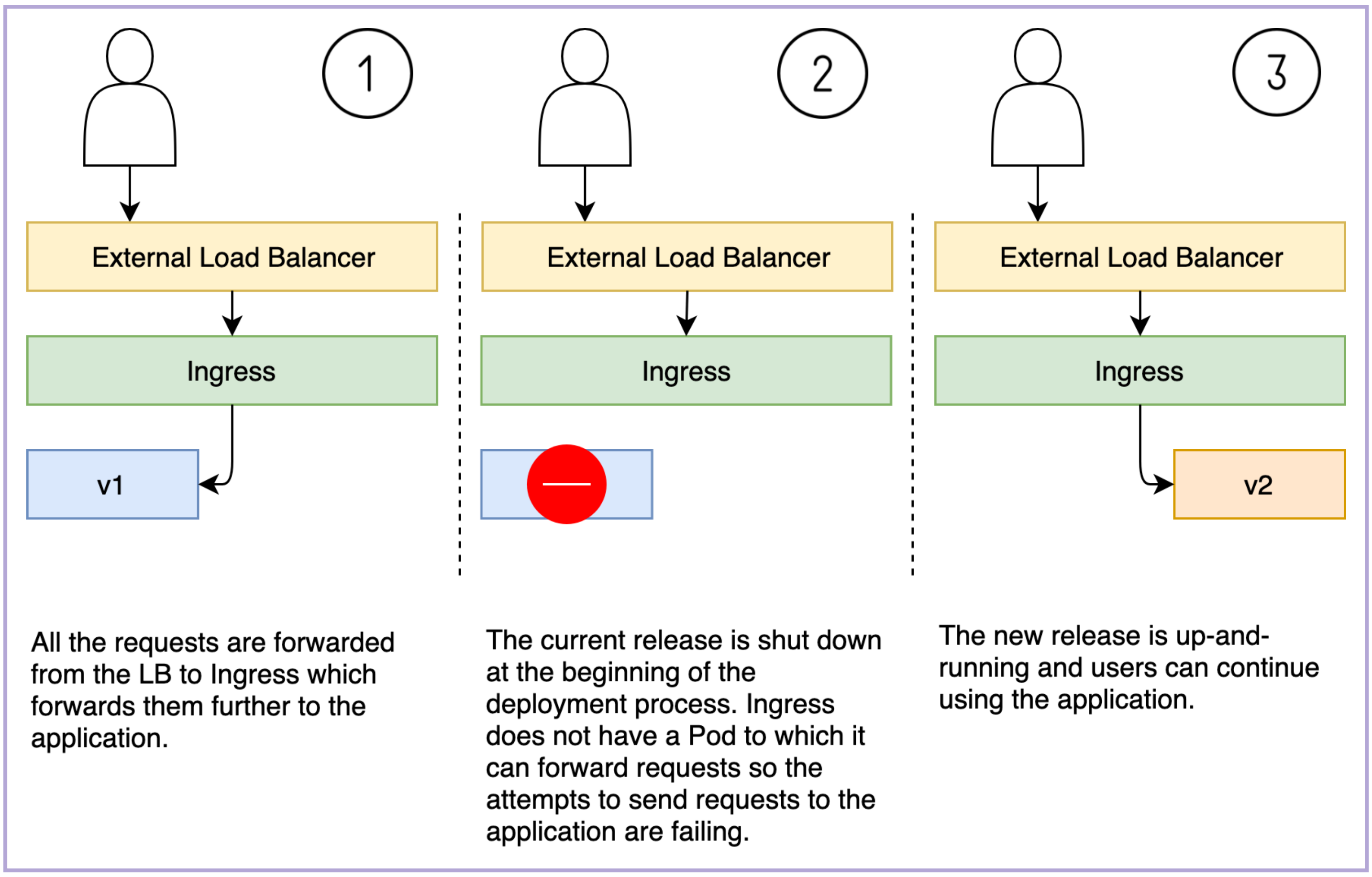
why we want to use the `Recreate` strategy even though it produces downtime. To answer why anyone would want something like that, we'll first explore why the outage was created in the first place.

When we deployed the second release using the `Recreate` strategy, Kubernetes first shut down all the instances of the old release. Only when they all ceased to work, it deployed the new release in its place. The downtime we experienced existed between the time the old release was shut down, and the time the new one became fully operational. The downtime lasted only for a couple of seconds, but that's because our application (*go-demo-6*) boots up very fast. Some other apps might be much slower, and the downtime would be much longer. It's not uncommon for the downtime in such cases to take minutes and sometimes even hours.

