**Workload Management**

Kubernetes provides several built-in APIs for declarative management of your workloads and the components of those workloads. Ultimately, your applications run as containers inside Pods; however, managing individual Pods would be a lot of effort. For example, if a Pod fails, you probably want to run a new Pod to replace it. Kubernetes can do that for you. You use the Kubernetes API to create a workload object that represents a higher abstraction level than a Pod, and then the Kubernetes control plane automatically manages Pod objects on your behalf, based on the specification for the workload object you defined.

**The built-in APIs for managing workloads are:**

Deployment (and, indirectly, ReplicaSet), the most common way to run an application on your cluster. Deployment is a good fit for managing a stateless application workload on your cluster, where any Pod in the Deployment is interchangeable and can be replaced if needed. (Deployments are a replacement for the legacy ReplicationController API).

A StatefulSet lets you manage one or more Pods – all running the same application code – where the Pods rely on having a distinct identity. This is different from a Deployment where the Pods are expected to be interchangeable. The most common use for a StatefulSet is to be able to make a link between its Pods and their persistent storage. For example, you can run a StatefulSet that associates each Pod with a PersistentVolume. If one of the Pods in the StatefulSet fails, Kubernetes makes a replacement Pod that is connected to the same PersistentVolume.

A DaemonSet defines Pods that provide facilities that are local to a specific node; for example, a driver that lets containers on that node access a storage system. You use a DaemonSet when the driver, or other node-level service, has to run on the node where it's useful. Each Pod in a DaemonSet performs a role similar to a system daemon on a classic Unix / POSIX server. A DaemonSet might be fundamental to the operation of your cluster, such as a plugin to let that node access cluster networking, it might help you to manage the node, or it could provide less essential facilities that enhance the container platform you are running. You can run DaemonSets (and their pods) across every node in your cluster, or across just a subset (for example, only install the GPU accelerator driver on nodes that have a GPU installed).

You can use a Job and / or a CronJob to define tasks that run to completion and then stop. A Job represents a one-off task, whereas each CronJob repeats according to a schedule.

**Automatic Cleanup for Finished Jobs**

A time-to-live mechanism to clean up old Jobs that have finished execution. When your Job has finished, it's useful to keep that Job in the API (and not immediately delete the Job) so that you can tell whether the Job succeeded or failed. Kubernetes' TTL-after-finished controller provides a TTL (time to live) mechanism to limit the lifetime of Job objects that have finished execution.

**Cleanup for finished Jobs**

The TTL-after-finished controller is only supported for Jobs. You can use this mechanism to clean up finished Jobs (either Complete or Failed) automatically by specifying the .spec.ttlSecondsAfterFinished field of a Job, as in this example.

The TTL-after-finished controller assumes that a Job is eligible to be cleaned up TTL seconds after the Job has finished. The timer starts once the status condition of the Job changes to show that the Job is either Complete or Failed; once the TTL has expired, that Job becomes eligible for cascading removal. When the TTL-after-finished controller cleans up a job, it will delete it cascadingly, that is to say it will delete its dependent objects together with it.

Kubernetes honors object lifecycle guarantees on the Job, such as waiting for finalizers. You can set the TTL seconds at any time. Here are some examples for setting the .spec.ttlSecondsAfterFinished field of a Job:

* Specify this field in the Job manifest, so that a Job can be cleaned up automatically sometime after it finishes.
* Manually set this field of existing, already finished Jobs, so that they become eligible for cleanup.
* Use a mutating admission webhook to set this field dynamically at Job creation time. Cluster administrators can use this to enforce a TTL policy for finished jobs.
* Use a mutating admission webhook to set this field dynamically after the Job has finished, and choose different TTL values based on job status, labels. For this case, the webhook needs to detect changes to the .status of the Job and only set a TTL when the Job is being marked as completed.
* Write your own controller to manage the cleanup TTL for Jobs that match a particular selector.

**Caveats**

**Updating TTL for finished Jobs**

You can modify the TTL period, e.g. .spec.ttlSecondsAfterFinished field of Jobs, after the job is created or has finished. If you extend the TTL period after the existing ttlSecondsAfterFinished period has expired, Kubernetes doesn't guarantee to retain that Job, even if an update to extend the TTL returns a successful API response.

**Time skew**

Because the TTL-after-finished controller uses timestamps stored in the Kubernetes jobs to determine whether the TTL has expired or not, this feature is sensitive to time skew in your cluster, which may cause the control plane to clean up Job objects at the wrong time. Clocks aren't always correct, but the difference should be very small. Please be aware of this risk when setting a non-zero TTL.

