StatefulSets

A StatefulSet runs a group of Pods, and maintains a sticky identity for each of those Pods. This is useful for managing applications that need persistent storage or a stable, unique network identity.

StatefulSet is the workload API object used to manage stateful applications.

Manages the deployment and scaling of a set of Pods, *and provides guarantees about the ordering and uniqueness* of these Pods.

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 its Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.

Using StatefulSets

StatefulSets are valuable for applications that require one or more of the following.

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.

In the above, stable is synonymous with persistence across Pod (re)scheduling. If an application doesn't require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application using a workload object that provides a set of stateless replicas. Deployment or ReplicaSet may be better suited to your stateless needs.

Limitations

- 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.
- 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.
- StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.
- 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.
- 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.

Components

The example below demonstrates the components of a StatefulSet.

```yaml
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:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: registry.k8s.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
```

Note: This example uses the `ReadWriteOnce` access mode, for simplicity. For production use, the Kubernetes project recommends using the `ReadWriteOncePod` access mode instead.

In the above example:

- A Headless Service, named `nginx`, is used to control the network domain.
- The StatefulSet, named `web`, has a Spec that indicates that 3 replicas of the nginx container will be launched in unique Pods.
- The `volumeClaimTemplates` will provide stable storage using PersistentVolumes provisioned by a PersistentVolume Provisioner.

The name of a StatefulSet object must be a valid DNS label.

Pod Selector

You must set the `.spec.selector` field of a StatefulSet to match the labels of its `.spec.template.metadata.labels`. Failing to specify a matching Pod Selector will result in a validation error during StatefulSet creation.

Volume Claim Templates

You can set the `.spec.volumeClaimTemplates` which can provide stable storage using PersistentVolumes provisioned by a PersistentVolume Provisioner.

Minimum ready seconds

FEATURE STATE: `Kubernetes v1.25 [stable]`

`.spec.minReadySeconds` is an optional field that specifies the minimum number of seconds for which a newly created Pod should be running and ready without any of its containers crashing, for it to be considered available. This is used to check progression of a rollout when using a Rolling Update strategy. This field defaults to 0 (the Pod will be considered available as soon as it is ready). To learn more about when a Pod is considered ready, see Container Probes.

Pod Identity

StatefulSet Pods have a unique identity that consists of an ordinal, a stable network identity, and stable storage. The identity sticks to the Pod, regardless of which node it's (re)scheduled on.

Ordinal Index

For a StatefulSet with N replicas, each Pod in the StatefulSet will be assigned an integer ordinal, that is unique over the Set. By default, pods will be assigned ordinals from 0 up through N-1. The StatefulSet controller will also add a pod label with this index: `apps.kubernetes.io/pod-index`.

Start ordinal

FEATURE STATE: `Kubernetes v1.27 [beta]`

`.spec.ordinals` is an optional field that allows you to configure the integer ordinals assigned to each Pod. It defaults to nil. You must enable the `StatefulSetStartOrdinal` feature gate to use this field. Once enabled, you can configure the following options:

- `.spec.ordinals.start`: If the `.spec.ordinals.start` field is set, Pods will be assigned ordinals from `.spec.ordinals.start` up through `.spec.ordinals.start + .spec.replicas - 1`.

Stable Network ID

Each Pod in a StatefulSet derives its hostname from the name of the StatefulSet and the ordinal of the Pod. The pattern for the constructed hostname is `$(statefulset name)-$(ordinal)`. The example above will create three Pods named `web-0,web-1,web-2`. A StatefulSet can use a Headless Service to control the domain of its Pods. The domain managed by this Service takes the form: `$(service name).$(namespace).svc.cluster.local`, where "cluster.local" is the cluster domain. As each Pod is created, it gets a matching DNS subdomain, taking the form: `$(podname).$(governing service domain)`, where the governing service is defined by the `serviceName` field on the StatefulSet.

Depending on how DNS is configured in your cluster, you may not be able to look up the DNS name for a newly-run Pod immediately. This behavior can occur when other clients in the cluster have already sent queries for the hostname of the Pod before it was created. Negative caching (normal in DNS) means that the results of previous failed lookups are remembered and reused, even after the Pod is running, for at least a few seconds.

If you need to discover Pods promptly after they are created, you have a few options:

- Query the Kubernetes API directly (for example, using a watch) rather than relying on DNS lookups.
- Decrease the time of caching in your Kubernetes DNS provider (typically this means editing the config map for CoreDNS, which currently caches for 30 seconds).

As mentioned in the limitations section, you are responsible for creating the Headless Service responsible for the network identity of the pods.

Here are some examples of choices for Cluster Domain, Service name, StatefulSet name, and how that affects the DNS names for the StatefulSet's Pods.

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	Pod DNS	Pod Hostname
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	web-{0..N-1}.nginx.default.svc.cluster.local	web-{0..N-1}
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	web-{0..N-1}.nginx.foo.svc.cluster.local	web-{0..N-1}
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	web-{0..N-1}.nginx.foo.svc.kube.local	web-{0..N-1}

Note: Cluster Domain will be set to `cluster.local` unless otherwise configured.

