Chapter 8. ReplicaSets

Previously, we covered how to run individual containers as Pods. But these pods are essentially one-off singletons. More often than not, you want multiple replicas of a container running at a particular time. There are a variety of reasons for this type of replication:

Redundancy
: Multiple running instances mean failure can be tolerated.

Scale
: Multiple running instances mean that more requests can be handled.

Sharding
: Different replicas can handle different parts of a computation in parallel.

Of course, you could manually create multiple copies of a Pod using multiple different (though largely similar) Pod manifests, but doing so is both tedious and error-prone. Logically, a user managing a replicated set of Pods considers them as a single entity to be defined and managed. This is precisely what a ReplicaSet is. A ReplicaSet acts as a cluster-wide Pod manager, ensuring that the right types and number of Pods are running at all times.

Because ReplicaSets make it easy to create and manage replicated sets of Pods, they are the building blocks used to describe common application deployment patterns and provide the underpinnings of self-healing for our applications at the infrastructure level. Pods managed by ReplicaSets are automatically rescheduled under certain failure conditions such as node failures and network partitions.

The easiest way to think of a ReplicaSet is that it combines a cookie cutter and a desired of number of cookies into a single API object. When we define a ReplicaSet, we define a specification for the Pods we want to create (the “cookie cutter”), and a desired number of replicas. Additionally, we need to define a way of finding Pods that the ReplicaSet should control. The actual act of managing the replicated Pods is an example of a *reconciliation loop*. Such loops are fundamental to most of the design and implementation of Kubernetes.

Reconciliation Loops

The central concept behind a reconciliation loop is the notion of *desired* state and *observed* or *current* state. Desired state is the state you want. With a ReplicaSet it is the desired number of replicas and the definition of the Pod to replicate. For example, the desired state is that there are three replicas of a Pod running the `kuard` server.

In contrast, current state is the currently observed state of the system. For example, there are only two `kuard` Pods currently running.

The reconciliation loop is constantly running, observing the current state of the world and taking action to try to make the observed state match the desired state. For example, given the previous example, the reconciliation loop creates a new `kuard` Pod in an effort to make the observed state match the desired state of three replicas.

There are many benefits to the reconciliation loop approach to managing state. It is an inherently goal-driven, self-healing system, yet it can often be easily expressed in a few lines of code.

As a concrete example of this, note that the reconciliation loop for ReplicaSets is a single loop, and yet it handles both user actions to scale up or scale down the ReplicaSet, as well as node failures or nodes rejoining the cluster after being absent.

Throughout the rest of the book we'll see numerous examples of reconciliation loops in action.

Relating Pods and ReplicaSets

One of the key themes that runs through Kubernetes is decoupling. In particular, it's important that all of the core concepts of Kubernetes are modular with respect to each other and that they are swappable and replaceable with other components. In this spirit, the relationship between ReplicaSets and Pods is loosely coupled. Though ReplicaSets create and manage Pods, they do not own the Pods they create. ReplicaSets use label queries to identify the set of Pods they should be managing. They then use the exact same Pod API that you used directly in Chapter 5 to create the Pods that they are managing. This notion of "coming in the front door" is another central design concept in Kubernetes. In a similar decoupling, ReplicaSets that create multiple Pods and the services that load-balance to those Pods are also totally separate, decoupled API objects. In addition to supporting modularity, the decoupling of Pods and ReplicaSets enables several important behaviors, discussed in the following sections.

Adopting Existing Containers

Despite the value placed on declarative configuration of software, there are times when it is easier to build something up imperatively. In particular, early on you may be simply deploying a single Pod with a container image without a ReplicaSet managing it. But at some point you may want to expand your singleton container into a replicated service and create and manage an array of similar containers. You may have even defined a load balancer that is serving traffic to that single Pod. If ReplicaSets owned the Pods they created, then the only way to start replicating your Pod would be to delete it and then relaunch it via a ReplicaSet. This might be disruptive, as there would be a moment in time when there would be no copies of your container running. However, because ReplicaSets are decoupled from the Pods they manage, you can simply create a ReplicaSet that will “adopt” the existing Pod, and scale out additional copies of those containers. In this way you can seamlessly move from a single imperative Pod to a replicated set of Pods managed by a ReplicaSet.

