# ReplicationController

**Note:** A [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) that configures a [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) is now the recommended way to set up replication.

A ReplicationController ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

## How a ReplicationController Works

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

ReplicationController is often abbreviated to "rc" in discussion, and as a shortcut in kubectl commands.

A simple case is to create one ReplicationController object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers.

## Running an example ReplicationController

This example ReplicationController config runs three copies of the nginx web server.

[controllers/replication.yaml](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/controllers/replication.yaml)

**apiVersion**: v1

**kind**: ReplicationController

**metadata**:

**name**: nginx

**spec**:

**replicas**: 3

**selector**:

**app**: nginx

**template**:

**metadata**:

**name**: nginx

**labels**:

**app**: nginx

**spec**:

**containers**:

- **name**: nginx

**image**: nginx

**ports**:

- **containerPort**: 80

Run the example job by downloading the example file and then running this command:

kubectl apply -f https://k8s.io/examples/controllers/replication.yaml

The output is similar to this:

replicationcontroller/nginx created

Check on the status of the ReplicationController using this command:

kubectl describe replicationcontrollers/nginx

The output is similar to this:

Name: nginx

Namespace: default

Selector: app=nginx

Labels: app=nginx

Annotations: <none>

Replicas: 3 current / 3 desired

Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed

Pod Template:

Labels: app=nginx

Containers:

nginx:

Image: nginx

Port: 80/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- ---- ------ -------

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-qrm3m

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-3ntk0

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-4ok8v

Here, three pods are created, but none is running yet, perhaps because the image is being pulled. A little later, the same command may show:

Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

To list all the pods that belong to the ReplicationController in a machine readable form, you can use a command like this:

pods=**$(**kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata.name}**)**

echo $pods

The output is similar to this:

nginx-3ntk0 nginx-4ok8v nginx-qrm3m

Here, the selector is the same as the selector for the ReplicationController (seen in the kubectl describe output), and in a different form in replication.yaml. The --output=jsonpath option specifies an expression with the name from each pod in the returned list.

## Writing a ReplicationController Spec

As with all other Kubernetes config, a ReplicationController needs apiVersion, kind, and metadata fields. The name of a ReplicationController object must be a valid [DNS subdomain name](https://kubernetes.io/docs/concepts/overview/working-with-objects/names#dns-subdomain-names). For general information about working with configuration files, see [object management](https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/).

A ReplicationController also needs a [.spec section](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status).

### Pod Template

The .spec.template is the only required field of the .spec.

The .spec.template is a [pod template](https://kubernetes.io/docs/concepts/workloads/pods/#pod-templates). It has exactly the same schema as a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/), except it is nested and does not have an apiVersion or kind.

In addition to required fields for a Pod, a pod template in a ReplicationController must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See [pod selector](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#pod-selector).

Only a [.spec.template.spec.restartPolicy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy) equal to Always is allowed, which is the default if not specified.

For local container restarts, ReplicationControllers delegate to an agent on the node, for example the [Kubelet](https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/).

### Labels on the ReplicationController

The ReplicationController can itself have labels (.metadata.labels). Typically, you would set these the same as the .spec.template.metadata.labels; if .metadata.labels is not specified then it defaults to .spec.template.metadata.labels. However, they are allowed to be different, and the .metadata.labels do not affect the behavior of the ReplicationController.

### Pod Selector

The .spec.selector field is a [label selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors). A ReplicationController manages all the pods with labels that match the selector. It does not distinguish between pods that it created or deleted and pods that another person or process created or deleted. This allows the ReplicationController to be replaced without affecting the running pods.

If specified, the .spec.template.metadata.labels must be equal to the .spec.selector, or it will be rejected by the API. If .spec.selector is unspecified, it will be defaulted to .spec.template.metadata.labels.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicationController, or with another controller such as Job. If you do so, the ReplicationController thinks that it created the other pods. Kubernetes does not stop you from doing this.

If you do end up with multiple controllers that have overlapping selectors, you will have to manage the deletion yourself (see [below](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#working-with-replicationcontrollers)).

### Multiple Replicas

You can specify how many pods should run concurrently by setting .spec.replicas to the number of pods you would like to have running concurrently. The number running at any time may be higher or lower, such as if the replicas were just increased or decreased, or if a pod is gracefully shutdown, and a replacement starts early.

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

## Working with ReplicationControllers

### Deleting a ReplicationController and its Pods

To delete a ReplicationController and all its pods, use [kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete). Kubectl will scale the ReplicationController to zero and wait for it to delete each pod before deleting the ReplicationController itself. If this kubectl command is interrupted, it can be restarted.

When using the REST API or [client library](https://kubernetes.io/docs/reference/using-api/client-libraries), you need to do the steps explicitly (scale replicas to 0, wait for pod deletions, then delete the ReplicationController).

### Deleting only a ReplicationController

You can delete a ReplicationController without affecting any of its pods.

Using kubectl, specify the --cascade=orphan option to [kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete).

