Workloads

Understand Pods, the smallest deployable compute object in Kubernetes, and the higher-level abstractions that help you to run them.

A workload is an application running on Kubernetes. Whether your workload is a single component or several that work together, on Kubernetes you run it inside a set of *pods*. In Kubernetes, a `Pod` represents a set of running containers on your cluster.

Kubernetes pods have a defined lifecycle. For example, once a pod is running in your cluster then a critical fault on the node where that pod is running means that all the pods on that node fail. Kubernetes treats that level of failure as final: you would need to create a new `Pod` to recover, even if the node later becomes healthy.

However, to make life considerably easier, you don't need to manage each `Pod` directly. Instead, you can use *workload resources* that manage a set of pods on your behalf. These resources configure controllers that make sure the right number of the right kind of pod are running, to match the state you specified.

Kubernetes provides several built-in workload resources:

- `Deployment` and `ReplicaSet` (replacing the legacy resource ReplicationController). `Deployment` is a good fit for managing a stateless application workload on your cluster, where any `Pod` in the `Deployment` is interchangeable and can be replaced if needed.
- `StatefulSet` lets you run one or more related Pods that do track state somehow. For example, if your workload records data persistently, you can run a `StatefulSet` that matches each `Pod` with a `PersistentVolume`. Your code, running in the `Pods` for that `StatefulSet`, can replicate data to other `Pods` in the same `StatefulSet` to improve overall resilience.
- `DaemonSet` defines `Pods` that provide node-local facilities. These might be fundamental to the operation of your cluster, such as a networking helper tool, or be part of an add-on. Every time you add a node to your cluster that matches the specification in a `DaemonSet`, the control plane schedules a `Pod` for that `DaemonSet` onto the new node.
- `Job` and `CronJob` define tasks that run to completion and then stop. Jobs represent one-off tasks, whereas `CronJobs` recur according to a schedule.

In the wider Kubernetes ecosystem, you can find third-party workload resources that provide additional behaviors. Using a custom resource definition, you can add in a third-party workload resource if you want a specific behavior that's not part of Kubernetes' core. For example, if you wanted to run a group of `Pods` for your application but stop work unless *all* the Pods are available (perhaps for some high-throughput distributed task), then you can implement or install an extension that does provide that feature.

What's next

As well as reading about each resource, you can learn about specific tasks that relate to them:

- Run a stateless application using a `Deployment`
- Run a stateful application either as a single instance or as a replicated set
- Run automated tasks with a `CronJob`

To learn about Kubernetes' mechanisms for separating code from configuration, visit Configuration.

There are two supporting concepts that provide backgrounds about how Kubernetes manages pods for applications:

- Garbage collection tidies up objects from your cluster after their *owning resource* has been removed.
- The *time-to-live after finished* controller removes Jobs once a defined time has passed since they completed.

Once your application is running, you might want to make it available on the internet as a `Service` or, for web application only, using an `Ingress`.

1 - Pods

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, 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 non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

As well as application containers, a Pod can contain init containers that run during Pod startup. You can also inject ephemeral containers for debugging if your cluster offers this.

What is a Pod?

> **Note:** While Kubernetes supports more container runtimes than just Docker, Docker is the most commonly known runtime, and it helps to describe Pods using some terminology from Docker.

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.

In terms of Docker concepts, a Pod is similar to a group of Docker containers with shared namespaces and shared filesystem volumes.

Using Pods

Usually you don't need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as Deployment or Job. If your Pods need to track state, consider the StatefulSet resource.

Pods in a Kubernetes cluster are 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; Kubernetes manages Pods rather than managing the containers directly.
- **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service —for example, one container serving data stored in a shared volume to the public, while a separate *sidecar* container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

  > **Note:** 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.

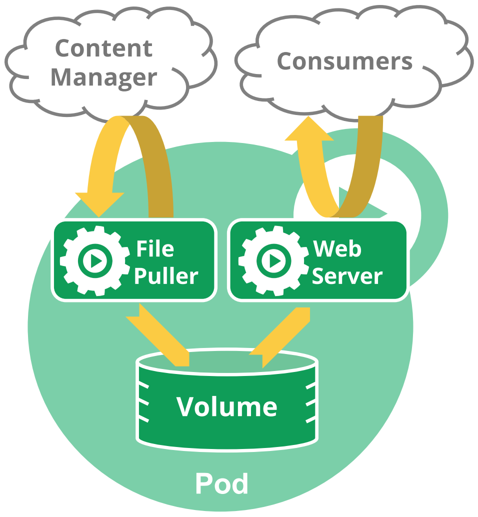
Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as *replication*. Replicated Pods are usually created and managed as a group by a workload resource and its controller.

See Pods and controllers for more information on how Kubernetes uses workload resources, and their controllers, to implement application scaling and auto-healing.

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.

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:



Some Pods have init containers as well as app containers. Init containers run and complete before the app containers are started.

Pods natively provide two kinds of shared resources for their constituent containers: networking and storage.

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), the new Pod is scheduled to run on a Node in your cluster. The Pod remains on that node until the Pod finishes execution, 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 a Pod. A Pod is not a process, but an environment for running container(s). A Pod persists until it is deleted.

When you create the manifest for a Pod object, make sure the name specified is a valid DNS subdomain name.

Pods and controllers

You can use workload resources to create and manage multiple Pods for you. A controller for the resource handles replication and rollout and automatic healing in case of Pod failure. For example, if a Node fails, a controller notices that Pods on that Node have stopped working and creates a replacement Pod. The scheduler places the replacement Pod onto a healthy Node.

Here are some examples of workload resources that manage one or more Pods:

- Deployment
- StatefulSet
- DaemonSet

Pod templates

Controllers for workload resources create Pods from a *pod template* and manage those Pods on your behalf.

