What is Statefulset and how is it different from Deployment?

Kubernetes - Beginners | Intermediate | Advanced

What is Statefulset and how is it different from Deployment?

A Statefulset is a Kubernetes controller that is used to manage and maintain one or more Pods. However, so do other controllers like ReplicaSets and, the more robust, Deployments. So what does Kubernetes use StatefulSets for? To answer this question, we need to discuss stateless versus stateful applications.

A stateless application is one that does not care which network it is using, and it does not need permanent storage. Examples of stateless apps may include web servers (Apache, Nginx, or Tomcat).

On the other hand, we have stateful apps. Let's say you have a Solr database cluster that 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. If you deploy Zookeeper on Kubernetes, you'll need to ensure that pods can reach each other through a unique identity that does not change (hostnames, IPs...etc.). Other examples of stateful applications include MySQL clusters, Redis, Kafka, MongoDB, and others.

Given this difference, Deployment is more suited to work with stateless applications. As far as a 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.

Exposing a StatefulSet

A Kubernetes Service acts as an abstraction layer. In a stateless application like an Nginx web server, the client does not (and should not) care which pod receives a response to the request. The connection reaches the Service, and it routes it to any backend pod. This is not the case in stateful apps. In the above diagram, a Solr pod may need to reach the Zookeeper master, not any pod. For this reason, part of the Statefulset definition entails a Headless Service. A Headless Service does not contain a ClusterIP. Instead, it creates several Endpoints that are used to produce DNS records. Each DNS record is bound to a pod. All of this is done internally by Kubernetes, but it's good to have an idea about how it does it.

Deploying a Stateful Application Using Kubernetes Statefulset

If you look at web_stateful.yaml file, you will find a snippet around how we are deploying a stateful application. For simplicity, are we using Nginx as the pod image. The deployment is made up of 2 Nginx web servers; both of them are connected to a persistent volume. For example, look at web_stateful.yaml file under the current location.

Before we start discussing the details of this definition, notice that the file actually contains two definitions: the storage class that the StatefulSet is using and the StatefulSet itself.

Storage Class

Storage classes are Kubernetes objects that let the users specify which type of storage they need from the cloud provider. Different storage classes represent various service quality, such as disk latency and throughput, and are selected depending on the scenario they are used for and the cloud provider's support. Persistent Volumes and Persistent Volume Claims use Storage Classes.

Persistent Volumes and Persistent Volume Claims

Persistent volumes act as an abstraction layer to save the user from going into the details of how storage is managed and provisioned by each cloud provider (in this example, we are using Google GCE). By definition, StatefulSets are the most frequent users of Persistent Volumes since they need permanent storage for their pods.

A Persistent Volume Claim is a request to use a Persistent Volume. If we are to use the Pods and Nodes analogy, then consider Persistent Volumes as the "nodes" and Persistent Volume Claims as the "pods" that use the node resources. The resources we are talking about here are storage properties, such as storage size, latency, throughput, etc.

Under this tutorial, we will see example of NFS server.

Deploying NFS Server

NFS is a protocol that allows nodes to read/write data over a network. The protocol works by having a master node running the NFS daemon and stores the data. This master node makes certain directories available over the network.

Clients access the masters shared via drive mounts. From the viewpoint of applications, they are writing to the local disk. Under the covers, the NFS protocol writes it to the master.

In this scenario, and for demonstration and learning purposes, the role of the NFS Server is handled by a customised container. The container makes directories available via NFS and stores the data inside the container. In production, it is recommended to configure a dedicated NFS Server.

Start the NFS using the command:

```
docker run -d --net=host \
   --privileged --name nfs-server \
   katacoda/contained-nfs-server:centos7 \
   /exports/data-0001 /exports/data-0002
```

The NFS server exposes two directories, data-0001 and data-0002. In the next steps, this is used to store data.

Deploying Persistent Volume

For Kubernetes to understand the available NFS shares, it requires a PersistentVolume configuration. The PersistentVolume supports different protocols for storing data, such as AWS EBS volumes, GCE storage, OpenStack Cinder, Glusterfs and NFS. The configuration provides an abstraction between storage and API allowing for a consistent experience.

