CHAPTER 10

Singleton Service

The *Singleton Service* pattern ensures that only one instance o an application is active at a time and yet is highly available. This pattern can be implemented from within the application or delegated fully to Kubernetes.

Problem

One of the main capabilities provided by Kubernetes is the ability to easily and transparently scale applications. Pods can scale imperatively with a single command such as `kubectl scale`, or declaratively through a controller definition such as ReplicaSet, or even dynamically based on the application load, as we describe in Chapter 29, "Elastic Scale". By running multiple instances of the same service (not a Kubernetes Service but a component of a distributed application represented by a Pod), the system usually increases throughput and availability. The availability increases because if one instance of a service becomes unhealthy, the request dispatcher forwards future requests to other healthy instances. In Kubernetes, multiple instances are the replicas of a Pod, and the Service resource is responsible for the request distribution and load balancing.

However, in some cases, only one instance of a service is allowed to run at a time. For example, if there is a periodically executed task in a service and multiple instances of the same service, every instance will trigger the task at the scheduled intervals, leading to duplicates rather than having only one task fired as expected. Another example is a service that performs polling on specific resources (a filesystem or database) and we want to ensure that only a single instance and maybe even a single thread performs the polling and processing. A third case occurs when we have to consume messages from a messages broker in an order-preserving manner with a single-threaded consumer that is also a singleton service.

In all these and similar situations, we need some control over how many instances \ of a service are active at a time (usually only one is required), while still ensuring high availability, regardless of how many instances have been started and kept running.

Solution

Running multiple replicas of the same Pod creates an *active-active* topology, where all instances of a service are active. What we need is an *active-passive* topology, where only one instance is active and all the other instances are passive. Fundamentally, this can be achieved at two possible levels: out-of-application and in-application locking.

Out-of-Application Locking

As the name suggests, this mechanism relies on a managing process that is outside of the application to ensure that only a single instance of the application is running. The application implementation itself is not aware of this constraint and is run as a singleton instance. From this perspective, it is similar to having a Java class that is instantiated only once by the managing runtime (such as the Spring Framework). The class implementation is not aware that it is run as a singleton, nor that it contains any code constructs to prevent instantiating multiple instances.

Figure 10-1 shows how to implement out-of-application locking with the help of a StatefulSet or ReplicaSet controller with one replica.

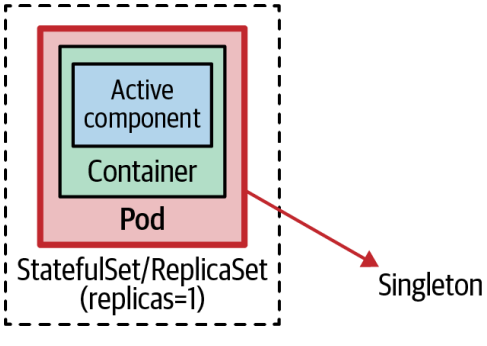

Figure 10-1. Out-of-application locking mechanism

The way to achieve this in Kubernetes is to start a single Pod. This activity alone does not ensure the singleton Pod is highly available. What we have to do is also back the Pod with a controller such as a ReplicaSet that turns the singleton Pod into a highly available singleton. This topology is not exactly *active-passive* (there is no passive instance), but it has the same effect, as Kubernetes ensures that one instance of the Pod is running at all times. In addition, the single Pod instance is highly available,

thanks to the controller performing health checks as described in Chapter 4, "Health Probe", and healing the Pod in case of failures.

The main thing to keep an eye on with this approach is the replica count, which should not be changed accidentally. In this section, you will see how we can voluntarily decrease the replica count through PodDisruptionBudget, but there is no platform-level mechanism to prevent an increase of the replica count.

It's not entirely true that only one instance is running at all times, especially when things go wrong. Kubernetes primitives such as ReplicaSet favor availability over consistency—a deliberate decision for achieving highly available and scalable distributed systems. That means a ReplicaSet applies "at least" rather than "at most" semantics for its replicas. If we configure a ReplicaSet to be a singleton with `replicas: 1`, the controller makes sure at least one instance is always running, but occasionally it can be more instances.

The most popular corner case here occurs when a node with a controller-managed Pod becomes unhealthy and disconnects from the rest of the Kubernetes cluster. In this scenario, a ReplicaSet controller starts another Pod instance on a healthy node (assuming there is enough capacity), without ensuring the Pod on the disconnected node is shut down. Similarly, when changing the number of replicas or relocating Pods to different nodes, the number of Pods can temporarily go above the desired number. That temporary increase is done with the intention of ensuring high availability and avoiding disruption, as needed for stateless and scalable applications.

Singletons can be resilient and recover, but by definition, they are not highly available. Singletons typically favor consistency over availability. The Kubernetes resource that also favors consistency over availability and provides the desired strict singleton guarantees is the StatefulSet. If ReplicaSets do not provide the desired guarantees for your application, and you have strict singleton requirements, StatefulSets might be the answer. StatefulSets are intended for stateful applications and offer many features, including stronger singleton guarantees, but they come with increased complexity as well. We discuss concerns around singletons and cover StatefulSets in more detail in Chapter 12, "Stateful Service".

Typically, singleton applications running in Pods on Kubernetes open outgoing connections to message brokers, relational databases, file servers, or other systems running on other Pods or external systems. However, occasionally, your singleton Pod may need to accept incoming connections, and the way to enable that on Kubernetes is through the Service resource.

We cover Kubernetes Services in depth in Chapter 13, "Service Discovery", but let's discuss briefly the part that applies to singletons here. A regular Service (with `type: ClusterIP`) creates a virtual IP and performs load balancing among all the Pod instances that its selector matches. However, a singleton Pod managed through

a StatefulSet has only one Pod and a stable network identity. In such a case, it is better to create a *headless Service* (by setting both `type: ClusterIP` and `clusterIP: None`). It is called *headless* because such a Service doesn't have a virtual IP address, kube-proxy doesn't handle these Services, and the platform performs no proxying.

However, such a Service is still useful because a headless Service with selectors creates endpoint records in the API Server and generates DNS A records for the matching Pod(s). With that, a DNS lookup for the Service does not return its virtual IP but instead the IP address(es) of the backing Pod(s). That enables direct access to the singleton Pod via the Service DNS record, and without going through the Service virtual IP. For example, if we create a headless Service with the name `my-singleton`, we can use it as `my-singleton.default.svc.cluster.local` to access the Pod's IP address directly.