When using the REST API or [client library](https://kubernetes.io/docs/reference/using-api/client-libraries), you can delete the ReplicationController object.

Once the original is deleted, you can create a new ReplicationController 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 [rolling update](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#rolling-updates).

### Isolating pods from a ReplicationController

Pods may be removed from a ReplicationController's target set by changing their labels. This technique may be used to remove pods from service for debugging and data recovery. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

## Common usage patterns

### Rescheduling

As mentioned above, whether you have 1 pod you want to keep running, or 1000, a ReplicationController will ensure that the specified number of pods exists, even in the event of node failure or pod termination (for example, due to an action by another control agent).

### Scaling

The ReplicationController enables scaling the number of replicas up or down, either manually or by an auto-scaling control agent, by updating the replicas field.

### Rolling updates

The ReplicationController is designed to facilitate rolling updates to a service by replacing pods one-by-one.

As explained in [#1353](https://issue.k8s.io/1353), the recommended approach is to create a new ReplicationController with 1 replica, scale the new (+1) and old (-1) controllers one by one, and then delete the old controller after it reaches 0 replicas. This predictably updates the set of pods regardless of unexpected failures.

Ideally, the rolling update controller would take application readiness into account, and would ensure that a sufficient number of pods were productively serving at any given time.

The two ReplicationControllers would need to create pods with at least one differentiating label, such as the image tag of the primary container of the pod, since it is typically image updates that motivate rolling updates.

### Multiple release tracks

In addition to running multiple releases of an application while a rolling update is in progress, it's common to run multiple releases for an extended period of time, or even continuously, using multiple release tracks. The tracks would be differentiated by labels.

For instance, a service might target all pods with tier in (frontend), environment in (prod). Now say you have 10 replicated pods that make up this tier. But you want to be able to 'canary' a new version of this component. You could set up a ReplicationController with replicas set to 9 for the bulk of the replicas, with labels tier=frontend, environment=prod, track=stable, and another ReplicationController with replicas set to 1 for the canary, with labels tier=frontend, environment=prod, track=canary. Now the service is covering both the canary and non-canary pods. But you can mess with the ReplicationControllers separately to test things out, monitor the results, etc.

### Using ReplicationControllers with Services

Multiple ReplicationControllers can sit behind a single service, so that, for example, some traffic goes to the old version, and some goes to the new version.

A ReplicationController will never terminate on its own, but it isn't expected to be as long-lived as services. Services may be composed of pods controlled by multiple ReplicationControllers, and it is expected that many ReplicationControllers may be created and destroyed over the lifetime of a service (for instance, to perform an update of pods that run the service). Both services themselves and their clients should remain oblivious to the ReplicationControllers that maintain the pods of the services.

## Writing programs for Replication

Pods created by a ReplicationController are intended to be fungible and semantically identical, though their configurations may become heterogeneous over time. This is an obvious fit for replicated stateless servers, but ReplicationControllers can also be used to maintain availability of master-elected, sharded, and worker-pool applications. Such applications should use dynamic work assignment mechanisms, such as the [RabbitMQ work queues](https://www.rabbitmq.com/tutorials/tutorial-two-python.html), as opposed to static/one-time customization of the configuration of each pod, which is considered an anti-pattern. Any pod customization performed, such as vertical auto-sizing of resources (for example, cpu or memory), should be performed by another online controller process, not unlike the ReplicationController itself.

## Responsibilities of the ReplicationController

The ReplicationController ensures that the desired number of pods matches its label selector and are operational. Currently, only terminated pods are excluded from its count. In the future, [readiness](https://issue.k8s.io/620) and other information available from the system may be taken into account, we may add more controls over the replacement policy, and we plan to emit events that could be used by external clients to implement arbitrarily sophisticated replacement and/or scale-down policies.

The ReplicationController is forever constrained to this narrow responsibility. It itself will not perform readiness nor liveness probes. Rather than performing auto-scaling, it is intended to be controlled by an external auto-scaler (as discussed in [#492](https://issue.k8s.io/492)), which would change its replicas field. We will not add scheduling policies (for example, [spreading](https://issue.k8s.io/367#issuecomment-48428019)) to the ReplicationController. Nor should it verify that the pods controlled match the currently specified template, as that would obstruct auto-sizing and other automated processes. Similarly, completion deadlines, ordering dependencies, configuration expansion, and other features belong elsewhere. We even plan to factor out the mechanism for bulk pod creation ([#170](https://issue.k8s.io/170)).

The ReplicationController is intended to be a composable building-block primitive. We expect higher-level APIs and/or tools to be built on top of it and other complementary primitives for user convenience in the future. The "macro" operations currently supported by kubectl (run, scale) are proof-of-concept examples of this. For instance, we could imagine something like [Asgard](https://netflixtechblog.com/asgard-web-based-cloud-management-and-deployment-2c9fc4e4d3a1) managing ReplicationControllers, auto-scalers, services, scheduling policies, canaries, etc.