**INDEX**

**Workloads**

**i)pods…………………… (2..44)**

**ii)containers………………(44..119)**

**Workloads**

**Pod overview:**

This page provides an overview of Pod, the smallest deployable object in the Kubernetes object model.

* [Understanding Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#understanding-pods)
* [Working with Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#working-with-pods)
* [Pod Templates](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates)
* [What's next](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#what-s-next)

*Pod* is the basic execution unit of a Kubernetes application–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents processes running on your [Cluster](https://kubernetes.io/docs/reference/glossary/?all=true#term-cluster).

A Pod encapsulates an application’s container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: *a single instance of an application in Kubernetes*, which might consist of either a single [container](https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#why-containers) or a small number of containers that are tightly coupled and that share resources.

[Docker](https://www.docker.com/) is the most common container runtime used in a Kubernetes Pod, but Pods support other [container runtimes](https://kubernetes.io/docs/setup/production-environment/container-runtimes/) as well.

Pods in a Kubernetes cluster can be used in two main ways:

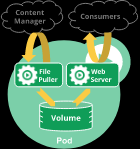
* Pods that run a single container. The “one-container-per-Pod” model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.
* Pods that run multiple containers that need to work together. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers might form a single cohesive unit of service–one container serving files from a shared volume to the public, while a separate “sidecar” container refreshes or updates those files. The Pod  wraps these containers and storage resources together as a single manageable entity. The [Kubernetes Blog](https://kubernetes.io/blog) has some additional information on Pod use cases. For more information, see:
* [The Distributed System Toolkit: Patterns for Composite Containers](https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns)
* [Container Design Patterns](https://kubernetes.io/blog/2016/06/container-design-patterns)

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (e.g., run multiple instances), you should use multiple Pods, one for each instance. In Kubernetes, this is generally referred to as replication. Replicated Pods are usually created and managed as a group by an abstraction called a Controller. See [Pods and Controllers](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pods-and-controllers) for more information.

### How Pods manage multiple Containers:

Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated.

Note that grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled. For example, you might have a container that acts as a web server for files in a shared volume, and a separate “sidecar” container that updates those files from a remote source, as in the following diagram:



#### Networking:

Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports. Containers inside a Pod can communicate with one another using localhost. When containers in a Pod communicate with entities outside the Pod, they must coordinate how they use the shared network resources (such as ports).

#### Storage:

A Pod can specify a set of shared storage [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/). All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/) for more information on how Kubernetes implements shared storage in a Pod.

Working with pods:

You’ll rarely create individual Pods directly in Kubernetes–even singleton Pods. This is because Pods are designed as relatively ephemeral, disposable entities. When a Pod gets created (directly by you, or indirectly by a Controller), it is scheduled to run on a [Node](https://kubernetes.io/docs/concepts/architecture/nodes/) in your cluster. The Pod remains on that Node until the process is terminated, the pod object is deleted, the Pod is evicted for lack of resources, or the Node fails.

**Note:** Restarting a container in a Pod should not be confused with restarting the Pod. The Pod itself does not run, but is an environment the containers run in and persists until it is deleted.

Pods do not, by themselves, self-heal. If a Pod is scheduled to a Node that fails, or if the scheduling operation itself fails, the Pod is deleted; likewise, a Pod won’t survive an eviction due to a lack of resources or Node maintenance. Kubernetes uses a higher-level abstraction, called a Controller that handles the work of managing the relatively disposable Pod instances. Thus, while it is possible to use Pod directly, it’s far more common in Kubernetes to manage your pods using a Controller. See [Pods and Controllers](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pods-and-controllers) for more information on how Kubernetes uses Controllers to implement Pod scaling and healing.

### Pods and Controllers:

A Controller can create and manage multiple Pods for you, handling replication and rollout and providing self-healing capabilities at cluster scope. For example, if a Node fails, the Controller might automatically replace the Pod by scheduling an identical replacement on a different Node.

Some examples of Controllers that contain one or more pods include:

* [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* [Statefulset](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
* [Daemonset](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)

In general, Controllers use a Pod Template that you provide to create the Pods for which it is responsible

.

Pod templates:

Pod templates are pod specifications which are included in other objects, such as [Replication Controllers](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/), [Jobs](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/), and [daemonsets](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/). Controllers use Pod Templates to make actual pods. The sample below is a simple manifest for a Pod which contains a container that prints a message.

Apiversion: v1

Kind: Pod

Metadata:

Name: myapp-pod

Labels:

App: myapp

Spec:

Containers:

- name: myapp-container

Image: busybox

Command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']

Rather than specifying the current desired state of all replicas, pod templates are like cookie cutters. Once a cookie has been cut, the cookie has no relationship to the cutter. There is no “quantum entanglement”. Subsequent changes to the template or even switching to a new template has no direct effect on the pods already created. Similarly, pods created by a replication controller may subsequently be updated directly. This is in deliberate contrast to pods, which do specify the current desired state of all containers belonging to the pod. This approach radically simplifies system semantics and increases the flexibility of the primitive.

**Pods**

Pods:

*Pods* are the smallest deployable units of computing that can be created and managed in Kubernetes.

* [What is a Pod?](https://kubernetes.io/docs/concepts/workloads/pods/pod/#what-is-a-pod)
* [Motivation for Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/#motivation-for-pods)
* [Uses of pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/#uses-of-pods)
* [Alternatives considered](https://kubernetes.io/docs/concepts/workloads/pods/pod/#alternatives-considered)
* [Durability of pods (or lack thereof)](https://kubernetes.io/docs/concepts/workloads/pods/pod/#durability-of-pods-or-lack-thereof)
* [Termination of Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/#termination-of-pods)
* [Privileged mode for pod containers](https://kubernetes.io/docs/concepts/workloads/pods/pod/#privileged-mode-for-pod-containers)
* [API Object](https://kubernetes.io/docs/concepts/workloads/pods/pod/#api-object)

What is pod?

A Pod (as in a pod of whales or pea pod) is a group of one or more [containers](https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#why-containers) (such as Docker containers), with shared storage/network, and a specification for how to run the containers. A Pod’s contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific “logical host” - it contains one or more application containers which are relatively tightly coupled — in a pre-container world, being executed on the same physical or virtual machine would mean being executed on the same logical host.

While Kubernetes supports more container runtimes than just Docker, Docker is the most commonly known runtime, and it helps to describe Pods in Docker terms.

The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container. Within a Pod’s context, the individual applications may have further sub-isolations applied.

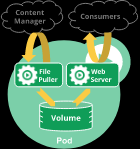
Containers within a Pod share an IP address and port space, and can find each other via localhost. They can also communicate with each other using standard inter-process communications like systemv semaphores or POSIX shared memory. Containers in different Pods have distinct IP addresses and can not communicate by IPC without [special configuration](https://kubernetes.io/docs/concepts/policy/pod-security-policy/). These containers usually communicate with each other via Pod IP addresses.

Applications within a Pod also have access to shared [volumes](https://kubernetes.io/docs/concepts/storage/volumes/), which are defined as part of a Pod and are made available to be mounted into each application’s filesystem.

In terms of [Docker](https://www.docker.com/) constructs, a Pod is modelled as a group of Docker containers with shared namespaces and shared filesystem volumes.

Like individual application containers, Pods are considered to be relatively ephemeral (rather than durable) entities. As discussed in [pod lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/), Pods are created, assigned a unique ID (UID), and scheduled to nodes where they remain until termination (according to restart policy) or deletion. If a [Node](https://kubernetes.io/docs/concepts/architecture/nodes/) dies, the Pods scheduled to that node are scheduled for deletion, after a timeout period. A given Pod (as defined by a UID) is not “rescheduled” to a new node; instead, it can be replaced by an identical Pod, with even the same name if desired, but with a new UID (see [replication controller](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/) for more details).

When something is said to have the same lifetime as a Pod, such as a volume, that means that it exists as long as that Pod (with that UID) exists. If that Pod is deleted for any reason, even if an identical replacement is created, the related thing (e.g. Volume) is also destroyed and created anew.



#### Pod diagram

A multi-container Pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

### Motivation for pods:

### Management:

Pods are a model of the pattern of multiple cooperating processes which form a cohesive unit of service. They simplify application deployment and management by providing a higher-level abstraction than the set of their constituent applications. Pods serve as unit of deployment, horizontal scaling, and replication. Colocation (co-scheduling), shared fate (e.g. Termination), coordinated replication, resource sharing, and dependency management are handled automatically for containers in a Pod.

### Resource sharing and communication:

Pods enable data sharing and communication among their constituents.

The applications in a Pod all use the same network namespace (same IP and port space), and can thus “find” each other and communicate using localhost. Because of this, applications in a Pod must coordinate their usage of ports. Each Pod has an IP address in a flat shared networking space that has full communication with other physical computers and Pods across the network.

Containers within the Pod see the system hostname as being the same as the configured name for the Pod. There’s more about this in the [networking](https://kubernetes.io/docs/concepts/cluster-administration/networking/) section.

In addition to defining the application containers that run in the Pod, the Pod specifies a set of shared storage volumes. Volumes enable data to survive container restarts and to be shared among the applications within the Pod.

Uses of pods:

Pods can be used to host vertically integrated application stacks (e.g. LAMP), but their primary motivation is to support co-located, co-managed helper programs, such as:

* Content management systems, file and data loaders, local cache managers, etc.
* Log and checkpoint backup, compression, rotation, snapshotting, etc.
* Data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
* Proxies, bridges, and adapters
* Controllers, managers, configurators, and updaters

Individual Pods are not intended to run multiple instances of the same application, in general.

For a longer explanation, see [The Distributed System toolkit: Patterns for Composite Containers](https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns).

Alternatives considered:

**Why not just run multiple programs in a single (Docker) container?**

1. Transparency. Making the containers within the Pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of conveniences for users.
2. Decoupling software dependencies. The individual containers may be versioned, rebuilt and redeployed independently. Kubernetes may even support live updates of individual containers someday.
3. Ease of use. Users don’t need to run their own process managers, worry about signal and exit-code propagation, etc.
4. Efficiency. Because the infrastructure takes on more responsibility, containers can be lighter weight.

**Why not support affinity-based co-scheduling of containers?**

That approach would provide co-location, but would not provide most of the benefits of Pods, such as resource sharing, IPC, guaranteed fate sharing, and simplified management.

Durability of pods (or lack thereof):

Pods aren’t intended to be treated as durable entities. They won’t survive scheduling failures, node failures, or other evictions, such as due to lack of resources, or in the case of node maintenance.

In general, users shouldn’t need to create Pods directly. They should almost always use controllers even for singletons, for example, [Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/). Controllers provide self-healing with a cluster scope, as well as replication and rollout management. Controllers like [statefulset](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset.md) can also provide support to stateful Pods.

The use of collective apis as the primary user-facing primitive is relatively common among cluster scheduling systems, including [Borg](https://research.google.com/pubs/pub43438.html), [Marathon](https://mesosphere.github.io/marathon/docs/rest-api.html), [Aurora](http://aurora.apache.org/documentation/latest/reference/configuration/#job-schema), and [Tupperware](https://www.slideshare.net/Docker/aravindnarayanan-facebook140613153626phpapp02-37588997).

Pod is exposed as a primitive in order to facilitate:

* Scheduler and controller pluggability
* Support for pod-level operations without the need to “proxy” them via controller apis
* Decoupling of Pod lifetime from controller lifetime, such as for bootstrapping
* Clean composition of Kubelet-level functionality with cluster-level functionality — Kubelet is effectively the “pod controller”
* High-availability applications, which will expect Pods to be replaced in advance of their termination and certainly in advance of deletion, such as in the case of planned evictions or image prefetching.

Temmination of pods:

Because Pods represent running processes on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (vs being violently killed with a KILL signal and having no chance to clean up). Users should be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When a user requests deletion of a Pod, the system records the intended grace period before the Pod is allowed to be forcefully killed, and a TERM signal is sent to the main process in each container. Once the grace period has expired, the KILL signal is sent to those processes, and the Pod is then deleted from the API server. If the Kubelet or the container manager is restarted while waiting for processes to terminate, the termination will be retried with the full grace period.

An example flow:

* User sends command to delete Pod, with default grace period (30s)
* The Pod in the API server is updated with the time beyond which the Pod is considered “dead” along with the grace period.
* Pod shows up as “Terminating” when listed in client commands
* (simultaneous with 3) When the Kubelet sees that a Pod has been marked as terminating because the time in 2 has been set, it begins the Pod shutdown process.
  + If one of the Pod’s containers has defined a [prestop hook](https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/" \l "hook-details), it is invoked inside of the container. If the prestop hook is still running after the grace period expires, step 2 is then invoked with a small (2 second) extended grace period.
  + The container is sent the TERM signal. Note that not all containers in the Pod will receive the TERM signal at the same time and may each require a prestop hook if the order in which they shut down matters.
* (simultaneous with 3) Pod is removed from endpoints list for service, and are no longer considered part of the set of running Pods for replication controllers. Pods that shutdown slowly cannot continue to serve traffic as load balancers (like the service proxy) remove them from their rotations.
* When the grace period expires, any processes still running in the Pod are killed with SIGKILL.
* The Kubelet will finish deleting the Pod on the API server by setting grace period 0 (immediate deletion). The Pod disappears from the API and is no longer visible from the client.

By default, all deletes are graceful within 30 seconds. The kubectl delete command supports the --grace-period=<seconds> option which allows a user to override the default and specify their own value. The value 0 [force deletes](https://kubernetes.io/docs/concepts/workloads/pods/pod/#force-deletion-of-pods) the Pod. You must specify an additional flag --force along with --grace-period=0 in order to perform force deletions.

### Force deletion of pods:

Force deletion of a Pod is defined as deletion of a Pod from the cluster state and etcd immediately. When a force deletion is performed, the API server does not wait for confirmation from the kubelet that the Pod has been terminated on the node it was running on. It removes the Pod in the API immediately so a new Pod can be created with the same name. On the node, Pods that are set to terminate immediately will still be given a small grace period before being force killed.

Force deletions can be potentially dangerous for some Pods and should be performed with caution. In case of statefulset Pods, please refer to the task documentation for [deleting Pods from a statefulset](https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/).

Privileged mode for pod container:

Any container in a Pod can enable privileged mode, using the privileged flag on the [security context](https://kubernetes.io/docs/tasks/configure-pod-container/security-context/) of the container spec. This is useful for containers that want to use Linux capabilities like manipulating the network stack and accessing devices. Processes within the container get almost the same privileges that are available to processes outside a container. With privileged mode, it should be easier to write network and volume plugins as separate Pods that don’t need to be compiled into the kubelet.

**Note:** Your container runtime must support the concept of a privileged container for this setting to be relevant.

API object:

Pod is a top-level resource in the Kubernetes REST API. The [Pod API object](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#pod-v1-core) definition describes the object in detail.

**Pods life cycle**

This page describes the lifecycle of a Pod.

* [Pod phase](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-phase)
* [Pod conditions](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-conditions)
* [Container probes](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes)
* [Pod and Container status](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-and-container-status)
* [Container States](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-states)
* [Pod readiness gate](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-readiness-gate)
* [Restart policy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy)
* [Pod lifetime](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-lifetime)
* [Examples](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#examples)
* [What's next](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#what-s-next)

Pod phase:

A Pod’s status field is a [podstatus](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "podstatus-v1-core) object, which has a phase field.

The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. The phase is not intended to be a comprehensive rollup of observations of Container or Pod state, nor is it intended to be a comprehensive state machine.

The number and meanings of Pod phase values are tightly guarded. Other than what is documented here, nothing should be assumed about Pods that have a given phase value.

Here are the possible values for phase:

| Value | Description |
| --- | --- |
| Pending | The Pod has been accepted by the Kubernetes system, but one or more of the Container images has not been created. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while. |
| Running | The Pod has been bound to a node, and all of the Containers have been created. At least one Container is still running, or is in the process of starting or restarting. |
| Succeeded | All Containers in the Pod have terminated in success, and will not be restarted. |
| Failed | All Containers in the Pod have terminated, and at least one Container has terminated in failure. That is, the Container either exited with non-zero status or was terminated by the system. |
| Unknown | For some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Pod. |

Pod condition:

A Pod has a podstatus, which has an array of [podconditions](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "podcondition-v1-core) through which the Pod has or has not passed. Each element of the podcondition array has six possible fields:

* The lastprobetime field provides a timestamp for when the Pod condition was last probed.
* The lasttransitiontime field provides a timestamp for when the Pod last transitioned from one status to another.
* The message field is a human-readable message indicating details about the transition.
* The reason field is a unique, one-word, camelcase reason for the condition’s last transition.
* The status field is a string, with possible values “True”, “False”, and “Unknown”.
* The type field is a string with the following possible values:
* Podscheduled: the Pod has been scheduled to a node;
* Ready: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services;
* Initialized: all [init containers](https://kubernetes.io/docs/concepts/workloads/pods/init-containers) have started successfully;
* Unschedulable: the scheduler cannot schedule the Pod right now, for example due to lack of resources or other constraints;
* Containersready: all containers in the Pod are ready.

Container probes:

A [Probe](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#probe-v1-core) is a diagnostic performed periodically by the [kubelet](https://kubernetes.io/docs/admin/kubelet/) on a Container. To perform a diagnostic, the kubelet calls a [Handler](https://godoc.org/k8s.io/kubernetes/pkg/api/v1#Handler) implemented by the Container. There are three types of handlers:

* [Execaction](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#execaction-v1-core): Executes a specified command inside the Container. The diagnostic is considered successful if the command exits with a status code of 0.
* [Tcpsocketaction](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#tcpsocketaction-v1-core): Performs a TCP check against the Container’s IP address on a specified port. The diagnostic is considered successful if the port is open.
* [Httpgetaction](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#httpgetaction-v1-core): Performs an HTTP Get request against the Container’s IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.
* Each probe has one of three results:
* Success: The Container passed the diagnostic.
* Failure: The Container failed the diagnostic.
* Unknown: The diagnostic failed, so no action should be taken.
* The kubelet can optionally perform and react to three kinds of probes on running Containers:
* Livenessprobe: Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its [restart policy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy). If a Container does not provide a liveness probe, the default state is Success.
* Readinessprobe: Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod’s IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is Failure. If a Container does not provide a readiness probe, the default state is Success.
* Startupprobe: Indicates whether the application within the Container is started. All other probes are disabled if a startup probe is provided, until it succeeds. If the startup probe fails, the kubelet kills the Container, and the Container is subjected to its [restart policy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy). If a Container does not provide a startup probe, the default state is Success.

### When should you use a liveness probe?

If the process in your Container is able to crash on its own whenever it encounters an issue or becomes unhealthy, you do not necessarily need a liveness probe; the kubelet will automatically perform the correct action in accordance with the Pod’s restartpolicy.

If you’d like your Container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a restartpolicy of Always or onfailure.

### When should you use a readiness probe?

If you’d like to start sending traffic to a Pod only when a probe succeeds, specify a readiness probe. In this case, the readiness probe might be the same as the liveness probe, but the existence of the readiness probe in the spec means that the Pod will start without receiving any traffic and only start receiving traffic after the probe starts succeeding. If your Container needs to work on loading large data, configuration files, or migrations during startup, specify a readiness probe.

If you want your Container to be able to take itself down for maintenance, you can specify a readiness probe that checks an endpoint specific to readiness that is different from the liveness probe.

Note that if you just want to be able to drain requests when the Pod is deleted, you do not necessarily need a readiness probe; on deletion, the Pod automatically puts itself into an unready state regardless of whether the readiness probe exists. The Pod remains in the unready state while it waits for the Containers in the Pod to stop.

