Deployments vs StatefulSets vs DaemonSets

Kubernetes (K8s) is an open-source container orchestration system for automating deployment, scaling, and management of containerized applications.

Kubernetes provides a basic resource called Pod. A pod is the smallest deployable unit in Kubernetes which is actually a wrapper around containers. A pod can have one or more containers and you can pass different configuration to the container(s) using the pod's configuration e.g. passing environment variables, mounting volumes, having health checks, etc. For more details about pods, check Pod.

In this post, I will be discussing three different ways to deploy your application(pods) on Kubernetes using different Kubernetes resources. Below are 3 different resources that Kubernetes provides for deploying pods.

1. Deployments
2. StatefulSets
3. DaemonSets

There is one other type ReplicationController but Kubernetes now favors Deployments as Deployments configure ReplicaSets to support replication.

For detailed differences between the 3 resources, I will be deploying a sample counter app, which logs and increments the count from a counter file like 1,2,3,…. I am using the counter file from a Persistent Volume to detail the differences between the Deployments, StatefulSets and DaemonSets. The manifests files to deploy the following resources can be found in the counter app.

Deployments

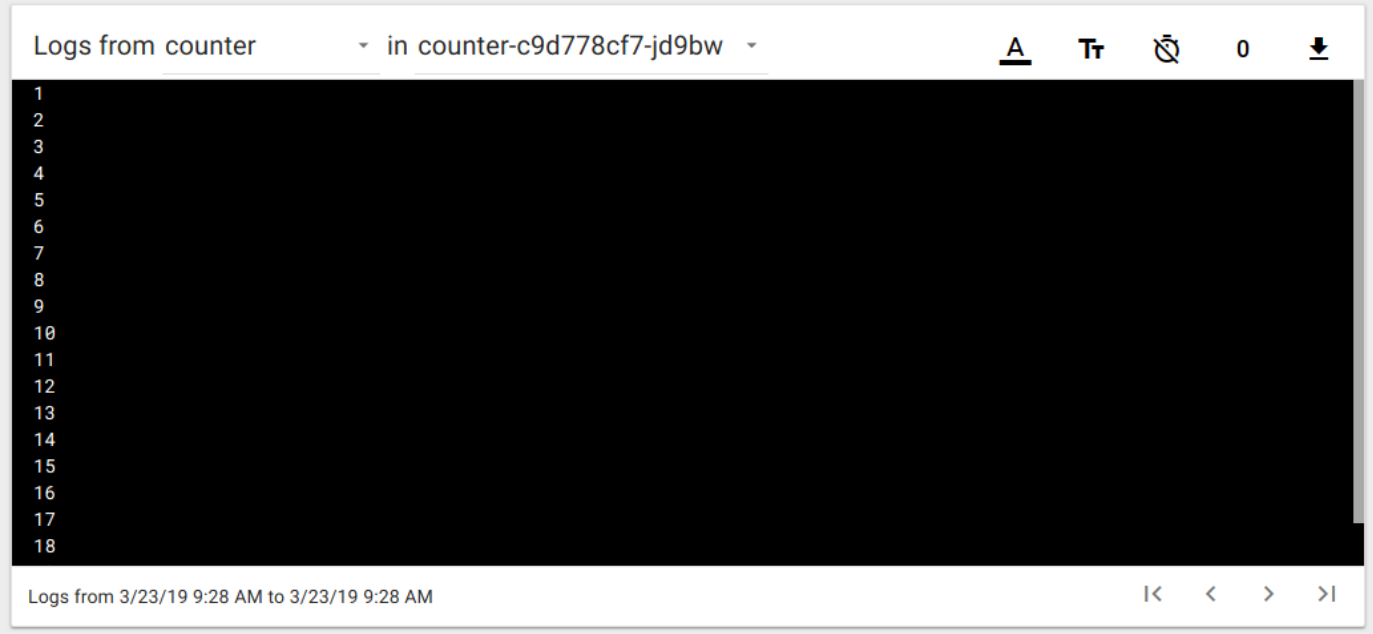
Deployment is the easiest and most used resource for deploying your application. It is a Kubernetes controller that matches the current state of your cluster to the desired state mentioned in the Deployment manifest. e.g. If you create a deployment with 1 replica, it will check that the desired state of ReplicaSet is 1 and current state is 0, so it will create a ReplicaSet, which will further create the pod. If you create a deployment with name **counter**, it will create a ReplicaSet with name **counter-<replica-set-id>**, which will further create a Pod with name **counter-<replica-set->-<pod-id>.**

Deployments are usually used for stateless applications. However, you can save the state of deployment by attaching a Persistent Volume to it and make it stateful, but all the pods of a deployment will be sharing the same Volume and data across all of them will be same.