To sum up, for nonstrict singletons with at least one instance requirement, defining a ReplicaSet with one replica would suffice. This configuration favors availability and ensures there is at least one available instance, and possibly more in some corner cases. For a strict singleton with an At-Most-One requirement and better performant service discovery, a StatefulSet and a headless Service would be preferred. Using StatefulSet will favor consistency and ensure there is an At-Most-One instance and occasionally none in some corner cases. You can find a complete example of this in Chapter 12, "Stateful Service", where you have to change the number of replicas to one to make it a singleton.

In-Application Locking

In a distributed environment, one way to control the service instance count is through a distributed lock, as shown in Figure 10-2. Whenever a service instance or a component inside the instance is activated, it can try to acquire a lock, and if it succeeds, the service becomes active. Any subsequent service instance that fails to acquire the lock waits and continuously tries to get the lock in case the currently active service releases it.

Many existing distributed frameworks use this mechanism for achieving high availability and resiliency. For example, the message broker Apache ActiveMQ can run in a highly available *active-passive* topology, where the data source provides the shared lock. The first broker instance that starts up acquires the lock and becomes active, and any other subsequently started instances become passive and wait for the lock to be released. This strategy ensures there is a single active broker instance that is also resilient to failures.

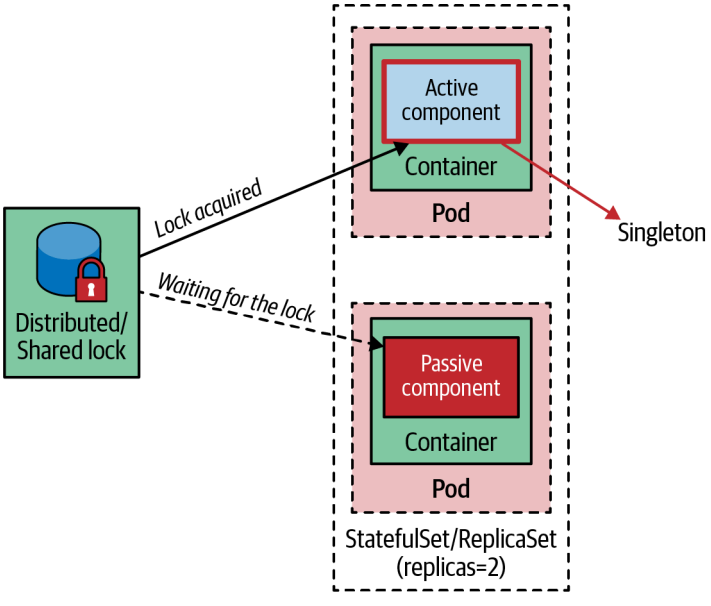

Figure 10-2. In-application locking mechanism

We can compare this strategy to a classic Singleton, as it is known in the object-oriented world: a *Singleton* is an object instance stored in a static class variable. In this instance, the class is aware of being a singleton, and it is written in a way that does not allow instantiation of multiple instances for the same process. In distributed systems, this would mean the containerized application itself has to be written in a way that does not allow more than one active instance at a time, regardless of the number of Pod instances that are started. To achieve this in a distributed environment, first we need a distributed lock implementation such as the one provided by Apache ZooKeeper, HashiCorp's Consul, Redis, or etcd.

The typical implementation with ZooKeeper uses ephemeral nodes, which exist as long as there is a client session and are deleted as soon as the session ends. The first service instance that starts up initiates a session in the ZooKeeper server and creates an ephemeral node to become active. All other service instances from the same cluster become passive and have to wait for the ephemeral node to be released. This is how a ZooKeeper-based implementation makes sure there is only one active service instance in the whole cluster, ensuring an active-passive failover behavior.

In the Kubernetes world, instead of managing a ZooKeeper cluster only for the locking feature, a better option would be to use etcd capabilities exposed through the Kubernetes API and running on the main nodes. etcd is a distributed key-value store that uses the Raft protocol to maintain its replicated state and provides the necessary building blocks for implementing leader election. For example, Kubernetes offers the

Lease object, which is used for node heartbeats and component-level leader election. For every node, there is a Lease object with a matching name, and the Kubelet on every node keeps running a heart beat by updating the Lease object's `renewTime` field. This information is used by the Kubernetes control plane to determine the availability of the nodes. Kubernetes Leases are also used in highly available cluster deployment scenarios for ensuring only single control plane components such as kube-controller-manager and kube-scheduler are active at a time and other instances remain on standby.

Another example is in Apache Camel, which has a Kubernetes connector that also provides leader election and singleton capabilities. This connector goes a step further, and rather than accessing the etcd API directly, it uses Kubernetes APIs to leverage ConfigMaps as a distributed lock. It relies on Kubernetes optimistic locking guarantees for editing resources such as ConfigMaps, where only one Pod can update a ConfigMap at a time. The Camel implementation uses this guarantee to ensure only one Camel route instance is active, and any other instance has to wait and acquire the lock before activating. It is a custom implementation of a lock but achieves the same goal: when there are multiple Pods with the same Camel application, only one of them becomes the active singleton, and the others wait in passive mode.

A more generic implementation of the *Singleton Service* pattern is provided by the Dapr project. Dapr's Distributed Lock building block provides APIs (HTTP and gRPC) with swappable implementations for mutually exclusive access to shared resources. The idea is that each application determines the resources the lock grants access to. Then, multiple instances of the same application use a named lock to exclusively access the shared resource. At any given moment, only one instance of an application can hold a named lock. All other instances of the application are unable to acquire the lock and therefore are not allowed to access the shared resource until the lock is released through unlock or the lock times out. Thanks to its lease-based locking mechanism, if an application acquires a lock, encounters an exception, and cannot free the lock, the lock is automatically released after a period of time using a lease. This prevents resource deadlocks in the event of application failures. Behind this generic distributed lock API, Dapr will be configured to use some kind of storage and lock implementation. This API can be used by applications to implement access to shared resources or in-application singletons.