Quarantining Containers

Oftentimes, when a server misbehaves, Pod-level health checks will automatically restart that Pod. But if your health checks are incomplete, a Pod can be misbehaving but still be part of the replicated set. In these situations, while it would work to simply kill the Pod, that would leave your developers with only logs to debug the problem. Instead, you can modify the set of labels on the sick Pod. Doing so will disassociate it from the ReplicaSet (and service) so that you can debug the Pod. The ReplicaSet controller will notice that a Pod is missing and create a new copy, but because the Pod is still running, it is available to developers for interactive debugging, which is significantly more valuable than debugging from logs.

Designing with ReplicaSets

ReplicaSets are designed to represent a single, scalable microservice inside your architecture. The key characteristic of ReplicaSets is that every Pod that is created by the ReplicaSet controller is entirely homogeneous. Typically, these Pods are then fronted by a Kubernetes service load balancer, which spreads traffic across the Pods that make up the service. Generally speaking, ReplicaSets are designed for stateles (or nearly stateless) services. The elements created by the ReplicaSet are interchangeable; when a ReplicaSet is scaled down, an arbitrary Pod is selected for deletion. Your application’s behavior shouldn’t change because of such a scale-down operation.

ReplicaSet Spec

Like all concepts in Kubernetes, ReplicaSets are defined using a specification. All ReplicaSets must have a unique name (defined using the `metadata.name` field), a `spec` section that describes the number of Pods (replicas) that should be running cluster-wide at a given time, and a Pod template that describes the Pod to be created when the defined number of replicas is not met. Example 8-1 shows a minimal ReplicaSet definition.

Example 8-1. kuard-rs.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: kuard
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: kuard
        version: "2"
    spec:
      containers:
        - name: kuard
          image: "gcr.io/kuar-demo/kuard-amd64:2"
```

Pod Templates

As mentioned previously, when the number of Pods in the current state is less than the number of Pods in the desired state, the ReplicaSet controller will create new Pods. The Pods are created using a Pod template that is contained in the ReplicaSet specification. The Pods are created in exactly the same manner as when you created a Pod from a YAML file in previous chapters. But instead of using a file, the Kubernetes ReplicaSet controller creates and submits a Pod manifest based on the Pod template directly to the API server. The following shows an example of a Pod template in a ReplicaSet:

```
template:
  metadata:
    labels:
      app: helloworld
      version: v1
  spec:
    containers:
      - name: helloworld
        image: kelseyhightower/helloworld:v1
        ports:
          - containerPort: 80
```

Labels

In any cluster of reasonable size, there are many different Pods running at any given time — so how does the ReplicaSet reconciliation loop discover the set of Pods for a particular ReplicaSet? ReplicaSets monitor cluster state using a set of Pod labels. Labels are used to filter Pod listings and track Pods running within a cluster. When ReplicaSets are initially created, the ReplicaSet fetches a Pod listing from the Kubernetes API and filters the results by labels. Based on the number of Pods returned by the query, the ReplicaSet deletes or creates Pods to meet the desired number of replicas. The labels used for filtering are defined in the ReplicaSet `spec` section and are the key to understanding how ReplicaSets work.

> **NOTE**
>
> The selector in the ReplicaSet `spec` should be a proper subset of the labels in the Pod template.

Creating a ReplicaSet

ReplicaSets are created by submitting a ReplicaSet object to the Kubernetes API. In this section we will create a ReplicaSet using a configuration file and the `kubectl apply` command.

The ReplicaSet configuration file in Example 8-1 will ensure one copy of the `gcr.io/kuar-demo/kuard-amd64:1` container is running at a given time.

Use the `kubectl apply` command to submit the `kuard` ReplicaSet to the Kubernetes API:

```
$ kubectl apply -f kuard-rs.yaml
replicaset "kuard" created
```

Once the `kuard` ReplicaSet has been accepted, the ReplicaSet controller will detect there are no `kuard` Pods running that match the desired state, and a new `kuard` Pod will be created based on the contents of the Pod template:

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
kuard-yvzgd   1/1       Running   0          11s
```