### When should you use a startup probe?

If your Container usually starts in more than initialdelayseconds + failurethreshold × periodseconds, you should specify a startup probe that checks the same endpoint as the liveness probe. The default for periodseconds is 30s. You should then set its failurethreshold high enough to allow the Container to start, without changing the default values of the liveness probe. This helps to protect against deadlocks.

For more information about how to set up a liveness, readiness, startup probe, see [Configure Liveness, Readiness and Startup Probes](https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/).

Pod and container status:

Fordetailed information about Pod Container status, see [podstatus](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "podstatus-v1-core) and [containerstatus](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "containerstatus-v1-core). Note that the information reported as Pod status depends on the current [containerstate](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "containerstatus-v1-core).

Container status:

Once Pod is assigned to a node by scheduler, kubelet starts creating containers using container runtime.There are three possible states of containers: Waiting, Running and Terminated. To check state of container, you can use kubectl describe pod [POD\_NAME]. State is displayed for each container within that Pod.

* Waiting: Default state of container. If container is not in either Running or Terminated state, it is in Waiting state. A container in Waiting state still runs its required operations, like pulling images, applying Secrets, etc. Along with this state, a message and reason about the state are displayed to provide more information.
* ...
* State: Waiting
* Reason: errimagepull

...

* Running: Indicates that the container is executing without issues. The poststart hook (if any) is executed prior to the container entering a Running state. This state also displays the time when the container entered Running state.
* ...
* State: Running
* Started: Wed, 30 Jan 2019 16:46:38 +0530

...

* Terminated: Indicates that the container completed its execution and has stopped running. A container enters into this when it has successfully completed execution or when it has failed for some reason. Regardless, a reason and exit code is displayed, as well as the container’s start and finish time. Before a container enters into Terminated, prestop hook (if any) is executed.
* ...
* State: Terminated
* Reason: Completed
* Exit Code: 0
* Started: Wed, 30 Jan 2019 11:45:26 +0530
* Finished: Wed, 30 Jan 2019 11:45:26 +0530

...

Pod readiness gate:

In order to add extensibility to Pod readiness by enabling the injection of extra feedback or signals into podstatus, Kubernetes 1.11 introduced a feature named [Pod ready++](https://github.com/kubernetes/enhancements/blob/master/keps/sig-network/0007-pod-ready%2B%2B.md). You can use the new field readinessgate in the podspec to specify additional conditions to be evaluated for Pod readiness. If Kubernetes cannot find such a condition in the status.conditions field of a Pod, the status of the condition is default to “False”. Below is an example:

Kind: Pod

...

Spec:

Readinessgates:

- conditiontype: "www.example.com/feature-1"

Status:

Conditions:

- type: Ready *# this is a builtin podcondition*

Status: "False"

Lastprobetime: null

Lasttransitiontime: 2018-01-01T00:00:00Z

- type: "www.example.com/feature-1" *# an extra podcondition*

Status: "False"

Lastprobetime: null

Lasttransitiontime: 2018-01-01T00:00:00Z

Containerstatuses:

- containerid: docker://abcd...

Ready: true

...

The new Pod conditions must comply with Kubernetes [label key format](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#syntax-and-character-set). Since the kubectl patch command still doesn’t support patching object status, the new Pod conditions have to be injected through the PATCH action using one of the [kubeclient libraries](https://kubernetes.io/docs/reference/using-api/client-libraries/).

With the introduction of new Pod conditions, a Pod is evaluated to be ready **only** when both the following statements are true:

* All containers in the Pod are ready.
* All conditions specified in readinessgates are “True”.

To facilitate this change to Pod readiness evaluation, a new Pod condition containersready is introduced to capture the old Pod Ready condition.

In K8s 1.11, as an alpha feature, the “Pod Ready++” feature has to be explicitly enabled by setting the podreadinessgates [feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) to true.

Restart policy:

A podspec has a restartpolicy field with possible values Always, onfailure, and Never. The default value is Always. Restartpolicy applies to all Containers in the Pod. Restartpolicy only refers to restarts of the Containers by the kubelet on the same node. Exited Containers that are restarted by the kubelet are restarted with an exponential back-off delay (10s, 20s, 40s …) capped at five minutes, and is reset after ten minutes of successful execution. As discussed in the [Pods document](https://kubernetes.io/docs/user-guide/pods/#durability-of-pods-or-lack-thereof), once bound to a node, a Pod will never be rebound to another node.

Pod lifetime:

In general, Pods remain until a human or controller process explicitly removes them. The control plane cleans up terminated Pods (with a phase of Succeeded or Failed), when the number of Pods exceeds the configured threshold (determined by terminated-pod-gc-threshold in the kube-controller-manager). This avoids a resource leak as Pods are created and terminated over time.

Three types of controllers are available:

* Use a [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/) for Pods that are expected to terminate, for example, batch computations. Jobs are appropriate only for Pods with restartpolicy equal to onfailure or Never.
* Use a [replicationcontroller](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/), [replicaset](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/), or [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) for Pods that are not expected to terminate, for example, web servers. Replicationcontrollers are appropriate only for Pods with a restartpolicy of Always.
* Use a [daemonset](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) for Pods that need to run one per machine, because they provide a machine-specific system service.

All three types of controllers contain a podtemplate. It is recommended to create the appropriate controller and let it create Pods, rather than directly create Pods yourself. That is because Pods alone are not resilient to machine failures, but controllers are.

If a node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the phase of all Pods on the lost node to Failed.

### Examples:

### Advanced liveness probe example:

Liveness probes are executed by the kubelet, so all requests are made in the kubelet network namespace.

Apiversion: v1

Kind: Pod

Metadata:

Labels:

Test: liveness

Name: liveness-http

Spec:

Containers:

- args:

- /server

Image: k8s.gcr.io/liveness

Livenessprobe:

Httpget:

*#* when *"host" is not defined, "podip" will be used*

*# host: my-host*

*#* when *"scheme" is not defined, "HTTP" scheme will be used. Only "HTTP" and "HTTPS" are allowed*

*#* scheme*: HTTPS*

Path: /healthz

Port: 8080

Httpheaders:

- name: X-Custom-Header

Value: Awesome

Initialdelayseconds: 15

Timeoutseconds: 1

Name: liveness

### Example states:

* Pod is running and has one Container. Container exits with success.
  + Log completion event.
  + If restartpolicy is:
    - Always: Restart Container; Pod phase stays Running.
    - Onfailure: Pod phase becomes Succeeded.
    - Never: Pod phase becomes Succeeded.
* Pod is running and has one Container. Container exits with failure.
  + Log failure event.
  + If restartpolicy is:
    - Always: Restart Container; Pod phase stays Running.
    - Onfailure: Restart Container; Pod phase stays Running.
    - Never: Pod phase becomes Failed.
* Pod is running and has two Containers. Container 1 exits with failure.
  + Log failure event.
  + If restartpolicy is:
    - Always: Restart Container; Pod phase stays Running.
    - Onfailure: Restart Container; Pod phase stays Running.
    - Never: Do not restart Container; Pod phase stays Running.
  + If Container 1 is not running, and Container 2 exits:
    - Log failure event.
    - If restartpolicy is:
      * Always: Restart Container; Pod phase stays Running.
      * Onfailure: Restart Container; Pod phase stays Running.
      * Never: Pod phase becomes Failed.
* Pod is running and has one Container. Container runs out of memory.
* Container terminates in failure.
* Log OOM event.
* If restartpolicy is:
  + - Always: Restart Container; Pod phase stays Running.
    - Onfailure: Restart Container; Pod phase stays Running.
    - Never: Log failure event; Pod phase becomes Failed.
* Pod is running, and a disk dies.
  + Kill all Containers.
  + Log appropriate event.
  + Pod phase becomes Failed.
  + If running under a controller, Pod is recreated elsewhere.
* Pod is running, and its node is segmented out.
  + Node controller waits for timeout.
  + Node controller sets Pod phase to Failed.
  + If running under a controller, Pod is recreated elsewhere.

**Init containers**

This page provides an overview of init containers: specialized containers that run before app containers in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/). Init containers can contain utilities or setup scripts not present in an app image.

You can specify init containers in the Pod specification alongside the containers array (which describes app containers).

* [Understanding init containers](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#understanding-init-containers)
* [Using init containers](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#using-init-containers)
* [Detailed behavior](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#detailed-behavior)
* [What's next](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#what-s-next)

Understanding init containers:

A [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) can have multiple containers running apps within it, but it can also have one or more init containers, which are run before the app containers are started.

Init containers are exactly like regular containers, except:

* Init containers always run to completion.
* Each init container must complete successfully before the next one starts.

If a Pod’s init container fails, Kubernetes repeatedly restarts the Pod until the init container succeeds. However, if the Pod has a restartpolicy of Never, Kubernetes does not restart the Pod.

To specify an init container for a Pod, add the initcontainers field into the Pod specification, as an array of objects of type [Container](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#container-v1-core), alongside the app containers array. The status of the init containers is returned in .status.initcontainerstatuses field as an array of the container statuses (similar to the .status.containerstatuses field).

### Differences from regular containers:

Init containers support all the fields and features of app containers, including resource limits, volumes, and security settings. However, the resource requests and limits for an init container are handled differently, as documented in [Resources](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#resources).

Also, init containers do not support readiness probes because they must run to completion before the Pod can be ready.

If you specify multiple init containers for a Pod, Kubelet runs each init container sequentially. Each init container must succeed before the next can run. When all of the init containers have run to completion, Kubelet initializes the application containers for the Pod and runs them as usual.

Using init containers:

Because init containers have separate images from app containers, they have some advantages for start-up related code:

* Init containers can contain utiilities or custom code for setup that are not present in an app image. For example, there is no need to make an image FROM another image just to use a tool like sed, awk, python, or dig during setup.
* The application image builder and deployer roles can work independently without the need to jointly build a single app image.
* Init containers can run with a different view of the filesystem than app containers in the same Pod. Consequently, they can be given access to [Secrets](https://kubernetes.io/docs/concepts/configuration/secret/) that app containers cannot access.
* Because init containers run to completion before any app containers start, init containers offer a mechanism to block or delay app container startup until a set of preconditions are met. Once preconditions are met, all of the app containers in a Pod can start in parallel.
* Init containers can securely run utilities or custom code that would otherwise make an app container image less secure. By keeping unnecessary tools separate you can limit the attack  
  surface of your app container image.

### Examples:

Here are some ideas for how to use init containers:

* Wait for a [Service](https://kubernetes.io/docs/concepts/services-networking/service/) to be created, using a shell one-line command like:

For i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; done; exit 1

* Register this Pod with a remote server from the downward API with a command like:

Curl -X POST http://$MANAGEMENT\_SERVICE\_HOST:$MANAGEMENT\_SERVICE\_PORT/register -d 'instance=$(<POD\_NAME>)&ip=$(<POD\_IP>)'

* Wait for some time before starting the app container with a command like

Sleep 60

* Clone a Git repository into a [Volume](https://kubernetes.io/docs/concepts/storage/volumes/)
* Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app container. For example, place the POD\_IP value in a configuration and generate the main app configuration file using Jinja.

#### Init containers in use:

This example defines a simple Pod that has two init containers. The first waits for myservice, and the second waits for mydb. Once both init containers complete, the Pod runs the app container from its spec section.

Apiversion: v1

Kind: Pod

Metadata:

Name: myapp-pod

Labels:

App: myapp

Spec:

Containers:

- name: myapp-container

Image: busybox:1.28

Command: ['sh', '-c', 'echo The app is running! && sleep 3600']

Initcontainers:

- name: init-myservice

Image: busybox:1.28

Command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']

- name: init-mydb

Image: busybox:1.28

Command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']

You can start this Pod by running:

Kubectl apply -f myapp.yaml

Pod/myapp-pod created

And check on its status with:

Kubectl get -f myapp.yaml

NAME READY STATUS RESTARTS AGE

Myapp-pod 0/1 Init:0/2 0 6m

Or for more details:

Kubectl describe -f myapp.yaml

Name: myapp-pod

Namespace: default

[...]

Labels: app=myapp

Status: Pending

[...]

Init Containers:

Init-myservice:

[...]

State: Running

[...]

Init-mydb:

[...]

State: Waiting

Reason: podinitializing

Ready: False

[...]

Containers:

Myapp-container:

[...]

State: Waiting

Reason: podinitializing

Ready: False

[...]

Events:

Firstseen lastseen Count From subobjectpath Type Reason Message

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

16s 16s 1 {default-scheduler } Normal Scheduled Successfully assigned myapp-pod to 172.17.4.201

16s 16s 1 {kubelet 172.17.4.201} spec.initcontainers{init-myservice} Normal Pulling pulling image "busybox"

13s 13s 1 {kubelet 172.17.4.201} spec.initcontainers{init-myservice} Normal Pulled Successfully pulled image "busybox"

13s 13s 1 {kubelet 172.17.4.201} spec.initcontainers{init-myservice} Normal Created Created container with docker id 5ced34a04634; Security:[seccomp=unconfined]

13s 13s 1 {kubelet 172.17.4.201} spec.initcontainers{init-myservice} Normal Started Started container with docker id 5ced34a04634

To see logs for the init containers in this Pod, run:

Kubectl logs myapp-pod -c init-myservice *# Inspect the first init container*

Kubectl logs myapp-pod -c init-mydb *# Inspect the second init container*

At this point, those init containers will be waiting to discover Services named mydb and myservice.

Here’s a configuration you can use to make those Services appear:

---

Apiversion: v1

Kind: Service

Metadata:

Name: myservice

Spec:

Ports:

- protocol: TCP

Port: 80

Targetport: 9376

---

Apiversion: v1

Kind: Service

Metadata:

Name: mydb

Spec:

Ports:

- protocol: TCP

Port: 80

Targetport: 9377

To create  the mydb and myservice services:

Kubectl apply -f services.yaml

Service/myservice created

Service/mydb created

You’ll then see that those init containers complete, and that the myapp-pod Pod moves into the Running state:

Kubectl get -f myapp.yaml

NAME READY STATUS RESTARTS AGE

Myapp-pod 1/1 Running 0 9m

This simple example should provide some inspiration for you to create your own init containers. [What’s next](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#what-s-next) contains a link to a more detailed example.

Detailed behaviour:

During the startup of a Pod, each init container starts in order, after the network and volumes are initialized. Each container must exit successfully before the next container starts. If a container fails to start due to the runtime or exits with failure, it is retried according to the Pod restartpolicy. However, if the Pod restartpolicy is set to Always, the init containers use restartpolicy onfailure.

A Pod cannot be Ready until all init containers have succeeded. The ports on an init container are not aggregated under a Service. A Pod that is initializing is in the Pending state but should have a condition Initialized set to true.

If the Pod [restarts](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#pod-restart-reasons), or is restarted, all init containers must execute again.

Changes to the init container spec are limited to the container image field. Altering an init container image field is equivalent to restarting the Pod.

Because init containers can be restarted, retried, or re-executed, init container code should be idempotent. In particular, code that writes to files on emptydirs should be prepared for the possibility that an output file already exists.

Init containers have all of the fields of an app container. However, Kubernetes prohibits readinessprobe from being used because init containers cannot define readiness distinct from completion. This is enforced during validation.

Use activedeadlineseconds on the Pod and livenessprobe on the container to prevent init containers from failing forever. The active deadline includes init containers.

The name of each app and init container in a Pod must be unique; a validation error is thrown for any container sharing a name with another.

### Resources:

Given the ordering and execution for init containers, the following rules for resource usage apply:

* The highest of any particular resource request or limit defined on all init containers is the effective init request/limit
* The Pod’s effective request/limit for a resource is the higher of:
  + The sum of all app containers request/limit for a resource
  + The effective init request/limit for a resource
* Scheduling is done based on effective requests/limits, which means init containers can reserve resources for initialization that are not used during the life of the Pod.
* The qos (quality of service) tier of the Pod’s effective qos tier is the qos tier for init containers and app containers alike.

Quota and limits are applied based on the effective Pod request and limit.

Pod level control groups (cgroups) are based on the effective Pod request and limit, the same as the scheduler.

### Pod restart reasons:

A Pod can restart, causing re-execution of init containers, for the following reasons:

* A user updates the Pod specification, causing the init container image to change. Any changes to the init container image restarts the Pod. App container image changes only restart the app container.
* The Pod infrastructure container is restarted. This is uncommon and would have to be done by someone with root access to nodes.
* All containers in a Pod are terminated while restartpolicy is set to Always, forcing a restart, and the init container completion record has been lost due to garbage collection.

#### Pods preset:

This page provides an overview of podpresets, which are objects for injecting certain information into pods at creation time. The information can include secrets, volumes, volume mounts, and environment variables.

* [Understanding Pod Presets](https://kubernetes.io/docs/concepts/workloads/pods/podpreset/#understanding-pod-presets)
* [How It Works](https://kubernetes.io/docs/concepts/workloads/pods/podpreset/#how-it-works)
* [Enable Pod Preset](https://kubernetes.io/docs/concepts/workloads/pods/podpreset/#enable-pod-preset)
* [What's next](https://kubernetes.io/docs/concepts/workloads/pods/podpreset/#what-s-next)

Understanding pod presets:

A Pod Preset is an API resource for injecting additional runtime requirements into a Pod at creation time. You use [label selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors) to specify the Pods to  which a given Pod Preset applies.

Using a Pod Preset allows pod template authors to not have to explicitly provide all information for every pod. This way, authors of pod templates consuming a specific service do not need to know all the details about that service.

For more information about the background, see the [design proposal for podpreset](https://git.k8s.io/community/contributors/design-proposals/service-catalog/pod-preset.md).

How it works:

Kubernetes provides an admission controller (podpreset) which, when enabled, applies Pod Presets to incoming pod creation requests. When a pod creation request occurs, the system does the following:

* Retrieve all podpresets available for use.
* Check if the label selectors of any podpreset matches the labels on the pod being created.
* Attempt to merge the various resources defined by the podpreset into the Pod being created.
* On error, throw an event documenting the merge error on the pod, and create the pod without any injected resources from the podpreset.
* Annotate the resulting modified Pod spec to indicate that it has been modified by a podpreset. The annotation is of the form podpreset.admission.kubernetes.io/podpreset-<pod-preset name>: "<resource version>".

Each Pod can be matched by zero or more Pod Presets; and each podpreset can be applied to zero or more pods. When a podpreset is applied to one or more Pods, Kubernetes modifies the Pod Spec. For changes to Env, envfrom, and volumemounts, Kubernetes modifies the container spec for all containers in the Pod; for changes to Volume, Kubernetes modifies the Pod Spec.

**Note:** A Pod Preset is capable of modifying the following fields in a Pod spec when appropriate: - The .spec.containers field. - The initcontainers field (requires Kubernetes version 1.14.0 or later).

### Disable Pod Preset for a Specific Pod:

There may be instances where you wish for a Pod to not be altered by any Pod Preset mutations. In these cases, you can add an annotation in the Pod Spec of the form: podpreset.admission.kubernetes.io/exclude: "true".

Enable pod preset:

In order to use Pod Presets in your cluster you must ensure the following:

1. You have enabled the API type settings.k8s.io/v1alpha1/podpreset. For example, this can be done by including settings.k8s.io/v1alpha1=true in the --runtime-config option for the API server. In minikube add this flag --extra-config=apiserver.runtime-config=settings.k8s.io/v1alpha1=true while starting the cluster.
2. You have enabled the admission controller podpreset. One way to doing this is to include podpreset in the --enable-admission-plugins option value specified for the API server. In minikube add this flag

--extra-config=apiserver.enable-admission-plugins=namespacelifecycle,limitranger,serviceaccount,defaultstorageclass,defaulttolerationseconds,noderestriction,mutatingadmissionwebhook,validatingadmissionwebhook,resourcequota,podpreset

While starting the cluster.