An implementation with Dapr, ZooKeeper, etcd, or any other distributed lock implementation would be similar to the one described: only one instance of the application becomes the leader and activates itself, and other instances are passive and wait for the lock. This ensures that even if multiple Pod replicas are started and all are healthy, up, and running, only one service is active and performs the business functionality as a singleton, and other instances wait to acquire the lock in case the leader fails or shuts down.

Pod Disruption Budget

While singleton service and leader election try to limit the maximum number of instances a service is running at a time, the PodDisruptionBudget functionality of Kubernetes provides a complementary and somewhat opposite functionality—limiting the number of instances that are simultaneously down for maintenance.

At its core, PodDisruptionBudget ensures a certain number or percentage of Pods will not voluntarily be evicted from a node at any one point in time. *Voluntarily* here means an eviction that can be delayed for a particular time—for example, when it is triggered by draining a node for maintenance or upgrade (`kubectl drain`), or a cluster scaling down, rather than a node becoming unhealthy, which cannot be predicted or controlled.

The PodDisruptionBudget in Example 10-1 applies to Pods that match its selector and ensures two Pods must be available all the time.

Example 10-1. PodDisruptionBudget

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: random-generator-pdb
spec:
  selector:
    matchLabels:                ❶
      app: random-generator
  minAvailable: 2               ❷
```

❶ Selector to count available Pods.

❷ At least two Pods have to be available. You can also specify a percentage, like 80%, to configure that only 20% of the matching Pods might be evicted.

In addition to `.spec.minAvailable`, there is also the option to use `.spec.maxUnavailable`, which specifies the number of Pods from that set that can be unavailable after the eviction. Similar to `.spec.minAvailable`, it can be either an absolute number or a percentage, but it has a few additional limitations. You can specify only either `.spec.minAvailable` or `.spec.maxUnavailable` in a single PodDisruptionBudget, and then it can be used only to control the eviction of Pods that have an associated controller such as ReplicaSet or StatefulSet. For Pods not managed by a controller (also referred to as *bare* or *naked* Pods), other limitations around PodDisruptionBudget should be considered.

PodDisruptionBudget is useful for quorum-based applications that require a minimum number of replicas running at all times to ensure a quorum. Or maybe when an application is serving critical traffic that should never go below a certain percentage of the total number of instances.

PodDisruptionBudget is useful in the context of singletons too. For example, setting `maxUnavailable` to `0` or setting `minAvailable` to `100%` will prevent any voluntary eviction. Setting voluntary eviction to zero for a workload will turn it into an unevictable Pod and will prevent draining the node forever. This can be used as a step in the process where a cluster operator has to contact the singleton workload owner for downtime before accidentally evicting a not highly available Pod. StatefulSet, combined with PodDisruptionBudget, and headless Service are Kubernetes primitives that control and help with the instance count at runtime and are worth mentioning in this chapter.

Discussion

If your use case requires strong singleton guarantees, you cannot rely on the out-of-application locking mechanisms of ReplicaSets. Kubernetes ReplicaSets are designed to preserve the availability of their Pods rather than to ensure At-Most-One semantics for Pods. As a consequence, there are many failure scenarios that have two copies of a Pod running concurrently for a short period (efor example, when a node that runs the singleton Pod is partitioned from the rest of the cluster—such as when replacing a deleted Pod instance with a new one). If that is not acceptable, use StatefulSets or investigate the in-application locking options that provide you more control over the leader election process with stronger guarantees. The latter also mitigates the risk of accidentally scaling Pods by changing the number of replicas. You can combine this with PodDisruptionBudget and prevent voluntary eviction and disruption of your singleton workloads.

In other scenarios, only a part of a containerized application should be a singleton. For example, there might be a containerized application that provides an HTTP endpoint that is safe to scale to multiple instances, but also a polling component that must be a singleton. Using the out-of-application locking approach would prevent scaling the whole service. In such a situation, we either have to split the singleton component in its deployment unit to keep it a singleton (good in theory but not always practical or worth the overhead) or use the in-application locking mechanism and lock only the component that has to be a singleton. This would allow us to scale the whole application transparently, have HTTP endpoints scaled, and have other parts as *active-passive* singletons.

More Information

- Singleton Service Example
- Leases
- Specifying a Disruption Budget for Your Application
- Leader Election in Go Client
- Dapr: Distributed Lock Overview
- Creating Clustered Singleton Services on Kubernetes
- Akka: Kubernetes Lease

CHAPTER 11

Stateless Service

The *Stateless Service* pattern describes how to create and operate applications that are composed of identical ephemeral replicas. These applications are best suited for dynamic cloud environments where they can be rapidly scaled and made highly available.

Problem

The microservices architecture style is the dominant choice for implementing new greenfield cloud native applications. Among the driving principles of this architecture are things such as how it addresses a single concern, how it owns its data, how it has a well-encapsulated deployment boundary, and others. Typically, such applications also follow the twelve-factor app principles, which makes them easy to operate with Kubernetes on dynamic cloud environments.

Applying some of these principles requires understanding the business domain, identifying the service boundary, or applying domain-driven design or a similar methodology during the service implementation. Implementing some of the other principles may involve making the services ephemeral, which means the service can be created, scaled, and destroyed with no side effects. These latter concerns are easier to address when a service is stateless rather than stateful.

A stateless service does not maintain any state internally within the instance across service interactions. In our context, it means a container is stateless if it does not hold any information from requests in its internal storage (memory or temporary filesystem) that is critical for serving future requests. A stateless process has no stored knowledge of or reference to past requests, so each request is made as if from scratch. Instead, if the process needs to store such information, it should store it in an external storage such as a database, message queue, mounted filesystem,

or some other data store that can be accessed by other instances. A good thought experiment is to imagine the instances of your services deployed on different nodes and a load-balancer that randomly distributes the requests to the instances without any sticky session (i.e., without an affinity between a client and a specific service instance). If the service can fulfill its purpose in this setup, it is likely a stateless service (or it has a mechanism for state distribution among the instances, such as a data grid).

Stateless services are made of identical, replaceable instances that often offload state to external permanent storage systems and use load-balancers for distributing incoming requests among themselves. In this chapter, we will see specifically which Kubernetes abstractions can help operate such stateless applications.

Solution

In Chapter 3, "Declarative Deployment", you learned how to use the concept of Deployment to control how an application should be updated to the next version, using the `RollingUpdate` and `Recreate` strategies. But this is only the upgrading aspect of Deployment. At a broader level, a Deployment represents an application deployed in the cluster. Kubernetes doesn't have the notion of an `Application` or a `Container` as top-level entities. Instead, an application is typically composed of a collection of Pods managed by a controller such as ReplicaSet, Deployment, or StatefulSet, combined with ConfigMap, Secret, Service, PersistentVolumeClaim, etc. The controller that is used for managing stateless Pods is ReplicaSet, but that is a lower-level internal control structure used by a Deployment. Deployment is the recommended user-facing abstraction for creating and updating stateless applications, which creates and manages the ReplicaSets behind the scene. A ReplicaSet should be used when the update strategies provided by Deployment are not suitable, or a custom mechanism is required, or no control over the update process is needed at all.

Instances

The primary purpose of a ReplicaSet is to ensure a specified number of identical Pod replicas running at any given time. The main sections of a ReplicaSet definition include the number of replicas indicating how many Pods it should maintain, a selector that specifies how to identify the Pods it manages, and a Pod template for creating new Pod replicas. Then, a ReplicaSet creates and deletes Pods as needed to maintain the desired replica count using the given Pod template, as demonstrated in Example 11-1.

Example 11-1. ReplicaSet definition for a stateless Pod

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rg
  labels:
    app: random-generator
spec:
  replicas: 3                    ❶
  selector:                      ❷
    matchLabels:
      app: random-generator
  template:                      ❸
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - name: random-generator
        image: k8spatterns/random-generator:1.0
```