**ReplicationController**

Legacy API for managing workloads that can scale horizontally. Superseded by the Deployment and ReplicaSet APIs. A Deployment that configures a 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.

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 Manifest**

As with all other Kubernetes config, a ReplicationController needs apiVersion, kind, and metadata fields. When the control plane creates new Pods for a ReplicationController, the .metadata.name of the ReplicationController is part of the basis for naming those Pods. The name of a ReplicationController 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. For general information about working with configuration files, see object management. A ReplicationController also needs a .spec section.

**Pod Template**

The .spec.template is the only required field of the .spec. The .spec.template is a pod template. It has exactly the same schema as a Pod, 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. Only a .spec.template.spec.restartPolicy 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.

**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. 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).

**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. 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, 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. When using the REST API or client library, 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.

**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, 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, 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 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), which would change its replicas field. We will not add scheduling policies (for example, spreading) 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). 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 managing ReplicationControllers, auto-scalers, services, scheduling policies, canaries, etc.

**API Object**

Replication controller is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: ReplicationController API object.

**Alternatives to ReplicationController**

**ReplicaSet**

ReplicaSet is the next-generation ReplicationController that supports the new set-based label selector. It's mainly used by Deployment as a mechanism to orchestrate pod creation, deletion and updates. Note that we recommend using Deployments instead of directly using Replica Sets, unless you require custom update orchestration or don't require updates at all.

**Deployment (Recommended)**

Deployment is a higher-level API object that updates its underlying Replica Sets and their Pods. Deployments are recommended if you want the rolling update functionality, because they are declarative, server-side, and have additional features.

**Bare Pods**

Unlike in the case where a user directly created pods, a ReplicationController 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 ReplicationController 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 ReplicationController delegates local container restarts to some agent on the node, such as the kubelet.

**Job**

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

**DaemonSet**

Use a DaemonSet instead of a ReplicationController 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.

**Deployments**

A Deployment manages a set of Pods to run an application workload, usually one that doesn't maintain state. A Deployment provides declarative updates for Pods and ReplicaSets. You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments. Do not manage ReplicaSets owned by a Deployment. Consider opening an issue in the main Kubernetes repository if your use case is not covered below.

**Use Case**

The following are typical use cases for Deployments:

* Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
* Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
* Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
* Scale up the Deployment to facilitate more load.
* Pause the rollout of a Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
* Use the status of the Deployment as an indicator that a rollout has stuck.
* Clean up older ReplicaSets that you don't need anymore.

**Creating a Deployment**

The following is an example of a Deployment. It creates a ReplicaSet to bring up three nginx Pods:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

In this example:

* A Deployment named nginx-deployment is created, indicated by the .metadata.name field. This name will become the basis for the ReplicaSets and Pods which are created later. See Writing a Deployment Spec for more details.
* The Deployment creates a ReplicaSet that creates three replicated Pods, indicated by the .spec.replicas field.
* The .spec.selector field defines how the created ReplicaSet finds which Pods to manage. In this case, you select a label that is defined in the Pod template (app: nginx). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule.

The .spec.selector.matchLabels field is a map of {key,value} pairs. A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is "key", the operator is "In", and the values array contains only "value". All of the requirements, from both matchLabels and matchExpressions, must be satisfied in order to match.

The .spec.template field contains the following sub-fields:

* The Pods are labeled app: nginxusing the .metadata.labels field.
* The Pod template's specification, or .spec field, indicates that the Pods run one container, nginx, which runs the nginx Docker Hub image at version 1.14.2.
* Create one container and name it nginx using the .spec.containers[0].name field.

Before you begin, make sure your Kubernetes cluster is up and running. Follow the steps given below to create the above Deployment:

1. Create the Deployment by running the following command:

kubectl apply -f <https://k8s.io/examples/controllers/nginx-deployment.yaml>

1. Run kubectl get deployments to check if the Deployment was created.

If the Deployment is still being created, the output is similar to the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
| nginx-deployment | 0/3 | 0 | 0 | 1s |

When you inspect the Deployments in your cluster, the following fields are displayed:

* NAME lists the names of the Deployments in the namespace.
* READY displays how many replicas of the application are available to your users. It follows the pattern ready/desired.
* UP-TO-DATE displays the number of replicas that have been updated to achieve the desired state.
* AVAILABLE displays how many replicas of the application are available to your users.
* AGE displays the amount of time that the application has been running.

Notice how the number of desired replicas is 3 according to .spec.replicas field.

1. To see the Deployment rollout status, run kubectl rollout status deployment/nginx-deployment.

The output is similar to:

Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

deployment "nginx-deployment" successfully rolled out