ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. Usually, you define a Deployment and let that Deployment manage ReplicaSets automatically.

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

How a ReplicaSet works

A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.

A ReplicaSet is linked to its Pods via the Pods' metadata.ownerReferences field, which specifies what resource the current object is owned by. All Pods acquired by a ReplicaSet have their owning ReplicaSet's identifying information within their ownerReferences field. It's through this link that the ReplicaSet knows of the state of the Pods it is maintaining and plans accordingly.

A ReplicaSet identifies new Pods to acquire by using its selector. If there is a Pod that has no OwnerReference or the OwnerReference is not a Controller and it matches a ReplicaSet's selector, it will be immediately acquired by said ReplicaSet.

When to use a ReplicaSet

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section.

Example

controllers/frontend.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Saving this manifest into `frontend.yaml` and submitting it to a Kubernetes cluster will create the defined ReplicaSet and the Pods that it manages.

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

You can then get the current ReplicaSets deployed:

```
kubectl get rs
```

And see the frontend one you created:

```
NAME       DESIRED   CURRENT   READY   AGE
frontend   3         3         3       6s
```

You can also check on the state of the ReplicaSet:

```
kubectl describe rs/frontend
```

And you will see output similar to:

```
Name:         frontend
Namespace:    default
Selector:     tier=frontend
Labels:       app=guestbook
              tier=frontend
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"apps/v1","kind":"ReplicaSet","metadata":{"annotati
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  tier=frontend
  Containers:
   php-redis:
    Image:        gcr.io/google_samples/gb-frontend:v3
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  117s  replicaset-controller  Created pod: frontend-wt
  Normal  SuccessfulCreate  116s  replicaset-controller  Created pod: frontend-b2
  Normal  SuccessfulCreate  116s  replicaset-controller  Created pod: frontend-vc
```

And lastly you can check for the Pods brought up:

```
kubectl get pods
```

You should see Pod information similar to:

```
NAME             READY   STATUS    RESTARTS   AGE
frontend-b2zdv   1/1     Running   0          6m36s
frontend-vcmts   1/1     Running   0          6m36s
frontend-wtsmm   1/1     Running   0          6m36s
```

You can also verify that the owner reference of these pods is set to the frontend ReplicaSet. To do this, get the yaml of one of the Pods running:

```
kubectl get pods frontend-b2zdv -o yaml
```

The output will look similar to this, with the frontend ReplicaSet's info set in the metadata's ownerReferences field:

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-02-12T07:06:16Z"
  generateName: frontend-
  labels:
    tier: frontend
  name: frontend-b2zdv
  namespace: default
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: frontend
    uid: f391f6db-bb9b-4c09-ae74-6a1f77f3d5cf
...
```

Non-Template Pod acquisitions

While you can create bare Pods with no problems, it is strongly recommended to make sure that the bare Pods do not have labels which match the selector of one of your ReplicaSets. The reason for this is because a ReplicaSet is not limited to owning Pods specified by its template-- it can acquire other Pods in the manner specified in the previous sections.

Take the previous frontend ReplicaSet example, and the Pods specified in the following manifest:

pods/pod-rs.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    tier: frontend
spec:
  containers:
  - name: hello1
    image: gcr.io/google-samples/hello-app:2.0

---

apiVersion: v1
kind: Pod
metadata:
  name: pod2
  labels:
    tier: frontend
spec:
  containers:
  - name: hello2
    image: gcr.io/google-samples/hello-app:1.0
```

As those Pods do not have a Controller (or any object) as their owner reference and match the selector of the frontend ReplicaSet, they will immediately be acquired by it.

Suppose you create the Pods after the frontend ReplicaSet has been deployed and has set up its initial Pod replicas to fulfill its replica count requirement:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

The new Pods will be acquired by the ReplicaSet, and then immediately terminated as the ReplicaSet would be over its desired count.