❶ Desired number of Pod replicas to maintain running.

❷ Label selector used to identify the Pods to manage.

❸ Template specifying the data for creating new Pods.

The template is used when the ReplicaSet needs to create new Pods to meet the desired number of replicas. But a ReplicaSet is not limited to managing the Pods specified by the template. If a bare Pod has no owner reference (meaning it is not managed by a controller), and it matches the label selector, it will be acquired by setting the owner reference and managed by the ReplicaSet. This setup can lead to a ReplicaSet owning a nonidentical set of Pods created by different means, and terminate existing bare Pods that exceed the declared replica count. To avoid such undesired side effects, it is recommended that you ensure bare Pods do not have labels matching ReplicaSet selectors.

Regardless of whether you create a ReplicaSet directly or through a Deployment, the end result will be that the desired number of identical Pod replicas are created and maintained. The added benefit of using Deployment is that we can control how the replicas are upgraded and rolled back, which we described in detail in Chapter 3, "Declarative Deployment". Next, the replicas are scheduled to the available nodes as per the policies we covered in Chapter 6, "Automated Placement". The ReplicaSet's job is to restart the containers if needed and scale out or in when the number of replicas is increased or decreased, respectively. With this behavior, Deployment and ReplicaSet can automate the lifecycle management of stateless applications.

Networking

Pods created by ReplicaSet are ephemeral and may disappear at any time, such as when a Pod is evicted because of resource starvation or because the node the Pod is running on fails. In such a situation, the ReplicaSet will create a new Pod that will have a new name, hostname, and IP address. If the application is stateless, as we've defined earlier in the chapter, new requests should be handled from the newly created Pod the same way as by any other Pod.

Depending on how the application within the container connects to the other systems to accept requests or poll for messages, for example, you may require a Kubernetes Service. If the application is starting an egress connection to a message broker or database, and that is the only way it exchanges data, then there is no need for a Kubernetes Service. But more often, stateless services are contacted by other services over synchronous request/response-driven protocols such as HTTP and gRPC. Since the Pod IP address changes with every Pod restart, it is better to use a permanent IP address based on a Kubernetes Service that service consumers can use. A Kubernetes Service has a fixed IP address that doesn't change during the lifetime of the Service, and it ensures the client requests are always load-balanced across instances and routed to the healthy and ready-to-accept-requests Pods. We cover different types of Kubernetes Services in Chapter 13, "Service Discovery". In Example 11-2, we use a simple Service to expose the Pods internally within the cluster to other Pods.

Example 11-2. Exposing a stateless service

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator      ❶
spec:
  selector:                   ❷
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
```

❶ Name of the service that can be used to reach the matching Pods.

❷ Selector matching the Pod labels from the ReplicaSet.

The definition in this example will create a Service named `random-generator` that accepts TCP connections on port 80 and routes them to port 8080 on all the matching Pods with selector `app: random-generator`. Once a Service is created, it is assigned a `clusterIP` that is accessible only from within the Kubernetes cluster, and that IP remains unchanged as long as the Service definition exists. This acts as a

permanent entrypoint to all matching Pods that are ephemeral and have changing IP addresses.

Notice that Deployment and the resulting ReplicaSet are only responsible for maintaining the desired number of stateless Pods that match the label selector. They are unaware of any Kubernetes Service that might be directing traffic to the same set of Pods or a different combination of Pods.

Storage

Few stateless services don't need any state and can process requests based only on the data provided in every request. Most stateless services require state, but they are stateless because they offload the state to some other stateful system or data store, such as a filesystem. Any Pod, whether it is created by a ReplicaSet or not, can declare and use file storage through volumes. Different types of volumes can be used to store state. Some of these are cloud-provider-specific storage, while others allow mounting network storage or even sharing filesystems from the node where the Pod is placed. In this section, we'll look at the `persistentVolumeClaim` volume type, which allows you to use manually or dynamically provisioned persistent storage.

A PersistentVolume (PV) represents a storage resource abstraction in a Kubernetes cluster that has a lifecycle independent of any Pod lifecycle that is using it. A Pod cannot directly refer to a PV; however, a Pod uses PersistentVolumeClaim (PVC) to request and bind to the PV, which points to the actual durable storage. This indirect connection allows for a separation of concerns and Pod lifecycle decoupling from PV. A cluster administrator can configure storage provisioning and define PVs. The developer creating Pod definitions can use PVC to use the storage. With this indirection, even if the Pod is deleted, the ownership of the PV remains attached to the PVC and continues to exist. Example 11-3 shows a storage claim that can be used in a Pod template.

Example 11-3. A claim for a PersistentVolume

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: random-generator-log  ❶
spec:
  storageClassName: "manual"
  accessModes:
   - ReadWriteOnce            ❷
  resources:
   requests:
    storage: 1Gi              ❸
```

❶ Name of the claim that can be referenced from a Pod template.