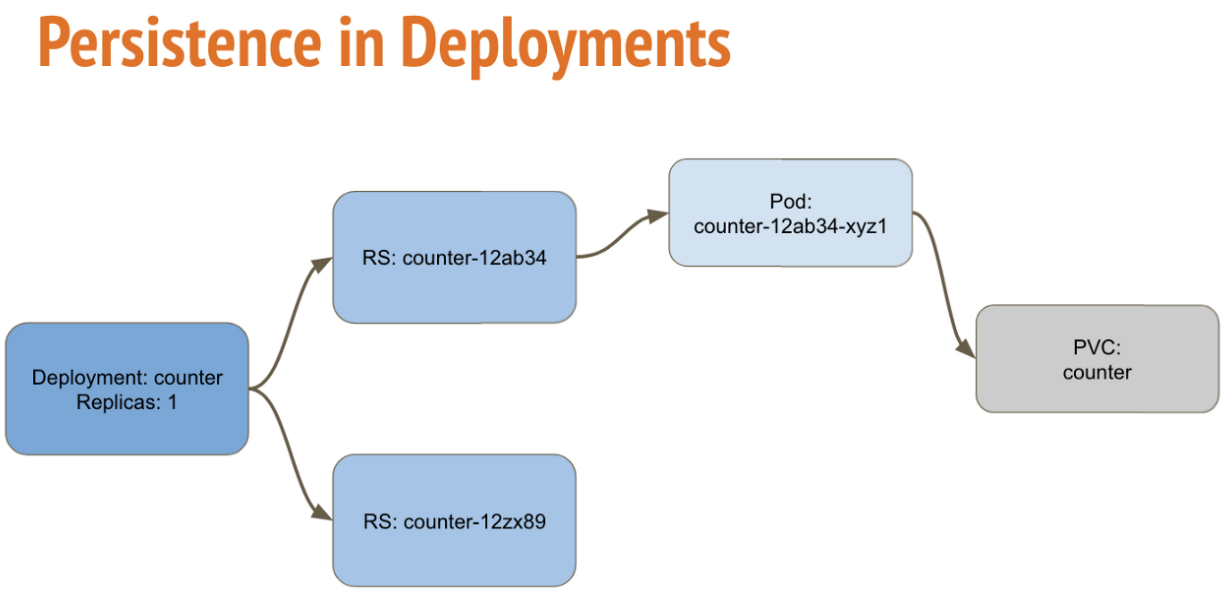
For deploying the sample counter app using a deployment, we will be using the following manifest, you can deploy it by copying the below manifest and saving it in a file e.g. deployment.yaml, and then applying by

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

If you deploy the above deployment, and see the logs of the pod, you will be able to see the log in order like 1,2,3,…



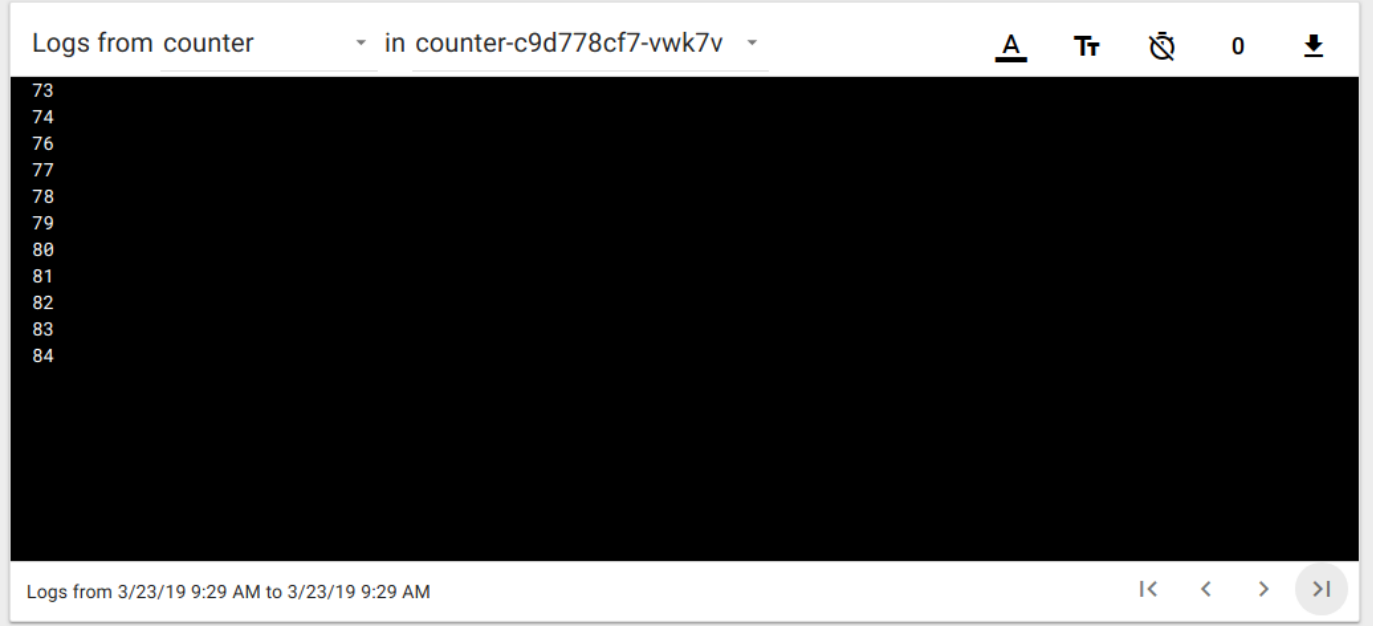
The logs from the 1st pod, Note the name of pod, counter-c9d778cf7-jd9bw, **counter** — deployment name, **c9d778cf7** — ReplicaSet id, **jd9b2** — Pod id