1. You have defined your Pod Presets by creating podpreset objects in the namespace you will use.

Pod Topology Spread Constraints:

. You can use topology spread constraints to control how [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) are spread across your cluster among failure-domains such as regions, zones, nodes, and other user-defined topology domains. This can help to achieve high availability as well as efficient resource utilization.

* [Prerequisites](https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/#prerequisites)
* [Spread Constraints for Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/#spread-constraints-for-pods)
* [Comparison with podaffinity/podantiaffinity](https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/#comparison-with-podaffinity-podantiaffinity)
* [Known Limitations](https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/#known-limitations)

### Prerequisites:

### Enable Feature Gate:

Ensure the evenpodsspread feature gate is enabled (it is disabled by default in 1.16). See [Feature Gates](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) for an explanation of enabling feature gates. The evenpodsspread feature gate must be enabled for the [API Server](https://kubernetes.io/docs/reference/generated/kube-apiserver/) **and** [scheduler](https://kubernetes.io/docs/reference/generated/kube-scheduler/).

### Node Labels:

Topology spread constraints rely on node labels to identify the topology domain(s) that each Node is in. For example, a Node might have labels: node=node1,zone=us-east-1a,region=us-east-1

Suppose you have a 4-node cluster with the following labels:

NAME STATUS ROLES AGE VERSION LABELS

Node1 Ready <none> 4m26s v1.16.0 node=node1,zone=zonea

Node2 Ready <none> 3m58s v1.16.0 node=node2,zone=zonea

Node3 Ready <none> 3m17s v1.16.0 node=node3,zone=zoneb

Node4 Ready <none> 2m43s v1.16.0 node=node4,zone=zoneb

Then the cluster is logically viewed as below:

+---------------+---------------+

| zonea | zoneb |

+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+

Instead of manually applying labels, you can also reuse the [well-known labels](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/) that are created and populated automatically on most clusters.

Spread constraints for pods:

### API

The field pod.spec.topologyspreadconstraints is introduced in 1.16 as below:

Apiversion: v1

Kind: Pod

Metadata:

Name: mypod

Spec:

Topologyspreadconstraints:

- maxskew: <integer*>*

*Topologykey: <string>*

Whenunsatisfiable: <string*>*

*Labelselector: <object>*

You can define one or multiple topologyspreadconstraint to instruct the kube-scheduler how to place each incoming Pod in relation to the existing Pods across your cluster. The fields are:

* Maxskew describes the degree to which Pods may be unevenly distributed. It’s the maximum permitted difference between the number of matching Pods in any two topology domains of a given topology type. It must be greater than zero.
* Topologykey is the key of node labels. If two Nodes are labelled with this key and have identical values for that label, the scheduler treats both Nodes as being in the same topology. The scheduler tries to place a balanced number of Pods into each topology domain.
* Whenunsatisfiiable indicates how to deal with a Pod if it doesn’t satisfy the spread constraint:
  + Donotschedule (default) tells the scheduler not to schedule it.
  + Scheduleanyway tells the scheduler to still schedule it while prioritizing nodes that minimize the skew.
* Labelselector is used to find matching Pods. Pods that match this label selector are counted to determine the number of Pods in their corresponding topology domain. See [Label Selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors) for more details.

You can read more about this field by running kubectl explain Pod.spec.topologyspreadconstraints.

### Example: One topologyspreadconstraint

Suppose you have a 4-node cluster where 3 Pods labeled foo:bar are located in node1, node2 and node3 respectively (P represents Pod):

+---------------+---------------+

| zonea | zoneb |

+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+

| P | P | P | |

+-------+-------+-------+-------+

If we want an incoming Pod to be evenly spread with existing Pods across zones,

|  |
| --- |
| E: pause  Image: k8s.gcr.io/pause:3.1 |

Topologykey: zone implies the even distribution will only be applied to the nodes which have label pair “zone:” present. Whenunsatisfiable: donotschedule tells the scheduler to let it stay pending if the incoming Pod can’t satisfy the constraint.

If the scheduler placed this incoming Pod into “zonea”, the Pods distribution would become [3, 1], hence the actual skew is 2 (3 - 1) - which violates maxskew: 1. In this example, the incoming Pod can only be placed onto “zoneb”:

+---------------+---------------+ +---------------+---------------+

| zonea | zoneb | | zonea | zoneb |

+-------+-------+-------+-------+ +-------+-------+-------+-------+

| node1 | node2 | node3 | node4 | OR | node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+ +-------+-------+-------+-------+

| P | P | P | P | | P | P | P P | |

+-------+-------+-------+-------+ +-------+-------+-------+-------+

You can tweak the Pod spec to meet various kinds of requirements:

* Change maxskew to a bigger value like “2” so that the incoming Pod can be placed onto “zonea” as well.
* Change topologykey to “node” so as to distribute the Pods evenly across nodes instead of zones. In the above example, if maxskew remains “1”, the incoming Pod can only be placed onto “node4”.
* Change whenunsatisfiable: donotschedule to whenunsatisfiable: scheduleanyway to ensure the incoming Pod to be always schedulable (suppose other scheduling apis are satisfied). However, it’s preferred to be placed onto the topology domain which has fewer matching Pods. (Be aware that this preferability is jointly normalized with other internal scheduling priorities like resource usage ratio, etc.)

### Example:

### Multiple topology spread constraints

This builds upon the previous example. Suppose you have a 4-node cluster where 3 Pods labeled foo:bar are located in node1, node2 and node3 respectively (P represents Pod):

+---------------+---------------+

| zonea | zoneb |

+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+

| P | P | P | |

+-------+-------+-------+-------+

You can use 2 topologyspreadconstraints to control the Pods spreading on both zone and node:

| [Pods/topology-spread-constraints/two-constraints.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/pods/topology-spread-constraints/two-constraints.yaml) |
| --- |
| Kind: Pod  Apiversion: v1  Metadata:  Name: mypod  Labels:  Foo: bar  Spec:  Topologyspreadconstraints:  - maxskew: 1  Topologykey: zone  Whenunsatisfiable: donotschedule  Labelselector:  Matchlabels:  Foo: bar  - maxskew: 1  Topologykey: node  Whenunsatisfiable: donotschedule  Labelselector:  Matchlabels:  Foo: bar  Containers:  - name: pause  Image: k8s.gcr.io/pause:3.1 |

In this case, to match the first constraint, the incoming Pod can only be placed onto “zoneb”; while in terms of the second constraint, the incoming Pod can only be placed onto “node4”. Then the results of 2 constraints are anded, so the only viable option is to place on “node4”.

Multiple constraints can lead to conflicts. Suppose you have a 3-node cluster across 2 zones:

+---------------+-------+

| zonea | zoneb |

+-------+-------+-------+

| node1 | node2 | node3 |

+-------+-------+-------+

| P P | P | P P |

+-------+-------+-------+

If you apply “two-constraints.yaml” to this cluster, you will notice “mypod” stays in Pending state. This is because: to satisfy the first constraint, “mypod” can only be put to “zoneb”; while in terms of the second constraint, “mypod” can only put to “node2”. Then a joint result of “zoneb” and “node2” returns nothing.

To overcome this situation, you can either increase the maxskew or modify one of the constraints to use whenunsatisfiable: scheduleanyway.

### Conventions:

There are some implicit conventions worth noting here:

* Only the Pods holding the same namespace as the incoming Pod can be matching candidates.
* Nodes without topologyspreadconstraints[\*].topologykey present will be bypassed. It implies that:
  1. The Pods located on those nodes do not impact maxskew calculation - in the above example, suppose “node1” does not have label “zone”, then the 2 Pods will be disregarded, hence the incomingpod will be scheduled into “zonea”.
  2. The incoming Pod has no chances to be scheduled onto this kind of nodes - in the above example, suppose a “node5” carrying label {zone-typo: zonec} joins the cluster, it will be bypassed due to the absence of label key “zone”.
* Be aware of what will happen if the incomingpod’s topologyspreadconstraints[\*].labelselector doesn’t match its own labels. In the above example, if we remove the incoming Pod’s labels, it can still be placed onto “zoneb” since the constraints are still satisfied. However, after the placement, the degree of imbalance of the cluster remains unchanged - it’s still zonea having 2 Pods which hold label {foo:bar}, and zoneb having 1 Pod which holds label {foo:bar}. So if this is not what you expect, we recommend the workload’s topologyspreadconstraints[\*].labelselector to match its own labels.
* If the incoming Pod has spec.nodeselector or spec.affinity.nodeaffinity defined, nodes not matching them will be bypassed.

Suppose you have a 5-node cluster ranging from zonea to zonec:

+---------------+---------------+-------+

| zonea | zoneb | zonec |

+-------+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 | node5 |

+-------+-------+-------+-------+-------+

| P | P | P | | |

+-------+-------+-------+-------+-------+

And you know that “zonec” must be excluded. In this case, you can compose the yaml as below, so that “mypod” will be placed onto “zoneb” instead of “zonec”. Similarly spec.nodeselector is also respected.

| [Pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml) |
| --- |
| Kind: Pod  Apiversion: v1  Metadata:  Name: mypod  Labels:  Foo: bar  Spec:  Topologyspreadconstraints:  - maxskew: 1  Topologykey: zone  Whenunsatisfiable: donotschedule  Labelselector:  Matchlabels:  Foo: bar  Affinity:  Nodeaffinity:  Requiredduringschedulingignoredduringexecution:  Nodeselectorterms:  - matchexpressions:  - key: zone  Operator: notin  Values:  - zonec  Containers:  - name: pause  Image: k8s.gcr.io/pause:3.1 |

Comparision With Podaffinity/Podantiaffinity:

In Kubernetes, directives related to “Affinity” control how Pods are scheduled - more packed or more scattered.

* For podaffinity, you can try to pack any number of Pods into qualifying topology domain(s).
* For podantiaffinity, only one Pod can be scheduled into a single topology domain.

The “evenpodsspread” feature provides flexible options to distribute Pods evenly across different topology domains - to achieve high availability or cost-saving. This can also help on rolling update workloads and scaling out replicas smoothly. See [Motivation](https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/20190221-even-pods-spreading.md#motivation) for more details.

**Disruptions**

This guide is for application owners who want to build highly available applications, and thus need to understand what types of Disruptions can happen to Pods.

It is also for Cluster Administrators who want to perform automated cluster actions, like upgrading and autoscaling clusters.

* [Voluntary and Involuntary Disruptions](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#voluntary-and-involuntary-disruptions)
* [Dealing with Disruptions](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#dealing-with-disruptions)
* [How Disruption Budgets Work](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#how-disruption-budgets-work)
* [PDB Example](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#pdb-example)
* [Separating Cluster Owner and Application Owner Roles](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#separating-cluster-owner-and-application-owner-roles)
* [How to perform Disruptive Actions on your Cluster](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#how-to-perform-disruptive-actions-on-your-cluster)
* [What's next](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#what-s-next)

Voluntary And Involuntary Disruptions:

Pods do not disappear until someone (a person or a controller) destroys them, or there is an unavoidable hardware or system software error.

We call these unavoidable cases *involuntary disruptions* to an application. Examples are:

* A hardware failure of the physical machine backing the node
* Cluster administrator deletes VM (instance) by mistake
* Cloud provider or hypervisor failure makes VM disappear
* A kernel panicthe
* Node disappears from the cluster due to cluster network partition
* Eviction of a pod due to the node being [out-of-resources](https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/).

Except for the out-of-resources condition, all these conditions should be familiar to most users; they are not specific to Kubernetes.

We call other cases *voluntary disruptions*. These include both actions initiated by the application owner and those initiated by a Cluster Administrator. Typical application owner actions include:

* Deleting the deployment or other controller that manages the pod
* Updating a deployment’s pod template causing a restart
* Directly deleting a pod (e.g. By accident)
* Cluster Administrator actions include:
* [Draining a node](https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/) for repair or upgrade.
* Draining a node from a cluster to scale the cluster down (learn about [Cluster Autoscaling](https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/#cluster-autoscaler) ).
* Removing a pod from a node to permit something else to fit on that node.

These actions might be taken directly by the cluster administrator, or by automation run by the cluster administrator, or by your cluster hosting provider.

Ask your cluster administrator or consult your cloud provider or distribution documentation to determine if any sources of voluntary disruptions are enabled for your cluster. If none are enabled, you can skip creating Pod Disruption Budgets.

Caution: Not all voluntary disruptions are constrained by Pod Disruption Budgets. For example, deleting deployments or pods bypasses Pod Disruption Budgets.

Dealing with disruptions:

Here are some ways to mitigate involuntary disruptions:

* Ensure you  pod [requests the resources](https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-ram-container) it needs.
* Replicate your application if you need higher availability. (Learn about running replicated [stateless](https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/) and [stateful](https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/) applications.)
* For even higher availability when running replicated applications, spread applications across racks (using [anti-affinity](https://kubernetes.io/docs/user-guide/node-selection/#inter-pod-affinity-and-anti-affinity-beta-feature)) or across zones (if using a [multi-zone cluster](https://kubernetes.io/docs/setup/multiple-zones).)

The frequency of voluntary disruptions varies. On a basic Kubernetes cluster, there are no voluntary disruptions at all. However, your cluster administrator or hosting provider may run some additional services which cause voluntary disruptions. For example, rolling out node software updates can cause voluntary disruptions. Also, some implementations of cluster (node) autoscaling may cause voluntary disruptions to defragment and compact nodes. Your cluster administrator or hosting provider should have documented what level of voluntary disruptions, if any, to expect.

Kubernetes offers features to help run highly available applications at the same time as frequent voluntary disruptions. We call this set of features *Disruption Budgets*.

How Disruption Budgets Work:

An Application Owner can create a poddisruptionbudget object (PDB) for each application. A PDB limits the number of pods of a replicated application that are down simultaneously from voluntary disruptions. For example, a quorum-based application would like to ensure that the number of replicas running is never brought below the number needed for a quorum. A web front end might want to ensure that the number of replicas serving load never falls below a certain percentage of the total.

Cluster managers and hosting providers should use tools which respect Pod Disruption Budgets by calling the [Eviction API](https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/#the-eviction-api) instead of directly deleting pods or deployments. Examples are the kubectl drain command and the Kubernetes-on-GCE cluster upgrade script (cluster/gce/upgrade.sh).

When a cluster administrator wants to drain a node they use the kubectl drain command. That tool tries to evict all the pods on the machine. The eviction request may be temporarily rejected, and the tool periodically retries all failed requests until all pods are terminated, or until a configurable timeout is reached.

A PDB specifies the number of replicas that an application can tolerate having, relative to how many it is intended to have. For example, a Deployment which has a .spec.replicas: 5 is supposed to have 5 pods at any given time. If its PDB allows for there to be 4 at a time, then the Eviction API will allow voluntary disruption of one, but not two pods, at a time.

The group of pods that comprise the application is specified using a label selector, the same as the one used by the application’s controller (deployment, stateful-set, etc).

The “intended” number of pods is computed from the .spec.replicas of the pods controller. The controller is discovered from the pods using the .metadata.ownerreferences of the object.

Pdbs cannot prevent [involuntary disruptions](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#voluntary-and-involuntary-disruptions) from occurring, but they do count against the budget.

Pods which are deleted or unavailable due to a rolling upgrade to an application do count against the disruption budget, but controllers (like deployment and stateful-set) are not limited by pdbs when doing rolling upgrades – the handling of failures during application updates is configured in the controller spec. (Learn about [updating a deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment).)

When a pod is evicted using the eviction API, it is gracefully terminated (see terminationgraceperiodseconds in [podspec](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "podspec-v1-core).)

PDB Example:

Consider a cluster with 3 nodes, node-1 through node-3. The cluster is running several applications. One of them has 3 replicas initially called pod-a, pod-b, and pod-c. Another, unrelated pod without a PDB, called pod-x, is also shown. Initially, the pods are laid out as follows:

| Node-1 | Node-2 | Node-3 |
| --- | --- | --- |
| Pod-a *available* | Pod-b *available* | Pod-c *available* |
| Pod-x *available* |  |  |

All 3 pods are part of a deployment, and they collectively have a PDB which requires there be at least 2 of the 3 pods to be available at all times.

For example, assume the cluster administrator wants to reboot into a new kernel version to fix a bug in the kernel. The cluster administrator first tries to drain node-1 using the kubectl drain command. That tool tries to evict pod-a and pod-x. This succeeds immediately. Both pods go into the terminating state at the same time. This puts the cluster in this state:

| Node-1 *draining* | Node-2 | Node-3 |
| --- | --- | --- |
| Pod-a *terminating* | Pod-b *available* | Pod-c *available* |
| Pod-x *terminating* |  |  |

The deployment notices that one of the pods is terminating, so it creates a replacement called pod-d. Since node-1 is cordoned, it lands on another node. Something has also created pod-y as a replacement for pod-x.

(Note: for a statefulset, pod-a, which would be called something like pod-0, would need to terminate completely before its replacement, which is also called pod-0 but has a different UID, could be created. Otherwise, the example applies to a statefulset as well.)

Now the cluster is in this state:

| Node-1 *draining* | Node-2 | Node-3 |
| --- | --- | --- |
| Pod-a *terminating* | Pod-b *available* | Pod-c *available* |
| Pod-x *terminating* | Pod-d *starting* | Pod-y |

At some point, the pods terminate, and the cluster looks like this:

| Node-1 *drained* | Node-2 | Node-3 |
| --- | --- | --- |
|  | Pod-b *available* | Pod-c *available* |
|  | Pod-d *starting* | Pod-y |

At this point, if an impatient cluster administrator tries to drain node-2 or node-3, the drain command will block, because there are only 2 available pods for the deployment, and its PDB requires at least 2. After some time passes, pod-d becomes available.

The cluster state now looks like this:

| Node-1 *drained* | Node-2 | Node-3 |
| --- | --- | --- |
|  | Pod-b *available* | Pod-c *available* |
|  | Pod-d *available* | Pod-y |

Now, the cluster administrator tries to drain node-2. The drain command will try to evict the two pods in some order, say pod-b first and then pod-d. It will succeed at evicting pod-b. But, when it tries to evict pod-d, it will be refused because that would leave only one pod available for the deployment.

The deployment creates a replacement for pod-b called pod-e. Because there are not enough resources in the cluster to schedule pod-e the drain will again block. The cluster may end up in this state:

| Node-1 *drained* | Node-2 | Node-3 | *No node* |
| --- | --- | --- | --- |
|  | Pod-b *available* | Pod-c *available* | Pod-e *pending* |
|  | Pod-d *available* | Pod-y |  |

At this point, the cluster administrator needs to add a node back to the cluster to proceed with the upgrade.

You can see how Kubernetes varies the rate at which disruptions can happen, according to:

* How many replicas an application needs
* How long it takes to gracefully shutdown an instance
* How long it takes a new instance to start up
* The type of controller
* The cluster’s resource capacity

Separating Cluster Owner And Application Owner Roles:

Often, it is useful to think of the Cluster Manager and Application Owner as separate roles with limited knowledge of each other. This separation of responsibilities may make sense in these scenarios:

* When there are many application teams sharing a Kubernetes cluster, and there is natural specialization of roles
* When third-party tools or services are used to automate cluster management

Pod Disruption Budgets support this separation of roles by providing an interface between the roles.

If you do not have such a separation of responsibilities in your organization, you may not need to use Pod Disruption Budgets.

How To Perform Disruptive Actions On Your Cluster:

If you are a Cluster Administrator, and you need to perform a disruptive action on all the nodes in your cluster, such as a node or system software upgrade, here are some options:

* Accept downtime during the upgrade.
* Failover to another complete replica cluster.
  + No downtime, but may be costly both for the duplicated nodes and for human effort to orchestrate the switchover.
* Write disruption tolerant applications and use pdbs.
  + No downtime.
  + Minimal resource duplication.
  + Allows more automation of cluster administration.
  + Writing disruption-tolerant applications is tricky, but the work to tolerate voluntary disruptions largely overlaps with work to support autoscaling and tolerating involuntary disruptions

**Ephemeral Containers**

This page provides an overview of ephemeral containers: a special type of container that runs temporarily in an existing [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) to accomplish user-initiated actions such as troubleshooting. You use ephemeral containers to inspect services rather than to build applications.