PodTemplates are specifications for creating Pods, and are included in workload resources such as Deployments, Jobs, and DaemonSets.

Each controller for a workload resource uses the `PodTemplate` inside the workload object to make actual Pods. The `PodTemplate` is part of the desired state of whatever workload resource you used to run your app.

The sample below is a manifest for a simple Job with a `template` that starts one container. The container in that Pod prints a message then pauses.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```

Modifying the pod template or switching to a new pod template has no direct effect on the Pods that already exist. If you change the pod template for a workload resource, that resource needs to create replacement Pods that use the updated template.

For example, the StatefulSet controller ensures that the running Pods match the current pod template for each StatefulSet object. If you edit the StatefulSet to change its pod template, the StatefulSet starts to create new Pods based on the updated template. Eventually, all of the old Pods are replaced with new Pods, and the update is complete.

Each workload resource implements its own rules for handling changes to the Pod template. If you want to read more about StatefulSet specifically, read Update strategy in the StatefulSet Basics tutorial.

On Nodes, the kubelet does not directly observe or manage any of the details around pod templates and updates; those details are abstracted away. That abstraction and separation of concerns simplifies system semantics, and makes it feasible to extend the cluster's behavior without changing existing code.

Pod update and replacement

As mentioned in the previous section, when the Pod template for a workload resource is changed, the controller creates new Pods based on the updated template instead of updating or patching the existing Pods.

Kubernetes doesn't prevent you from managing Pods directly. It is possible to update some fields of a running Pod, in place. However, Pod update operations like `patch`, and `replace` have some limitations:

- Most of the metadata about a Pod is immutable. For example, you cannot change the `namespace`, `name`, `uid`, or `creationTimestamp` fields; the `generation` field is unique. It only accepts updates that increment the field's current value.
- If the `metadata.deletionTimestamp` is set, no new entry can be added to the `metadata.finalizers` list.

- Pod updates may not change fields other than `spec.containers[*].image`, `spec.initContainers[*].image`, `spec.activeDeadlineSeconds` or `spec.tolerations`. For `spec.tolerations`, you can only add new entries.
- When updating the `spec.activeDeadlineSeconds` field, two types of updates are allowed:
    1. setting the unassigned field to a positive number;
    2. updating the field from a positive number to a smaller, non-negative number.

Resource sharing and communication

Pods enable data sharing and communication among their constituent containers.

Storage in Pods

A Pod can specify a set of shared 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 Storage for more information on how Kubernetes implements shared storage and makes it available to Pods.

Pod networking

Each Pod is assigned a unique IP address for each address family. Every container in a Pod shares the network namespace, including the IP address and network ports. Inside a Pod (and **only** then), the containers that belong to the 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). Within a Pod, containers share an IP address and port space, and can find each other via `localhost`. The containers in a Pod 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. Containers that want to interact with a container running in a different Pod can use IP networking to communicate.

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 section.

Privileged mode for containers

Any container in a Pod can enable privileged mode, using the `privileged` flag on the security context of the container spec. This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices. Processes within a privileged container get almost the same privileges that are available to processes outside a container.

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

Static Pods

Static Pods are managed directly by the kubelet daemon on a specific node, without the API server observing them. Whereas most Pods are managed by the control plane (for example, a Deployment), for static Pods, the kubelet directly supervises each static Pod (and restarts it if it fails).

Static Pods are always bound to one Kubelet on a specific node. The main use for static Pods is to run a self-hosted control plane: in other words, using the kubelet to supervise the individual control plane components.

The kubelet automatically tries to create a mirror Pod on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there.

What's next

- Learn about the lifecycle of a Pod.
- Learn about RuntimeClass and how you can use it to configure different Pods with different container runtime configurations.
- Read about Pod topology spread constraints.
- Read about PodDisruptionBudget and how you can use it to manage application availability during disruptions.
- Pod is a top-level resource in the Kubernetes REST API. The Pod object definition describes the object in detail.
- The Distributed System Toolkit: Patterns for Composite Containers explains common layouts for Pods with more than one container.

To understand the context for why Kubernetes wraps a common Pod API in other resources (such as StatefulSets or Deployments), you can read about the prior art, including:

- Aurora
- Borg
- Marathon
- Omega
- Tupperware.

1.1 - Pod Lifecycle

This page describes the lifecycle of a Pod. Pods follow a defined lifecycle, starting in the `Pending` phase, moving through `Running` if at least one of its primary containers starts OK, and then through either the `Succeeded` or `Failed` phases depending on whether any container in the Pod terminated in failure.

Whilst a Pod is running, the kubelet is able to restart containers to handle some kind of faults. Within a Pod, Kubernetes tracks different container states and determines what action to take to make the Pod healthy again.

In the Kubernetes API, Pods have both a specification and an actual status. The status for a Pod object consists of a set of Pod conditions. You can also inject custom readiness information into the condition data for a Pod, if that is useful to your application.

Pods are only scheduled once in their lifetime. Once a Pod is scheduled (assigned) to a Node, the Pod runs on that Node until it stops or is terminated.

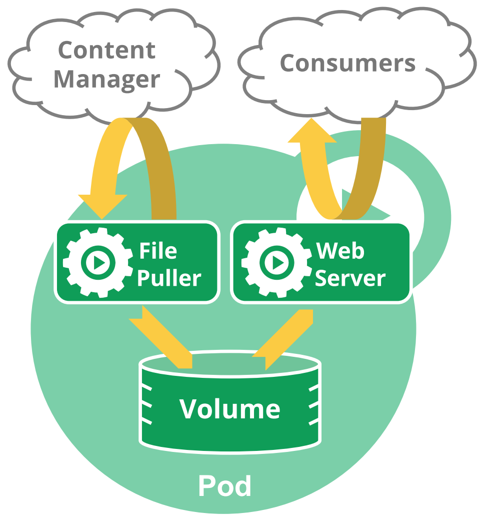
Pod lifetime

Like individual application containers, Pods are considered to be relatively ephemeral (rather than durable) entities. 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 dies, the Pods scheduled to that node are scheduled for deletion after a timeout period.

