KUBERNETES DEPLOYMENTS: AN INTRODUCTION

In my last two posts, I touched on Kubernetes pod and ReplicaSets (RS). I will now build on that with another controller concept called **deployment**. The main role of deployment is to provide declarative updates to both the pod and the RS. First, declare a state for a manifest (yaml file), then the controller makes sure the current state is reconciled to match the desired state.

To better understand, we'll explore how deployment works, how to define deployment in a manifest, and how to create a deployment demo.

Understanding K8s deployments

There are three stages a in a deployment lifecycle:

1. Progressing
2. Complete
3. Failed

In Kubernetes, deployment is the recommended way to deploy a pod or RS because of its advanced, built-in features. Below are some of the key features of deployment:

- Easily deploy a RS
- Update pods (PodTemplateSpec)
- Rollback to previous deployment versions
- Scale deployment
- Pause and resume deployment
- Determine the state of replicas (using deployment status)

- Clean up older RS that are no longer needed
- Deploy Canary

Now, let's look at some of these features in action.

Deployment strategies

Deployment strategies are used to replace old pods by new ones. There are two kinds you can use:

- **spec.strategy.type** to recreate in the manifest. All pods are killed and recreated. This is defined by setting.
- **Rolling update.** Pods are updated in a rolling fashion. This is defined by setting .spec.strategy.type to RollingUpdate. We can set maxUnavailable and maxSurge but, by default, it makes sure that only 25 percent of your pods are unavailable so we don't have to change it if it's not necessary.

How to create a deployment

To demonstrate, let's create a simple deployment of nginx with three replicas. Like any other object in Kubernetes, you need the apiVersion, kind, and metadata. The spec section is slightly different.

With this basic example, we will see what the manifest for a deployment looks like and see some of the benefits mentioned above in action.

Deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment # Name of our deployment
  labels:
    app: nginx
spec:
  replicas: 3      # number of pods
  Selector:        # this is how deployment knows what pod to manage
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.1     # Image version
        ports:
        - containerPort: 80
```

```
$ kubectl create -f deploy.yaml
deployment.apps "example-deployment" created
```

How to work with a deployment

Before working with your deployment, let's check to make sure it was successfully created. Run the **kubectl rollout status** and **kubectl get deployment** commands.

- **kubectl rollout status** tells us whether the deployment was successful.
- **kubectl get deployment** shows the desired and updated number of replicas, the number of nginx pod replicas are running, and how many are available to end users.

```
$ kubectl rollout status deployment example-deployment
deployment "example-deployment" successfully rolled out
```

```
$ kubectl get deployment
NAME                            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
example-deployment              3         3         3            3
2m
```

Run **kubectl get replicaset** to confirm that the deployment created a RS for the nginx pods. Since RS is automatically created, the status helps determine the state of replicas.

```
$ kubectl get deployment
$ kubectl get replicaset
NAME                                       DESIRED   CURRENT   READY
AGE
example-deployment-5d4fbdd945              3         3         3
10m
```

By default, deployment will add a pod-template-hash to the name of RS that it creates. Do not change this hash. For example, "example-deployment-64ff85b579".

Now, run **kubectl get pod** to see the pods that our RS created.

```
$ kubectl get pod
NAME                                          READY     STATUS
RESTARTS   AGE
example-deployment-5d4fbdd945-7hmfq           1/1       Running   0
29m
example-deployment-5d4fbdd945-nfwcx           1/1       Running   0
29m
example-deployment-5d4fbdd945-wzrr6           1/1       Running   0
29m
```

Let's stop here to note a few things:

- A special argument called **—record**. Append this to *kubectl* to record the commands you run. We'll try this argument from here on, to show you how it works.
- A rollout triggers if, and only if, the deployment's pod template (.spec.template) changes.
    - For example, if you've updated the labels or container images of the template. As a new RS is created, Existing RS-controlling pods whose labels match .spec.selector but whose template does not match .spec.template are scaled down.

- Other updates, like deployment scaling, do not trigger a rollout—it will not create a new RS. This is a key concept.

Now, run **kubectl set image** to change the nginx image version. This command is the same as updating the container image field in the manifest(yaml), then applying it. We can also just run "kubectl edit deployment" and edit our image version directly.

```
$ kubectl set image deployment  example-deployment nginx=nginx:latest --
record
deployment.apps "example-deployment" image updated
```

If we run **kubectl get replicaset** again, we see a new RS because of the image update we made.

```
$ kubectl get replicaset
NAME DESIRED CURRENT READY AGE
example-deployment-5d4fbdd945 0 0 0 31m
example-deployment-7d9f9876cc 3 3 3 3m
```

We also see that our first RS was scaled to 0, while the new RS with latest version of nginx now has three replicas. This is another key feature, scaling down the old RS without us doing any manual work.

How deployment handles rollouts

By default, deployment ensures that only 25% of your pods are unavailable. This is great because it makes sure that all our nginx pods are not all scaled down at the same time. It also makes sure that it does not create more that 25% of the desired number or replicas we specified while performing the rollout.

It does not kill old pods until/unless enough new pods come up. It does not create new pods until a sufficient number of old pods are killed. This is called 'rolling update strategy', another key benefit of using deployment.

To see a record of changes, run **kubectl rollout history deployment example-deployment**.