Now if you scale the deployment to 2 by running

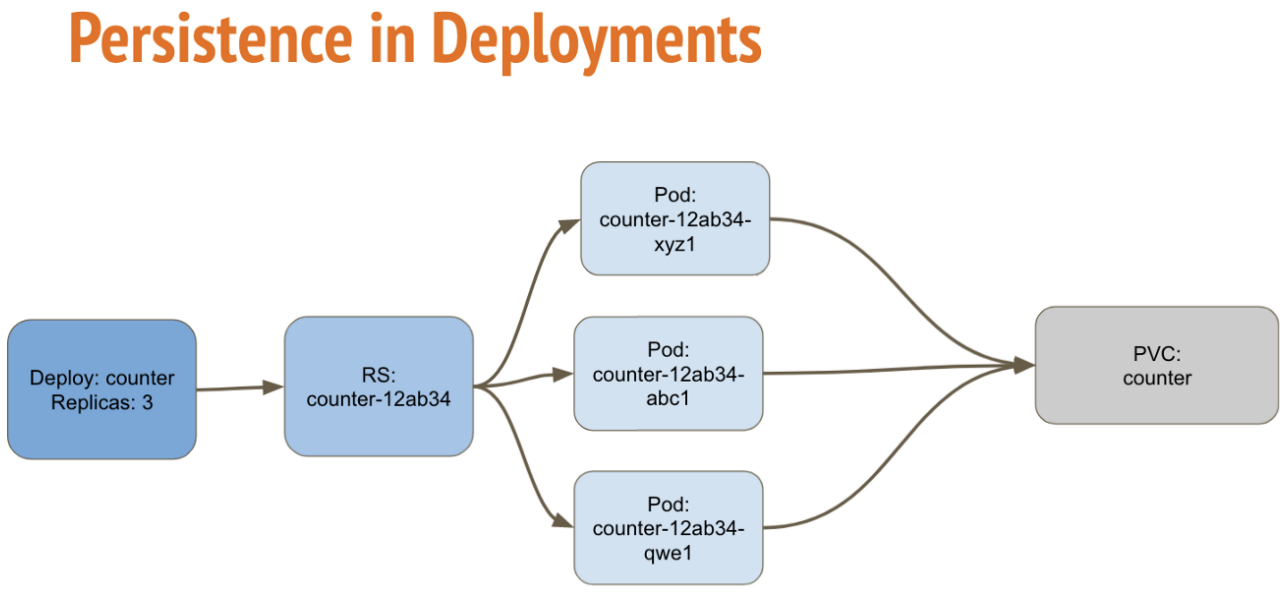
```
kubectl scale deployment counter --replicas=2
```

you can see a new pod created, if you check the logs of the new pod, its logs will not start from 1 rather it will start from the last number of the 1st pod.



The logs from the 2nd pod, Note the name of pod, **counter-c9d778cf7-jd9bw.** Deployment and replicaset id are same only pod id is different than previous pod.