Pods do not, by themselves, self-heal. If a Pod is scheduled to a node that then 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.

A given Pod (as defined by a UID) is never "rescheduled" to a different node; instead, that Pod can be replaced by a new, near-identical Pod, with even the same name if desired, but with a different UID.

When something is said to have the same lifetime as a Pod, such as a volume, that means that the thing exists as long as that specific Pod (with that exact UID) exists. If that Pod is deleted for any reason, and even if an identical replacement is created, the related thing (a volume, in this example) 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.

Pod phase

A Pod's `status` field is a PodStatus 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 cluster, but one or more of the containers has not been set up and made ready to run. This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network.
`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. This phase typically occurs due to an error in communicating with the node where the Pod should be running.

Note: When a Pod is being deleted, it is shown as `Terminating` by some kubectl commands. This `Terminating` status is not one of the Pod phases. A Pod is granted a term to terminate gracefully, which defaults to 30 seconds. You can use the flag `--force` to terminate a Pod by force.

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.

Container states

As well as the phase of the Pod overall, Kubernetes tracks the state of each container inside a Pod. You can use container lifecycle hooks to trigger events to run at certain points in a container's lifecycle.

Once the scheduler assigns a Pod to a Node, the kubelet starts creating containers for that Pod using a container runtime. There are three possible container states: `Waiting`, `Running`, and `Terminated`.

To check the state of a Pod's containers, you can use `kubectl describe pod <name-of-pod>`. The output shows the state for each container within that Pod.

Each state has a specific meaning:

`Waiting`

If a container is not in either the `Running` or `Terminated` state, it is `Waiting`. A container in the `Waiting` state is still running the operations it requires in order to complete start up: for example, pulling the container image from a container image registry, or applying Secret data.

When you use `kubectl` to query a Pod with a container that is `Waiting`, you also see a Reason field to summarize why the container is in that state.

`Running`

The `Running` status indicates that a container is executing without issues. If there was a `postStart` hook configured, it has already executed and finished. When you use `kubectl` to query a Pod with a container that is `Running`, you also see information about when the container entered the `Running` state.

`Terminated`

A container in the `Terminated` state began execution and then either ran to completion or failed for some reason. When you use `kubectl` to query a Pod with a container that is `Terminated`, you see a reason, an exit code, and the start and finish time for that container's period of execution.

If a container has a `preStop` hook configured, that runs before the container enters the `Terminated` state.

Container restart policy

The `spec` of a Pod has a `restartPolicy` field with possible values Always, OnFailure, and Never. The default value is Always.

The `restartPolicy` applies to all containers in the Pod. `restartPolicy` only refers to restarts of the containers by the kubelet on the same node. After containers in a Pod exit, the kubelet restarts them with an exponential back-off delay (10s, 20s, 40s, …), that is capped at five minutes. Once a container has executed for 10 minutes without any problems, the kubelet resets the restart backoff timer for that container.

Pod conditions

A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed:

- `PodScheduled`: the Pod has been scheduled to a node.
- `ContainersReady`: all containers in the Pod are ready.
- `Initialized`: all init containers have started successfully.
- `Ready`: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services.

Field name	Description
`type`	Name of this Pod condition.
`status`	Indicates whether that condition is applicable, with possible values "`True`", "`False`", or "`Unknown`".
`lastProbeTime`	Timestamp of when the Pod condition was last probed.
`lastTransitionTime`	Timestamp for when the Pod last transitioned from one status to another.
`reason`	Machine-readable, UpperCamelCase text indicating the reason for the condition's last transition.

Field name	Description
`message`	Human-readable message indicating details about the last status transition.

Pod readiness

FEATURE STATE: `Kubernetes v1.14 [stable]`

Your application can inject extra feedback or signals into PodStatus: *Pod readiness*. To use this, set `readinessGates` in the Pod's `spec` to specify a list of additional conditions that the kubelet evaluates for Pod readiness.

Readiness gates are determined by the current state of `status.condition` fields for the Pod. If Kubernetes cannot find such a condition in the `status.conditions` field of a Pod, the status of the condition is defaulted to "`False`".