Inspecting a ReplicaSet

As with Pods and other Kubernetes API objects, if you are interested in further details about a ReplicaSet, the `describe` command will provide much more information about its state. Here is an example of using `describe` to obtain the details of the ReplicaSet we previously created:

```
$ kubectl describe rs kuard
Name:         kuard
Namespace:    default
Image(s):     kuard:1.9.15
Selector:     app=kuard,version=2
Labels:       app=kuard,version=2
Replicas:     1 current / 1 desired
Pods Status:  1 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
```

You can see the label selector for the ReplicaSet, as well as the state of all of the replicas managed by the ReplicaSet.

Finding a ReplicaSet from a Pod

Sometimes you may wonder if a Pod is being managed by a ReplicaSet, and, if it is, which ReplicaSet.

To enable this kind of discovery, the ReplicaSet controller adds an annotation to every Pod that it creates. The key for the annotation is `kubernetes.io/created-by`. If you run the following, look for the `kubernetes.io/created-by` entry in the annotations section:

```
$ kubectl get pods <pod-name> -o yaml
```

If applicable, this will list the name of the ReplicaSet that is managing this Pod. Note that such annotations are best-effort; they are only created when the Pod is created by the ReplicaSet, and can be removed by a Kubernetes user at any time.

Finding a Set of Pods for a ReplicaSet

You can also determine the set of Pods managed by a ReplicaSet. First, you can get the set of labels using the `kubectl describe` command. In the previous example, the label selector was `app=kuard,version=2`. To find the Pods that match this selector, use the `--selector` flag or the shorthand `-l`:

```
$ kubectl get pods -l app=kuard,version=2
```

This is exactly the same query that the ReplicaSet executes to determine the current number of Pods.

Scaling ReplicaSets

ReplicaSets are scaled up or down by updating the `spec.replicas` key on the ReplicaSet object stored in Kubernetes. When a ReplicaSet is scaled up, new Pods are submitted to the Kubernetes API using the Pod template defined on the ReplicaSet.

Imperative Scaling with kubectl Scale

The easiest way to achieve this is using the `scale` command in `kubectl`. For example, to scale up to four replicas you could run:

```
$ kubectl scale kuard --replicas=4
```

While such imperative commands are useful for demonstrations and quick reactions to emergency situations (e.g., in response to a sudden increase in load), it is important to also update any text-file configurations to match the number of replicas that you set via the imperative `scale` command. The reason for this becomes obvious when you consider the following scenario:

> Alice is on call, when suddenly there is a large increase in load on the service she is managing. Alice uses the +scale+ command to increase the number of servers responding to requests to 10, and the situation is resolved. However, Alice forgets to update the ReplicaSet configurations checked into source control. Several days later, Bob is preparing the weekly rollouts. Bob edits the ReplicaSet configurations stored in version control to use the new container image, but he doesn't notice that the number of replicas in the file is currently 5, not the 10 that Alice set in response to the increased load. Bob proceeds with the rollout, which both updates the container image and reduces the number of replicas by half, causing an immediate overload or outage.

Hopefully, this illustrates the need to ensure that any imperative changes are immediately followed by a declarative change in source control. Indeed, if the need is not acute, we generally recommend only making declarative changes as described in the following section.

Declaratively Scaling with kubectl apply

In a declarative world, we make changes by editing the configuration file in version control and then applying those changes to our cluster. To scale the `kuard` ReplicaSet, edit the *kuard-rs.yaml* configuration file and set the `replicas` count to `3`:

```
...
spec:
  replicas: 3
...
```

In a multiuser setting, you would like to have a documented code review of this change and eventually check the changes into version control. Either way, you can then use the `kubectl apply` command to submit the updated `kuard` ReplicaSet to the API server:

```
$ kubectl apply -f kuard-rs.yaml
replicaset "kuard" configured
```

