Kubernetes Deployment vs. StatefulSets

1. Overview

Kubernetes (K8s) is an open-source container orchestration system. It allows us to automate deployments, scale, and manage containerized applications.

In this tutorial, we'll discuss two different ways to deploy our application(pods) on Kubernetes using different Kubernetes resources. Below are two different resources that Kubernetes provides for deploying pods:

- Deployment
- StatefulSet

Let's start by looking at the difference between a stateful and stateless application.

2. Stateful and Stateless Applications

The key difference between stateful and stateless applications is that **stateless applications don't "store" data. On the other hand, stateful applications require backing storage**. For example, applications like the Cassandra, MongoDB, and MySQL databases require some type of persistent storage to survive service restarts.

Keeping a state is critical for running a stateful application. But for a stateless service, any data flow is typically transitory. Also, the state is stored only in a separate back-end service like a database. Any associated storage is typically ephemeral. For instance, if the container restarts, anything stored is lost. As organizations adopt containers, they tend to begin with stateless containers as they are easier to adopt.

Kubernetes is well-known for managing stateless services. The deployment workload is more suited to work with stateless applications. As far as deployment is concerned, pods are interchangeable. While a *StatefulSet* keeps a unique identity for each pod it manages. It uses the same identity whenever it needs to reschedule those pods. In this article, we'll discuss this further.

3. *Deployment*

3.1. Understanding *Deployment*: the Basics

A Kubernetes Deployment provides means for managing a set of pods. These could be one or more running containers or a group of duplicate pods, known as *ReplicaSets*. *Deployment* allows us to easily keep a group of identical pods running with a common configuration.

First, we define our Kubernetes *Deployment* and then deploy it. Kubernetes will then work to make sure all pods managed by the deployment meet whatever requirements we have set. ***Deployment* is a supervisor for pods. It gives us fine-grained control over how and when a new pod version is rolled out. It also provides control when we have to rollback to a previous version.**

In Kubernetes *Deployment* with a replica of 1, the controller will verify whether the current state is equal to the desired state of *ReplicaSet,* i.e., 1. If the current state is 0, it will create a *ReplicaSet.* The ReplicaSet will further create the pods. When we create a Kubernetes *Deployment* with the name *web-app*, it will create a *ReplicaSet* with the name *web-app-<replica-set-id>.* This replica will further create a pod with name *web-app-<replica-set->-<pod-id>*.

Kubernetes *Deployment* is usually used for stateless applications. **However, we can save the state of *Deployment* by attaching a Persistent Volume to it and make it stateful.** The deployed pods will share the same Volume, and the data will be the same across all of them.

3.2. *Deployment* Components in Kubernetes

The following are the major components of a Kubernetes *Deployment* :

- *Deployment* Template
- PersistentVolume
- Service

First, let's make our deployment template and save it as '*deployment.yaml*'. In the template below, we're also attaching a persistent volume:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app-deployment
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 1
  selector:
    matchLabels:
      app: web-app
  replicas: 3
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
        - name: web-app
          image: hello-world:nanoserver-1809
          volumeMounts:
          - name: counter
            mountPath: /app/
      volumes:
       - name: counter
         persistentVolumeClaim:
          claimName: counter
```

In the template below, we have our *PersistentVolumeClaim*:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: counter
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 50Mi
  storageClassName: default
```

3.3. Executing a *Deployment* in Kubernetes

Before we execute our *Deployment*, we need a service to access the above *Deployment*. Let's create a service of type *NodePort* and save it as '*service.yaml*' :

```
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
spec:
  ports:
    - name: http
      port: 80
      nodePort: 30080
  selector:
    name: web-app
  type: NodePort
```

First, we run the service template with the below kubectl *apply* command:

```
```

`kubectl apply -f service.yaml`

Then we run the same command for the deployment template:

```
kubectl apply -f deployment.yaml
```

In addition, to get a detailed description of the deployment, let's run the kubectl *describe* command:

```
kubectl describe deployment web-app-deployment
```