Here is an example:

```yaml
kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.example.com/feature-1"
status:
  conditions:
    - type: Ready                              # a built in 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 Pod conditions you add must have names that meet the Kubernetes label key format.

Status for Pod readiness

The `kubectl patch` command does not support patching object status. To set these `status.conditions` for the pod, applications and operators should use the `PATCH` action. You can use a Kubernetes client library to write code that sets custom Pod conditions for Pod readiness.

For a Pod that uses custom conditions, that Pod is evaluated to be ready **only** when both the following statements apply:

- All containers in the Pod are ready.
- All conditions specified in `readinessGates` are `True`.

When a Pod's containers are Ready but at least one custom condition is missing or `False`, the kubelet sets the Pod's condition to `ContainersReady`.

Container probes

A Probe is a diagnostic performed periodically by the kubelet on a Container. To perform a diagnostic, the kubelet calls a Handler implemented by the container. There are three types of handlers:

- ExecAction: Executes a specified command inside the container. The diagnostic is considered successful if the command exits with a status code of 0.
- TCPSocketAction: Performs a TCP check against the Pod's IP address on a specified port. The diagnostic is considered successful if the port is open.
- HTTPGetAction: Performs an HTTP `GET` request against the Pod'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. If a Container does not provide a liveness probe, the default state is `Success`.
- `readinessProbe`: Indicates whether the container is ready to respond to 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. If a Container does not provide a startup probe, the default state is `Success`.

For more information about how to set up a liveness, readiness, or startup probe, see Configure Liveness, Readiness and Startup Probes.

When should you use a liveness probe?

FEATURE STATE: `Kubernetes v1.0 [stable]`

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?

FEATURE STATE: `Kubernetes v1.0 [stable]`

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: If you 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?

FEATURE STATE: `Kubernetes v1.20 [stable]`

Startup probes are useful for Pods that have containers that take a long time to come into service. Rather than set a long liveness interval, you can configure a separate configuration for probing the container as it starts up, allowing a time longer than the liveness interval would allow.

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 10s. 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.

Termination of Pods

Because Pods represent processes running on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (rather than being abruptly stopped with a `KILL` signal and having no chance to clean up).

The design aim is for you to be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When you request deletion of a Pod, the cluster records and tracks the intended grace period before the Pod is allowed to be forcefully killed. With that forceful shutdown tracking in place, the kubelet attempts graceful shutdown.

Typically, the container runtime sends a TERM signal to the main process in each container. Many container runtimes respect the `STOPSIGNAL` value defined in the container image and send this instead of TERM. Once the grace period has expired, the KILL signal is sent to any remaining processes, and the Pod is then deleted from the API Server. If the kubelet or the container runtime's management service is restarted while waiting for processes to terminate, the cluster retries from the start including the full original grace period.

An example flow:

1. You use the `kubectl` tool to manually delete a specific Pod, with the default grace period (30 seconds).
2. The Pod in the API server is updated with the time beyond which the Pod is considered "dead" along with the grace period. If you use `kubectl describe` to check on the Pod you're deleting, that Pod shows up as "Terminating". On the node where the Pod is running: as soon as the kubelet sees that a Pod has been marked as terminating (a graceful shutdown duration has been set), the kubelet begins the local Pod shutdown process.
    1. If one of the Pod's containers has defined a `preStop` hook, the kubelet runs that hook inside of the container. If the `preStop` hook is still running after the grace period expires, the kubelet requests a small, one-off grace period extension of 2 seconds.

       Note: If the `preStop` hook needs longer to complete than the default grace period allows, you must modify `terminationGracePeriodSeconds` to suit this.

    2. The kubelet triggers the container runtime to send a TERM signal to process 1 inside each container.

       Note: The containers in the Pod receive the TERM signal at different times and in an arbitrary order. If the order of shutdowns matters, consider using a `preStop` hook to synchronize.

3. At the same time as the kubelet is starting graceful shutdown, the control plane removes that shutting-down Pod from Endpoints (and, if enabled, EndpointSlice) objects where these represent a Service with a configured selector. ReplicaSets and other workload resources no longer treat the shutting-down Pod as a valid, in-service replica. Pods that shut down slowly cannot continue to serve traffic as load balancers (like the service proxy) remove the Pod from the list of endpoints as soon as the termination grace period *begins*.
4. When the grace period expires, the kubelet triggers forcible shutdown. The container runtime sends `SIGKILL` to any processes still running in any container in the Pod. The

kubelet also cleans up a hidden `pause` container if that container runtime uses one.
5. The kubelet triggers forcible removal of Pod object from the API server, by setting grace period to 0 (immediate deletion).
6. The API server deletes the Pod's API object, which is then no longer visible from any client.

Forced Pod termination

Caution: Forced deletions can be potentially disruptive for some workloads and their Pods.

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows you to override the default and specify your own value.

Setting the grace period to `0` forcibly and immediately deletes the Pod from the API server. If the pod was still running on a node, that forcible deletion triggers the kubelet to begin immediate cleanup.

Note: You must specify an additional flag `--force` along with `--grace-period=0` in order to perform force deletions.

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.

If you need to force-delete Pods that are part of a StatefulSet, refer to the task documentation for deleting Pods from a StatefulSet.

Garbage collection of failed Pods

For failed Pods, the API objects remain in the cluster's API 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.

What's next

- Get hands-on experience attaching handlers to Container lifecycle events.
- Get hands-on experience configuring Liveness, Readiness and Startup Probes.
- Learn more about container lifecycle hooks.
- For detailed information about Pod / Container status in the API, see PodStatus and ContainerStatus.

1.2 - Init Containers

This page provides an overview of init containers: specialized containers that run before app containers in a Pod. 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

A Pod 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, the kubelet repeatedly restarts that init container until it succeeds. However, if the Pod has a `restartPolicy` of Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.

To specify an init container for a Pod, add the `initContainers` field into the Pod specification, as an array of objects of type Container, 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.

Also, init containers do not support `lifecycle`, `livenessProbe`, `readinessProbe`, or `startupProbe` 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 utilities 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 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 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
```

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

```
sleep 60
```

- Clone a Git repository into a Volume
- 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.$(cat /var/run/secrets/kubernete
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup mydb.$(cat /var/run/secrets/kubernetes.io/
```

You can start this Pod by running:

```
kubectl apply -f myapp.yaml
```

The output is similar to this:

```
pod/myapp-pod created
```

And check on its status with:

```
kubectl get -f myapp.yaml
```

The output is similar to this:

```
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0          6m
```

or for more details:

```
kubectl describe -f myapp.yaml
```

The output is similar to this:

```
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
  ---------    --------    -----    ----                      -------------
  16s          16s         1        {default-scheduler }
  16s          16s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{ir
```

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:

```yaml
---
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
```

The output is similar to this:

```
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
```

The output is similar to this:

```
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 contains a link to a more detailed example.

Detailed behavior

During Pod startup, the kubelet delays running init containers until the networking and storage are ready. Then the kubelet runs the Pod's init containers in the order they appear in the Pod's spec.