Stable Storage

For each VolumeClaimTemplate 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. 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. 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.

Pod Name Label

When the StatefulSet controller creates a Pod, it adds a label, `statefulset.kubernetes.io/pod-name`, that is set to the name of the Pod. This label allows you to attach a Service to a specific Pod in the StatefulSet.

Pod index label

FEATURE STATE: `Kubernetes v1.28 [beta]`

When the StatefulSet controller creates a Pod, the new Pod is labelled with `apps.kubernetes.io/pod-index`. The value of this label is the ordinal index of the Pod. This label allows you to route traffic to a particular pod index, filter logs/metrics using the pod index label, and more. Note the feature gate `PodIndexLabel` must be enabled for this feature, and it is enabled by default.

Deployment and Scaling Guarantees

- For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from {0..N-1}.
- When Pods are being deleted, they are terminated in reverse order, from {N-1..0}.
- Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready.
- Before a Pod is terminated, all of its successors must be completely shutdown.

The StatefulSet should not specify a `pod.Spec.TerminationGracePeriodSeconds` of 0. This practice is unsafe and strongly discouraged. For further explanation, please refer to force deleting StatefulSet Pods.

When the nginx example above is created, three Pods will be deployed in the order web-0, web-1, web-2. web-1 will not be deployed before web-0 is Running and Ready, and web-2 will not be deployed until web-1 is Running and Ready. If web-0 should fail, after web-1 is Running and Ready, but before web-2 is launched, web-2 will not be launched until web-0 is successfully relaunched and becomes Running and Ready.

If a user were to scale the deployed example by patching the StatefulSet such that `replicas=1`, web-2 would be terminated first. web-1 would not be terminated until web-2 is fully shutdown and deleted. If web-0 were to fail after web-2 has been terminated and is completely shutdown, but prior to web-1's termination, web-1 would not be terminated until web-0 is Running and Ready.

Pod Management Policies

StatefulSet allows you to relax its ordering guarantees while preserving its uniqueness and identity guarantees via its `.spec.podManagementPolicy` field.

OrderedReady Pod Management

`OrderedReady` pod management is the default for StatefulSets. It implements the behavior described above.

Parallel Pod Management

`Parallel` pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod. This option only affects the behavior for scaling operations. Updates are not affected.

Update strategies

A StatefulSet's `.spec.updateStrategy` field allows you to configure and disable automated rolling updates for containers, labels, resource request/limits, and annotations for the Pods in a StatefulSet. There are two possible values:

`OnDelete`

When a StatefulSet's `.spec.updateStrategy.type` is set to `OnDelete`, the StatefulSet controller will not automatically update the Pods in a StatefulSet. Users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a StatefulSet's `.spec.template`.

`RollingUpdate`

The `RollingUpdate` update strategy implements automated, rolling updates for the Pods in a StatefulSet. This is the default update strategy.

Rolling Updates

When a StatefulSet's `.spec.updateStrategy.type` is set to `RollingUpdate`, the StatefulSet controller will delete and recreate each Pod in the StatefulSet. It will proceed in the same order as Pod termination (from the largest ordinal to the smallest), updating each Pod one at a time.

The Kubernetes control plane waits until an updated Pod is Running and Ready prior to updating its predecessor. If you have set `.spec.minReadySeconds` (see Minimum Ready Seconds), the control plane additionally waits that amount of time after the Pod turns ready, before moving on.

Partitioned rolling updates

The `RollingUpdate` update strategy can be partitioned, by specifying a `.spec.updateStrategy.rollingUpdate.partition`. If a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the StatefulSet's `.spec.template` is updated. All Pods with an ordinal that is less than the partition will not be updated, and, even if they are deleted, they will be recreated at the previous version. If a StatefulSet's `.spec.updateStrategy.rollingUpdate.partition` is greater than its `.spec.replicas`, updates to its `.spec.template` will not be propagated to its Pods. In most cases you will not need to use a partition, but they are useful if you want to stage an update, roll out a canary, or perform a phased roll out.

Maximum unavailable Pods

FEATURE STATE: `Kubernetes v1.24 [alpha]`

You can control the maximum number of Pods that can be unavailable during an update by specifying the `.spec.updateStrategy.rollingUpdate.maxUnavailable` field. The value can be an absolute number (for example, `5`) or a percentage of desired Pods (for example, `10%`). Absolute number is calculated from the percentage value by rounding it up. This field cannot be 0. The default setting is 1.

This field applies to all Pods in the range `0` to `replicas - 1`. If there is any unavailable Pod in the range `0` to `replicas - 1`, it will be counted towards `maxUnavailable`.

> **Note:** The `maxUnavailable` field is in Alpha stage and it is honored only by API servers that are running with the `MaxUnavailableStatefulSet` feature gate enabled.

Forced rollback

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.

If you update the Pod template to a configuration that never becomes Running and Ready (for example, due to a bad binary or application-level configuration error), StatefulSet will stop the rollout and wait.