If you see the logs, they are starting from 73, meaning that the previous pod had written till 72 in the file and they both are sharing the same file and volume and data is shared across all pods of a Deployment. Also if you check the Persistent Volume Claims(PVCs), only one PVC will be created that both the pods will be sharing so it can cause Data Inconsistency.



Persistence for Deployments sharing single Volume can cause Data Inconsistency

Deployments, as discussed, creates a ReplicaSet which then creates a Pod so whenever you update the deployment using RollingUpdate (default) strategy, a new ReplicaSet is created and the Deployment moves the Pods from the old ReplicaSet to the new one at a controlled rate. Rolling Update means that the previous ReplicaSet doesn't scale to 0 unless the new ReplicaSet is up & running ensuring 100% uptime. If an error occurs while updating, the new ReplicaSet will never be in **Ready** state, so old ReplicaSet will not terminate again ensuring 100% uptime in case of a failed update. In Deployments, you can also manually roll back to a previous ReplicaSet, if needed in case if your new feature is not working as expected.

StatefulSets

StatefulSet(stable-GA in k8s v1.9) is a Kubernetes resource used to manage stateful applications. It manages the deployment and scaling of a set of Pods, and provides guarantee about the ordering and uniqueness of these Pods.

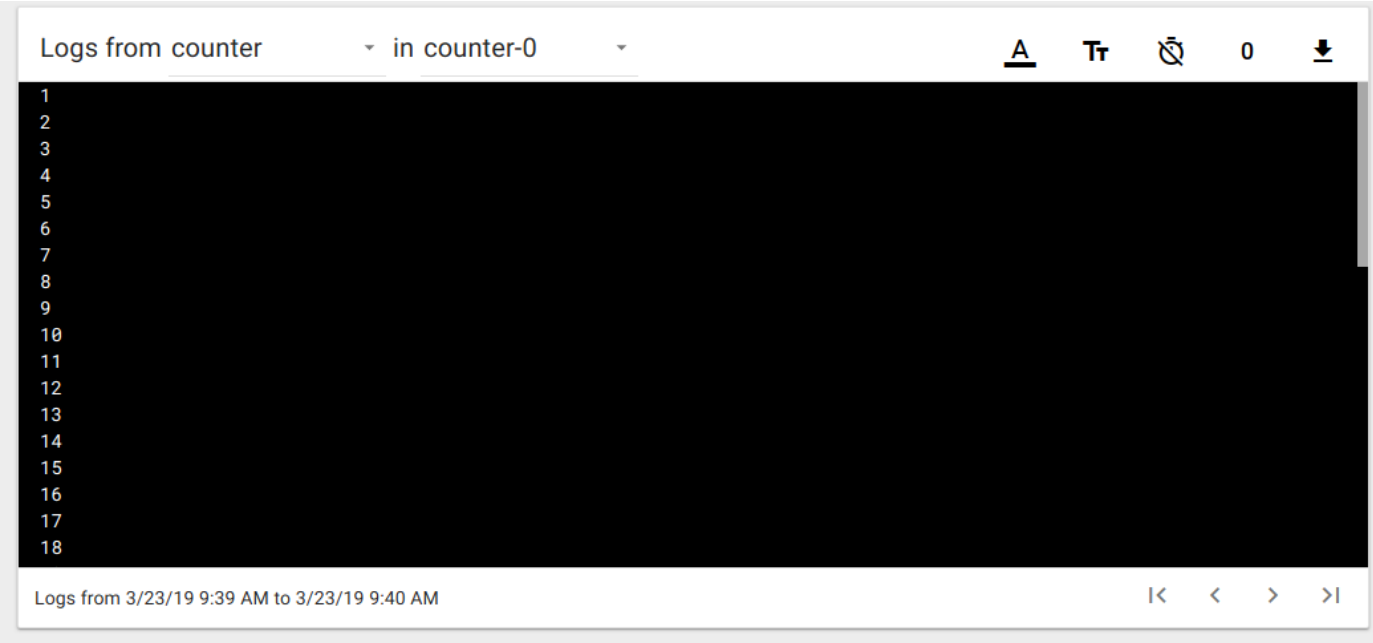
StatefulSet is also a Controller but unlike Deployments, it doesn't create ReplicaSet rather itself creates the Pod with a unique naming convention. e.g. If you create a StatefulSet with name **counter,** it will create a pod with name **counter-0,** and for multiple replicas of a statefulset, their names will increment like **counter-0**, **counter-1**, **counter-2, etc**

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.