Warning: Ephemeral containers are in early alpha state and are not suitable for production clusters. You should expect the feature not to work in some situations, such as when targeting the namespaces of a container. In accordance with the [Kubernetes Deprecation Policy](https://kubernetes.io/docs/reference/using-api/deprecation-policy/), this alpha feature could change significantly in the future or be removed entirely.

* [Understanding ephemeral containers](https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/#understanding-ephemeral-containers)
* [Uses for ephemeral containers](https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/#uses-for-ephemeral-containers)

Understanding Ephemeral Containers

[Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) are the fundamental building block of Kubernetes applications. Since Pods are intended to be disposable and replaceable, you cannot add a container to a Pod once it has been created. Instead, you usually delete and replace Pods in a controlled fashion using [deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/).

Sometimes it’s necessary to inspect the state of an existing Pod, however, for example to troubleshoot a hard-to-reproduce bug. In these cases you can run an ephemeral container in an existing Pod to inspect its state and run arbitrary commands.

### What is an ephemeral container?

Ephemeral containers differ from other containers in that they lack guarantees for resources or execution, and they will never be automatically restarted, so they are not appropriate for building applications. Ephemeral containers are described using the same containerspec as regular containers, but many fields are incompatible and disallowed for ephemeral containers.

* Ephemeral containers may not have ports, so fields such as ports, livenessprobe, readinessprobe are disallowed.
* Pod resource allocations are immutable, so setting resources is disallowed.
* For a complete list of allowed fields, see the [ephemeralcontainer reference documentation](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "ephemeralcontainer-v1-core).

Ephemeral containers are created using a special ephemeralcontainers handler in the API rather than by adding them directly to pod.spec, so it’s not possible to add an ephemeral container using kubectl edit.

Like regular containers, you may not change or remove an ephemeral container after you have added it to a Pod.

Uses for ephemeral containers:

Ephemeral containers are useful for interactive troubleshooting when kubectl exec is insufficient because a container has crashed or a container image doesn’t include debugging utilities.

In particular, [distroless images](https://github.com/GoogleContainerTools/distroless" \t "_blank) enable you to deploy minimal container images that reduce attack surface and exposure to bugs and vulnerabilities. Since distroless images do not include a shell or any debugging utilities, it’s difficult to troubleshoot distroless images using kubectl exec alone.

When using ephemeral containers, it’s helpful to enable [process namespace sharing](https://kubernetes.io/docs/tasks/configure-pod-container/share-process-namespace/) so you can view processes in other containers.

### Examples:

**Note:** The examples in this section require the ephemeralcontainers [feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) to be enabled, and Kubernetes client and server version v1.16 or later.

The examples in this section demonstrate how ephemeral containers appear in the API. You would normally use a kubectl plugin for troubleshooting that automates these steps.

Ephemeral containers are created using the ephemeralcontainers subresource of Pod, which can be demonstrated using kubectl --raw. First describe the ephemeral container to add as an ephemeralcontainers list:

{

"apiversion": "v1",

"kind": "ephemeralcontainers",

"metadata": {

"name": "example-pod"

},

"ephemeralcontainers": [{

"command": [

"sh"

],

"image": "busybox",

"imagepullpolicy": "ifnotpresent",

"name": "debugger",

"stdin": true,

"tty": true,

"terminationmessagepolicy": "File"

}]

}

To update the ephemeral containers of the already running example-pod:

Kubectl replace --raw /api/v1/namespaces/default/pods/example-pod/ephemeralcontainers -f ec.json

This will return the new list of ephemeral containers:

{

"kind":"ephemeralcontainers",

"apiversion":"v1",

"metadata":{

"name":"example-pod",

"namespace":"default",

"selflink":"/api/v1/namespaces/default/pods/example-pod/ephemeralcontainers",

"uid":"a14a6d9b-62f2-4119-9d8e-e2ed6bc3a47c",

"resourceversion":"15886",

"creationtimestamp":"2019-08-29T06:41:42Z"

},

"ephemeralcontainers":[

{

"name":"debugger",

"image":"busybox",

"command":[

"sh"

],

"resources":{

},

"terminationmessagepolicy":"File",

"imagepullpolicy":"ifnotpresent",

"stdin":true,

"tty":true

}

]

}

You can view the state of the newly created ephemeral container using kubectl describe:

Kubectl describe pod example-pod

...

Ephemeral Containers:

Debugger:

Container ID: docker://cf81908f149e7e9213d3c3644eda55c72efaff67652a2685c1146f0ce151e80f

Image: busybox

Image ID: docker-pullable://busybox@sha256:9f1003c480699be56815db0f8146ad2e22efea85129b5b5983d0e0fb52d9ab70

Port: <none>

Host Port: <none>

Command:

Sh

State: Running

Started: Thu, 29 Aug 2019 06:42:21 +0000

Ready: False

Restart Count: 0

Environment: <none>

Mounts: <none>

...

You can attach to the new ephemeral container using kubectl attach:

Kubectl attach -it example-pod -c debugger

If process namespace sharing is enabled, you can see processes from all the containers in that Pod. For example, after attaching, you run ps in the debugger container:

*# Run this in a shell inside the "debugger" ephemeral container*

Ps auxww

The output is similar to:

PID USER TIME COMMAND

1 root 0:00 /pause

6 root 0:00 nginx: master process nginx -g daemon off;

11 101 0:00 nginx: worker process

12 101 0:00 nginx: worker process

13 101 0:00 nginx: worker process

14 101 0:00 nginx: worker process

15 101 0:00 nginx: worker process

16 101 0:00 nginx: worker process

17 101 0:00 nginx: worker process

18 101 0:00 nginx: worker process

19 root 0:00 /pause

24 root 0:00 sh

29 root 0:00 ps auxww

**Controllers**

**Replicaset:**

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**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#how-a-replicaset-works)
* [**When to use a replicaset**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#when-to-use-a-replicaset)
* [**Example**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#example)
* [**Non-Template Pod acquisitions**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#non-template-pod-acquisitions)
* [**Writing a replicaset manifest**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#writing-a-replicaset-manifest)
* [**Working with replicasets**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#working-with-replicasets)
* [**Alternatives to replicaset**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#alternatives-to-replicaset)

**How a replicaset works:**

 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.

The link a replicaset has to its Pods is via the Pods’ [metadata.ownerreferences](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/" \l "owners-and-dependents) 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](https://kubernetes.io/docs/concepts/architecture/controller/) and it matches a replicaset’s selector, it will be immediately acquired by said replicaset.

When to use a replicaset:

A Replica Set 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:

|  |
| --- |
| 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: <none>

Replicas: 3 current / 3 desired

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

Pod Template:

Labels: app=guestbook

Tier=frontend

Containers:

Php-redis:

Image: gcr.io/google\_samples/gb-frontend:v3

Port: 80/TCP

Requests:

Cpu: 100m

Memory: 100Mi

Environment:

GET\_HOSTS\_FROM: dns

Mounts: <none>

Volumes: <none>

Events:

Firstseen lastseen Count From subobjectpath Type Reason Message

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

1m 1m 1 {replicaset-controller } Normal successfulcreate Created pod: frontend-qhloh

1m 1m 1 {replicaset-controller } Normal successfulcreate Created pod: frontend-dnjpy

1m 1m 1 {replicaset-controller } Normal successfulcreate Created pod: frontend-9si5l

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-9si5l 1/1 Running 0 1m

Frontend-dnjpy 1/1 Running 0 1m

Frontend-qhloh 1/1 Running 0 1m

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-9si5l -o yaml

The output will look similar to this, with the frontend replicaset’s info set in the metadata’s ownerreferences field:

Apiversion: v1

Kind: Pod

Metadata:

Creationtimestamp: 2019-01-31T17:20:41Z

Generatename: frontend-

Labels:

Tier: frontend

Name: frontend-9si5l

Namespace: default

Ownerreferences:

- apiversion: apps/v1

Blockownerdeletion: true

Controller: true

Kind: replicaset

Name: frontend

Uid: 892a2330-257c-11e9-aecd-025000000001

...

Non template pod acquisition:

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](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/pods/pod-rs.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-9si5l 1/1 Running 0 1m

Frontend-dnjpy 1/1 Running 0 1m

Frontend-qhloh 1/1 Running 0 1m

Pod2 0/1 Terminating 0 4s

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-pxj4r 1/1 Running 0 5s

Pod1 1/1 Running 0 13s

Pod2 1/1 Running 0 13s

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 just replicaset. In Kubernetes 1.9 the API version apps/v1 on the replicaset kind is the current version and is enabled by default. The API version apps/v1beta2 is deprecated. Refer to the first lines of the frontend.yaml example for guidance.

A replicaset 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 a [pod template](https://kubernetes.io/docs/concepts/workloads/Pods/pod-overview/#pod-templates) 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](https://kubernetes.io/docs/concepts/workloads/Pods/pod-lifecycle/#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](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/). As discussed [earlier](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#how-a-replicaset-works) 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 replicaset:**

### Deleting a replicaset and its Pods:

To delete a replicaset and all of its Pods, use [kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "delete). The [Garbage collector](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/) 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/frontend' **\**

> -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](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "delete) with the --cascade=false 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/frontend' **\**

> -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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-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.

### Replicaset as a Horizontal Pod Autoscaler Target:

A replicaset can also be a target for [Horizontal Pod Autoscalers (HPA)](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/). 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](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/controllers/hpa-rs.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

### Alternatives Replicaset:

### Deployment (recommended)

[Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/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 (for example, Kubelet or Docker).

### Job:

Use a [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/) instead of a replicaset for Pods that are expected to terminate on their own (that is, batch jobs).

### Daemonset:

Use a [daemonset](https://kubernetes.io/docs/concepts/workloads/controllers/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.

### Replication Controller:

Replicasets are the successors to [replicationcontrollers](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/). 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](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors). As such, replicasets are preferred over replicationcontrollers.

**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**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#how-a-replicationcontroller-works)
* [**Running an example replicationcontroller**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#running-an-example-replicationcontroller)
* [**Writing a replicationcontroller Spec**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#writing-a-replicationcontroller-spec)
* [**Working with replicationcontrollers**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#working-with-replicationcontrollers)
* [**Common usage patterns**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#common-usage-patterns)
* [**Writing programs for Replication**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#writing-programs-for-replication)
* [**Responsibilities of the replicationcontroller**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#responsibilities-of-the-replicationcontroller)
* [**API Object**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#api-object)
* [**Alternatives to replicationcontroller**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#alternatives-to-replicationcontroller)
* [**For more information**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#for-more-information)

How replication controller works:

If there are too many p ods, 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 of replication:

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

| [Controllers/replication.yaml](https://raw.githubusercontent.com/kubernetes/website/master/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 |

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

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

Replicationcontroller/nginx created

Check on the status of the replicationcontroller using this command:

Kubectl describe replicationcontrollers/nginx

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

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

20 s 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

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 that just gets 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. For general information about working with config 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-overview/#pod-templates). It has exactly the same schema as a [pod](https://kubernetes.io/docs/concepts/workloads/pods/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](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/admin/kubelet/) or Docker.

### 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 replication controllers:**

### 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" \l "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 go client library, you need to do the steps explicitly (scale replicas to 0, wait for pod deletions, then delete the replicationcontroller).

### Deleting just a replicationcontroller:

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

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

When using the REST API or go client library, simply 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, data recovery, etc. 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 makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply 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](http://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.

Rolling update is implemented in the client tool [kubectl rolling-update](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "rolling-update). Visit [kubectl rolling-update task](https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/) for more concrete examples.

### 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" \t "_blank), 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 replication controller:

The replicationcontroller simply 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](http://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](http://issue.k8s.io/492)), which would change its replicas field. We will not add scheduling policies (for example, [spreading](http://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](http://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, rolling-update) are proof-of-concept examples of this. For instance, we could imagine something like [Asgard](http://techblog.netflix.com/2012/06/asgard-web-based-cloud-management-and.html" \t "_blank) 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](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/" \l "replicationcontroller-v1-core).

**Alternatives to replication controller :**

[Replicaset](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) is the next-generation replicationcontroller that supports the new [set-based label selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#set-based-requirement). It’s mainly used by [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) is a higher-level API object that updates its underlying Replica Sets and their Pods in a similar fashion as kubectl rolling-update. Deployments are recommended if you want this rolling update functionality, because unlike kubectl rolling-update, 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 (for example, Kubelet or Docker).

### Job:

Use a [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/) instead of a replicationcontroller for pods that are expected to terminate on their own (that is, batch jobs).

### Daemonset:

Use a [daemonset](https://kubernetes.io/docs/concepts/workloads/controllers/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 provides declarative updates for [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/) and [replicasets](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/).

You describe a desired state in a Deployment, and the Deployment [Controller](https://kubernetes.io/docs/concepts/architecture/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.

**Note:** 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**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#use-case)
* [**Creating a Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment)
* [**Updating a Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment)
* [**Rolling Back a Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment)
* [**Scaling a Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment)
* [**Pausing and Resuming a Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#pausing-and-resuming-a-deployment)
* [**Deployment status**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#deployment-status)
* [**Clean up Policy**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#clean-up-policy)
* [**Canary Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#canary-deployment)
* [**Writing a Deployment Spec**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#writing-a-deployment-spec)
* [**Alternative to Deployments**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#alternative-to-deployments)

Top of Form

Use case:

The following are typical use cases for Deployments:

* [Create a Deployment to rollout a replicaset](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment). 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment) 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment) 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment).
* [Pause the Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#pausing-and-resuming-a-deployment) to apply multiple fixes to its podtemplatespec and then resume it to start a new rollout.
* [Use the status of the Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#deployment-status) as an indicator that a rollout has stuck.
* [Clean up older replicasets](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#clean-up-policy) 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:

| [Controllers/nginx-deployment.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/controllers/nginx-deployment.yaml) |
| --- |
| 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:1.7.9  Ports:  - containerport: 80 |

In this example:

* A Deployment named nginx-deployment is created, indicated by the .metadata.name field.
* The Deployment creates three replicated Pods, indicated by the replicas field.
* The selector field defines how the Deployment finds which Pods to manage. In this case, you simply 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.

**Note:** The 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 template field contains the following sub-fields:
  + The Pods are labeled app: nginxusing the labels field.
  + The Pod template’s specification, or .template.spec field, indicates that the Pods run one container, nginx, which runs the nginx [Docker Hub](https://hub.docker.com/) image at version 1.7.9.
  + Create one container and name it nginx using the name field.

Follow the steps given below to create the above Deployment:

Before you begin, make sure your Kubernetes cluster is up and running.

1. Create the Deployment by running the following command:

**Note:** You may specify the –record flag to write the command executed in the resource annotation kubernetes.io/change-cause. It is useful for future introspection. For example, to see the commands executed in each Deployment revision.

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:
2. 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 cluster.
  + DESIRED displays the desired number of replicas of the application, which you define when you create the Deployment. This is the desired state.
  + CURRENT displays how many replicas are currently running.
  + 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.v1.apps/nginx-deployment. The output is similar to this:
2. Waiting **for** rollout to finish: 2 out of 3 new replicas have been updated...

Deployment.apps/nginx-deployment successfully rolled out

1. Run the kubectl get deployments again a few seconds later. The output is similar to this:
2. NAME READY UP-TO-DATE AVAILABLE AGE

Nginx-deployment 3/3 3 3 18s

Notice that the Deployment has created all three replicas, and all replicas are up-to-date (they contain the latest Pod template) and available.

1. To see the replicaset (rs) created by the Deployment, run kubectl get rs. The output is similar to this:
2. NAME DESIRED CURRENT READY AGE

Nginx-deployment-75675f5897 3 3 3 18s

Notice that the name of the replicaset is always formatted as [DEPLOYMENT-NAME]-[RANDOM-STRING]. The random string is randomly generated and uses the pod-template-hash as a seed.

1. To see the labels automatically generated for each Pod, run kubectl get pods --show-labels. The following output is returned:
2. NAME READY STATUS RESTARTS AGE LABELS
3. Nginx-deployment-75675f5897-7ci7o 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453
4. Nginx-deployment-75675f5897-kzszj 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453

Nginx-deployment-75675f5897-qqcnn 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453

The created replicaset ensures that there are three nginx Pods.

**Note:** You must specify an appropriate selector and Pod template labels in a Deployment (in this case, app: nginx). Do not overlap labels or selectors with other controllers (including other Deployments and statefulsets). Kubernetes doesn’t stop you from overlapping, and if multiple controllers have overlapping selectors those controllers might conflict and behave unexpectedly.

### Pod-template-hash label:

The pod-template-hash label is added by the Deployment controller to every replicaset that a Deployment creates or adopts.

This label ensures that child replicasets of a Deployment do not overlap. It is generated by hashing the podtemplate of the replicaset and using the resulting hash as the label value that is added to the replicaset selector, Pod template labels, and in any existing Pods that the replicaset might have.

Updating a deployment:

**Note:** A Deployment’s rollout is triggered if and only if the Deployment’s Pod template (that is, .spec.template) is changed, for example if the labels or container images of the template are updated. Other updates, such as scaling the Deployment, do not trigger a rollout.

Follow the steps given below to update your Deployment:

1. Let’s update the nginx Pods to use the nginx:1.9.1 image instead of the nginx:1.7.9 image.

Kubectl --record deployment.apps/nginx-deployment set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1

Or simply use the following command:

Kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1 --record

The output is similar to this:

Deployment.apps/nginx-deployment image updated

Alternatively, you can edit the Deployment and change .spec.template.spec.containers[0].image from nginx:1.7.9 to nginx:1.9.1:

Kubectl edit deployment.v1.apps/nginx-deployment

The output is similar to this:

Deployment.apps/nginx-deployment edited

1. To see the rollout status, run:

Kubectl rollout status deployment.v1.apps/nginx-deployment

The output is similar to this:

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

Or

Deployment.apps/nginx-deployment successfully rolled out

Get more details on your updated Deployment:

* After the rollout succeeds, you can view the Deployment by running kubectl get deployments. The output is similar to this:
* NAME READY UP-TO-DATE AVAILABLE AGE
* Nginx-deployment 3/3 3 3 36s
* Run kubectl get rs to see that the Deployment updated the Pods by creating a new replicaset and scaling it up to 3 replicas, as well as scaling down the old replicaset to 0 replicas.

Kubectl get rs

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-deployment-1564180365 3 3 3 6s

Nginx-deployment-2035384211 0 0 0 36s

* Running get pods should now show only the new Pods:

Kubectl get pods

The output is similar to this:

NAME READY STATUS RESTARTS AGE

Nginx-deployment-1564180365-khku8 1/1 Running 0 14s

Nginx-deployment-1564180365-nacti 1/1 Running 0 14s

Nginx-deployment-1564180365-z9gth 1/1 Running 0 14s

Next time you want to update these Pods, you only need to update the Deployment’s Pod template again.

Deployment ensures that only a certain number of Pods are down while they are being updated. By default, it ensures that at least 75% of the desired number of Pods are up (25% max unavailable).

Deployment also ensures that only a certain number of Pods are created above the desired number of Pods. By default, it ensures that at most 125% of the desired number of Pods are up (25% max surge).

For example, if you look at the above Deployment closely, you will see that it first created a new Pod, then deleted some old Pods, and created new ones. It does not kill old Pods until a sufficient number of new Pods have come up, and does not create new Pods until a sufficient number of old Pods have been killed. It makes sure that at least 2 Pods are available and that at max 4 Pods in total are available.