```
$ kubectl rollout history deployment example-deployment
deployments "example-deployment"
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deploy.yaml --record=true
2         kubectl set image deployment example-deployment nginx=nginx:latest
--record=true
```

Here, we see that all our deployment changes are recorded and what commands changed which components. Always pay attention to the revision number.

How to rollback changes

Let's say the image version was bad and we want to go back to a previous version. We achieved this with the help of the **record** argument and the **revision number**. Once we have the revision number we want to rollback to, make sure that the revision contains what we want. We can then append the revision number to the rollout history command to show more information about the revision.

```
$ kubectl rollout history deployment example-deployment --revision=1
deployments "example-deployment" with revision #1
Pod Template:
  Labels:  app=nginx
     pod-template-hash=1809688501
  Annotations:  Kubernetes.io/change-cause=kubectl create --
filename=deploy.yaml --record=true
  Containers:
   nginx:
    Image: nginx:1.7.1
    Port:  80/TCP
    Host Port:  0/TCP
    Environment: <none>
    Mounts:     <none>
  Volumes: <none>
```

Once we confirm this is the correct revision, we can then rollback to it.

```
$ kubectl rollout undo  deployment example-deployment --to-revision=1
deployment.apps "example-deployment"
```

Now, we run the status command to make sure it was successfully rolled out. Once we are sure, we run the rollout history command again. Notice how we no longer have revision 1, we now have 2 and 3. If we describe the deployment, we will see in the event section how the rollback happened.

```
Events:

Type    Reason              Age                From                    Message

----    ------              ----               ----                    -------

Normal  ScalingReplicaSet   13m                deployment-controller  Scaled
up replica set example-deployment-7d9f9876cc to 1

Normal  ScalingReplicaSet   13m                deployment-controller  Scaled
down replica set example-deployment-5d4fbdd945 to 2

Normal  ScalingReplicaSet   13m                deployment-controller  Scaled
up replica set example-deployment-7d9f9876cc to 2

Normal  ScalingReplicaSet   13m                deployment-controller  Scaled
down replica set example-deployment-5d4fbdd945 to 1

Normal  ScalingReplicaSet   13m                deployment-controller  Scaled
up replica set example-deployment-7d9f9876cc to 3

Normal  ScalingReplicaSet   13m                deployment-controller  Scaled
down replica set example-deployment-5d4fbdd945 to 0
```

```
Normal  DeploymentRollback  2m                deployment-controller  Rolled
back deployment "example-deployment" to revision 1

Normal  ScalingReplicaSet   2m                deployment-controller  Scaled
up replica set example-deployment-5d4fbdd945 to 1

Normal  ScalingReplicaSet   2m                deployment-controller  Scaled
up replica set example-deployment-5d4fbdd945 to 2

Normal  ScalingReplicaSet   2m                deployment-controller  Scaled
down replica set example-deployment-7d9f9876cc to 2

Normal  ScalingReplicaSet   1m (x2 over 19m)  deployment-controller  Scaled
up replica set example-deployment-5d4fbdd945 to 3

Normal  ScalingReplicaSet   1m                deployment-controller  Scaled
down replica set example-deployment-7d9f9876cc to 1

Normal  ScalingReplicaSet   1m                deployment-controller  Scaled
down replica set example-deployment-7d9f9876cc to 0
```

How to change the amount of replicas

We can scale our deployment by simply changing the replica number in the manifest, then applying it. We can also simply run **kubectl scale** and append **—replicas**.

```
$ kubectl scale --replicas=4 deployment example-deployment
deployment.extensions "example-deployment" scaled
```

```
$ kubectl get deploy
NAME                            DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
example-deployment              4         4         4
4           23m
```

We now have four replicas.

How to pause and resume a deployment

The **kubectl pause** command allows us to make changes and fixes without triggering a new RS rollout.

```
$ kubectl rollout pause deploy example-deployment
deployment.apps "example-deployment" paused
```

Let's change the image again:

```
$ kubectl set image deployment  example-deployment nginx=nginx:1.9.1 --record
deployment.apps "example-deployment" image updated
```

```
$ kubectl get replicaset
NAME                                    DESIRED   CURRENT   READY
AGE
example-deployment-5d4fbdd945           4         4         4
28m
example-deployment-7d9f9876cc           0         0         0
22m
```

Notice how deployment did not rollout a new RS, but if we resume the rollout by running **kubectl rollout resume deploy example-deployment**, a new RS is created.

Additional resources

For more on Kubernetes, explore these resources:

- Kubernetes Guide, with 20+ articles and tutorials
- BMC DevOps Blog
- The State of Kubernetes in 2020
- Bring Kubernetes to the Serverless Party
- How eBay is Reinventing Their IT with Kubernetes & Replatforming Program