❷ Indicates that only a single node can mount the volume for reading and writing.

❸ Requesting 1 GiB of storage.

Once a PVC is defined, it can be referenced from a Pod template through the `persistentVolumeClaim` field. One of the interesting fields of PersistentVolumeClaim is `accessModes`. It controls how the storage is mounted to the nodes and consumed by the Pods. For example, network filesystems can be mounted to multiple nodes and can allow reading and writing to multiple applications at the same time. Other storage implementations can be mounted to only a single node at a time and can be accessed only by the Pods scheduled on that node. Let's look at different `accessModes` offered by Kubernetes:

ReadWriteOnce
: This represents a volume that can be mounted to a single node at a time. In this mode, one or multiple Pods running on the node could carry out read and write operations.

ReadOnlyMany
: The volume can be mounted to multiple nodes, but it allows read-only operations to all Pods.

ReadWriteMany
: In this mode, the volume can be mounted by many nodes and allows both read and write operations.

ReadWriteOncePod
: Notice that all of the access modes described so far offer per-node granularity. Even `ReadWriteOnce` allows multiple Pods on the same node to read from and write to the same volume simultaneously. Only `ReadWriteOncePod` access mode guarantees that only a single Pod has access to a volume. This is invaluable in scenarios where at most one writer application is allowed to access data for data-consistency guarantees. Use this mode with caution as it will turn your services into a singleton and prevent scaling out. If another Pod replica uses the same PVC, the Pod will fail to start because the PVC is already in use by another Pod. As of this writing, `ReadWriteOncePod` doesn't honor preemption either, which means a lower-priority Pod will hold on to the storage and not be preempted from the node in favor of a higher-priority Pod waiting on the same `ReadWriteOncePod` claim.

In a ReplicaSet, all Pods are identical; they share the same PVC and refer to the same PV. This is in contrast to StatefulSets covered in the next chapter, where PVCs are created dynamically for each stateful Pod replica. This is one of the major differences between how stateless and stateful workloads are handled in Kubernetes.

Discussion

A complex distributed system is usually composed of multiple services, some of which will be stateful and perform some form of distributed coordination, some of which might be short-lived jobs, and some of which might be highly scalable stateless services. Stateless services are composed of identical, swappable, ephemeral, and replaceable instances. They are ideal for handling short-lived requests and can scale up and down rapidly without having any dependencies among the instances. As shown in Figure 11-1, Kubernetes offers a number of useful primitives to manage such applications.

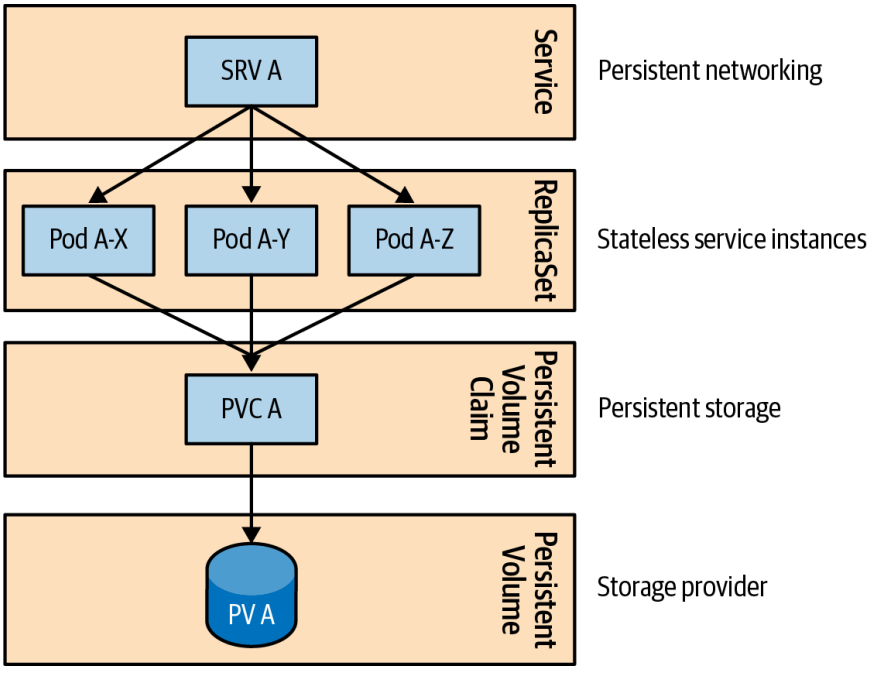

Figure 11-1. A distributed stateless application on Kubernetes

At the lowest level, the Pod abstraction ensures that one or more containers are observed with liveness checks and are always up and running. Building on that, the ReplicaSet also ensures that the desired number of stateless Pods are always running on the healthy nodes. Deployments automate the upgrade and rollback mechanism of Pod replicas. When there is incoming traffic, the Service abstraction discovers and distributes traffic to healthy Pod instances with passing readiness probes. When a persistent file storage is required, PVCs can request and mount storage.

Although Kubernetes offers these building blocks, it will not enforce any direct relationship between them. It is your responsibility to combine them to match the application nature. You have to understand how liveness checks and ReplicaSet control Pods' lifecycles, and how they relate to readiness probes and Service definitions controlling how the traffic is directed to the Pods. You should also understand how PVCs and `accessMode` control where the storage is mounted and how it is accessed. When Kubernetes primitives are not sufficient, you should know how to combine it with other frameworks such as Knative and KEDA and how to autoscale and even turn stateless applications into serverless. The latter frameworks are covered in Chapter 29, "Elastic Scale".

More Information

- Stateless Service Example
- ReplicaSet
- Persistent Volumes
- Storage Classes
- Access Modes

CHAPTER 12

Stateful Service

Distributed stateful applications require features such as persistent identity, networking, storage, and ordinality. The *Stateful Service* pattern describes the StatefulSet primitive that provides these building blocks with strong guarantees ideal for the management of stateful applications.

Problem

We have seen many Kubernetes primitives for creating distributed applications: containers with health checks and resource limits, Pods with multiple containers, dynamic cluster-wide placements, batch jobs, scheduled jobs, singletons, and more. The common characteristic of these primitives is that they treat the managed application as a stateless application composed of identical, swappable, and replaceable containers and comply with the twelve-factor app principles.