Each init 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 false.

If the Pod restarts, 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.

What's next

- Read about creating a Pod that has an init container
- Learn how to debug init containers

1.3 - Pod Topology Spread Constraints

FEATURE STATE: `Kubernetes v1.19 [stable]`

You can use *topology spread constraints* to control how Pods 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.

> **Note:** In versions of Kubernetes before v1.19, you must enable the `EvenPodsSpread` feature gate on the API server and the scheduler in order to use Pod topology spread constraints.

Prerequisites

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:

graph TB subgraph "zoneB" n3(Node3) n4(Node4) end subgraph "zoneA" n1(Node1) n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4 k8s; class zoneA,zoneB cluster;

Instead of manually applying labels, you can also reuse the well-known labels that are created and populated automatically on most clusters.

Spread Constraints for Pods

API

The API field `pod.spec.topologySpreadConstraints` is defined as below:

```yaml
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. Its semantics differs according to the value of `whenUnsatisfiable`:
  - when `whenUnsatisfiable` equals to "DoNotSchedule", `maxSkew` is the maximum permitted difference between the number of matching pods in the target topology and the global minimum.
  - when `whenUnsatisfiable` equals to "ScheduleAnyway", scheduler gives higher precedence to topologies that would help reduce the skew.
- **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.
- **whenUnsatisfiable** 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 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:

```
graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph
"zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain
fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s
fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster
fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class
n1,n2,n3,n4,p1,p2,p3 k8s; class zoneA,zoneB cluster;
```

If we want an incoming Pod to be evenly spread with existing Pods across zones, the spec can be given as:

pods/topology-spread-constraints/one-constraint.yaml

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: mypod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - name: 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:<any value>" 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":

graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) p4(mypod) --> n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;

OR

graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) p4(mypod) --> n3 n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;

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 TopologySpreadConstraints

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:

graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;

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

pods/topology-spread-constraints/two-constraints.yaml

```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:

```
graph BT subgraph "zoneB" p4(Pod) --> n3(Node3) p5(Pod) --> n3 end
subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n1 p3(Pod) -->
n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-
width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-
width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-
width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3,p4,p5 k8s; class
zoneA,zoneB cluster;
```

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 incoming Pod 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:

  graph BT subgraph "zoneB" p3(Pod) --> n3(Node3) n4(Node4) end subgraph "zoneA" p1(Pod) --> n1(Node1) p2(Pod) --> n2(Node2) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n1,n2,n3,n4,p1,p2,p3 k8s; class p4 plain; class zoneA,zoneB cluster;

  graph BT subgraph "zoneC" n5(Node5) end classDef plain fill:#ddd,stroke:#fff,stroke-width:4px,color:#000; classDef k8s fill:#326ce5,stroke:#fff,stroke-width:4px,color:#fff; classDef cluster fill:#fff,stroke:#bbb,stroke-width:2px,color:#326ce5; class n5 k8s; class zoneC cluster;

  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

```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
```

Cluster-level default constraints

It is possible to set default topology spread constraints for a cluster. Default topology spread constraints are applied to a Pod if, and only if:

- It doesn't define any constraints in its `.spec.topologySpreadConstraints`.

- It belongs to a service, replication controller, replica set or stateful set.

Default constraints can be set as part of the `PodTopologySpread` plugin args in a scheduling profile. The constraints are specified with the same API above, except that `labelSelector` must be empty. The selectors are calculated from the services, replication controllers, replica sets or stateful sets that the Pod belongs to.

An example configuration might look like follows:

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - pluginConfig:
      - name: PodTopologySpread
        args:
          defaultConstraints:
            - maxSkew: 1
              topologyKey: topology.kubernetes.io/zone
              whenUnsatisfiable: ScheduleAnyway
          defaultingType: List
```

Note: The score produced by default scheduling constraints might conflict with the score produced by the `SelectorSpread` plugin. It is recommended that you disable this plugin in the scheduling profile when using default constraints for `PodTopologySpread`.

Internal default constraints

FEATURE STATE: `Kubernetes v1.20 [beta]`

With the `DefaultPodTopologySpread` feature gate, enabled by default, the legacy `SelectorSpread` plugin is disabled. kube-scheduler uses the following default topology constraints for the `PodTopologySpread` plugin configuration:

```yaml
defaultConstraints:
  - maxSkew: 3
    topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: ScheduleAnyway
  - maxSkew: 5
    topologyKey: "topology.kubernetes.io/zone"
    whenUnsatisfiable: ScheduleAnyway
```

Also, the legacy `SelectorSpread` plugin, which provides an equivalent behavior, is disabled.

Note:
If your nodes are not expected to have **both** `kubernetes.io/hostname` and `topology.kubernetes.io/zone` labels set, define your own constraints instead of using the Kubernetes defaults.

The `PodTopologySpread` plugin does not score the nodes that don't have the topology keys specified in the spreading constraints.

If you don't want to use the default Pod spreading constraints for your cluster, you can disable those defaults by setting `defaultingType` to `List` and leaving empty `defaultConstraints` in the `PodTopologySpread` plugin configuration:

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - pluginConfig:
      - name: PodTopologySpread
        args:
          defaultConstraints: []
          defaultingType: List
```

Comparison 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.

For finer control, you can specify topology spread constraints to distribute Pods across different topology domains - to achieve either high availability or cost-saving. This can also help on rolling update workloads and scaling out replicas smoothly. See Motivation for more details.

Known Limitations

- Scaling down a Deployment may result in imbalanced Pods distribution.
- Pods matched on tainted nodes are respected. See Issue 80921

What's next

- Blog: Introducing PodTopologySpread explains `maxSkew` in details, as well as bringing up some advanced usage examples.