For deploying the sample counter app using a statefulset, we will be using the following manifest. you can deploy it by copying the below manifest and saving it in a file e.g. statefulset.yaml, and then applying by

```
kubectl apply -f statefulset.yaml
```

If you deploy the above statefulset, and see the logs of the pod, you will be able to see the log in order like 1,2,3,…

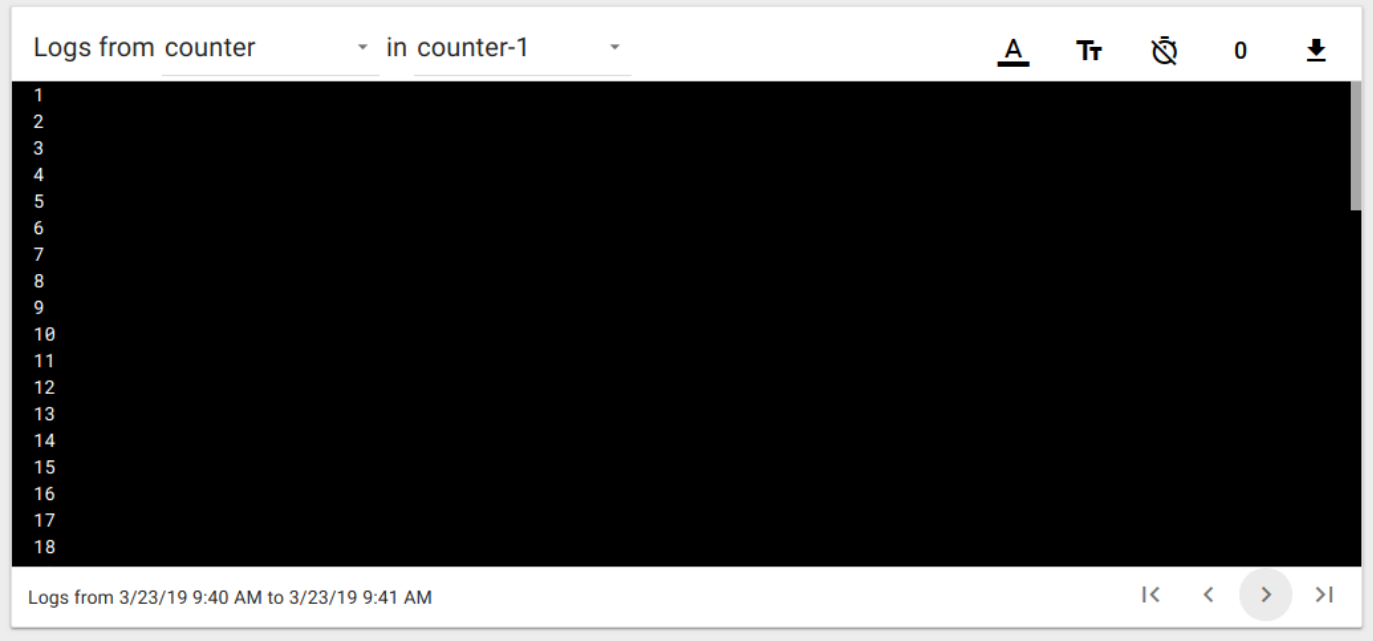


The logs from the 1st pod. Note the name of the pod is **counter-0**

Here, you can see the logs start from 1. Now if we scale up the statefulset to 3 replicas by run

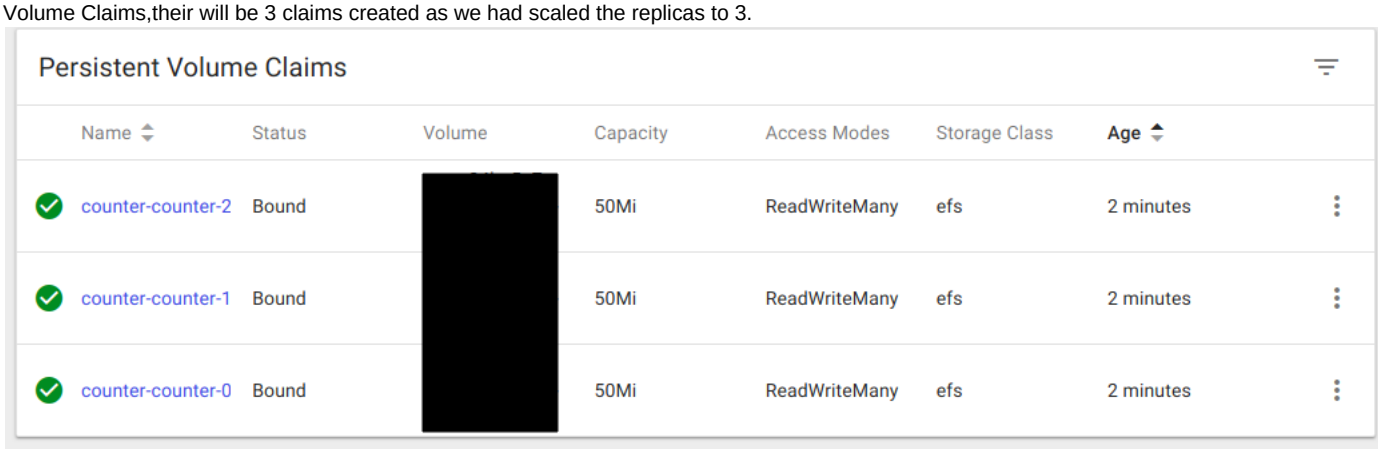
```
kubectl scale statefulsets counter --replicas=3
```