* Get details of your Deployment:

Kubectl describe deployments

The output is similar to this:

Name: nginx-deployment

Namespace: default

Creationtimestamp: Thu, 30 Nov 2017 10:56:25 +0000

Labels: app=nginx

Annotations: deployment.kubernetes.io/revision=2

Selector: app=nginx

Replicas: 3 desired | 3 updated | 3 total | 3 available | 0 unavailable

Strategytype: rollingupdate

Minreadyseconds: 0

Rollingupdatestrategy: 25% max unavailable, 25% max surge

Pod Template:

Labels: app=nginx

Containers:

Nginx:

Image: nginx:1.9.1

Port: 80/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Conditions:

Type Status Reason

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

Available True minimumreplicasavailable

Progressing True newreplicasetavailable

Oldreplicasets: <none>

Newreplicaset: nginx-deployment-1564180365 (3/3 replicas created)

Events:

Type Reason Age From Message

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

Normal scalingreplicaset 2m deployment-controller Scaled up replica set nginx-deployment-2035384211 to 3

Normal scalingreplicaset 24s deployment-controller Scaled up replica set nginx-deployment-1564180365 to 1

Normal scalingreplicaset 22s deployment-controller Scaled down replica set nginx-deployment-2035384211 to 2

Normal scalingreplicaset 22s deployment-controller Scaled up replica set nginx-deployment-1564180365 to 2

Normal scalingreplicaset 19s deployment-controller Scaled down replica set nginx-deployment-2035384211 to 1

Normal scalingreplicaset 19s deployment-controller Scaled up replica set nginx-deployment-1564180365 to 3

Normal scalingreplicaset 14s deployment-controller Scaled down replica set nginx-deployment-2035384211 to 0

Here you see that when you first created the Deployment, it created a replicaset (nginx-deployment-2035384211) and scaled it up to 3 replicas directly. When you updated the Deployment, it created a new replicaset (nginx-deployment-1564180365) and scaled it up to 1 and then scaled down the old replicaset to 2, so that at least 2 Pods were available and at most 4 Pods were created at all times. It then continued scaling up and down the new and the old replicaset, with the same rolling update strategy. Finally, you’ll have 3 available replicas in the new replicaset, and the old replicaset is scaled down to 0.

### Rollover (aka multiple updates in-flight):

Each time a new Deployment is observed by the Deployment controller, a replicaset is created to bring up the desired Pods. If the Deployment is updated, the existing replicaset that controls Pods whose labels match .spec.selector but whose template does not match .spec.template are scaled down. Eventually, the new replicaset is scaled to .spec.replicas and all old replicasets is scaled to 0.

If you update a Deployment while an existing rollout is in progress, the Deployment creates a new replicaset as per the update and start scaling that up, and rolls over the replicaset that it was scaling up previously – it will add it to its list of old replicasets and start scaling it down.

For example, suppose you create a Deployment to create 5 replicas of nginx:1.7.9, but then update the Deployment to create 5 replicas of nginx:1.9.1, when only 3 replicas of nginx:1.7.9 had been created. In that case, the Deployment immediately starts killing the 3 nginx:1.7.9 Pods that it had created, and starts creating nginx:1.9.1 Pods. It does not wait for the 5 replicas of nginx:1.7.9 to be created before changing course.

### Label selector updates:

It is generally discouraged to make label selector updates and it is suggested to plan your selectors up front. In any case, if you need to perform a label selector update, exercise great caution and make sure you have grasped all of the implications.

**Note:** In API version apps/v1, a Deployment’s label selector is immutable after it gets created.

* Selector additions require the Pod template labels in the Deployment spec to be updated with the new label too, otherwise a validation error is returned. This change is a non-overlapping one, meaning that the new selector does not select replicasets and Pods created with the old selector, resulting in orphaning all old replicasets and creating a new replicaset.
* Selector updates changes the existing value in a selector key – result in the same behavior as additions.
* Selector removals removes an existing key from the Deployment selector – do not require any changes in the Pod template labels. Existing replicasets are not orphaned, and a new replicaset is not created, but note that the removed label still exists in any existing Pods and replicasets.

Rolling back a deployment:

Sometimes, you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping. By default, all of the Deployment’s rollout history is kept in the system so that you can rollback anytime you want (you can change that by modifying revision history limit).

**Note:** A Deployment’s revision is created when a Deployment’s rollout is triggered. This means that the new revision is created if and only if the Deployment’s Pod template (.spec.template) is changed, for example if you update the labels or container images of the template. Other updates, such as scaling the Deployment, do not create a Deployment revision, so that you can facilitate simultaneous manual- or auto-scaling. This means that when you roll back to an earlier revision, only the Deployment’s Pod template part is rolled back.

* Suppose that you made a typo while updating the Deployment, by putting the image name as nginx:1.91 instead of nginx:1.9.1:

Kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.91 --record=true

The output is similar to this:

Deployment.apps/nginx-deployment image updated

* The rollout gets stuck. You can verify it by checking the rollout status:

Kubectl rollout status deployment.v1.apps/nginx-deployment

The output is similar to this:

Waiting for rollout to finish: 1 out of 3 new replicas have been updated...

* Press Ctrl-C to stop the above rollout status watch. For more information on stuck rollouts, [read more here](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#deployment-status).
* You see that the number of old replicas (nginx-deployment-1564180365 and nginx-deployment-2035384211) is 2, and new replicas (nginx-deployment-3066724191) is 1.

Kubectl get rs

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-deployment-1564180365 3 3 3 25s

Nginx-deployment-2035384211 0 0 0 36s

Nginx-deployment-3066724191 1 1 0 6s

* Looking at the Pods created, you see that 1 Pod created by new replicaset is stuck in an image pull loop.

Kubectl get pods

The output is similar to this:

NAME READY STATUS RESTARTS AGE

Nginx-deployment-1564180365-70iae 1/1 Running 0 25s

Nginx-deployment-1564180365-jbqqo 1/1 Running 0 25s

Nginx-deployment-1564180365-hysrc 1/1 Running 0 25s

Nginx-deployment-3066724191-08mng 0/1 imagepullbackoff 0 6s

**Note:** The Deployment controller stops the bad rollout automatically, and stops scaling up the new replicaset. This depends on the rollingupdate parameters (maxunavailable specifically) that you have specified. Kubernetes by default sets the value to 25%.

* Get the description of the Deployment:

Kubectl describe deployment

The output is similar to this:

Name: nginx-deployment

Namespace: default

Creationtimestamp: Tue, 15 Mar 2016 14:48:04 -0700

Labels: app=nginx

Selector: app=nginx

Replicas: 3 desired | 1 updated | 4 total | 3 available | 1 unavailable

Strategytype: rollingupdate

Minreadyseconds: 0

Rollingupdatestrategy: 25% max unavailable, 25% max surge

Pod Template:

Labels: app=nginx

Containers:

Nginx:

Image: nginx:1.91

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Conditions:

Type Status Reason

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

Available True minimumreplicasavailable

Progressing True replicasetupdated

Oldreplicasets: nginx-deployment-1564180365 (3/3 replicas created)

Newreplicaset: nginx-deployment-3066724191 (1/1 replicas created)

Events:

Firstseen lastseen Count From subobjectpath Type Reason Message

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

1m 1m 1 {deployment-controller } Normal scalingreplicaset Scaled up replica set nginx-deployment-2035384211 to 3

22s 22s 1 {deployment-controller } Normal scalingreplicaset Scaled up replica set nginx-deployment-1564180365 to 1

22s 22s 1 {deployment-controller } Normal scalingreplicaset Scaled down replica set nginx-deployment-2035384211 to 2

22s 22s 1 {deployment-controller } Normal scalingreplicaset Scaled up replica set nginx-deployment-1564180365 to 2

21s 21s 1 {deployment-controller } Normal scalingreplicaset Scaled down replica set nginx-deployment-2035384211 to 1

21s 21s 1 {deployment-controller } Normal scalingreplicaset Scaled up replica set nginx-deployment-1564180365 to 3

13s 13s 1 {deployment-controller } Normal scalingreplicaset Scaled down replica set nginx-deployment-2035384211 to 0

13s 13s 1 {deployment-controller } Normal scalingreplicaset Scaled up replica set nginx-deployment-3066724191 to 1

To fix this, you need to rollback to a previous revision of Deployment that is stable.

### Checking Rollout History of a Deployment:

Follow the steps given below to check the rollout history:

1. First, check the revisions of this Deployment:

Kubectl rollout history deployment.v1.apps/nginx-deployment

The output is similar to this:

Deployments "nginx-deployment"

REVISION CHANGE-CAUSE

1 kubectl apply --filename=https://k8s.io/examples/controllers/nginx-deployment.yaml --record=true

2 kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true

3 kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.91 --record=true

CHANGE-CAUSE is copied from the Deployment annotation kubernetes.io/change-cause to its revisions upon creation. You can specify thechange-CAUSE message by:

* + Annotating the Deployment with kubectl annotate deployment.v1.apps/nginx-deployment kubernetes.io/change-cause="image updated to 1.9.1"
  + Append the --record flag to save the kubectl command that is making changes to the resource.
  + Manually editing the manifest of the resource.

1. To see the details of each revision, run:

Kubectl rollout history deployment.v1.apps/nginx-deployment --revision=2

The output is similar to this:

Deployments "nginx-deployment" revision 2

Labels: app=nginx

Pod-template-hash=1159050644

Annotations: kubernetes.io/change-cause=kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true

Containers:

Nginx:

Image: nginx:1.9.1

Port: 80/TCP

Qos Tier:

Cpu: besteffort

Memory: besteffort

Environment Variables: <none>

No volumes.

### Rolling Back to a Previous Revision:

Follow the steps given below to rollback the Deployment from the current version to the previous version, which is version 2.

1. Now you’ve decided to undo the current rollout and rollback to the previous revision:

Kubectl rollout undo deployment.v1.apps/nginx-deployment

The output is similar to this:

Deployment.apps/nginx-deployment

Alternatively, you can rollback to a specific revision by specifying it with --to-revision:

Kubectl rollout undo deployment.v1.apps/nginx-deployment --to-revision=2

The output is similar to this:

Deployment.apps/nginx-deployment

For more details about rollout related commands, read [kubectl rollout](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "rollout).

The Deployment is now rolled back to a previous stable revision. As you can see, a deploymentrollback event for rolling back to revision 2 is generated from Deployment controller.

1. Check if the rollback was successful and the Deployment is running as expected, run:

Kubectl get deployment nginx-deployment

The output is similar to this:

NAME READY UP-TO-DATE AVAILABLE AGE

Nginx-deployment 3/3 3 3 30m

1. Get the description of the Deployment:

Kubectl describe deployment nginx-deployment

The output is similar to this:

Name: nginx-deployment

Namespace: default

Creationtimestamp: Sun, 02 Sep 2018 18:17:55 -0500

Labels: app=nginx

Annotations: deployment.kubernetes.io/revision=4

Kubernetes.io/change-cause=kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true

Selector: app=nginx

Replicas: 3 desired | 3 updated | 3 total | 3 available | 0 unavailable

Strategytype: rollingupdate

Minreadyseconds: 0

Rollingupdatestrategy: 25% max unavailable, 25% max surge

Pod Template:

Labels: app=nginx

Containers:

Nginx:

Image: nginx:1.9.1

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Conditions:

Type Status Reason

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

Available True minimumreplicasavailable

Progressing True newreplicasetavailable

Oldreplicasets: <none>

Newreplicaset: nginx-deployment-c4747d96c (3/3 replicas created)

Events:

Type Reason Age From Message

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

Normal scalingreplicaset 12m deployment-controller Scaled up replica set nginx-deployment-75675f5897 to 3

Normal scalingreplicaset 11m deployment-controller Scaled up replica set nginx-deployment-c4747d96c to 1

Normal scalingreplicaset 11m deployment-controller Scaled down replica set nginx-deployment-75675f5897 to 2

Normal scalingreplicaset 11m deployment-controller Scaled up replica set nginx-deployment-c4747d96c to 2

Normal scalingreplicaset 11m deployment-controller Scaled down replica set nginx-deployment-75675f5897 to 1

Normal scalingreplicaset 11m deployment-controller Scaled up replica set nginx-deployment-c4747d96c to 3

Normal scalingreplicaset 11m deployment-controller Scaled down replica set nginx-deployment-75675f5897 to 0

Normal scalingreplicaset 11m deployment-controller Scaled up replica set nginx-deployment-595696685f to 1

Normal deploymentrollback 15s deployment-controller Rolled back deployment "nginx-deployment" to revision 2

Normal scalingreplicaset 15s deployment-controller Scaled down replica set nginx-deployment-595696685f to 0

Scaling a deployment:

You can scale a Deployment by using the following command:

Kubectl scale deployment.v1.apps/nginx-deployment --replicas=10

The output is similar to this:

Deployment.apps/nginx-deployment scaled

Assuming [horizontal Pod autoscaling](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/) is enabled in your cluster, you can setup an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.

Kubectl autoscale deployment.v1.apps/nginx-deployment --min=10 --max=15 --cpu-percent=80

The output is similar to this:

Deployment.apps/nginx-deployment scaled

### Proportional scaling:

Rollingupdate Deployments support running multiple versions of an application at the same time. When you or an autoscaler scales a rollingupdate Deployment that is in the middle of a rollout (either in progress or paused), the Deployment controller balances the additional replicas in the existing active replicasets (replicasets with Pods) in order to mitigate risk. This is called proportional scaling.

For example, you are running a Deployment with 10 replicas, [maxsurge](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/" \l "max-surge)=3, and [maxunavailable](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/" \l "max-unavailable)=2.

* Ensure that the 10 replicas in your Deployment are running.

Kubectl get deploy

The output is similar to this:

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

Nginx-deployment 10 10 10 10 50s

* You update to a new image which happens to be unresolvable from inside the cluster.

Kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:sometag

The output is similar to this:

Deployment.apps/nginx-deployment image updated

* The image update starts a new rollout with replicaset nginx-deployment-1989198191, but it’s blocked due to the maxunavailable requirement that you mentioned above. Check out the rollout status:

Kubectl get rs

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-deployment-1989198191 5 5 0 9s

Nginx-deployment-618515232 8 8 8 1m

* Then a new scaling request for the Deployment comes along. The autoscaler increments the Deployment replicas to 15. The Deployment controller needs to decide where to add these new 5 replicas. If you weren’t using proportional scaling, all 5 of them would be added in the new replicaset. With proportional scaling, you spread the additional replicas across all replicasets. Bigger proportions go to the replicasets with the most replicas and lower proportions go to replicasets with less replicas. Any leftovers are added to the replicaset with the most replicas. Replicasets with zero replicas are not scaled up.

In our example above, 3 replicas are added to the old replicaset and 2 replicas are added to the new replicaset. The rollout process should eventually move all replicas to the new replicaset, assuming the new replicas become healthy. To confirm this, run:

Kubectl get deploy

The output is similar to this:

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

Nginx-deployment 15 18 7 8 7m

The rollout status confirms how the replicas were added to each replicaset.

Kubectl get rs

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-deployment-1989198191 7 7 0 7m

Nginx-deployment-618515232 11 11 11 7m

Pausing and resuming a deployment:

You can pause a Deployment before triggering one or more updates and then resume it. This allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

* For example, with a Deployment that was just created: Get the Deployment details:

Kubectl get deploy

The output is similar to this:

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

Nginx 3 3 3 3 1m

Get the rollout status:

Kubectl get rs

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-2142116321 3 3 3 1m

* Pause by running the following command:

Kubectl rollout pause deployment.v1.apps/nginx-deployment

The output is similar to this:

Deployment.apps/nginx-deployment paused

* Then update the image of the Deployment:

Kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1

The output is similar to this:

Deployment.apps/nginx-deployment image updated

* Notice that no new rollout started:

Kubectl rollout history deployment.v1.apps/nginx-deployment

The output is similar to this:

Deployments "nginx"

REVISION CHANGE-CAUSE

1 <none>

* Get the rollout status to ensure that the Deployment is updates successfully:

Kubectl get rs

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-2142116321 3 3 3 2m

* You can make as many updates as you wish, for example, update the resources that will be used:

Kubectl set resources deployment.v1.apps/nginx-deployment -c=nginx --limits=cpu=200m,memory=512Mi

The output is similar to this:

Deployment.apps/nginx-deployment resource requirements updated

The initial state of the Deployment prior to pausing it will continue its function, but new updates to the Deployment will not have any effect as long as the Deployment is paused.

* Eventually, resume the Deployment and observe a new replicaset coming up with all the new updates:

Kubectl rollout resume deployment.v1.apps/nginx-deployment

The output is similar to this:

Deployment.apps/nginx-deployment resumed

* Watch the status of the rollout until it’s done.

Kubectl get rs -w

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-2142116321 2 2 2 2m

Nginx-3926361531 2 2 0 6s

Nginx-3926361531 2 2 1 18s

Nginx-2142116321 1 2 2 2m

Nginx-2142116321 1 2 2 2m

Nginx-3926361531 3 2 1 18s

Nginx-3926361531 3 2 1 18s

Nginx-2142116321 1 1 1 2m

Nginx-3926361531 3 3 1 18s

Nginx-3926361531 3 3 2 19s

Nginx-2142116321 0 1 1 2m

Nginx-2142116321 0 1 1 2m

Nginx-2142116321 0 0 0 2m

Nginx-3926361531 3 3 3 20s

* Get the status of the latest rollout:

Kubectl get rs

The output is similar to this:

NAME DESIRED CURRENT READY AGE

Nginx-2142116321 0 0 0 2m

Nginx-3926361531 3 3 3 28s

**Note:** You cannot rollback a paused Deployment until you resume it.

**Deployment status:**

A Deployment enters various states during its lifecycle. It can be [progressing](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#progressing-deployment) while rolling out a new replicaset, it can be [complete](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#complete-deployment), or it can [fail to progress](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#failed-deployment).

### Progressing Deployment:

Kubernetes marks a Deployment as progressing when one of the following tasks is performed:

* The Deployment creates a new replicaset.
* The Deployment is scaling up its newest replicaset.
* The Deployment is scaling down its older replicaset(s).
* New Pods become ready or available (ready for at least [minreadyseconds](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/" \l "min-ready-seconds)).
* You can monitor the progress for a Deployment by using kubectl rollout status.

### Complete Deployment:

Kubernetes marks a Deployment as complete when it has the following characteristics:

* All of the replicas associated with the Deployment have been updated to the latest version you’ve specified, meaning any updates you’ve requested have been completed.
* All of the replicas associated with the Deployment are available.
* No old replicas for the Deployment are running.

You can check if a Deployment has completed by using kubectl rollout status. If the rollout completed successfully, kubectl rollout status returns a zero exit code.

Kubectl rollout status deployment.v1.apps/nginx-deployment

The output is similar to this:

Waiting for rollout to finish: 2 of 3 updated replicas are available...

Deployment.apps/nginx-deployment successfully rolled out

$ echo $?

0

### Failed Deployment:

Your Deployment may get stuck trying to deploy its newest replicaset without ever completing. This can occur due to some of the following factors:

* Insufficient quota
* Readiness probe failures
* Image pull errors
* Insufficient permissions
* Limit ranges
* Application runtime misconfiguration

One way you can detect this condition is to specify a deadline parameter in your Deployment spec: ([.spec.progressdeadlineseconds](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#progress-deadline-seconds)). .spec.progressdeadlineseconds denotes the number of seconds the Deployment controller waits before indicating (in the Deployment status) that the Deployment progress has stalled.