Alternatively, you might not have seen a single 5xx error. If that's the case, you were fortunate because that means that the old release was shut down and the new was up-and-running within 200 milliseconds (iteration between two requests in the loop9. If that's what happened to you, rest assured that it is highly unlikely it'll happen again. You just experienced once in a lifetime event. As you can probably guess, we cannot rely on users being that lucky.

The recreate deployments strategy

We can think of the `Recreate` strategy as a "big bang". There is no transition period, there are no rolling updates, nor there are any other"modern" deployment practices. The old release is shut down, and the new one is put in its place. It's simple and straightforward, but it results in inevitable downtime.



1
2
3
External Load Balancer
Ingress
v1
v2
All the requests are forwarded from the LB to Ingress which forwards them further to the application.
The current release is shut down at the beginning of the deployment process. Ingress does not have a Pod to which it can forward requests so the attempts to send requests to the application are failing.
The new release is up-and-running and users can continue using the application.

The recreate deployments strategy

Still, the initial question stands. *Who would ever want to use the* `Recreate` *strategy?* The answer is not that much who wants it, **but rather who must use it.**

Does recreate strategy fulfill our needs?

High Availability

Let's take another look at static Jenkins. It is a stateful application that cannot scale. So, replacing one replica at a time as a way to avoid downtime is out of the question. When applications cannot scale, there is no way we could ever accomplish deployments without downtime. Two replicas are a minimum. Otherwise, if only one replica is allowed to run at any given moment, we have to shut it down to make room for the other replica (the one from the new release). So, when there is no scaling, there is no high availability. Downtime, at least related to new releases, is unavoidable.

Why can static Jenkins not scale?

There can be many answers to that question, but the main culprit is its state. It is a stateful application unable to share that state across multiple instances. Even if you deploy various Jenkins instances, they will operate independently from each other. Each would have a different state and manage different pipelines. That, dear reader, is not scaling. Having multiple independent instances of an application is not replication. For an application to be scalable, each replica needs to work together with others and share the load. As a rule of thumb, for an application to be scalable, it needs to be stateless (e.g., *go-demo-6*) or to be able to replicate state across all replicas (e.g., MongoDB). Jenkins does not fulfill either of the two criteria and, therefore, it cannot scale. Each instance has its separate file storage where it keeps the state unique to that instance. The best we can do with static Jenkins is to give an instance to each team. That solves quite a few Jenkins-specific problems, but it does not make it scalable. As a result, it is impossible to upgrade Jenkins without downtime.

Upgrades are not the only source of downtime with unscalable applications. If we have only one replica, Kubernetes will recreate it when it fails. But that will also result in downtime. As a matter of fact, failure and upgrades of single-replica applications are more or less the same processes. In both cases, the only replica is shut down, and a new one is put in its place. There is downtime between those two actions.

All that might lead you to conclude that only single-replica applications should use the `Recreate` strategy. That's not true. There are many other reasons why the *"big bang"* deployment strategy should be applied. We won't have time to discuss all. Instead, I'll mention only one more example.

The only way to avoid downtime when upgrading applications is to run multiple replicas and start replacing them one by one or in batches. It does not matter much how many replicas we shut down and replace with those of the new release. We should be able to avoid downtime as long as there is at least one replica running. So, we are likely going to run new releases in parallel with the old ones, at least for a while. We'll go through that scenario soon. For now, trust me when I say that running multiple releases of an application in parallel is unavoidable if we are to perform deployments without downtime. That means that our releases must be backward compatible, that our applications need to version APIs, and that clients need to take that versioning into account when working with our apps. Backward compatibility is usually the main stumbling block that prevents teams from applying zero-downtime deployments. It extends everywhere. Database schemas, APIs, clients, and many other components need to be backward compatible.

compatible.

All in all, inability to scale, statefulness, lack of backward compatibility, and quite a few other things might prevent us from running two releases in parallel. As a result, we are forced to use the `Recreate` strategy or something similar.

So, the real question is not whether anyone wants to use the `Recreate` strategy, but rather who is forced to apply it due to the problems usually related to the architecture of an application. If you have a stateful application, the chances are that you have to use that strategy. Similarly, if your application cannot scale, you are probably forced to use it as well.