It is a significant boost to have a platform taking care of the placement, resiliency, and scaling of stateless applications, but there is still a large part of the workload to consider: stateful applications in which every instance is unique and has long-lived characteristics.

In the real world, behind every highly scalable stateless service is a stateful service, typically in the shape of a data store. In the early days of Kubernetes, when it lacked support for stateful workloads, the solution was placing stateless applications on Kubernetes to get the benefits of the cloud native model and keeping stateful components outside the cluster, either on a public cloud or on-premises hardware, managed with the traditional noncloud native mechanisms. Considering that every enterprise has a multitude of stateful workloads (legacy and modern), the lack of support for stateful workloads was a significant limitation in Kubernetes, which was known as a universal cloud native platform.

But what are the typical requirements of a stateful application? We could deploy a stateful application such as Apache ZooKeeper, MongoDB, Redis, or MySQL by using a Deployment, which could create a ReplicaSet with `replicas=1` to make it reliable, use a Service to discover its endpoint, and use PersistentVolumeClaim (PVC) and PersistentVolume (PV) as permanent storage for its state.

While that is mostly true for a single-instance stateful application, it is not entirely true, as a ReplicaSet does not guarantee At-Most-One semantics, and the number of replicas can vary temporarily. Such a situation can be disastrous and lead to data loss for distributed stateful applications. Also, the main challenges arise when it is a distributed stateful service that is composed of multiple instances. A stateful application composed of multiple clustered services requires multifaceted guarantees from the underlying infrastructure. Let's see some of the most common long-lived persistent prerequisites for distributed stateful applications.

Storage

We could easily increase the number of `replicas` in a ReplicaSet and end up with a distributed stateful application. However, how do we define the storage requirements in such a case? Typically, a distributed stateful application such as those mentioned previously would require dedicated, persistent storage for every instance. A ReplicaSet with `replicas=3` and a PVC definition would result in all three Pods attached to the same PV. While the ReplicaSet and the PVC ensure the instances are up and the storage is attached to whichever node the instances are scheduled on, the storage is not dedicated but shared among all Pod instances.

A workaround is for the application instances to share storage and have an in-app mechanism to split the storage into subfolders and use it without conflicts. While doable, this approach creates a single point of failure with the single storage. Also, it is error-prone as the number of Pods changes during scaling, and it may cause severe challenges around preventing data corruption or loss during scaling.

Another workaround is to have a separate ReplicaSet (with `replicas=1`) for every instance of the distributed stateful application. In this scenario, every ReplicaSet would get its PVC and dedicated storage. The downside of this approach is that it is intensive in manual labor: scaling up requires creating a new set of ReplicaSet, PVC, or Service definitions. This approach lacks a single abstraction for managing all instances of the stateful application as one.

Networking

Similar to the storage requirements, a distributed stateful application requires a stable network identity. In addition to storing application-specific data into the storage space, stateful applications also store configuration details such as hostname and

connection details of their peers. That means every instance should be reachable in a predictable address that should not change dynamically, as is the case with Pod IP addresses in a ReplicaSet. Here we could address this requirement again through a workaround: create a Service per ReplicaSet and have `replicas=1`. However, managing such a setup is manual work, and the application itself cannot rely on a stable hostname because it changes after every restart and is also not aware of the Service name it is accessed from.

Identity

As you can see from the preceding requirements, clustered stateful applications depend heavily on every instance having a hold of its long-lived storage and network identity. That is because in a stateful application, every instance is unique and knows its own identity, and the main ingredients of that identity are the long-lived storage and the networking coordinates. To this list, we could also add the identity/name of the instance (some stateful applications require unique persistent names), which in Kubernetes would be the Pod name. A Pod created with ReplicaSet would have a random name and would not preserve that identity across a restart.

Ordinality

In addition to a unique and long-lived identity, the instances of clustered stateful applications have a fixed position in the collection of instances. This ordering typically impacts the sequence in which the instances are scaled up and down. However, it can also be used for data distribution or access and in-cluster behavior positioning such as locks, singletons, or leaders.

Other Requirements

Stable and long-lived storage, networking, identity, and ordinality are among the collective needs of clustered stateful applications. Managing stateful applications also carries many other specific requirements that vary case by case. For example, some applications have the notion of a quorum and require a minimum number of instances to always be available; some are sensitive to ordinality, and some are fine with parallel Deployments; and some tolerate duplicate instances, and some don't. Planning for all these one-off cases and providing generic mechanisms is an impossible task, and that's why Kubernetes also allows you to create CustomResourceDefinitions (CRDs) and *Operators* for managing applications with bespoke requirements. The *Operator* pattern is explained in Chapter 28.

We have seen some common challenges of managing distributed stateful applications and a few less-than-ideal workarounds. Next, let's check out the Kubernetes native mechanism for addressing these requirements through the StatefulSet primitive.

Solution

To explain what StatefulSet provides for managing stateful applications, we occasionally compare its behavior to the already-familiar ReplicaSet primitive that Kubernetes uses for running stateless workloads. In many ways, StatefulSet is for managing pets, and ReplicaSet is for managing cattle. Pets versus cattle is a famous (but also a controversial) analogy in the DevOps world: identical and replaceable servers are referred to as cattle, and nonfungible unique servers that require individual care are referred to as pets. Similarly, StatefulSet (initially inspired by the analogy and named PetSet) is designed for managing nonfungible Pods, as opposed to ReplicaSet, which is for managing identical replaceable Pods.

Let's explore how StatefulSets work and how they address the needs of stateful applications. Example 12-1 is our random-generator service as a StatefulSet.[1]

Example 12-1. StatefulSet definition for a stateful application

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rg                                  ❶
spec:
  serviceName: random-generator             ❷
  replicas: 2                               ❸
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        ports:
        - containerPort: 8080
          name: http
        volumeMounts:
        - name: logs
          mountPath: /logs
  volumeClaimTemplates:                     ❹
  - metadata:
      name: logs
```

1 Let's assume we have invented a highly sophisticated way of generating random numbers in a distributed Random Number Generator (RNG) cluster with several instances of our service as nodes. Of course, that's not true, but for this example's sake, it's a good enough story.