The following kubectl command sets the spec with progressdeadlineseconds to make the controller report lack of progress for a Deployment after 10 minutes:

Kubectl patch deployment.v1.apps/nginx-deployment -p '{"spec":{"progressdeadlineseconds":600}}'

The output is similar to this:

Deployment.apps/nginx-deployment patched

Once the deadline has been exceeded, the Deployment controller adds a deploymentcondition with the following attributes to the Deployment’s .status.conditions:

* Type=Progressing
* Status=False
* Reason=progressdeadlineexceeded

See the [Kubernetes API conventions](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#typical-status-properties) for more information on status conditions.

**Note:** Kubernetes takes no action on a stalled Deployment other than to report a status condition with Reason=progressdeadlineexceeded. Higher level orchestrators can take advantage of it and act accordingly, for example, rollback the Deployment to its previous version.

**Note:** If you pause a Deployment, Kubernetes does not check progress against your specified deadline. You can safely pause a Deployment in the middle of a rollout and resume without triggering the condition for exceeding the deadline.

You may experience transient errors with your Deployments, either due to a low timeout that you have set or due to any other kind of error that can be treated as transient. For example, let’s suppose you have insufficient quota. If you describe the Deployment you will notice the following section:

Kubectl describe deployment nginx-deployment

The output is similar to this:

<...>

Conditions:

Type Status Reason

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

Available True minimumreplicasavailable

Progressing True replicasetupdated

Replicafailure True failedcreate

<...>

If you run kubectl get deployment nginx-deployment -o yaml, the Deployment status is similar to this:

Status:

Availablereplicas: 2

Conditions:

- lasttransitiontime: 2016-10-04T12:25:39Z

Lastupdatetime: 2016-10-04T12:25:39Z

Message: Replica set "nginx-deployment-4262182780" is progressing.

Reason: replicasetupdated

Status: "True"

Type: Progressing

- lasttransitiontime: 2016-10-04T12:25:42Z

Lastupdatetime: 2016-10-04T12:25:42Z

Message: Deployment has minimum availability.

Reason: minimumreplicasavailable

Status: "True"

Type: Available

- lasttransitiontime: 2016-10-04T12:25:39Z

Lastupdatetime: 2016-10-04T12:25:39Z

Message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden: exceeded quota:

Object-counts, requested: pods=1, used: pods=3, limited: pods=2'

Reason: failedcreate

Status: "True"

Type: replicafailure

Observedgeneration: 3

Replicas: 2

Unavailablereplicas: 2

Eventually, once the Deployment progress deadline is exceeded, Kubernetes updates the status and the reason for the Progressing condition:

Conditions:

Type Status Reason

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

Available True minimumreplicasavailable

Progressing False progressdeadlineexceeded

Replicafailure True failedcreate

You can address an issue of insufficient quota by scaling down your Deployment, by scaling down other controllers you may be running, or by increasing quota in your namespace. If you satisfy the quota conditions and the Deployment controller then completes the Deployment rollout, you’ll see the Deployment’s status update with a successful condition (Status=True and Reason=newreplicasetavailable).

Conditions:

Type Status Reason

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

Available True minimumreplicasavailable

Progressing True newreplicasetavailable

Type=Available with Status=True means that your Deployment has minimum availability. Minimum availability is dictated by the parameters specified in the deployment strategy. Type=Progressing with Status=True means that your Deployment is either in the middle of a rollout and it is progressing or that it has successfully completed its progress and the minimum required new replicas are available (see the Reason of the condition for the particulars - in our case Reason=newreplicasetavailable means that the Deployment is complete).

You can check if a Deployment has failed to progress by using kubectl rollout status. Kubectl rollout status returns a non-zero exit code if the Deployment has exceeded the progression deadline.

Kubectl rollout status deployment.v1.apps/nginx-deployment

The output is similar to this:

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

Error: deployment "nginx" exceeded its progress deadline

$ echo $?

1

### Operating on a failed deployment:

All actions that apply to a complete Deployment also apply to a failed Deployment. You can scale it up/down, roll back to a previous revision, or even pause it if you need to apply multiple tweaks in the Deployment Pod template.

Clean up policy:

You can set .spec.revisionhistorylimit field in a Deployment to specify how many old replicasets for this Deployment you want to retain. The rest will be garbage-collected in the background. By default, it is 10.

**Note:** Explicitly setting this field to 0, will result in cleaning up all the history of your Deployment thus that Deployment will not be able to roll back.

Canary deployment:

If you want to roll out releases to a subset of users or servers using the Deployment, you can create multiple Deployments, one for each release, following the canary pattern described in [managing resources](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments).

Writing a deployment spec:

As with all other Kubernetes configs, a Deployment needs apiversion, kind, and metadata fields. For general information about working with config files, see [deploying applications](https://kubernetes.io/docs/tutorials/stateless-application/run-stateless-application-deployment/), configuring containers, and [using kubectl to manage resources](https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/) documents.

A Deployment 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 and .spec.selector are the only required field of the .spec.

The .spec.template is a [Pod template](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates). It has exactly the same schema as a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/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 Deployment must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See [selector](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#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.

### Replicas:

.spec.replicas is an optional field that specifies the number of desired Pods. It defaults to 1.

### Selector:

.spec.selector is an required field that specifies a [label selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) for the Pods targeted by this Deployment.

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

In API version apps/v1, .spec.selector and .metadata.labels do not default to .spec.template.metadata.labels if not set. So they must be set explicitly. Also note that .spec.selector is immutable after creation of the Deployment in apps/v1.

A Deployment may terminate Pods whose labels match the selector if their template is different from .spec.template or if the total number of such Pods exceeds .spec.replicas. It brings up new Pods with .spec.template if the number of Pods is less than the desired number.

**Note:** You should not create other Pods whose labels match this selector, either directly, by creating another Deployment, or by creating another controller such as a replicaset or a replicationcontroller. If you do so, the first Deployment thinks that it created these other Pods. Kubernetes does not stop you from doing this.

If you have multiple controllers that have overlapping selectors, the controllers will fight with each other and won’t behave correctly.

### Strategy:

.spec.strategy specifies the strategy used to replace old Pods by new ones. .spec.strategy.type can be “Recreate” or “rollingupdate”. “rollingupdate” is the default value.

#### Recreate Deployment:

All existing Pods are killed before new ones are created when .spec.strategy.type==Recreate

.

#### Rolling Update Deployment:

The Deployment updates Pods in a [rolling update](https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/) fashion when .spec.strategy.type==rollingupdate. You can specify maxunavailable and maxsurge to control the rolling update process.

##### Max Unavailable

.spec.strategy.rollingupdate.maxunavailable is an optional field that specifies the maximum number of Pods that can be unavailable during the update process. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The absolute number is calculated from percentage by rounding down. The value cannot be 0 if .spec.strategy.rollingupdate.maxsurge is 0. The default value is 25%.

For example, when this value is set to 30%, the old replicaset can be scaled down to 70% of desired Pods immediately when the rolling update starts. Once new Pods are ready, old replicaset can be scaled down further, followed by scaling up the new replicaset, ensuring that the total number of Pods available at all times during the update is at least 70% of the desired Pods.

##### Max Surge

.spec.strategy.rollingupdate.maxsurge is an optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The value cannot be 0 if maxunavailable is 0. The absolute number is calculated from the percentage by rounding up. The default value is 25%.

For example, when this value is set to 30%, the new replicaset can be scaled up immediately when the rolling update starts, such that the total number of old and new Pods does not exceed 130% of desired Pods. Once old Pods have been killed, the new replicaset can be scaled up further, ensuring that the total number of Pods running at any time during the update is at most 130% of desired Pods.

### Progress Deadline Seconds:

.spec.progressdeadlineseconds is an optional field that specifies the number of seconds you want to wait for your Deployment to progress before the system reports back that the Deployment has [failed progressing](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#failed-deployment) - surfaced as a condition with Type=Progressing, Status=False. And Reason=progressdeadlineexceeded in the status of the resource. The Deployment controller will keep retrying the Deployment. In the future, once automatic rollback will be implemented, the Deployment controller will roll back a Deployment as soon as it observes such a condition.

If specified, this field needs to be greater than .spec.minreadyseconds.

### Min Ready Seconds:

.spec.minreadyseconds is an optional field that specifies the minimum number of seconds for which a newly created Pod should be ready without any of its containers crashing, for it to be considered available. This defaults to 0 (the Pod will be considered available as soon as it is ready). To learn more about when a Pod is considered ready, see [Container Probes](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes).

### Rollback To:

Field .spec.rollbackto has been deprecated in API versions extensions/v1beta1 and apps/v1beta1, and is no longer supported in API versions starting apps/v1beta2. Instead, kubectl rollout undo as introduced in [Rolling Back to a Previous Revision](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-to-a-previous-revision) should be used.

### Revision History Limit:

A Deployment’s revision history is stored in the replicasets it controls.

.spec.revisionhistorylimit is an optional field that specifies the number of old replicasets to retain to allow rollback. These old replicasets consume resources in etcd and crowd the output of kubectl get rs. The configuration of each Deployment revision is stored in its replicasets; therefore, once an old replicaset is deleted, you lose the ability to rollback to that revision of Deployment. By default, 10 old replicasets will be kept, however its ideal value depends on the frequency and stability of new Deployments.

More specifically, setting this field to zero means that all old replicasets with 0 replicas will be cleaned up. In this case, a new Deployment rollout cannot be undone, since its revision history is cleaned up.

### Paused:

.spec.paused is an optional boolean field for pausing and resuming a Deployment. The only difference between a paused Deployment and one that is not paused, is that any changes into the podtemplatespec of the paused Deployment will not trigger new rollouts as long as it is paused. A Deployment is not paused by default when it is created.

Alternative to deployments:

### Kubectl rolling-update:

[Kubectl rolling-update](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#rolling-update) updates Pods and replicationcontrollers in a similar fashion. But Deployments are recommended, since they are declarative, server side, and have additional features, such as rolling back to any previous revision even after the rolling update is done

**Statefulsets:**

Statefulset is the workload API object used to manage stateful applications.

Manages the deployment and scaling of a set of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/), and provides guarantees about the ordering and uniqueness of these Pods.

Like a [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), a statefulset manages Pods that are based on an identical container spec. Unlike a Deployment, a statefulset maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

* [**Using statefulsets**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#using-statefulsets)
* [**Limitations**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#limitations)
* [**Components**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#components)
* [**Pod Selector**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#pod-selector)
* [**Pod Identity**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#pod-identity)
* [**Deployment and**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#deployment-and-scaling-guarantees) [**Scaling Guarantees**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#deployment-and-scaling-guarantees)
* [**Update Strategies**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#update-strategies)
* [**What's next**](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#what-s-next)

Sing statefulnets:

Statefulsets are valuable for applications that require one or more of the following.

* Stable, unique network identifiers.
* Stable, persistent storage.
* Ordered, graceful deployment and scaling.
* Ordered, automated rolling updates.
  + In the above, stable is synonymous with persistence across Pod (re)scheduling. If an application doesn’t require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application using a workload object that provides a set of stateless replicas.  [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) or [replicaset](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) may be better suited to your stateless needs.

Limitations:

* The storage for a given Pod must either be provisioned by a [persistentvolume Provisioner](https://github.com/kubernetes/examples/tree/master/staging/persistent-volume-provisioning/README.md" \t "_blank) based on the requested storage class, or pre-provisioned by an admin.
* Deleting and/or scaling a statefulset down will not delete the volumes associated with the statefulset. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related statefulset resources.
* Statefulsets currently require a [Headless Service](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) to be responsible for the network identity of the Pods. You are responsible for creating this Service.
* Statefulsets do not provide any guarantees on the termination of pods when a statefulset is deleted. To achieve ordered and graceful termination of the pods in the statefulset, it is possible to scale the statefulset down to 0 prior to deletion.
* When using [Rolling Updates](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#rolling-updates) with the default [Pod Management Policy](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#pod-management-policies) (orderedready), it’s possible to get into a broken state that requires [manual intervention to repair](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#forced-rollback).

**Components:**

The example below demonstrates the components of a statefulset.

Apiversion: v1

Kind: Service

Metadata:

Name: nginx

Labels:

App: nginx

Spec:

Ports:

- port: 80

Name: web

Clusterip: None

Selector:

App: nginx

---

Apiversion: apps/v1

Kind: statefulset

Metadata:

Name: web

Spec:

Selector:

Matchlabels:

App: nginx *# has to match .spec.template.metadata.labels*

Servicename: "nginx"

Replicas: 3 *# by default is 1*

Template:

Metadata:

Labels:

App: nginx *# has to match .spec.selector.matchlabels*

Spec:

Terminationgraceperiodseconds: 10

Containers:

- name: nginx

Image: k8s.gcr.io/nginx-slim:0.8

Ports:

- containerport: 80

Name: web

Volumemounts:

- name: www

Mountpath: /usr/share/nginx/html

Volumeclaimtemplates:

- metadata:

Name: www

Spec:

Accessmodes: [ "readwriteonce" ]

Storageclassname: "my-storage-class"

Resources:

Requests:

Storage: 1Gi

In the above example:

* A Headless Service, named nginx, is used to control the network domain.
* The statefulset, named web, has a Spec that indicates that 3 replicas of the nginx container will be launched in unique Pods.
* The volumeclaimtemplates will provide stable storage using [persistentvolumes](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) provisioned by a persistentvolume Provisioner.

Pod selector:

You must set the .spec.selector field of a statefulset to match the labels of its .spec.template.metadata.labels. Prior to Kubernetes 1.8, the .spec.selector field was defaulted when omitted. In 1.8 and later versions, failing to specify a matching Pod Selector will result in a validation error during statefulset creation.

Pod identity:

Statefulset Pods have a unique identity that is comprised of an ordinal, a stable network identity, and stable storage. The identity sticks to the Pod, regardless of which node it’s (re)scheduled on.

### Ordinal Index:

For a statefulset with N replicas, each Pod in the statefulset will be assigned an integer ordinal, from 0 up through N-1, that is unique over the Set.

### Stable Network ID:

Each Pod in a statefulset derives its hostname from the name of the statefulset and the ordinal of the Pod. The pattern for the constructed hostname is $(statefulset name)-$(ordinal). The example above will create three Pods named web-0,web-1,web-2. A statefulset can use a [Headless Service](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) to control the domain of its Pods. The domain managed by this Service takes the form: $(service name).$(namespace).svc.cluster.local, where “cluster.local” is the cluster domain. As each Pod is created, it gets a matching DNS subdomain, taking the form: $(podname).$(governing service domain), where the governing service is defined by the servicename field on the statefulset.

As mentioned in the [limitations](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#limitations) section, you are responsible for creating the [Headless Service](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) responsible for the network identity of the pods.

Here are some examples of choices for Cluster Domain, Service name, statefulset name, and how that affects the DNS names for the statefulset’s Pods.

| Cluster Domain | Service (ns/name) | Statefulset (ns/name) | Statefulset Domain | Pod DNS | Pod Hostname |
| --- | --- | --- | --- | --- | --- |
| Cluster.local | Default/nginx | Default/web | Nginx.default.svc.cluster.local | Web-{0..N-1}.nginx.default.svc.cluster.local | Web-{0..N-1} |
| Cluster.local | Foo/nginx | Foo/web | Nginx.foo.svc.cluster.local | Web-{0..N-1}.nginx.foo.svc.cluster.local | Web-{0..N-1} |
| Kube.local | Foo/nginx | Foo/web | Nginx.foo.svc.kube.local | Web-{0..N-1}.nginx.foo.svc.kube.local | Web-{0..N-1} |

**Note:** Cluster Domain will be set to cluster.local unless [otherwise configured](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/#how-it-works).

### Stable Storage:

Kubernetes creates one [persistentvolume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) for each volumeclaimtemplate. In the nginx example above, each Pod will receive a single persistentvolume with a storageclass of my-storage-class and 1 Gib of provisioned storage. If no storageclass is specified, then the default storageclass will be used. When a Pod is (re)scheduled onto a node, its volumemounts mount the persistentvolumes associated with its persistentvolume Claims. Note that, the persistentvolumes associated with the Pods’ persistentvolume Claims are not deleted when the Pods, or statefulset are deleted. This must be done manually.

### Pod Name Label:

When the statefulset [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) creates a Pod, it adds a label, statefulset.kubernetes.io/pod-name, that is set to the name of the Pod. This label allows you to attach a Service to a specific Pod in the statefulset.

Deployment and scaling guarantees:

* For a statefulset with N replicas, when Pods are being deployed, they are created sequentially, in order from {0..N-1}.
* When Pods are being deleted, they are terminated in reverse order, from {N-1..0}.
* Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready.
* Before a Pod is terminated, all of its successors must be completely shutdown.
* The statefulset should not specify a pod.Spec.terminationgraceperiodseconds of 0. This practice is unsafe and strongly discouraged. For further explanation, please refer to [force deleting statefulset Pods](https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/)*.*

When the nginx example above is created, three Pods will be deployed in the order web-0, web-1, web-2. Web-1 will not be deployed before web-0 is [Running and Ready](https://kubernetes.io/docs/user-guide/pod-states/), and web-2 will not be deployed until web-1 is Running and Ready. If web-0 should fail, after web-1 is Running and Ready, but before web-2 is launched, web-2 will not be launched until web-0 is successfully relaunched and becomes Running and Ready.

If a user were to scale the deployed example by patching the statefulset such that replicas=1, web-2 would be terminated first. Web-1 would not be terminated until web-2 is fully shutdown and deleted. If web-0 were to fail after web-2 has been terminated and is completely shutdown, but prior to web-1’s termination, web-1 would not be terminated until web-0 is Running and Ready.

### Pod Management Policies*:*

In Kubernetes 1.7 and later, statefulset allows you to relax its ordering guarantees while preserving its uniqueness and identity guarantees via its .spec.podmanagementpolicy field.

#### Orderedready Pod Management:

Orderedready pod management is the default for statefulsets. It implements the behavior described [above](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#deployment-and-scaling-guarantees).

#### Parallel Pod Management:

Parallel pod management tells the statefulset controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod. This option only affects the behavior for scaling operations. Updates are not affected.

Update strategies:

In Kubernetes 1.7 and later, statefulset’s .spec.updatestrategy field allows you to configure and disable automated rolling updates for containers, labels, resource request/limits, and annotations for the Pods in a statefulset.

### On Delete:

The ondelete update strategy implements the legacy (1.6 and prior) behavior. When a statefulset’s .spec.updatestrategy.type is set to ondelete, the statefulset controller will not automatically update the Pods in a statefulset. Users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a statefulset’s .spec.template.

### Rolling Updates:

The rollingupdate update strategy implements automated, rolling update for the Pods in a statefulset. It is the default strategy when .spec.updatestrategy is left unspecified. When a statefulset’s .spec.updatestrategy.type is set to rollingupdate, the statefulset controller will delete and recreate each Pod in the statefulset. It will proceed in the same order as Pod termination (from the largest ordinal to the smallest), updating each Pod one at a time. It will wait until an updated Pod is Running and Ready prior to updating its predecessor.

#### Partitions:

The rollingupdate update strategy can be partitioned, by specifying a .spec.updatestrategy.rollingupdate.partition. If a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the statefulset’s .spec.template is updated. All Pods with an ordinal that is less than the partition will not be updated, and, even if they are deleted, they will be recreated at the previous version. If a statefulset’s .spec.updatestrategy.rollingupdate.partition is greater than its .spec.replicas, updates to its .spec.template will not be propagated to its Pods. In most cases you will not need to use a partition, but they are useful if you want to stage an update, roll out a canary, or perform a phased roll out.

#### Forced Rollback:

When using [Rolling Updates](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#rolling-updates) with the default [Pod Management Policy](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#pod-management-policies) (orderedready), it’s possible to get into a broken state that requires manual intervention to repair.

If you update the Pod template to a configuration that never becomes Running and Ready (for example, due to a bad binary or application-level configuration error), statefulset will stop the rollout and wait.