Fetching the Pods:

```
kubectl get pods
```

The output shows that the new Pods are either already terminated, or in the process of being terminated:

```
NAME             READY   STATUS        RESTARTS   AGE
frontend-b2zdv   1/1     Running       0          10m
frontend-vcmts   1/1     Running       0          10m
frontend-wtsmm   1/1     Running       0          10m
pod1             0/1     Terminating   0          1s
pod2             0/1     Terminating   0          1s
```

If you create the Pods first:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

And then create the ReplicaSet however:

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

You shall see that the ReplicaSet has acquired the Pods and has only created new ones according to its spec until the number of its new Pods and the original matches its desired count. As fetching the Pods:

```
kubectl get pods
```

Will reveal in its output:

```
NAME             READY   STATUS    RESTARTS   AGE
frontend-hmmj2   1/1     Running   0          9s
pod1             1/1     Running   0          36s
pod2             1/1     Running   0          36s
```

In this manner, a ReplicaSet can own a non-homogenous set of Pods

Writing a ReplicaSet manifest

As with all other Kubernetes API objects, a ReplicaSet needs the `apiVersion`, `kind`, and `metadata` fields. For ReplicaSets, the `kind` is always a ReplicaSet.

When the control plane creates new Pods for a ReplicaSet, the `.metadata.name` of the ReplicaSet is part of the basis for naming those Pods. The name of a ReplicaSet must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostnames. For best compatibility, the name should follow the more restrictive rules for a DNS label.

A ReplicaSet also needs a `.spec` section.

Pod Template

The `.spec.template` is a pod template which is also required to have labels in place. In our `frontend.yaml` example we had one label: `tier: frontend`. Be careful not to overlap with the selectors of other controllers, lest they try to adopt this Pod.

For the template's restart policy field, `.spec.template.spec.restartPolicy`, the only allowed value is `Always`, which is the default.

Pod Selector

The `.spec.selector` field is a label selector. As discussed earlier these are the labels used to identify potential Pods to acquire. In our `frontend.yaml` example, the selector was:

```
matchLabels:
  tier: frontend
```

In the ReplicaSet, `.spec.template.metadata.labels` must match `spec.selector`, or it will be rejected by the API.

> **Note:** For 2 ReplicaSets specifying the same `.spec.selector` but different `.spec.template.metadata.labels` and `.spec.template.spec` fields, each ReplicaSet ignores the Pods created by the other ReplicaSet.

Replicas

You can specify how many Pods should run concurrently by setting `.spec.replicas`. The ReplicaSet will create/delete its Pods to match this number.

If you do not specify `.spec.replicas`, then it defaults to 1.

Working with ReplicaSets

Deleting a ReplicaSet and its Pods

To delete a ReplicaSet and all of its Pods, use `kubectl delete`. The Garbage collector automatically deletes all of the dependent Pods by default.

When using the REST API or the `client-go` library, you must set `propagationPolicy` to `Background` or `Foreground` in the `-d` option. For example:

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/apps/v1/namespaces/default/replicasets/front
  -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}
  -H "Content-Type: application/json"
```

Deleting just a ReplicaSet

You can delete a ReplicaSet without affecting any of its Pods using `kubectl delete` with the `--cascade=orphan` option. When using the REST API or the `client-go` library, you must set `propagationPolicy` to `Orphan`. For example:

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/apps/v1/namespaces/default/replicasets/front
  -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' \
  -H "Content-Type: application/json"
```

Once the original is deleted, you can create a new ReplicaSet to replace it. As long as the old and new `.spec.selector` are the same, then the new one will adopt the old Pods. However, it will not make any effort to make existing Pods match a new, different pod template. To update Pods to a new spec in a controlled way, use a Deployment, as ReplicaSets do not support a rolling update directly.

Isolating Pods from a ReplicaSet

You can remove Pods from a ReplicaSet by changing their labels. This technique may be used to remove Pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

Scaling a ReplicaSet

A ReplicaSet can be easily scaled up or down by simply updating the `.spec.replicas` field. The ReplicaSet controller ensures that a desired number of Pods with a matching label selector are available and operational.

When scaling down, the ReplicaSet controller chooses which pods to delete by sorting the available pods to prioritize scaling down pods based on the following general algorithm:

1. Pending (and unschedulable) pods are scaled down first
2. If `controller.kubernetes.io/pod-deletion-cost` annotation is set, then the pod with the lower value will come first.
3. Pods on nodes with more replicas come before pods on nodes with fewer replicas.
4. If the pods' creation times differ, the pod that was created more recently comes before the older pod (the creation times are bucketed on an integer log scale when the `LogarithmicScaleDown` feature gate is enabled)

If all of the above match, then selection is random.