it will first create a new pod **counter-1,** and once that pod is ready, then another pod **counter-2.** The new pods will have their own Volume and if you check the logs, the count will again start from 1 for the new pods, unlike in Deployments as we saw earlier.

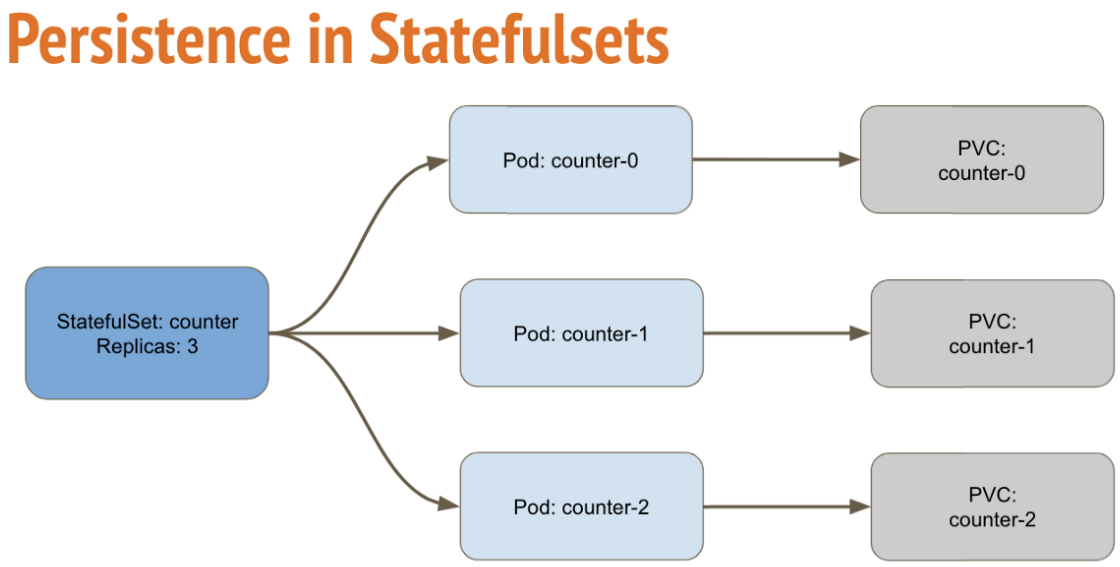


The logs from 2nd pod. Note the name is **counter-1.**

Here, the logs are again starting from 1, as this pod has its own Volume, so it doesn't read the file of 1st pod. And if we see the Persistent Volume Claims,their will be 3 claims created as we had scaled the replicas to 3.