To a [known issue](https://github.com/kubernetes/kubernetes/issues/67250), statefulset will continue to wait for the broken Pod to become Ready (which never happens) before it will attempt to revert it back to the working configuration.

After reverting the template, you must also delete any Pods that statefulset had already attempted to run with the bad configuration. Statefulset will then begin to recreate the Pods using the reverted template.

**Daemonset**

D

A daemonset ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a daemonset will clean up the Pods it created.

Some typical uses of a daemonset are:

* Running a cluster storage daemon, such as glusterd, ceph, on each node.
* Running a logs collection daemon on every node, such as fluentd or logstash.
* Running a node monitoring daemon on every node, such as [Prometheus Node Exporter](https://github.com/prometheus/node_exporter), [Flowmill](https://github.com/Flowmill/flowmill-k8s/" \t "_blank), [Sysdig Agent](https://docs.sysdig.com/" \t "_blank), collectd, [Dynatrace oneagent](https://www.dynatrace.com/technologies/kubernetes-monitoring/), [appdynamics Agent](https://docs.appdynamics.com/display/CLOUD/Container+Visibility+with+Kubernetes" \t "_blank), [Datadog agent](https://docs.datadoghq.com/agent/kubernetes/daemonset_setup/), [New Relic agent](https://docs.newrelic.com/docs/integrations/kubernetes-integration/installation/kubernetes-installation-configuration), Ganglia gmond or [Instana Agent](https://www.instana.com/supported-integrations/kubernetes-monitoring/" \t "_blank).

In a simple case, one daemonset, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple daemonsets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

* [**Writing a DaemonSet Spec**](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/#writing-a-daemonset-spec)
* [**How Daemon Pods are Scheduled**](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/#how-daemon-pods-are-scheduled)
* [**Communicating with Daemon Pods**](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/#communicating-with-daemon-pods)
* [**Updating a DaemonSet**](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/#updating-a-daemonset)
* [**Alternatives to DaemonSet**](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/#alternatives-to-daemonset)

Writing a daemonset spec:

Creating daemonset:

You can describe a DaemonSet in a YAML file. For example, the daemonset.yaml file below describes a DaemonSet that runs the fluentd-elasticsearch Docker image:

|  |
| --- |
| apiVersion: apps/v1  kind: DaemonSet  metadata:  name: fluentd-elasticsearch  namespace: kube-system  labels:  k8s-app: fluentd-logging  spec:  selector:  matchLabels:  name: fluentd-elasticsearch  template:  metadata:  labels:  name: fluentd-elasticsearch  spec:  tolerations:  - key: node-role.kubernetes.io/master  effect: NoSchedule  containers:  - name: fluentd-elasticsearch  image: quay.io/fluentd\_elasticsearch/fluentd:v2.5.2  resources:  limits:  memory: 200Mi  requests:  cpu: 100m  memory: 200Mi  volumeMounts:  - name: varlog  mountPath: /var/log  - name: varlibdockercontainers  mountPath: /var/lib/docker/containers  readOnly: **true**  terminationGracePeriodSeconds: 30  volumes:  - name: varlog  hostPath:  path: /var/log  - name: varlibdockercontainers  hostPath:  path: /var/lib/docker/containers |

* Create a DaemonSet based on the YAML file:
* kubectl apply -f https://k8s.io/examples/controllers/daemonset.yaml

### Required Fields:

As with all other Kubernetes config, a DaemonSet needs apiVersion, kind, and metadata fields. For general information about working with config files, see [deploying applications](https://kubernetes.io/docs/user-guide/deploying-applications/), [configuring containers](https://kubernetes.io/docs/tasks/), and [object management using kubectl](https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/) documents.

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

### Pod Template:

The .spec.template is one of the required fields in .spec.

The .spec.template is a [pod template](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates). It has exactly the same schema as a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/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 DaemonSet has to specify appropriate labels (see [pod selector](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/#pod-selector)).

A Pod Template in a DaemonSet must have a [RestartPolicy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/" \l "restart-policy) equal to Always, or be unspecified, which defaults to Always.

### Pod Selector:

The .spec.selector field is a pod selector. It works the same as the .spec.selector of a [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/).

As of Kubernetes 1.8, you must specify a pod selector that matches the labels of the .spec.template. The pod selector will no longer be defaulted when left empty. Selector defaulting was not compatible with kubectl apply. Also, once a DaemonSet is created, its .spec.selector can not be mutated. Mutating the pod selector can lead to the unintentional orphaning of Pods, and it was found to be confusing to users.

The .spec.selector is an object consisting of two fields:

* matchLabels - works the same as the .spec.selector of a [ReplicationController](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/).
* matchExpressions - allows to build more sophisticated selectors by specifying key, list of values and an operator that relates the key and values.

When the two are specified the result is ANDed.

If the .spec.selector is specified, it must match the .spec.template.metadata.labels. Config with these not matching will be rejected by the API.

Also you should not normally create any Pods whose labels match this selector, either directly, via another DaemonSet, or via another workload resource such as ReplicaSet. Otherwise, the DaemonSet [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) will think that those Pods were created by it. Kubernetes will not stop you from doing this. One case where you might want to do this is manually create a Pod with a different value on a node for testing.

### Running Pods on Only Some Nodes:

If you specify a .spec.template.spec.nodeSelector, then the DaemonSet controller will create Pods on nodes which match that [node selector](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/). Likewise if you specify a .spec.template.spec.affinity, then DaemonSet controller will create Pods on nodes which match that [node affinity](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/). If you do not specify either, then the DaemonSet controller will create Pods on all nodes.

**How daemon pods are scheduled**

### Scheduled by default scheduler:

A DaemonSet ensures that all eligible nodes run a copy of a Pod. Normally, the node that a Pod runs on is selected by the Kubernetes scheduler. However, DaemonSet pods are created and scheduled by the DaemonSet controller instead. That introduces the following issues:

* Inconsistent Pod behavior: Normal Pods waiting to be scheduled are created and in Pending state, but DaemonSet pods are not created in Pending state. This is confusing to the user.
* [Pod preemption](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/) is handled by default scheduler. When preemption is enabled, the DaemonSet controller will make scheduling decisions without considering pod priority and preemption.

ScheduleDaemonSetPods allows you to schedule DaemonSets using the default scheduler instead of the DaemonSet controller, by adding the NodeAffinity term to the DaemonSet pods, instead of the .spec.nodeName term. The default scheduler is then used to bind the pod to the target host. If node affinity of the DaemonSet pod already exists, it is replaced. The DaemonSet controller only performs these operations when creating or modifying DaemonSet pods, and no changes are made to the spec.template of the DaemonSet.

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchFields:

- key: metadata.name

operator: In

values:

- target-host-name

In addition, node.kubernetes.io/unschedulable:NoSchedule toleration is added automatically to DaemonSet Pods. The default scheduler ignores unschedulable Nodes when scheduling DaemonSet Pods.

### Taints and Tolerations:

Although Daemon Pods respect [taints and tolerations](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration), the following tolerations are added to DaemonSet Pods automatically according to the related features.

| Toleration Key | Effect | Version | Description |
| --- | --- | --- | --- |
| node.kubernetes.io/not-ready | NoExecute | 1.13+ | DaemonSet pods will not be evicted when there are node problems such as a network partition. |
| node.kubernetes.io/unreachable | NoExecute | 1.13+ | DaemonSet pods will not be evicted when there are node problems such as a network partition. |
| node.kubernetes.io/disk-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/memory-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/unschedulable | NoSchedule | 1.12+ | DaemonSet pods tolerate unschedulable attributes by default scheduler. |
| node.kubernetes.io/network-unavailable | NoSchedule | 1.12+ | DaemonSet pods, who uses host network, tolerate network-unavailable attributes by default scheduler. |

**Communicating with daemon pods**:

Some possible patterns for communicating with Pods in a DaemonSet are:

* **Push**: Pods in the DaemonSet are configured to send updates to another service, such as a stats database. They do not have clients.
* **NodeIP and Known Port**: Pods in the DaemonSet can use a hostPort, so that the pods are reachable via the node IPs. Clients know the list of node IPs somehow, and know the port by convention.
* **DNS**: Create a [headless service](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) with the same pod selector, and then discover DaemonSets using the endpoints resource or retrieve multiple A records from DNS.
* **Service**: Create a service with the same Pod selector, and use the service to reach a daemon on a random node. (No way to reach specific node.)

**Updating a daemonset:**

If node labels are changed, the DaemonSet will promptly add Pods to newly matching nodes and delete Pods from newly not-matching nodes.

You can modify the Pods that a DaemonSet creates. However, Pods do not allow all fields to be updated. Also, the DaemonSet controller will use the original template the next time a node (even with the same name) is created.

You can delete a DaemonSet. If you specify --cascade=false with kubectl, then the Pods will be left on the nodes. If you subsequently create a new DaemonSet with the same selector, the new DaemonSet adopts the existing Pods. If any Pods need replacing the DaemonSet replaces them according to its updateStrategy.

You can [perform a rolling update](https://kubernetes.io/docs/tasks/manage-daemon/update-daemon-set/) on a DaemonSet.

**Alternatives to daemonset:**

### Init Scripts:

It is certainly possible to run daemon processes by directly starting them on a node (e.g. using init, upstartd, or systemd). This is perfectly fine. However, there are several advantages to running such processes via a DaemonSet:

* Ability to monitor and manage logs for daemons in the same way as applications.
* Same config language and tools (e.g. Pod templates, kubectl) for daemons and applications.
* Running daemons in containers with resource limits increases isolation between daemons from app containers. However, this can also be accomplished by running the daemons in a container but not in a Pod (e.g. start directly via Docker).

### Bare Pods:

It is possible to create Pods directly which specify a particular node to run on. However, a DaemonSet 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, you should use a DaemonSet rather than creating individual Pods.

### Static Pods:

It is possible to create Pods by writing a file to a certain directory watched by Kubelet. These are called [static pods](https://kubernetes.io/docs/concepts/cluster-administration/static-pod/). Unlike DaemonSet, static Pods cannot be managed with kubectl or other Kubernetes API clients. Static Pods do not depend on the apiserver, making them useful in cluster bootstrapping cases. Also, static Pods may be deprecated in the future.

### Deployments:

DaemonSets are similar to [Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) in that they both create Pods, and those Pods have processes which are not expected to terminate (e.g. web servers, storage servers).

Use a Deployment for stateless services, like frontends, where scaling up and down the number of replicas and rolling out updates are more important than controlling exactly which host the Pod runs on. Use a DaemonSet when it is important that a copy of a Pod always run on all or certain hosts, and when it needs to start before other Pods.

**Garbage collection**

The role of the Kubernetes garbage collector is to delete certain objects that once had an owner, but no longer have an owner.

* [**Owners and dependents**](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/#owners-and-dependents)
* [**Controlling how the garbage collector deletes dependents**](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/#controlling-how-the-garbage-collector-deletes-dependents)
* [**Known issues**](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/#known-issues)
* [**What's next**](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/#what-s-next)

Owners dependents:

Some Kubernetes objects are owners of other objects. For example, a ReplicaSet is the owner of a set of Pods. The owned objects are called dependents of the owner object. Every dependent object has a metadata.ownerReferences field that points to the owning object.

Sometimes, Kubernetes sets the value of ownerReference automatically. For example, when you create a ReplicaSet, Kubernetes automatically sets the ownerReference field of each Pod in the ReplicaSet. In 1.8, Kubernetes automatically sets the value of ownerReference for objects created or adopted by ReplicationController, ReplicaSet, StatefulSet, DaemonSet, Deployment, Job and CronJob.

You can also specify relationships between owners and dependents by manually setting the ownerReference field.

Here’s a configuration file for a ReplicaSet that has three Pods:

| [controllers/replicaset.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/controllers/replicaset.yaml) |
| --- |
| apiVersion: apps/v1  kind: ReplicaSet  metadata:  name: my-repset  spec:  replicas: 3  selector:  matchLabels:  pod-is-for: garbage-collection-example  template:  metadata:  labels:  pod-is-for: garbage-collection-example  spec:  containers:  - name: nginx  image: nginx |

If you create the ReplicaSet and then view the Pod metadata, you can see OwnerReferences field:

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

kubectl get pods --output=yaml

The output shows that the Pod owner is a ReplicaSet named my-repset:

apiVersion: v1

kind: Pod

metadata:

...

ownerReferences:

- apiVersion: apps/v1

controller: **true**

blockOwnerDeletion: **true**

kind: ReplicaSet

name: my-repset

uid: d9607e19-f88f-11e6-a518-42010a800195

...

**Note:** Cross-namespace owner references are disallowed by design. This means: 1) Namespace-scoped dependents can only specify owners in the same namespace, and owners that are cluster-scoped. 2) Cluster-scoped dependents can only specify cluster-scoped owners, but not namespace-scoped owners.

Controlling how the garbage collector deletes dependents:

When you delete an object, you can specify whether the object’s dependents are also deleted automatically. Deleting dependents automatically is called cascading deletion. There are two modes of cascading deletion: background and foreground.

If you delete an object without deleting its dependents automatically, the dependents are said to be orphaned.

### Foreground cascading deletion:

In foreground cascading deletion, the root object first enters a “deletion in progress” state. In the “deletion in progress” state, the following things are true:

* The object is still visible via the REST API
* The object’s deletionTimestamp is set
* The object’s metadata.finalizers contains the value “foregroundDeletion”.

Once the “deletion in progress” state is set, the garbage collector deletes the object’s dependents. Once the garbage collector has deleted all “blocking” dependents (objects with ownerReference.blockOwnerDeletion=true), it deletes the owner object.

Note that in the “foregroundDeletion”, only dependents with ownerReference.blockOwnerDeletion=true block the deletion of the owner object. Kubernetes version 1.7 added an [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#ownerreferencespermissionenforcement) that controls user access to set blockOwnerDeletion to true based on delete permissions on the owner object, so that unauthorized dependents cannot delay deletion of an owner object.

If an object’s ownerReferences field is set by a controller (such as Deployment or ReplicaSet), blockOwnerDeletion is set automatically and you do not need to manually modify this field.

### Background cascading deletion:

In background cascading deletion, Kubernetes deletes the owner object immediately and the garbage collector then deletes the dependents in the background.

### Setting the cascading deletion policy:

To control the cascading deletion policy, set the propagationPolicy field on the deleteOptions argument when deleting an Object. Possible values include “Orphan”, “Foreground”, or “Background”.

Prior to Kubernetes 1.9, the default garbage collection policy for many controller resources was orphan. This included ReplicationController, ReplicaSet, StatefulSet, DaemonSet, and Deployment. For kinds in the extensions/v1beta1, apps/v1beta1, and apps/v1beta2 group versions, unless you specify otherwise, dependent objects are orphaned by default. In Kubernetes 1.9, for all kinds in the apps/v1 group version, dependent objects are deleted by default.

Here’s an example that deletes dependents in background:

kubectl proxy --port=8080

curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/replicasets/my-repset **\**

-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Background"}' **\**

-H "Content-Type: application/json"

Here’s an example that deletes dependents in foreground:

kubectl proxy --port=8080

curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/replicasets/my-repset **\**

-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' **\**

-H "Content-Type: application/json"

Here’s an example that orphans dependents:

kubectl proxy --port=8080

curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/replicasets/my-repset **\**

-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' **\**

-H "Content-Type: application/json"

kubectl also supports cascading deletion. To delete dependents automatically using kubectl, set --cascade to true. To orphan dependents, set --cascade to false. The default value for --cascade is true.

Here’s an example that orphans the dependents of a ReplicaSet:

kubectl delete replicaset my-repset --cascade=false

### Additional note on Deployments:

Prior to 1.7, When using cascading deletes with Deployments you must use propagationPolicy: Foreground to delete not only the ReplicaSets created, but also their Pods. If this type of propagationPolicy is not used, only the ReplicaSets will be deleted, and the Pods will be orphaned. See [kubeadm/#149](https://github.com/kubernetes/kubeadm/issues/149" \l "issuecomment-284766613" \t "_blank) for more information.

TTL Controller for finished resources:

The TTL controller provides a TTL (time to live) mechanism to limit the lifetime of resource objects that have finished execution. TTL controller only handles [Jobs](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/) for now, and may be expanded to handle other resources that will finish execution, such as Pods and custom resources.

Alpha Disclaimer: this feature is currently alpha, and can be enabled with both kube-apiserver and kube-controller-manager [feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) TTLAfterFinished.

* [**TTL Controller**](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/#ttl-controller)
* [**Caveat**](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/#caveat)
* [**What's next**](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/#what-s-next)

TTL controller:

The TTL controller only supports Jobs for now. A cluster operator can use this feature to clean up finished Jobs (either Complete or Failed) automatically by specifying the .spec.ttlSecondsAfterFinished field of a Job, as in this [example](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#clean-up-finished-jobs-automatically). The TTL controller will assume that a resource is eligible to be cleaned up TTL seconds after the resource has finished, in other words, when the TTL has expired. When the TTL controller cleans up a resource, it will delete it cascadingly, i.e. delete its dependent objects together with it. Note that when the resource is deleted, its lifecycle guarantees, such as finalizers, will be honored.

The TTL seconds can be set at any time. Here are some examples for setting the .spec.ttlSecondsAfterFinished field of a Job:

* Specify this field in the resource manifest, so that a Job can be cleaned up automatically some time after it finishes.
* Set this field of existing, already finished resources, to adopt this new feature.
* Use a [mutating admission webhook](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#admission-webhooks) to set this field dynamically at resource creation time. Cluster administrators can use this to enforce a TTL policy for finished resources.
* Use a [mutating admission webhook](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#admission-webhooks) to set this field dynamically after the resource has finished, and choose different TTL values based on resource status, labels, etc.

### Cavent

### Updating TTL Seconds:

Note that the TTL period, e.g. .spec.ttlSecondsAfterFinished field of Jobs, can be modified after the resource is created or has finished. However, once the Job becomes eligible to be deleted (when the TTL has expired), the system won’t guarantee that the Jobs will be kept, even if an update to extend the TTL returns a successful API response.

### Time Skew:

Because TTL controller uses timestamps stored in the Kubernetes resources to determine whether the TTL has expired or not, this feature is sensitive to time skew in the cluster, which may cause TTL controller to clean up resource objects at the wrong time.

In Kubernetes, it’s required to run NTP on all nodes (see [#6159](https://github.com/kubernetes/kubernetes/issues/6159#issuecomment-93844058)) to avoid time skew. Clocks aren’t always correct, but the difference should be very small. Please be aware of this risk when setting a non-zero TTL.

**Jobs –Run Ro Completion:**

A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

You can also use a Job to run multiple Pods in parallel.

* [**Running an example Job**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#running-an-example-job)
* [**Writing a Job Spec**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#writing-a-job-spec)
* [**Handling Pod and Container Failures**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#handling-pod-and-container-failures)
* [**Job Termination and Cleanup**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#job-termination-and-cleanup)
* [**Clean Up Finished Jobs Automatically**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#clean-up-finished-jobs-automatically)
* [**Job Patterns**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#job-patterns)
* [**Advanced Usage**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#advanced-usage)
* [**Alternatives**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#alternatives)
* [**Cron Jobs**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#cron-jobs)

**Running an example of job:**

Here is an example Job config. It computes π to 2000 places and prints it out. It takes around 10s to complete.

| [controllers/job.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/controllers/job.yaml) |
| --- |
| apiVersion: batch/v1  kind: Job  metadata:  name: pi  spec:  template:  spec:  containers:  - name: pi  image: perl  command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]  restartPolicy: Never  backoffLimit: 4 |

You can run the example with this command:

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

job.batch/pi created

Check on the status of the Job with kubectl:

kubectl describe jobs/pi

Name: pi

Namespace: default

Selector: controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c

Labels: controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c

job-name=pi

Annotations: kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"batch/v1","kind":"Job","metadata":{"annotations":{},"name":"pi","namespace":"default"},"spec":{"backoffLimit":4,"template":...