```
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Mi
```

❶ Name of the StatefulSet is used as prefix for the generated node names.

❷ References the mandatory Service defined in Example 12-2.

❸ Two Pod members in the StatefulSet named *rg-0* and *rg-1*.

❹ Template for creating a PVC for each Pod (similar to the Pod's template).

Rather than going through the definition in Example 12-1 line by line, we explore the overall behavior and the guarantees provided by this StatefulSet definition.

Storage

While it is not always necessary, the majority of stateful applications store state and thus require per-instance-based dedicated persistent storage. The way to request and associate persistent storage with a Pod in Kubernetes is through PVs and PVCs. To create PVCs the same way it creates Pods, StatefulSet uses a `volumeClaimTemplates` element. This extra property is one of the main differences between a StatefulSet and a ReplicaSet, which has a `persistentVolumeClaim` element.

Rather than referring to a predefined PVC, StatefulSets create PVCs by using `volumeClaimTemplates` on the fly during Pod creation. This mechanism allows every Pod to get its own dedicated PVC during initial creation as well as during scaling up by changing the `replicas` count of the StatefulSets.

As you probably realize, we said PVCs are created and associated with the Pods, but we didn't say anything about PVs. That is because StatefulSets do not manage PVs in any way. The storage for the Pods must be provisioned in advance by an admin or provisioned on demand by a PV provisioner based on the requested storage class and ready for consumption by the stateful Pods.

Note the asymmetric behavior here: scaling up a StatefulSet (increasing the `replicas` count) creates new Pods and associated PVCs. Scaling down deletes the Pods, but it does not delete any PVCs (or PVs), which means the PVs cannot be recycled or deleted, and Kubernetes cannot free the storage. This behavior is by design and driven by the presumption that the storage of stateful applications is critical and that an accidental scale-down should not cause data loss. If you are sure the stateful application has been scaled down on purpose and has replicated/drained the data to other instances, you can delete the PVC manually, which allows subsequent PV recycling.

Networking

Each Pod created by a StatefulSet has a stable identity generated by the StatefulSet's name and an ordinal index (starting from 0). Based on the preceding example, the two Pods are named `rg-0` and `rg-1`. The Pod names are generated in a predictable format that differs from the ReplicaSet's Pod-name-generation mechanism, which contains a random suffix.

Dedicated scalable persistent storage is an essential aspect of stateful applications and so is networking.

In Example 12-2, we define a *headless* Service. In a headless Service, `clusterIP` is set to `None`, which means we don't want a kube-proxy to handle the Service, and we don't want a cluster IP allocation or load balancing. Then why do we need a Service?

Example 12-2. Service for accessing StatefulSet

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  clusterIP: None          ❶
  selector:
    app: random-generator
  ports:
  - name: http
    port: 8080
```

❶ Declares this Service as headless.

Stateless Pods created through a ReplicaSet are assumed to be identical, and it doesn't matter on which one a request lands (hence the load balancing with a regular Service). But stateful Pods differ from one another, and we may need to reach a specific Pod by its coordinates.

A headless Service with selectors (notice `.selector.app == random-generator`) enables exactly this. Such a Service creates endpoint records in the API Server and creates DNS entries to return A records (addresses) that point directly to the Pods backing the Service. Long story short, each Pod gets a DNS entry where clients can directly reach out to it in a predictable way. For example, if our `random-generator` Service belongs to the `default` namespace, we can reach our `rg-0` Pod through its fully qualified domain name: `rg-0.random-generator.default.svc.cluster.local`, where the Pod's name is prepended to the Service name. This mapping allows other members of the clustered application or other clients to reach specific Pods if they wish to.

We can also perform DNS lookup for Service (SRV) records (e.g., through `dig SRV random-generator.default.svc.cluster.local`) and discover all running Pods registered with the StatefulSet's governing Service. This mechanism allows dynamic cluster member discovery if any client application needs to do so. The association between the headless Service and the StatefulSet is not only based on the selectors, but the StatefulSet should also link back to the Service by its name as `serviceName: "random-generator"`.

Having dedicated storage defined through `volumeClaimTemplates` is not mandatory, but linking to a Service through `serviceName` field is. The governing Service must exist before the StatefulSet is created and is responsible for the network identity of the set. You can always create other types of Services that also load balance across your stateful Pods if that is what you want.

As Figure 12-1 shows, StatefulSets offer a set of building blocks and guaranteed behavior needed for managing stateful applications in a distributed environment. Your job is to choose and use them in a meaningful way for your stateful use case.

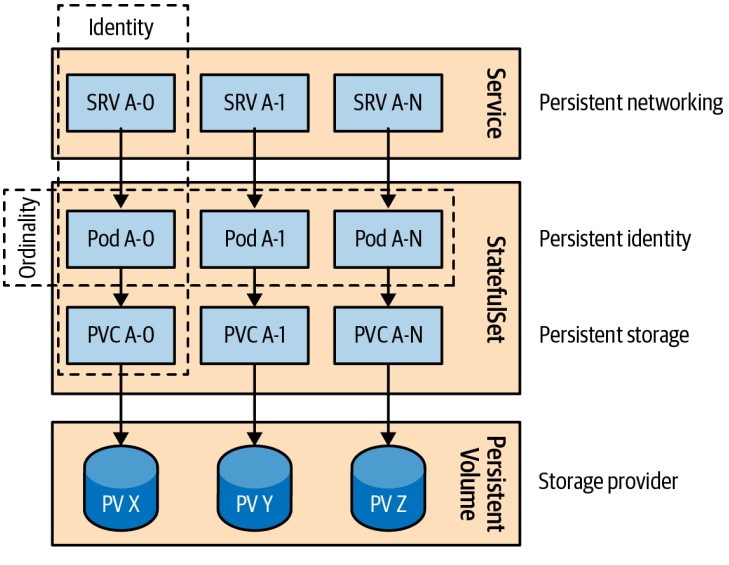

Figure 12-1. A distributed stateful application on Kubernetes

Identity

Identity is the meta building block all other StatefulSet guarantees are built upon. A predictable Pod name and identity is generated based on StatefulSet's name. We then use that identity to name PVCs, reach out to specific Pods through headless Services, and more. You can predict the identity of every Pod before creating it and use that knowledge in the application itself if needed.