Now that the updated `kuard` ReplicaSet is in place, the ReplicaSet controller will detect that the number of desired Pods has changed and that it needs to take action to realize that desired state. If you used the imperative `scale` command in the previous section, the ReplicaSet controller will destroy one Pod to get the number to three. Otherwise, it will submit two new Pods to the Kubernetes API using the Pod template defined on the `kuard` ReplicaSet. Regardless, use the `kubectl get pods` command to list the running `kuard` Pods. You should see output like the following:

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
kuard-3a2sb   1/1       Running   0          26s
kuard-wuq9v   1/1       Running   0          26s
kuard-yvzgd   1/1       Running   0          2m
```

Autoscaling a ReplicaSet

While there will be times when you want to have explicit control over the number of replicas in a ReplicaSet, often you simply want to have "enough" replicas. The definition varies depending on the needs of the containers in the ReplicaSet. For example, with a web server like nginx, you may want to scale due to CPU usage. For an in-memory cache, you may want to scale with memory consumption. In some cases you may want to scale in response to custom application metrics. Kubernetes can handle all of these scenarios via *horizontal pod autoscaling* (HPA).

> **NOTE**
>
> HPA requires the presence of the `heapster` Pod on your cluster. `heapster` keeps track of metrics and provides an API for consuming metrics HPA uses when making scaling decisions. Most installations of Kubernetes include `heapster` by default. You can validate its presence by listing the Pods in the `kube-system` namespace:
>
> ```
> $ kubectl get pods --namespace=kube-system
> ```
>
> You should see a Pod named `heapster` somewhere in that list. If you do not see it, autoscaling will not work correctly.

"Horizontal pod autoscaling" is kind of a mouthful, and you might wonder why it is not simply called "autoscaling." Kubernetes makes a distinction between *horizontal* scaling, which involves creating additional replicas of a Pod, and *vertical* scaling, which involves increasing the resources required for a particular Pod (e.g., increasing the CPU required for the Pod). Vertical scaling is not currently implemented in Kubernetes, but it is planned. Additionally, many solutions also enable *cluster* autoscaling, where the number of machines in the cluster is scaled in response to resource needs, but this solution is not covered here.

Autoscaling based on CPU

Scaling based on CPU usage is the most common use case for Pod autoscaling. Generally it is most useful for request-based systems that consume CPU

proportionally to the number of requests they are receiving, while using a relatively static amount of memory.

To scale a ReplicaSet, you can run a command like the following:

```
$ kubectl autoscale rs kuard --min=2 --max=5 --cpu-percent=80
```

This command creates an autoscaler that scales between two and five replicas with a CPU threshold of 80%. To view, modify, or delete this resource you can use the standard `kubectl` commands and the `horizontalpodautoscalers` resource. `horizontalpodautoscalers` is quite a bit to type, but it can be shortened to `hpa`:

```
$ kubectl get hpa
```

> **WARNING**
>
> Because of the decoupled nature of Kubernetes, there is no direct link between the horizontal pod autoscaler and the ReplicaSet. While this is great for modularity and composition, it also enables some antipatterns. In particular, it's a bad idea to combine both autoscaling and imperative or declarative management of the number of replicas. If both you and an autoscaler are attempting to modify the number of replicas, it's highly likely that you will clash, resulting in unexpected behavior.

Deleting ReplicaSets

When a ReplicaSet set is no longer required it can be deleted using the `kubectl delete` command. By default, this also deletes the Pods that are managed by the ReplicaSet:

```
$ kubectl delete rs kuard
replicaset "kuard" deleted
```

Running the `kubectl get pods` command shows that all the `kuard` Pods created by the `kuard` ReplicaSet have also been deleted:

```
$ kubectl get pods
```

If you don’t want to delete the Pods that are being managed by the ReplicaSet you can set the `--cascade` flag to `false` to ensure only the ReplicaSet object is deleted and not the Pods:

```
$ kubectl delete rs kuard --cascade=false
```

Summary

Composing Pods with ReplicaSets provides the foundation for building robust applications with automatic failover, and makes deploying those applications a breeze by enabling scalable and sane deployment patterns. ReplicaSets should be used for any Pod you care about, even if it is a single Pod! Some people even default to using ReplicaSets instead of Pods. A typical cluster will have many ReplicaSets, so apply liberally to the affected area.