**StatefulSet:**

1. StatefulSet is the workload API object used to manage stateful applications.
2. Manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods
3. Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

**Example:**

1. Let’s just start with a simple example and we are tasked to deploy a database server. So we install and setup MySQL on the server and create a database. Our database is now operational. Other applications can now write data to our database.
2. To withstand failures we are tasked to deploy a High Availability solution. So we deploy additional servers and install MySQL on those as well. We have a blank database on the new servers.
3. So how do we replicate the data from the original database to the new databases on the new servers. Before we get into that
4. So back to our question on how do we replicate the database to the databases in the new server.
5. There are different topologies available. The most straight forward one being a **single master multi slave topology**, where all writes come in to the master server and reads can be served by either the master or any of the slaves servers.
6. So the master server should be setup first, before deploying the slaves. Once the slaves are deployed, perform an initial clone of the database from the master server to the first slave.
7. After the initial copy enable continuous replication from the master to that slave so that the database on the slave node is always in sync with the database on the master.
8. Note that both these slaves are configured with the address of the master host. When replication is initialized you point the slave to the master with the master’s hostname or address.
9. That way the slaves know where the master is.

**Kubernetes deployment problem with master slave:** g

* 1. Let us now go back to the world of Kubernetes and containers and try to deploy this setup.
  2. In the Kubernetes world, each of these instances, including the master and slaves are a POD part of a deployment.
  3. In step 1, we want the master to come up first and then the slaves. And in case of the slaves we want slave 1 to come up first,
  4. perform initial clone of data from the master, and we want slave 2 to come up next and clone data from slave 1.
  5. With deployments you can’t guarantee that order.
  6. With deployments all pods part of the deployment come up at the same time.
  7. So the first step can’t be implemented with a Deployment.
  8. As we have seen while working with deployments the pods come up with random names.
  9. So that won’t help us here. Even if we decide to designate one of these pods as the master,
  10. and use its name to configure the slaves,
  11. if that POD crashes and the deployment creates a new pod in it’s place, it’s going to come up with a completely new pod name.
  12. And now the slaves are pointing to an address that does not exist.
  13. And because of all of these, the remaining steps can’t be executed.

**StatefulSets fix the master slave architecture Problem:**

* 1. that’s where stateful sets come into play. Stateful sets are similar to deployment sets, as in they create PODs based on a template. But with some differences.
  2. With stateful sets, pods are created in a sequential order.
  3. After the first pod is deployed, it must be in a running and ready state before the next pod is deployed.
  4. So that helps us ensure master is deployed first and then slave 1 and then slave 2.
  5. Stateful sets assign a unique ordinal index to each POD – a number starting from 0 for the first pod and increments by
  6. Each Pod gets a name derived from this index, combined with the stateful set name.
  7. So the first pod gets mysql-0, the second pod mysql-1 and third mysql-2.
  8. SO no more random names. You can rely on these names going forward.
  9. We can designate the pod with the name mysql-0 as the master, and any other pods as slaves.
  10. Pod mysql-2 knows that it has to perform an initial clone of data from the pod mysql-1. If you scale up by deploying another pod, mysql-3, then it would know that it can perform a clone from mysql-2.
  11. To enable continuous replication, you can now point the slaves to the master at mysql-0. Even if the master fails, and the pod is recreated is created, it would still come up with the same name.
  12. Stateful sets maintain a sticky identity for each of their pods.
  13. The master is now always the master and available at the address mysql-0.
  14. And that is why you need stateful sets

**Limitations**

* 1. The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner based on the requested storage class, or pre-provisioned by an admin.
  2. Deleting and/or scaling a StatefulSet down will not delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources.
  3. StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.
  4. StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the StatefulSet down to 0 prior to deletion.
  5. When using Rolling Updates with the default Pod Management Policy (OrderedReady), it's possible to get into a broken state that requires manual intervention to repair.

**Stable Storage**

* 1. For each VolumeClaimTemplates entry defined in a StatefulSet, each Pod receives one PersistentVolumeClaim. In the nginx example above, each Pod receives a single PersistentVolume with a StorageClass of my-storage-class and 1 Gib of provisioned storage.
  2. If no StorageClass is specified, then the default StorageClass will be used. When a Pod is (re)scheduled onto a node, its volumeMounts mount the PersistentVolumes associated with its PersistentVolume Claims.
  3. Note that, the PersistentVolumes associated with the Pods' PersistentVolume Claims are not deleted when the Pods, or StatefulSet are deleted. This must be done manually.

**Example:**

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: nginx

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

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

selector:

matchLabels:

app: nginx # has to match .spec.template.metadata.labels

serviceName: "nginx"

replicas: 3 # by default is 1

minReadySeconds: 10 # by default is 0

template:

metadata:

labels:

app: nginx # has to match .spec.selector.matchLabels

spec:k

terminationGracePeriodSeconds: 10

containers:

- name: nginx

image: k8s.gcr.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "my-storage-class"

resources:

requests:

storage: 1Gi