Ordinality

By definition, a distributed stateful application consists of multiple instances that are unique and nonswappable. In addition to their uniqueness, instances may also be related to one another based on their instantiation order/position, and this is where the *ordinality* requirement comes in.

From a StatefulSet point of view, the only place where ordinality comes into play is during scaling. Pods have names that have an ordinal suffix (starting from 0), and that Pod creation order also defines the order in which Pods are scaled up and down (in reverse order, from n – 1 to 0).

If we create a ReplicaSet with multiple replicas, Pods are scheduled and started together without waiting for the first one to start successfully (running and ready status, as described in Chapter 4, "Health Probe"). The order in which Pods are starting and are ready is not guaranteed. It is the same when we scale down a ReplicaSet (either by changing the `replicas` count or deleting it). All Pods belonging to a ReplicaSet start shutting down simultaneously without any ordering and dependency among them. This behavior may be faster to complete but is not preferred for stateful applications, especially if data partitioning and distribution are involved among the instances.

To allow proper data synchronization during scale-up and -down, StatefulSet by default performs sequential startup and shutdown. That means Pods start from the first one (with index 0), and only when that Pod has successfully started is the next one scheduled (with index 1), and the sequence continues. During scaling down, the order reverses—first shutting down the Pod with the highest index, and only when it has shut down successfully is the Pod with the next lower index stopped. This sequence continues until the Pod with index 0 is terminated.

Other Features

StatefulSets have other aspects that are customizable to suit the needs of stateful applications. Each stateful application is unique and requires careful consideration while trying to fit it into the StatefulSet model. Let's see a few more Kubernetes features that may turn out to be useful while taming stateful applications:

Partitioned updates
: We described earlier the sequential ordering guarantees when scaling a StatefulSet. As for updating an already-running stateful application (e.g., by altering the `.spec.template` element), StatefulSets allow phased rollout (such as a canary release), which guarantees a certain number of instances to remain intact while applying updates to the rest of the instances.

By using the default rolling update strategy, you can partition instances by specifying a `.spec.updateStrategy.rollingUpdate.partition` number. The parameter (with a default value of 0) indicates the ordinal at which the StatefulSet should be partitioned for updates. If the parameter is specified, all Pods with an ordinal index greater than or equal to the `partition` are updated, while all Pods with an ordinal less than that are not updated. That is true even if the Pods are deleted; Kubernetes recreates them at the previous version. This feature can enable partial updates to clustered stateful applications (ensuring the quorum is preserved, for example) and then roll out the changes to the rest of the cluster by setting the `partition` back to 0.

Parallel deployments

When we set `.spec.podManagementPolicy` to `Parallel`, the StatefulSet launches or terminates all Pods in parallel and does not wait for Pods to run and become ready or completely terminated before moving to the next one. If sequential processing is not a requirement for your stateful application, this option can speed up operational procedures.

At-Most-One Guarantee

Uniqueness is among the fundamental attributes of stateful application instances, and Kubernetes guarantees that uniqueness by making sure no two Pods of a StatefulSet have the same identity or are bound to the same PV. In contrast, ReplicaSet offers the *At-Least-X-Guarantee* for its instances. For example, a ReplicaSet with two replicas tries to keep at least two instances up and running at all times. Even if there is occasionally a chance for that number to go higher, the controller's priority is not to let the number of Pods go below the specified number. It is possible to have more than the specified number of replicas running when a Pod is being replaced by a new one and the old Pod is still not fully terminated. Or, it can go higher if a Kubernetes node is unreachable with `NotReady` state but still has running Pods. In this scenario, the ReplicaSet's controller would start new Pods on healthy nodes, which could lead to more running Pods than desired. That is all acceptable within the semantics of At-Least-X.

A StatefulSet controller, on the other hand, makes every possible check to ensure there are no duplicate Pods—hence the *At-Most-One Guarantee*. It does not start a Pod again unless the old instance is confirmed to be shut down completely. When a node fails, it does not schedule new Pods on a different node unless Kubernetes can confirm that the Pods (and maybe the whole node) are shut down. The At-Most-One semantics of StatefulSets dictates these rules.

It is still possible to break these guarantees and end up with duplicate Pods in a StatefulSet, but this requires active human intervention. For example, deleting an unreachable node resource object from the API Server while the physical node is still running would break this guarantee. Such an action should be performed

only when the node is confirmed to be dead or powered down and no Pod processes are running on it. Or, for example, when you are forcefully deleting a Pod with `kubectl delete pods <pod> --grace-period=0 --force`, which does not wait for a confirmation from the Kubelet that the Pod is terminated. This action immediately clears the Pod from the API Server and causes the StatefulSet controller to start a replacement Pod that could lead to duplicates.

We discuss other approaches to achieving singletons in more depth in Chapter 10, "Singleton Service".

Discussion

In this chapter, we saw some of the standard requirements and challenges in managing distributed stateful applications on a cloud native platform. We discovered that handling a single-instance stateful application is relatively easy, but handling distributed state is a multidimensional challenge. While we typically associate the notion of "state" with "storage," here we have seen multiple facets of state and how it requires different guarantees from different stateful applications. In this space, StatefulSets is an excellent primitive for implementing distributed stateful applications generically. It addresses the need for persistent storage, networking (through Services), identity, ordinality, and a few other aspects. It provides a good set of building blocks for managing stateful applications in an automated fashion, making them first-class citizens in the cloud native world.

StatefulSets are a good start and a step forward, but the world of stateful applications is unique and complex. In addition to the stateful applications designed for a cloud native world that can fit into a StatefulSet, a ton of legacy stateful applications exist that have not been designed for cloud native platforms and have even more needs. Luckily Kubernetes has an answer for that too. The Kubernetes community has realized that rather than modeling different workloads through Kubernetes resources and implementing their behavior through generic controllers, it should allow users to implement their custom controllers and even go one step further and allow modeling application resources through custom resource definitions and behavior through operators.

In Chapters 27 and 28, you will learn about the related *Controller* and *Operator* patterns, which are better suited for managing complex stateful applications in cloud native environments.

More Information

- Stateful Service Example
- StatefulSet Basics
- StatefulSets
- Example: Deploying Cassandra with a Stateful Set
- Running ZooKeeper, a Distributed System Coordinator
- Headless Services
- Force Delete StatefulSet Pods
- Graceful Scaledown of Stateful Apps in Kubernetes