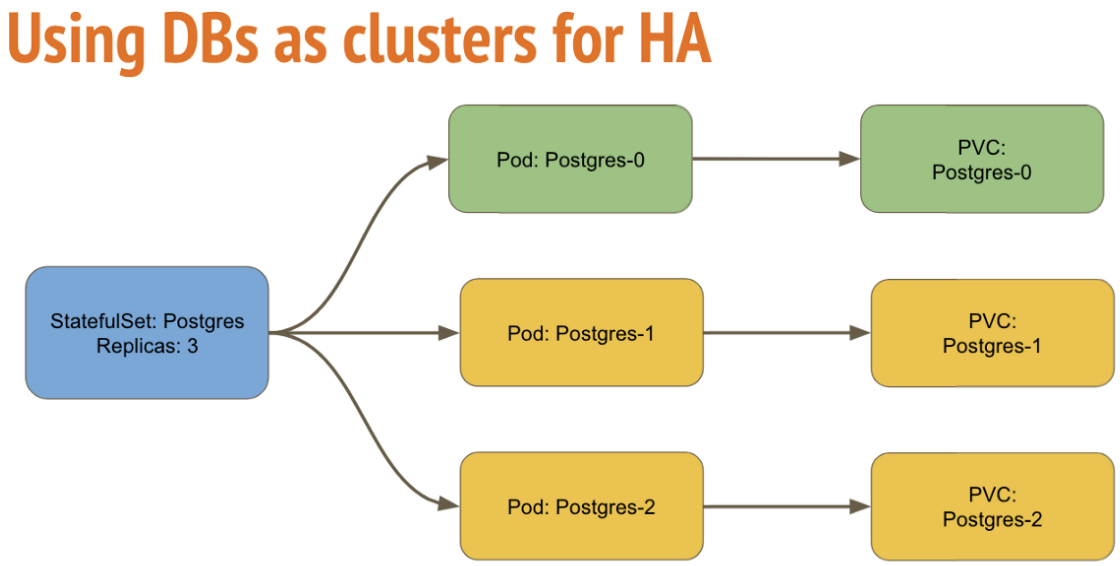
PVC



Persistence for StatefulSets each having its own Volume

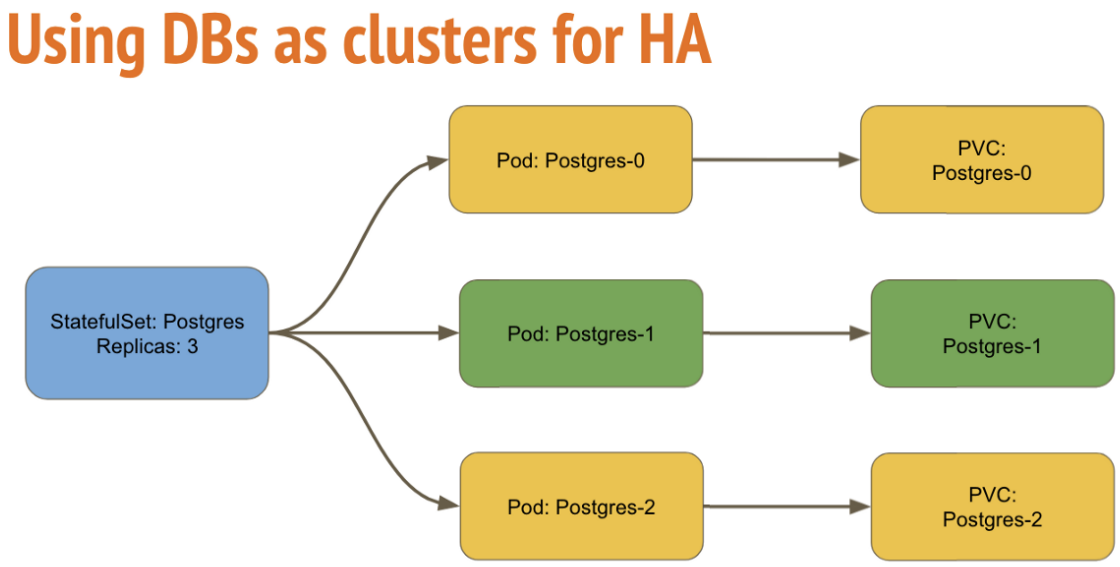
StatefulSets don't create ReplicaSet or anything of that sort, so you cant rollback a StatefulSet to a previous version. You can only delete or scale up/down the Statefulset. If you update a StatefulSet, it also performs RollingUpdate i.e. one replica pod will go down and the updated pod will come up, then the next replica pod will go down in same manner e.g. If I change the image of the above StatefulSet, the **counter-2** will terminate and once it terminates completely, then **counter-2** will be recreated and **counter-1** will be terminated at the same time, similarly for next replica i. e. counter-0. If an error occurs while updating, so only **counter-2** will be down, counter-1 & counter-0 will still be up, running on previous stable version. Unlike Deployments, you cannot roll back to any previous version of a StatefulSet.

StatefulSets are useful in case of Databases especially when we need Highly Available Databases in production as we create a cluster of Database replicas with one being the primary replica and others being the secondary replicas. The primary will be responsible for read/write operations and secondary for read only operations and they will be syncing data with the primary one.



Using StatefulSets to provision Postgres as Highly Available Database

If the primary goes down, any of the secondary replica will become primary and the StatefulSet controller will create a new replica in account of the one that went down, which will now become a secondary replica.