In the case of NFS, one PersistentVolume relates to one NFS directory. When a container has finished with the volume, the data can either be Retained for future use or the volume can be Recycled meaning all the data is deleted. The policy is defined by the persistentVolumeReclaimPolicy option.

Spec File:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: <friendly-name>
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    server: <server-name>
```

```
    path: <shared-path>
```

The spec defines additional metadata about the persistent volume, including how much space is available and if it has read/write access.

Task

Create two new PersistentVolume definitions to point at the two available NFS shares.

```
kubectl create -f nfs-0001.yaml
```

```
kubectl create -f nfs-0002.yaml
```

View the contents of the files using cat nfs-0001.yaml nfs-0002.yaml

Once created, view all PersistentVolumes in the cluster using kubectl get pv

Deploying Persistent Volume Claim

Once a Persistent Volume is available, applications can claim the volume for their use. The claim is designed to stop applications accidentally writing to the same volume and causing conflicts and data corruption.

The claim specifies the requirements for a volume. This includes read/write access and storage space required. An example is as follows:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: claim-mysql
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
```

```
      storage: 3Gi
```

Task

Create two claims for two different applications. A MySQL Pod will use one claim, the other used by an HTTP server.

```
kubectl create –f pvc–mysql.yaml
```

```
kubectl create –f pvc–http.yaml
```

View the contents of the files using cat pvc-mysql.yaml pvc-http.yaml

Once created, view all PersistentVolumesClaims in the cluster using

```
kubectl get pvc.
```

The claim will output which Volume the claim is mapped to claim.

Using Volume

When a deployment is defined, it can assign itself to a previous claim. The following snippet defines a volume mount for the directory /var/lib/mysql/data which is mapped to the storage mysql-persistent-storage. The storage called mysql-persistent-storage is mapped to the claim called claim-mysql.

```
spec:
     volumeMounts:
       – name: mysql–persistent–storage
         mountPath: /var/lib/mysql/data
 volumes:
   – name: mysql–persistent–storage
     persistentVolumeClaim:
       claimName: claim–mysql
```

```
```

Task

Launch two new Pods with Persistent Volume Claims. Volumes are mapped to the correct directory when the Pods start allowing applications to read/write as if it was a local directory.

```
kubectl create -f pod-mysql.yaml
```

```
kubectl create -f pod-www.yaml
```

Use the command below to view the definition of the Pods.

```
cat pod-mysql.yaml pod-www.yaml
```

You can see the status of the Pods starting using

```
kubectl get pods
```

If a Persistent Volume Claim is not assigned to a Persistent Volume, then the Pod will be in Pending mode until it becomes available. In the next step, we'll read/write data to the volume.

Read/Write Data

tore all database changes to the NFS Server while the HTTP Server will serve static from the NFS drive. When upgrading, restarting or moving containers to a different machine the data will still be accessible.

To test the HTTP server, write a 'Hello World' index.html homepage. In this scenario, we know the HTTP directory will be based on data-0001 as the volume definition hasn't driven enough space to satisfy the MySQL size requirement.

```
docker exec -it nfs-server bash -c "echo 'Hello World' > /exports/data-0001/
```

Based on the IP of the Pod, when accessing the Pod, it should return the expected response.

```
ip=$(kubectl get pod www -o yaml |grep podIP | awk '{split($0,a,":"); print
```

```
curl $ip
```

Update Data

When the data on the NFS share changes, then the Pod will read the newly updated data.

```
docker exec -it nfs-server bash -c "echo 'Hello NFS World' > /exports/data-0
```

```
curl $ip
```

Recreate Pod

Because a remote NFS server stores the data, if the Pod or the Host were to go down, then the data will still be available.

Task

Deleting a Pod will cause it to remove claims to any persistent volumes. New Pods can pick up and re-use the NFS share.

```
kubectl delete pod www
```

```
kubectl create -f pod-www2.yaml
```

```
ip=$(kubectl get pod www2 -o yaml |grep podIP | awk '{split($0,a,":"); print
```

The applications now use a remote NFS for their data storage. Depending on requirements, this same approach works with other storage engines such as GlusterFS, AWS EBS, GCE storage or OpenStack Cinder.

Join KubeDaily

kubelabs is maintained by collabnix.

This page was generated by GitHub Pages.