The output will be similar to this:

```
Name:                   web-app-deployment
 Namespace:             default
 CreationTimestamp:     Tue, 30 Aug 2016 18:11:37 -0700
 Labels:                app=web-app
 Annotations:           deployment.kubernetes.io/revision=1
 Selector:              app=web-app
 Replicas:              3 desired | 3 updated | 3 total | 3 available | 0
unavailable
 StrategyType:          RollingUpdate
 MinReadySeconds:       0
 RollingUpdateStrategy:  1 max unavailable, 2 max surge
 Pod Template:
   Labels:              app=web-app
   Containers:
    web-app:
     Image:             spring-boot-docker-project:1
     Port:              80/TCP
     Environment:       <none>
     Mounts:            <none>
   Volumes:             <none>
 Conditions:
   Type          Status  Reason
   ----          ------  ------
   Available     True    MinimumReplicasAvailable
   Progressing   True    NewReplicaSetAvailable
 OldReplicaSets:   <none>
 NewReplicaSet:    web-app-deployment-1771418926 (3/3 replicas created)
 No events.
```

In the above section, we observe that *Deployment* internally creates a *ReplicaSet*. Then, it internally creates *Pods* inside that *ReplicaSet*. In the future, when we update the current deployment, it will create a new *ReplicaSet*. Then, it'll gradually move the *Pods* from the old *ReplicaSet* to the new one at a controlled rate.

If an error occurs while updating, the new *ReplicaSet* will never be in Ready state. The old *ReplicaSet* will not terminate again, ensuring 100% uptime in case of a failed update. In Kubernetes *Deployment*, we can also manually rollback to a previous *ReplicaSet* in case our new feature is not working as expected.

4. *StatefulSet*s

4.1. Understanding *StatefulSet*s: the Basics

In Kubernetes *Deployment*, we treat our pods like cattle, not like pets. If one of the cattle members gets sick or dies, we can easily replace it by purchasing a new head. Such an action is not noticeable. Similarly, if one pod goes down in deployment, it brings up another one. **In *StatefulSet*s, pods are given names and are treated like pets.** If one of your pets got sick, it's immediately noticeable. The same is in the case of *StatefulSet*s, as it interacts with pods by their name.

*StatefulSet*s provides to each pod in it two stable unique identities. First, the **Network Identity enables us to assign the same DNS name to the pod regardless of the number of restarts**. The IP addresses might still be different, so consumers should depend on the DNS name (or watch for changes and update the internal cache).

Secondly, **the Storage Identity remains the same**. The Network Identity always receives the same instance of Storage, regardless of which node it's rescheduled on.

StatefulSet is also a Controller, but unlike Kubernetes *Deployment*, it doesn't create *ReplicaSet* rather, it creates the *pod* with a unique naming convention. Each *pod* receives DNS name according to the pattern: *<statefulset name>-<ordinal index>.* For example, for *StatefulSet* with the name *mysql,* it will be *mysql-0*.

Every replica of a stateful set will have its own state, and each of the pods will be creating its own PVC(Persistent Volume Claim). So a *StatefulSet* with 3 replicas will create 3 pods, each having its own Volume, so total 3 PVCs. As *StatefulSet*s works with data, we should be careful while stopping pod instances by allowing the required time to persist data from memory to disk. There still might be valid reasons to perform force deletion, for example, when Kubernetes Node fails.

4.2. Headless Service

A stateful application requires pods with a unique identity (hostname). One pod should be able to reach other pods with well-defined names. A *StatefulSet* needs a Headless Service to work. **A headless service is a service with a service IP**. Thus, it directly returns the IPs of our associated pods. This allows us to interact directly with the pods instead of a proxy. It's as simple as specifying *None* for *.spec.clusterIP.*

A Headless Service does not have an IP address. Internally, it creates the necessary endpoints to expose pods with DNS names. The *StatefulSet* definition includes a reference to the Headless Service, but we have to create it separately.