1.4 - 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

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 panic
- the node disappears from the cluster due to cluster network partition
- eviction of a pod due to the node being out-of-resources.

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 for repair or upgrade.
- Draining a node from a cluster to scale the cluster down (learn about Cluster Autoscaling).
- 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 your pod requests the resources it needs.
- Replicate your application if you need higher availability. (Learn about running replicated stateless and stateful applications.)
- For even higher availability when running replicated applications, spread applications across racks (using anti-affinity) or across zones (if using a multi-zone cluster.)

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.

Pod disruption budgets

FEATURE STATE: `Kubernetes v1.5 [beta]`

Kubernetes offers features to help you run highly available applications even when you introduce frequent voluntary disruptions.

As an application owner, you can create a PodDisruptionBudget (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 PodDisruptionBudgets by calling the Eviction API instead of directly deleting pods or deployments.

For example, the `kubectl drain` subcommand lets you mark a node as going out of service. When you run `kubectl drain`, the tool tries to evict all of the Pods on the Node you're taking out of service. The eviction request that `kubectl` submits on your behalf may be temporarily rejected, so the tool periodically retries all failed requests until all Pods on the target node 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 workload resource that is managing those pods. The control plane discovers the owning workload resource by examining the `.metadata.ownerReferences` of the Pod.

PDBs cannot prevent 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 workload resources (such as Deployment and StatefulSet) are not limited by PDBs when doing rolling upgrades. Instead, the handling of failures during application updates is configured in the spec for the specific workload resource.

When a pod is evicted using the eviction API, it is gracefully terminated, honoring the `terminationGracePeriodSeconds` setting in its PodSpec.)

PodDisruptionBudget 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***

node-1 *drained*	node-2	node-3	*no node*
	pod-b *terminating*	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.

What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.

- Learn more about draining nodes

- Learn about updating a deployment including steps to maintain its availability during the rollout.

1.5 - Ephemeral Containers

FEATURE STATE: `Kubernetes v1.16 [alpha]`

This page provides an overview of ephemeral containers: a special type of container that runs temporarily in an existing Pod 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. In accordance with the Kubernetes Deprecation Policy, this alpha feature could change significantly in the future or be removed entirely.

Understanding ephemeral containers

Pods 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.

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.

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 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 so you can view processes in other containers.

See Debugging with Ephemeral Debug Container for examples of troubleshooting using ephemeral containers.

Ephemeral containers API

Note: The examples in this section require the `EphemeralContainers` feature gate 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 `kubectl debug` or another `kubectl` plugin to automate these steps rather than invoking the API directly.

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/ephemeralcontainer
```

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://cf81908f149e7e9213d3c3644eda55c72efaff67652a2685c1146f0c
    Image:         busybox
    Image ID:      docker-pullable://busybox@sha256:9f1003c480699be56815db0f8146ad2e
    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 interact with the new ephemeral container in the same way as other containers using `kubectl attach`, `kubectl exec`, and `kubectl logs`, for example:

```
kubectl attach -it example-pod -c debugger
```

2 - Workload Resources

2.1 - Deployments

A *Deployment* provides declarative updates for Pods and ReplicaSets.

You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

> **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

The following are typical use cases for Deployments:

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

Creating a Deployment

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

`controllers/nginx-deployment.yaml`

```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:1.14.2
        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 `.spec.replicas` field.
- The `.spec.selector` field defines how the Deployment finds which Pods to manage. In this case, you select a label that is defined in the Pod template (`app: nginx`). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule.

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

- The `template` field contains the following sub-fields:
  - The Pods are labeled `app: nginx` using the `.metadata.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 image at version 1.14.2.
  - Create one container and name it `nginx` using the `.spec.template.spec.containers[0].name` field.

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

1. Create the Deployment by running the following command:

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

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

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

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

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

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

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

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

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

The output is similar to:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "nginx-deployment" successfully rolled out
```

4. Run the `kubectl get deployments` again a few seconds later. The output is similar to this:

```
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.

5. To see the ReplicaSet (`rs`) created by the Deployment, run `kubectl get rs` . The output is similar to this:

```
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-75675f5897   3         3         3       18s
```

   ReplicaSet output shows the following fields:

   - `NAME` lists the names of the ReplicaSets in the namespace.
   - `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.
   - `READY` 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 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.

6. To see the labels automatically generated for each Pod, run `kubectl get pods --show-labels` . The output is similar to:

```
NAME                                READY     STATUS    RESTARTS   AGE       LA
nginx-deployment-75675f5897-7ci7o   1/1       Running   0          18s       ap
nginx-deployment-75675f5897-kzszj   1/1       Running   0          18s       ap
nginx-deployment-75675f5897-qqcnn   1/1       Running   0          18s       ap
```

   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

Caution: Do not change this 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:1.16.1` image instead of the `nginx:1.14.2` image.

   ```
   kubectl --record deployment.apps/nginx-deployment set image deployment.v1.apps/
   ```

   or use the following command:

   ```
   kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1 --record
   ```

   The output is similar to:

   ```
   deployment.apps/nginx-deployment image updated
   ```

   Alternatively, you can `edit` the Deployment and change `.spec.template.spec.containers[0].image` from `nginx:1.14.2` to `nginx:1.16.1`:

   ```
   kubectl edit deployment.v1.apps/nginx-deployment
   ```

   The output is similar to:

   ```
   deployment.apps/nginx-deployment edited
   ```

2. To see the rollout status, run:

   ```
   kubectl rollout status deployment/nginx-deployment
   ```

   The output is similar to this:

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

   or

   ```
   deployment "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 unava
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
   Containers:
    nginx:
      Image:        nginx:1.16.1
      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 s
    Normal  ScalingReplicaSet  24s   deployment-controller  Scaled up replica s
    Normal  ScalingReplicaSet  22s   deployment-controller  Scaled down replica
    Normal  ScalingReplicaSet  22s   deployment-controller  Scaled up replica s
    Normal  ScalingReplicaSet  19s   deployment-controller  Scaled down replica
    Normal  ScalingReplicaSet  19s   deployment-controller  Scaled up replica s
    Normal  ScalingReplicaSet  14s   deployment-controller  Scaled down replica
```

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:1.14.2`, but then update the Deployment to create 5 replicas of `nginx:1.16.1`, when only 3 replicas of `nginx:1.14.2` had been created. In that case, the Deployment immediately starts killing the 3 `nginx:1.14.2` Pods that it had created, and starts creating `nginx:1.16.1` Pods. It does not wait for the 5 replicas of `nginx:1.14.2` 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:1.161` instead of `nginx:1.16.1`:

  ```
  kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.161 --recor
  ```

  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/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.

- 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:1.161
    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
  --------- --------    -----   ----                    -------------   -------
  1m        1m          1       {deployment-controller }                Normal
  22s       22s         1       {deployment-controller }                Normal
  22s       22s         1       {deployment-controller }                Normal
  22s       22s         1       {deployment-controller }                Normal
  21s       21s         1       {deployment-controller }                Normal
  21s       21s         1       {deployment-controller }                Normal
  13s       13s         1       {deployment-controller }                Normal
  13s       13s         1       {deployment-controller }                Normal
```

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-
2           kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1
3           kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1
```

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

- Annotating the Deployment with `kubectl annotate deployment.v1.apps/nginx-deployment kubernetes.io/change-cause="image updated to 1.16.1"`
- Append the `--record` flag to save the `kubectl` command that is making changes to the resource.

   - Manually editing the manifest of the resource.

2. 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
     Containers:
      nginx:
       Image:      nginx:1.16.1
       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 rolled back
   ```

   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 rolled back
   ```

   For more details about rollout related commands, read `kubectl 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.

2. 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
```

3. 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
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unava
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.16.1
    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 se
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica se
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica se
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica se
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica se
  Normal  DeploymentRollback  15s   deployment-controller  Rolled back deployme
  Normal  ScalingReplicaSet   15s   deployment-controller  Scaled down replica
```

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 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-percer
```

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=3, and maxUnavailable=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 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:1.16.1
```

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 updated 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
```

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 while rolling out a new ReplicaSet, it can be complete, or it can fail to progress.

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).

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/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "nginx-deployment" successfully rolled out
```

and the exit status from `kubectl rollout` is 0 (success):

```
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`). `.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":{"progressDeadlineSecc
```

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 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: exce
      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/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
```

and the exit status from `kubectl rollout` is 1 (indicating an error):

```
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.

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, configuring containers, and using kubectl to manage resources documents. The name of a Deployment object must be a valid DNS subdomain name.

A Deployment also needs a `.spec` section.

Pod Template

The `.spec.template` and `.spec.selector` are the only required field of the `.spec`.

The `.spec.template` is a Pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a Deployment must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See selector.

Only a `.spec.template.spec.restartPolicy` 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 a required field that specifies a label selector 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`.

> **Note:** This will only guarantee Pod termination previous to creation for upgrades. If you upgrade a Deployment, all Pods of the old revision will be terminated immediately. Successful removal is awaited before any Pod of the new revision is created. If you manually delete a Pod, the lifecycle is controlled by the ReplicaSet and the replacement will be created immediately (even if the old Pod is still in a Terminating state). If you need an "at most" guarantee for your Pods, you should consider using a StatefulSet.

Rolling Update Deployment

The Deployment updates Pods in a rolling update 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 - 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. This defaults to 600. 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.

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.

2.2 - 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

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

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

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

When to use a ReplicaSet

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

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

Example

controllers/frontend.yaml

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

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

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

You can then get the current ReplicaSets deployed:

```
kubectl get rs
```

And see the frontend one you created:

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

You can also check on the state of the ReplicaSet:

```
kubectl describe rs/frontend
```

And you will see output similar to:

```
Name:         frontend
Namespace:    default
Selector:     tier=frontend
Labels:       app=guestbook
              tier=frontend
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"apps/v1","kind":"ReplicaSet","metadata":{"annotations
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  tier=frontend
  Containers:
   php-redis:
    Image:        gcr.io/google_samples/gb-frontend:v3
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  117s  replicaset-controller  Created pod: frontend-wtsmm
  Normal  SuccessfulCreate  116s  replicaset-controller  Created pod: frontend-b2zdv
  Normal  SuccessfulCreate  116s  replicaset-controller  Created pod: frontend-vcmts
```

And lastly you can check for the Pods brought up:

```
kubectl get pods
```

You should see Pod information similar to:

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

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

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

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

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

Non-Template Pod acquisitions

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

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

pods/pod-rs.yaml

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

---

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

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

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

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

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

Fetching the Pods:

```
kubectl get pods
```

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

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

If you create the Pods first:

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

And then create the ReplicaSet however:

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

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

```
kubectl get pods
```

Will reveal in its output:

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

```
pod1             1/1     Running   0          36s
pod2             1/1     Running   0          36s
```

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

Writing a ReplicaSet manifest

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

The name of a ReplicaSet object must be a valid DNS subdomain name.

A ReplicaSet also needs a `.spec` section.