In case if postgres-0 went down, now postgres-1 became the primary replica

DaemonSet

A DaemonSet is a controller that ensures that the pod runs on all the nodes of the cluster. If a node is added/removed from a cluster, DaemonSet automatically adds/deletes the pod.

Some typical use cases of a DaemonSet is to run cluster level applications like:

- **Monitoring Exporters**: You would want to monitor all the nodes of your cluster so you will need to run a monitor on all the nodes of the cluster like NodeExporter.
- **Logs Collection Daemon:** You would want to export logs from all nodes so you would need a DaemonSet of log collector like Fluentd to export logs from all your nodes.

However, Daemonset automatically doesn't run on nodes which have a taint e.g. Master. You will have to specify the tolerations for it on the pod.

Taints are a way of telling the nodes to repel the pods i.e. no pods will be schedule on this node unless the pod tolerates the node with the same toleration. The master node is already tainted by:

Which means it will repel all pods that do not tolerate this taint, so for daemonset to run on all nodes, you would have to add following tolerations on DaemonSet

which means that it should tolerate all nodes.

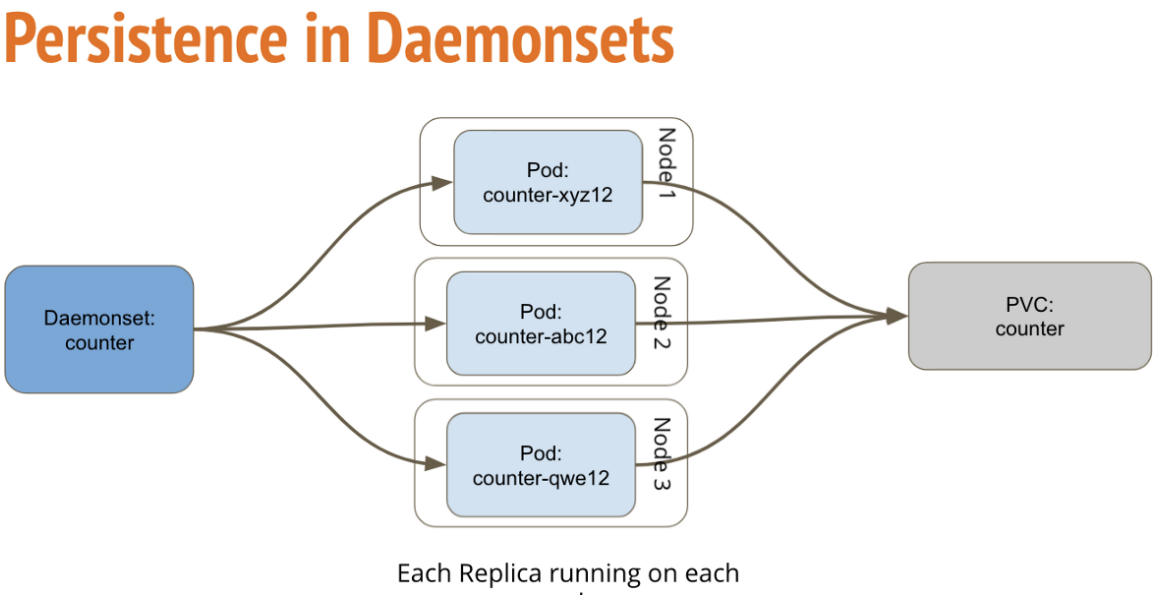
For deploying the sample counter app using a daemonset, we will be using the following manifest. you can deploy it by copying the below manifest and saving it in a file e.g. daemonset.yaml, and then applying by

```
kubectl apply -f daemonset.yaml
```

When you deploy the daemonset, it will create pods equal to the number of nodes. In terms of behavior, it will behave the same as Deployments i. e. all pods will share the same Persistent Volume.

Logs of a pod of DaemonSet

These are the logs of a pod of DaemonSet, you can see the logs are not in order, meaning that all pods are sharing the same Volume. Also only one PVC will be created that all pods will be sharing.



Each Replica running on each

Similar to ReplicaSet, but DaemonSets run one replica per node in the cluster

If you update a DaemonSet, it also performs RollingUpdate i.e. one pod will go down and the updated pod will come up, then the next replica pod will go down in same manner e.g. If I change the image of the above DaemonSet, one pod will go down, and when it comes back up with the updated image, only then the next pod will terminate and so on. If an error occurs while updating, so only one pod will be down, all other pods will still be up, running on previous stable version. Unlike Deployments, you cannot roll back your DaemonSet to a previous version.

That's all, these are the main resources to deploy your applications (containers) on Kubernetes.