Parallelism: 1

Completions: 1

Start Time: Mon, 02 Dec 2019 15:20:11 +0200

Completed At: Mon, 02 Dec 2019 15:21:16 +0200

Duration: 65s

Pods Statuses: 0 Running / 1 Succeeded / 0 Failed

Pod Template:

Labels: controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c

job-name=pi

Containers:

pi:

Image: perl

Port: <none>

Host Port: <none>

Command:

perl

-Mbignum=bpi

-wle

print bpi(2000)

Environment: <none>

Mounts: <none>

Volumes: <none>

Events:

Type Reason Age From Message

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

Normal SuccessfulCreate 14m job-controller Created pod: pi-5rwd7

To view completed Pods of a Job, use kubectl get pods.

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

pods=**$(**kubectl get pods --selector=job-name=pi --output=jsonpath='{.items[\*].metadata.name}'**)**

echo $pods

pi-5rwd7

Here, the selector is the same as the selector for the Job. The --output=jsonpath option specifies an expression that just gets the name from each Pod in the returned list.

View the standard output of one of the pods:

kubectl logs $pods

The output is similar to this:

3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109756659334461284756482337867831652712019091456485669234603486104543266482133936072602491412737245870066063155881748815209209628292540917153643678925903600113305305488204665213841469519415116094330572703657595919530921861173819326117931051185480744623799627495673518857527248912279381830119491298336733624406566430860213949463952247371907021798609437027705392171762931767523846748184676694051320005681271452635608277857713427577896091736371787214684409012249534301465495853710507922796892589235420199561121290219608640344181598136297747713099605187072113499999983729780499510597317328160963185950244594553469083026425223082533446850352619311881710100031378387528865875332083814206171776691473035982534904287554687311595628638823537875937519577818577805321712268066130019278766111959092164201989380952572010654858632788659361533818279682303019520353018529689957736225994138912497217752834791315155748572424541506959508295331168617278558890750983817546374649393192550604009277016711390098488240128583616035637076601047101819429555961989467678374494482553797747268471040475346462080466842590694912933136770289891521047521620569660240580381501935112533824300355876402474964732639141992726042699227967823547816360093417216412199245863150302861829745557067498385054945885869269956909272107975093029553211653449872027559602364806654991198818347977535663698074265425278625518184175746728909777727938000816470600161452491921732172147723501414419735685481613611573525521334757418494684385233239073941433345477624168625189835694855620992192221842725502542568876717904946016534668049886272327917860857843838279679766814541009538837863609506800642251252051173929848960841284886269456042419652850222106611863067442786220391949450471237137869609563643719172874677646575739624138908658326459958133904780275901

**Writing a job spec:**

As with all other Kubernetes config, a Job needs apiVersion, kind, and metadata fields.

A Job 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-overview/#pod-templates). It has exactly the same schema as a [pod](https://kubernetes.io/docs/user-guide/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 Job must specify appropriate labels (see [pod selector](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#pod-selector)) and an appropriate restart policy.

Only a [RestartPolicy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/" \l "restart-policy) equal to Never or OnFailure is allowed.

### Pod Selector:

The .spec.selector field is optional. In almost all cases you should not specify it. See section [specifying your own pod selector](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#specifying-your-own-pod-selector).

### Parallel Jobs:

There are three main types of task suitable to run as a Job:

1. Non-parallel Jobs
   * normally, only one Pod is started, unless the Pod fails.
   * the Job is complete as soon as its Pod terminates successfully.
2. Parallel Jobs with a fixed completion count:
   * specify a non-zero positive value for .spec.completions.
   * the Job represents the overall task, and is complete when there is one successful Pod for each value in the range 1 to .spec.completions.
   * **not implemented yet:** Each Pod is passed a different index in the range 1 to .spec.completions.
3. Parallel Jobs with a work queue:
   * do not specify .spec.completions, default to .spec.parallelism.
   * the Pods must coordinate amongst themselves or an external service to determine what each should work on. For example, a Pod might fetch a batch of up to N items from the work queue.
   * each Pod is independently capable of determining whether or not all its peers are done, and thus that the entire Job is done.
   * when any Pod from the Job terminates with success, no new Pods are created.
   * once at least one Pod has terminated with success and all Pods are terminated, then the Job is completed with success.
   * once any Pod has exited with success, no other Pod should still be doing any work for this task or writing any output. They should all be in the process of exiting.

For a non-parallel Job, you can leave both .spec.completions and .spec.parallelism unset. When both are unset, both are defaulted to 1.

For a fixed completion count Job, you should set .spec.completions to the number of completions needed. You can set .spec.parallelism, or leave it unset and it will default to 1.

For a work queue Job, you must leave .spec.completions unset, and set .spec.parallelism to a non-negative integer.

For more information about how to make use of the different types of job, see the [job patterns](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#job-patterns) section.

#### Controlling Parallelism:

The requested parallelism (.spec.parallelism) can be set to any non-negative value. If it is unspecified, it defaults to 1. If it is specified as 0, then the Job is effectively paused until it is increased.

Actual parallelism (number of pods running at any instant) may be more or less than requested parallelism, for a variety of reasons:

* For fixed completion count Jobs, the actual number of pods running in parallel will not exceed the number of remaining completions. Higher values of .spec.parallelism are effectively ignored.
* For work queue Jobs, no new Pods are started after any Pod has succeeded – remaining Pods are allowed to complete, however.
* If the Job [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) has not had time to react.
* If the Job controller failed to create Pods for any reason (lack of ResourceQuota, lack of permission, etc.), then there may be fewer pods than requested.
* The Job controller may throttle new Pod creation due to excessive previous pod failures in the same Job.
* When a Pod is gracefully shut down, it takes time to stop.

**Handling pod containers failure:**

A container in a Pod may fail for a number of reasons, such as because the process in it exited with a non-zero exit code, or the container was killed for exceeding a memory limit, etc. If this happens, and the .spec.template.spec.restartPolicy = "OnFailure", then the Pod stays on the node, but the container is re-run. Therefore, your program needs to handle the case when it is restarted locally, or else specify .spec.template.spec.restartPolicy = "Never". See [pod lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#example-states) for more information on restartPolicy.

An entire Pod can also fail, for a number of reasons, such as when the pod is kicked off the node (node is upgraded, rebooted, deleted, etc.), or if a container of the Pod fails and the .spec.template.spec.restartPolicy = "Never". When a Pod fails, then the Job controller starts a new Pod. This means that your application needs to handle the case when it is restarted in a new pod. In particular, it needs to handle temporary files, locks, incomplete output and the like caused by previous runs.

Note that even if you specify .spec.parallelism = 1 and .spec.completions = 1 and .spec.template.spec.restartPolicy = "Never", the same program may sometimes be started twice.

If you do specify .spec.parallelism and .spec.completions both greater than 1, then there may be multiple pods running at once. Therefore, your pods must also be tolerant of concurrency.

### Pod backoff failure policy:

There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so, set .spec.backoffLimit to specify the number of retries before considering a Job as failed. The back-off limit is set by default to 6. Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s …) capped at six minutes. The back-off count is reset if no new failed Pods appear before the Job’s next status check.

**Note:** Issue [#54870](https://github.com/kubernetes/kubernetes/issues/54870) still exists for versions of Kubernetes prior to version 1.12

**Note:** If your job has restartPolicy = "OnFailure", keep in mind that your container running the Job will be terminated once the job backoff limit has been reached. This can make debugging the Job’s executable more difficult. We suggest setting restartPolicy = "Never" when debugging the Job or using a logging system to ensure output from failed Jobs is not lost inadvertently.

**Job termination and cleanup:**

When a Job completes, no more Pods are created, but the Pods are not deleted either. Keeping them around allows you to still view the logs of completed pods to check for errors, warnings, or other diagnostic output. The job object also remains after it is completed so that you can view its status. It is up to the user to delete old jobs after noting their status. Delete the job with kubectl (e.g. kubectl delete jobs/pi or kubectl delete -f ./job.yaml). When you delete the job using kubectl, all the pods it created are deleted too.

By default, a Job will run uninterrupted unless a Pod fails (restartPolicy=Never) or a Container exits in error (restartPolicy=OnFailure), at which point the Job defers to the .spec.backoffLimit described above. Once .spec.backoffLimit has been reached the Job will be marked as failed and any running Pods will be terminated.

Another way to terminate a Job is by setting an active deadline. Do this by setting the .spec.activeDeadlineSeconds field of the Job to a number of seconds. The activeDeadlineSeconds applies to the duration of the job, no matter how many Pods are created. Once a Job reaches activeDeadlineSeconds, all of its running Pods are terminated and the Job status will become type: Failed with reason: DeadlineExceeded.

Note that a Job’s .spec.activeDeadlineSeconds takes precedence over its .spec.backoffLimit. Therefore, a Job that is retrying one or more failed Pods will not deploy additional Pods once it reaches the time limit specified by activeDeadlineSeconds, even if the backoffLimit is not yet reached.

Example:

apiVersion: batch/v1

kind: Job

metadata:

name: pi-with-timeout

spec:

backoffLimit: 5

activeDeadlineSeconds: 100

template:

spec:

containers:

- name: pi

image: perl

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

Note that both the Job spec and the [Pod template spec](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#detailed-behavior) within the Job have an activeDeadlineSeconds field. Ensure that you set this field at the proper level.

Keep in mind that the restartPolicy applies to the Pod, and not to the Job itself: there is no automatic Job restart once the Job status is type: Failed. That is, the Job termination mechanisms activated with .spec.activeDeadlineSeconds and .spec.backoffLimit result in a permanent Job failure that requires manual intervention to resolve.

**Clean up finished jobs automatically:**

Finished Jobs are usually no longer needed in the system. Keeping them around in the system will put pressure on the API server. If the Jobs are managed directly by a higher level controller, such as [CronJobs](https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/), the Jobs can be cleaned up by CronJobs based on the specified capacity-based cleanup policy.

### TTL Mechanism for Finished Jobs:

Another way to clean up finished Jobs (either Complete or Failed) automatically is to use a TTL mechanism provided by a [TTL controller](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/) for finished resources, by specifying the .spec.ttlSecondsAfterFinished field of the Job.

When the TTL controller cleans up the Job, it will delete the Job cascadingly, i.e. delete its dependent objects, such as Pods, together with the Job. Note that when the Job is deleted, its lifecycle guarantees, such as finalizers, will be honored.

For example:

apiVersion: batch/v1

kind: Job

metadata:

name: pi-with-ttl

spec:

ttlSecondsAfterFinished: 100

template:

spec:

containers:

- name: pi

image: perl

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

The Job pi-with-ttl will be eligible to be automatically deleted, 100 seconds after it finishes.

If the field is set to 0, the Job will be eligible to be automatically deleted immediately after it finishes. If the field is unset, this Job won’t be cleaned up by the TTL controller after it finishes.

Note that this TTL mechanism is alpha, with feature gate TTLAfterFinished. For more information, see the documentation for [TTL controller](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/) for finished resources.

**Job pattrens:**

The Job object can be used to support reliable parallel execution of Pods. The Job object is not designed to support closely-communicating parallel processes, as commonly found in scientific computing. It does support parallel processing of a set of independent but related work items. These might be emails to be sent, frames to be rendered, files to be transcoded, ranges of keys in a NoSQL database to scan, and so on.

In a complex system, there may be multiple different sets of work items. Here we are just considering one set of work items that the user wants to manage together — a batch job.

There are several different patterns for parallel computation, each with strengths and weaknesses. The tradeoffs are:

* One Job object for each work item, vs. a single Job object for all work items. The latter is better for large numbers of work items. The former creates some overhead for the user and for the system to manage large numbers of Job objects.
* Number of pods created equals number of work items, vs. each Pod can process multiple work items. The former typically requires less modification to existing code and containers. The latter is better for large numbers of work items, for similar reasons to the previous bullet.
* Several approaches use a work queue. This requires running a queue service, and modifications to the existing program or container to make it use the work queue. Other approaches are easier to adapt to an existing containerised application.

The tradeoffs are summarized here, with columns 2 to 4 corresponding to the above tradeoffs. The pattern names are also links to examples and more detailed description.

| Pattern | Single Job object | Fewer pods than work items? | Use app unmodified? | Works in Kube 1.1? |
| --- | --- | --- | --- | --- |
| [Job Template Expansion](https://kubernetes.io/docs/tasks/job/parallel-processing-expansion/) |  |  | ✓ | ✓ |
| [Queue with Pod Per Work Item](https://kubernetes.io/docs/tasks/job/coarse-parallel-processing-work-queue/) | ✓ |  | sometimes | ✓ |
| [Queue with Variable Pod Count](https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/) | ✓ | ✓ |  | ✓ |
| Single Job with Static Work Assignment | ✓ |  | ✓ |  |

When you specify completions with .spec.completions, each Pod created by the Job controller has an identical [spec](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status). This means that all pods for a task will have the same command line and the same image, the same volumes, and (almost) the same environment variables. These patterns are different ways to arrange for pods to work on different things.

This table shows the required settings for .spec.parallelism and .spec.completions for each of the patterns. Here, W is the number of work items.

| Pattern | .spec.completions | .spec.parallelism |
| --- | --- | --- |
| [Job Template Expansion](https://kubernetes.io/docs/tasks/job/parallel-processing-expansion/) | 1 | should be 1 |
| [Queue with Pod Per Work Item](https://kubernetes.io/docs/tasks/job/coarse-parallel-processing-work-queue/) | W | any |
| [Queue with Variable Pod Count](https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/) | 1 | any |
| Single Job with Static Work Assignment | W | any |

### Advanced usage:

### Specifying your own pod selector:

Normally, when you create a Job object, you do not specify .spec.selector. The system defaulting logic adds this field when the Job is created. It picks a selector value that will not overlap with any other jobs.

However, in some cases, you might need to override this automatically set selector. To do this, you can specify the .spec.selector of the Job.

Be very careful when doing this. If you specify a label selector which is not unique to the pods of that Job, and which matches unrelated Pods, then pods of the unrelated job may be deleted, or this Job may count other Pods as completing it, or one or both Jobs may refuse to create Pods or run to completion. If a non-unique selector is chosen, then other controllers (e.g. ReplicationController) and their Pods may behave in unpredictable ways too. Kubernetes will not stop you from making a mistake when specifying .spec.selector.

Here is an example of a case when you might want to use this feature.

Say Job old is already running. You want existing Pods to keep running, but you want the rest of the Pods it creates to use a different pod template and for the Job to have a new name. You cannot update the Job because these fields are not updatable. Therefore, you delete Job old but leave its pods running, using kubectl delete jobs/old --cascade=false. Before deleting it, you make a note of what selector it uses:

kubectl get job old -o yaml

kind: Job

metadata:

name: old

...

spec:

selector:

matchLabels:

controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002

...

Then you create a new Job with name new and you explicitly specify the same selector. Since the existing Pods have label controller-uid=a8f3d00d-c6d2-11e5-9f87-42010af00002, they are controlled by Job new as well.

You need to specify manualSelector: true in the new Job since you are not using the selector that the system normally generates for you automatically.

kind: Job

metadata:

name: new

...

spec:

manualSelector: true

selector:

matchLabels:

controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002

...

The new Job itself will have a different uid from a8f3d00d-c6d2-11e5-9f87-42010af00002. Setting manualSelector: true tells the system to that you know what you are doing and to allow this mismatch.

**Alternatives:**

### Bare Pods:

When the node that a Pod is running on reboots or fails, the pod is terminated and will not be restarted. However, a Job will create new Pods to replace terminated ones. For this reason, we recommend that you use a Job rather than a bare Pod, even if your application requires only a single Pod.

### Replication Controller:

Jobs are complementary to [Replication Controllers](https://kubernetes.io/docs/user-guide/replication-controller). A Replication Controller manages Pods which are not expected to terminate (e.g. web servers), and a Job manages Pods that are expected to terminate (e.g. batch tasks).

As discussed in [Pod Lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/), Job is only appropriate for pods with RestartPolicy equal to OnFailure or Never. (Note: If RestartPolicy is not set, the default value is Always.)

### Single Job starts Controller Pod:

Another pattern is for a single Job to create a Pod which then creates other Pods, acting as a sort of custom controller for those Pods. This allows the most flexibility, but may be somewhat complicated to get started with and offers less integration with Kubernetes.

One example of this pattern would be a Job which starts a Pod which runs a script that in turn starts a Spark master controller (see [spark example](https://github.com/kubernetes/examples/tree/master/staging/spark/README.md)), runs a spark driver, and then cleans up.

An advantage of this approach is that the overall process gets the completion guarantee of a Job object, but complete control over what Pods are created and how work is assigned to them.

Cron jobs:

You can use a [CronJob](https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/) to create a Job that will run at specified times/dates, similar to the Unix tool cron.

**Cron Job**

A Cron Job creates [Jobs](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/) on a time-based schedule.

One CronJob object is like one line of a crontab (cron table) file. It runs a job periodically on a given schedule, written in [Cron](https://en.wikipedia.org/wiki/Cron) format.

**Note:** All **CronJob** schedule: times are based on the timezone of the master where the job is initiated.

When creating the manifest for a CronJob resource, make sure the name you provide is no longer than 52 characters. This is because the CronJob controller will automatically append 11 characters to the job name provided and there is a constraint that the maximum length of a Job name is no more than 63 characters.

For instructions on creating and working with cron jobs, and for an example of a spec file for a cron job, see [Running automated tasks with cron jobs](https://kubernetes.io/docs/tasks/job/automated-tasks-with-cron-jobs).

**Cron job limitations:**

A cron job creates a job object about once per execution time of its schedule. We say “about” because there are certain circumstances where two jobs might be created, or no job might be created. We attempt to make these rare, but do not completely prevent them. Therefore, jobs should be idempotent.

If startingDeadlineSeconds is set to a large value or left unset (the default) and if concurrencyPolicy is set to Allow, the jobs will always run at least once.

For every CronJob, the CronJob [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) checks how many schedules it missed in the duration from its last scheduled time until now. If there are more than 100 missed schedules, then it does not start the job and logs the error

Cannot determine if job needs to be started. Too many missed start time (> 100). Set or decrease .spec.startingDeadlineSeconds or check clock skew.

It is important to note that if the startingDeadlineSeconds field is set (not nil), the controller counts how many missed jobs occurred from the value of startingDeadlineSeconds until now rather than from the last scheduled time until now. For example, if startingDeadlineSeconds is 200, the controller counts how many missed jobs occurred in the last 200 seconds.

A CronJob is counted as missed if it has failed to be created at its scheduled time. For example, If concurrencyPolicy is set to Forbid and a CronJob was attempted to be scheduled when there was a previous schedule still running, then it would count as missed.

For example, suppose a CronJob is set to schedule a new Job every one minute beginning at 08:30:00, and its startingDeadlineSeconds field is not set. If the CronJob controller happens to be down from 08:29:00 to 10:21:00, the job will not start as the number of missed jobs which missed their schedule is greater than 100.

To illustrate this concept further, suppose a CronJob is set to schedule a new Job every one minute beginning at 08:30:00, and its startingDeadlineSeconds is set to 200 seconds. If the CronJob controller happens to be down for the same period as the previous example (08:29:00 to 10:21:00,) the Job will still start at 10:22:00. This happens as the controller now checks how many missed schedules happened in the last 200 seconds (ie, 3 missed schedules), rather than from the last scheduled time until now.

The CronJob is only responsible for creating Jobs that match its schedule, and the Job in turn is responsible for the management of the Pods it represents.