Given that deployment with the `Recreate` strategy inevitably produces downtime, most teams tend to have less frequent releases. The impact of, let's say, one minute of downtime is not that big if we produce it only a couple of times a year. But, if we would increase the release frequency, that negative impact would increase as well. Having downtime a couple of times a year is much better than once a month, which is still better than if we'd have it once a day. High-velocity iterations are out of the question. We cannot deploy releases frequently if we experience downtime each time we do that. In other words, zero-downtime deployments are a prerequisite for high-frequency releases to production. Given that the `Recreate` strategy does produce downtime, it stands to reason that it fosters less frequent releases to production as a way to reduce the impact of downtime.

Before we proceed, it might be important to note that there was no particular reason to use the `Recreate` deployment strategy. The *jx-progressive* application is scalable, stateless, and it is designed to be backward compatible. Any other strategy would be better suited given that zero-downtime deployment is probably the most important requirement we can have. We used the `Recreate` strategy only to demonstrate how that deployment type works and to be consistent with other examples in this chapter.

Now that we saw how the `Recreate` strategy works, let's see which requirements it fulfills, and which it fails to address. As you can probably guess, what follows is not going to be a pretty picture.

When there is downtime, there is no high-availability. One excludes the other, so we failed with that one.

Responsiveness

Responsiveness

Is our application responsive? If we used an application that is more appropriate for that type of deployment, we would probably discover that it would not be responsive or that it would be using more resources than it needs. Likely we'd experience both side effects.

Cost-effectiveness

If we go back to static Jenkins as an excellent example of the `Recreate` deployment strategy, we would quickly discover that it is expensive to run it. Now, I do not mean expensive in terms of licensing costs but rather in resource usage. We'd need to set it up to always use memory and CPU required for its peak load. We'd probably take a look at metrics and try to figure out how much memory and CPU it uses when the most concurrent builds are running. Then, we'd increase those values to be on the safe side and set them as requested resources. That would mean that we'd use more CPU and memory than what is required for the peak load, even if most of the time we need much less. In the case of some other applications, we'd let them scale up and down and, in that way, balance the load while using only the resources they need. But, if that would be possible with Jenkins, we would not use the `Recreate` strategy. Instead, we'd have to waste resources to be on the safe side, knowing that it can handle any load. That's very costly. The alternative would be to be cheaper and give it fewer resources than the peak load. However, in that case, it would not be responsive given that the builds at the peak load would need to be queued. Or, even worse, it would just bleed out and fail under a lot of pressure. In any case, a typical application used with the `Recreate` deployment strategy is often unresponsive, or it is expensive. More often than not, it is both.

Progressive rollout

The only thing left is to see whether the `Recreate` strategy allows progressive rollout and automated rollbacks. In both cases, the answer is a resounding no. Given that most of the time only one replica is allowed to run, the progressive rollout is impossible.

Rollback

On the other hand, there is no mechanism to roll back in case of a failure. That is not to say that it is not possible to do that, but that it is not incorporated into the deployment process itself. We'd need to modify our pipelines to accomplish that.

Given that we're focused only on deployments, we can say that rollbacks are not available.

Conclusion

What's the score? Does the `Recreate` *strategy fulfill all our requirements?* The answer to that question is a huge **"no"**. *Did we manage to get at least one of the requirements?* The answer is still no. *"Big bang"* deployments do **not provide high-availability**. They are **not cost-effective**. They are **rarely responsive**. There is **no possibility to perform progressive rollouts**, and they come with **no automated rollbacks**.

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

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

As you can see, that was a very depressing outcome. Still, the architecture of our applications often forces us to apply it. We need to learn to live with it, at least until the day we are allowed to redesign those applications or throw them to thrash and start over.

I hope that you never worked with such applications. If you didn't, you are either very young, or you always worked in awesome companies. I, for example, spent most of my career with applications that had to be put down for hours every time we deploy a new release. I had to come to the office during weekends because that’s when the least number of people were using our applications. I had to spend hours or even days doing deployments. I spent too many nights sleeping in the office over weekends. Luckily, we had only a few releases a year. Those days now feel like a nightmare that I never want to experience again. That might be the reason why I got interested in automation and architecture. I wanted to make sure that I replace myself with scripts.

So far, we saw two deployment strategies. We probably started with the inverted order, at least from the historical perspective. We can say that serverless deployments are one of the most advanced and modern strategies. At the same time, `Recreate` or, to use a better name, “big bang” deployments are the ghosts of the past that are still haunting us. It’s no wonder that Kubernetes does not use it as a default deployment type.

From here on, the situation can only be more positive. Brace yourself for an increased level of happiness.

Next, we are going to discuss another deployment strategy called the rolling update strategy.