4.3. *StatefulSet*s Components in Kubernetes

The following are the major components of *StatefulSet*s :

- StatefulSet
- PersistentVolume
- Headless Service

First, we create a *StatefulSet* template:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
      volumes:
      - name: www
        persistentVolumeClaim:
          claimName: myclaim
```

Secondly, we create a PersistentVolume mentioned in the *StatefulSet* template:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Lastly, we now create a Headless Service for the above *StatefulSet*:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
```

4.4. Executing *StatefulSet*s in Kubernetes

We have templates ready for all three components. Now, let's run the *create* kubectl command to create the *StatefulSet*:

```
kubectl create -f statefulset.yaml
```

It will create three pods named *web-0,web-1,web-2*. We can verify the correct creation with *get pods*:

```
kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          1m
web-1     1/1       Running   0          46s
web-2     1/1       Running   0          18s
```

We can also verify the running service:

```
kubectl get svc nginx
NAME      TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     ClusterIP   None         < none >      80/TCP    2m
```

*StatefulSet*s don't create *ReplicaSet*, so we can't rollback a StatefulSet to a previous version. **We can only delete or scale up/down the *StatefulSet*.** If we update a *StatefulSet*, it also performs RollingUpdate, i.e., one replica pod will go down, and the updated pod will come up. Similarly, then the next replica pod will go down in the same manner.

For instance, we change the image of the above *StatefulSet*. The *web-2* will terminate, and once it terminates completely, then *web-2* will be recreated, and *web-1* will be terminated at the same time. The same happens for the next replica, i.e., *web-0*. If an error occurs while updating, so only *web-2* will be down, *web-1* & *web-0* will still be up, running on the previous stable version. Unlike *Deployment*, we cannot rollback to any previous version of a *StatefulSet*.

Deleting and/or scaling a *StatefulSet* down will not delete the volumes associated with the *StatefulSet*. This ensures data safety, which is generally more valuable than an automatic purge of all related *StatefulSet* resources. After deleting *StatefulSets* there are no guarantees on the termination of pods. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the *StatefulSet* down to 0 before deletion.

4.5. *StatefulSet*s Usage

*StatefulSet*s enable us to deploy stateful applications and clustered applications. They save data to persistent storage, such as Compute Engine persistent disks. They are suitable for deploying Kafka, MySQL, Redis, ZooKeeper, and other applications (needing unique, persistent identities and stable hostnames).

For instance, a Solr database cluster is managed by several Zookeeper instances. For such an application to function correctly, each Solr instance must be aware of the Zookeeper instances that are controlling it. Similarly, the Zookeeper instances themselves establish connections between each other to elect a master node. Due to such a design, Solr clusters are an example of stateful applications. Other examples of stateful applications include MySQL clusters, Redis, Kafka, MongoDB, and others. In such scenarios, *StatefulSet*s are used.

5. *Deployment* vs. *StatefulSet*s

Let's see the main differences between *Deployment* vs. *StatefulSet*:

Deployment	**StatefulSet**
Deployment is used to deploy stateless applications	*StatefulSet*s is used to deploy stateful applications
Pods are interchangeable	Pods are not interchangeable. Each pod has a persistent identifier that it maintains across any rescheduling
Pod names are unique	Pod names are in sequential order
Service is required to interact with pods in a deployment	A Headless Service is responsible for the network identity of the pods
The specified PersistentVolumeClaim is shared by all pod replicas. In other words, shared volume	The specified volumeClaimTemplates so that each replica pod gets a unique PersistentVolumeClaim associated with it. In other words, no shared volume

A Kubernetes *Deployment* is a resource object in Kubernetes that provides declarative updates to applications. A deployment allows us to describe an application's life cycle. Such as, which images to use for the app, the number of pods there should be, and how they should be updated.

A *StatefulSet*s are more suited for stateful apps. A stateful application requires pods with a unique identity (for instance, hostname). A pod will be able to reach other pods with well-defined names. It needs a Headless Service to connect with the pods. A Headless Service does not have an IP address. Internally, it creates the necessary endpoints to expose pods with DNS names.

The *StatefulSet* definition includes a reference to the Headless Service, but we have to create it separately. *StatefulSet* needs persistent storage so that the hosted application saves its state and data across restarts. Once the *StatefulSet* and the Headless Service are created, a pod can access another one by name prefixed with the service name.

6. Conclusion

In this tutorial, we've looked at the two ways to make deployments in Kubernetes: *Statefulset* and *Deployment*. We've seen their main characteristics and components and finally compared them.