In this state, it's not enough to revert the Pod template to a good configuration. Due to a known issue, StatefulSet will continue to wait for the broken Pod to become Ready (which never happens) before it will attempt to revert it back to the working configuration.

After reverting the template, you must also delete any Pods that StatefulSet had already attempted to run with the bad configuration. StatefulSet will then begin to recreate the Pods using the reverted template.

PersistentVolumeClaim retention

FEATURE STATE: `Kubernetes v1.27 [beta]`

The optional `.spec.persistentVolumeClaimRetentionPolicy` field controls if and how PVCs are deleted during the lifecycle of a StatefulSet. You must enable the `StatefulSetAutoDeletePVC` feature gate on the API server and the controller manager to use this field. Once enabled, there are two policies you can configure for each StatefulSet:

`whenDeleted`

configures the volume retention behavior that applies when the StatefulSet is deleted

`whenScaled`

configures the volume retention behavior that applies when the replica count of the StatefulSet is reduced; for example, when scaling down the set.

For each policy that you can configure, you can set the value to either `Delete` or `Retain`.

`Delete`

The PVCs created from the StatefulSet `volumeClaimTemplate` are deleted for each Pod affected by the policy. With the `whenDeleted` policy all PVCs from the `volumeClaimTemplate` are deleted after their Pods have been deleted. With the `whenScaled` policy, only PVCs corresponding to Pod replicas being scaled down are deleted, after their Pods have been deleted.

`Retain` (default)

PVCs from the `volumeClaimTemplate` are not affected when their Pod is deleted. This is the behavior before this new feature.

Bear in mind that these policies **only** apply when Pods are being removed due to the StatefulSet being deleted or scaled down. For example, if a Pod associated with a StatefulSet fails due to node failure, and the control plane creates a replacement Pod, the StatefulSet retains the existing PVC. The existing volume is unaffected, and the cluster will attach it to the node where the new Pod is about to launch.

The default for policies is `Retain`, matching the StatefulSet behavior before this new feature.

Here is an example policy.

```yaml
apiVersion: apps/v1
kind: StatefulSet
...
spec:
  persistentVolumeClaimRetentionPolicy:
    whenDeleted: Retain
    whenScaled: Delete
...
```

The StatefulSet controller adds owner references to its PVCs, which are then deleted by the garbage collector after the Pod is terminated. This enables the Pod to cleanly unmount all volumes before the PVCs are deleted (and before the backing PV and volume are deleted, depending on the retain policy). When you set the `whenDeleted` policy to `Delete`, an owner reference to the StatefulSet instance is placed on all PVCs associated with that StatefulSet.

The `whenScaled` policy must delete PVCs only when a Pod is scaled down, and not when a Pod is deleted for another reason. When reconciling, the StatefulSet controller compares its desired replica count to the actual Pods present on the cluster. Any StatefulSet Pod whose id greater than the replica count is condemned and marked for deletion. If the `whenScaled` policy is `Delete`, the condemned Pods are first set as owners to the associated StatefulSet template PVCs, before the Pod is deleted. This causes the PVCs to be garbage collected after only the condemned Pods have terminated.

This means that if the controller crashes and restarts, no Pod will be deleted before its owner reference has been updated appropriate to the policy. If a condemned Pod is force-deleted while the controller is down, the owner reference may or may not have been set up, depending on when the controller crashed. It may take several reconcile loops to update the owner references, so some condemned Pods may have set up owner references and others may not. For this reason we recommend waiting for the controller to come back up, which will verify owner references before terminating Pods. If that is not possible, the operator should verify the owner references on PVCs to ensure the expected objects are deleted when Pods are force-deleted.

Replicas

`.spec.replicas` is an optional field that specifies the number of desired Pods. It defaults to 1.

Should you manually scale a deployment, example via `kubectl scale statefulset statefulset --replicas=X`, and then you update that StatefulSet based on a manifest (for example: by running `kubectl apply -f statefulset.yaml`), then applying that manifest overwrites the manual scaling that you previously did.

If a HorizontalPodAutoscaler (or any similar API for horizontal scaling) is managing scaling for a Statefulset, don't set `.spec.replicas`. Instead, allow the Kubernetes control plane to manage the `.spec.replicas` field automatically.

What's next

- Learn about Pods.
- Find out how to use StatefulSets
  - Follow an example of deploying a stateful application.
  - Follow an example of deploying Cassandra with Stateful Sets.
  - Follow an example of running a replicated stateful application.
  - Learn how to scale a StatefulSet.
  - Learn what's involved when you delete a StatefulSet.
  - Learn how to configure a Pod to use a volume for storage.
  - Learn how to configure a Pod to use a PersistentVolume for storage.
- `StatefulSet` is a top-level resource in the Kubernetes REST API. Read the StatefulSet object definition to understand the API for stateful sets.
- Read about PodDisruptionBudget and how you can use it to manage application availability during disruptions.

Feedback

Was this page helpful?

Yes No

Last modified October 10, 2023 at 7:55 AM PST: Graduate ReadWriteOncePod to GA (c07ce392e4)