Pod deletion cost

FEATURE STATE: `Kubernetes v1.22 [beta]`

Using the `controller.kubernetes.io/pod-deletion-cost` annotation, users can set a preference regarding which pods to remove first when downscaling a ReplicaSet.

The annotation should be set on the pod, the range is [-2147483647, 2147483647]. It represents the cost of deleting a pod compared to other pods belonging to the same ReplicaSet. Pods with lower deletion cost are preferred to be deleted before pods with higher deletion cost.

The implicit value for this annotation for pods that don't set it is 0; negative values are permitted. Invalid values will be rejected by the API server.

This feature is beta and enabled by default. You can disable it using the feature gate `PodDeletionCost` in both kube-apiserver and kube-controller-manager.

Note:

- This is honored on a best-effort basis, so it does not offer any guarantees on pod deletion order.
- Users should avoid updating the annotation frequently, such as updating it based on a metric value, because doing so will generate a significant number of pod updates on the apiserver.

Example Use Case

The different pods of an application could have different utilization levels. On scale down, the application may prefer to remove the pods with lower utilization. To avoid frequently updating the pods, the application should update `controller.kubernetes.io/pod-deletion-cost` once before issuing a scale down (setting the annotation to a value proportional to pod utilization level). This works if the application itself controls the down scaling; for example, the driver pod of a Spark deployment.

ReplicaSet as a Horizontal Pod Autoscaler Target

A ReplicaSet can also be a target for Horizontal Pod Autoscalers (HPA). That is, a ReplicaSet can be auto-scaled by an HPA. Here is an example HPA targeting the ReplicaSet we created in the previous example.

controllers/hpa-rs.yaml

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: frontend
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Saving this manifest into `hpa-rs.yaml` and submitting it to a Kubernetes cluster should create the defined HPA that autoscales the target ReplicaSet depending on the CPU usage of the replicated Pods.

```
kubectl apply -f https://k8s.io/examples/controllers/hpa-rs.yaml
```

Alternatively, you can use the `kubectl autoscale` command to accomplish the same (and it's easier!)

```
kubectl autoscale rs frontend --max=10 --min=3 --cpu-percent=50
```

Alternatives to ReplicaSet

Deployment (recommended)

`Deployment` is an object which can own ReplicaSets and update them and their Pods via declarative, server-side rolling updates. While ReplicaSets can be used independently, today they're mainly used by Deployments as a mechanism to orchestrate Pod creation, deletion and updates. When you use Deployments you don't have to worry about managing the ReplicaSets that they create. Deployments own and manage their ReplicaSets. As such, it is recommended to use Deployments when you want ReplicaSets.

Bare Pods

Unlike the case where a user directly created Pods, a ReplicaSet replaces Pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, we recommend that you use a ReplicaSet even if your application requires only a single Pod. Think of it similarly to a process supervisor, only it supervises multiple Pods across multiple nodes instead of individual processes on a single node. A ReplicaSet delegates local container restarts to some agent on the node such as Kubelet.

Job

Use a `Job` instead of a ReplicaSet for Pods that are expected to terminate on their own (that is, batch jobs).

DaemonSet

Use a `DaemonSet` instead of a ReplicaSet for Pods that provide a machine-level function, such as machine monitoring or machine logging. These Pods have a lifetime that is tied to a machine lifetime: the Pod needs to be running on the machine before other Pods start, and are safe to terminate when the machine is otherwise ready to be rebooted/shutdown.

ReplicationController

ReplicaSets are the successors to ReplicationControllers. The two serve the same purpose, and behave similarly, except that a ReplicationController does not support set-based selector requirements as described in the labels user guide. As such, ReplicaSets are preferred over ReplicationControllers

What's next

- Learn about Pods.
- Learn about Deployments.
- Run a Stateless Application Using a Deployment, which relies on ReplicaSets to work.
- `ReplicaSet` is a top-level resource in the Kubernetes REST API. Read the ReplicaSet object definition to understand the API for replica 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 29, 2022 at 6:33 PM PST: Improve overview for workload APIs (50635afc37)