Pod Template

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

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

Pod Selector

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

```
matchLabels:
  tier: frontend
```

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

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

Replicas

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

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

Working with ReplicaSets

Deleting a ReplicaSet and its Pods

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

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

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

Deleting just a ReplicaSet

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

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

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

Isolating Pods from a ReplicaSet

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

Scaling a ReplicaSet

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

ReplicaSet as a Horizontal Pod Autoscaler Target

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

controllers/hpa-rs.yaml

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

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

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

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

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

Alternatives to ReplicaSet

Deployment (recommended)

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

Bare Pods

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

Job

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

DaemonSet

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

ReplicationController

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

2.3 - StatefulSets

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

Manages the deployment and scaling of a set of Pods, *and provides guarantees about the ordering and uniqueness* of these Pods.

Like a 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.

If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.

Using StatefulSets

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 or ReplicaSet may be better suited to your stateless needs.

Limitations

- The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner 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 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 with the default Pod Management Policy (`OrderedReady`), it's possible to get into a broken state that requires manual intervention to repair.

Components

The example below demonstrates the components of a StatefulSet.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
```

```yaml
    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 provisioned by a PersistentVolume Provisioner.

The name of a StatefulSet object must be a valid DNS subdomain name.

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 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.

Depending on how DNS is configured in your cluster, you may not be able to look up the DNS name for a newly-run Pod immediately. This behavior can occur when other clients in the cluster have already sent queries for the hostname of the Pod before it was created. Negative caching (normal in DNS) means that the results of previous failed lookups are remembered and reused, even after the Pod is running, for at least a few seconds.

If you need to discover Pods promptly after they are created, you have a few options:

- Query the Kubernetes API directly (for example, using a watch) rather than relying on DNS lookups.
- Decrease the time of caching in your Kubernetes DNS provider (typically this means editing the config map for CoreDNS, which currently caches for 30 seconds).

As mentioned in the limitations section, you are responsible for creating the Headless Service 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.

Stable Storage

Kubernetes creates one PersistentVolume 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 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.

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, 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.

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 with the default Pod Management Policy (`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.

In this state, it's not enough to revert the Pod template to a good configuration. Due to a known issue, 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.

What's next

- Follow an example of deploying a stateful application.
- Follow an example of deploying Cassandra with Stateful Sets.
- Follow an example of running a replicated stateful application.

2.4 - DaemonSet

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 on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

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

Create a 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:

controllers/daemonset.yaml

```yaml
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:
      # this toleration is to have the daemonset runnable on master nodes
      # remove it if your masters can't run pods
      - 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 running stateless applications, configuring containers, and object management using kubectl documents.

The name of a DaemonSet object must be a valid DNS subdomain name.

A DaemonSet also needs a `.spec` section.

Pod Template

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

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a DaemonSet has to specify appropriate labels (see pod selector).

A Pod Template in a DaemonSet must have a `RestartPolicy` 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.

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.
- `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.

Running Pods on select Nodes

If you specify a `.spec.template.spec.nodeSelector`, then the DaemonSet controller will create Pods on nodes which match that node selector. Likewise if you specify a `.spec.template.spec.affinity`, then DaemonSet controller will create Pods on nodes which match that node affinity. 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

FEATURE STATE: `Kubernetes v1.20 [stable]`

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 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 original node affinity was taken into account before selecting the target host). 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, 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+	DaemonSet pods tolerate disk-pressure attributes by default scheduler.
`node.kubernetes.io/memory-pressure`	NoSchedule	1.8+	DaemonSet pods tolerate memory-pressure attributes by default scheduler.

Toleration Key	Effect	Version	Description
`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 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 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. 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 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.

2.5 - Jobs

A Job creates one or more Pods and will continue to retry execution of the Pods until 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

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

```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://kubernetes.io/examples/controllers/job.yaml
```

The output is similar to this:

```
job.batch/pi created
```

Check on the status of the Job with `kubectl`:

```
kubectl describe jobs/pi
```

The output is similar to this:

```
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":{}
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[*].metadat
echo $pods
```

The output is similar to this:

```
pi-5rwd7
```

Here, the selector is the same as the selector for the Job. The `--output=jsonpath` option specifies an expression with 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:

```
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986
```

Writing a Job spec

As with all other Kubernetes config, a Job needs `apiVersion`, `kind`, and `metadata` fields. Its name must be a valid DNS subdomain name.

A Job also needs a `.spec` section.

Pod Template

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

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a Job must specify appropriate labels (see pod selector) and an appropriate restart policy.

Only a `RestartPolicy` 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.

Parallel execution for 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 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 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 and container failures

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 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 when a Job's Pod is deleted or successful without any other Pods for the Job failing around that time.

> **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:

```yaml
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 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, the Jobs can be cleaned up by CronJobs based on the specified capacity-based cleanup policy.

TTL mechanism for finished Jobs

FEATURE STATE: `Kubernetes v1.12 [alpha]`

Another way to clean up finished Jobs (either `Complete` or `Failed`) automatically is to use a TTL mechanism provided by a TTL controller 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 for finished resources.

Job patterns

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			✓	✓

Pattern	Single Job object	Fewer pods than work items?	Use app unmodified?	Works in Kube 1.1?
Queue with Pod Per Work Item	✓		sometimes	✓
Queue with Variable Pod Count	✓	✓		✓
Single Job with Static Work Assignment	✓		✓	

When you specify completions with `.spec.completions`, each Pod created by the Job controller has an identical `spec`. 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	1	should be 1
Queue with Pod Per Work Item	W	any
Queue with Variable Pod Count	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
```

The output is similar to this:

```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.

```yaml
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. 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, `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), 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 maintains complete control over what Pods are created and how work is assigned to them.

Cron Jobs

You can use a `CronJob` to create a Job that will run at specified times/dates, similar to the Unix tool `cron`.

2.6 - 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

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

```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`:

```yaml
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.

Namespaced dependents can specify cluster-scoped or namespaced owners. A namespaced owner **must** exist in the same namespace as the dependent. If it does not, the owner reference is treated as absent, and the dependent is subject to deletion once all owners are verified absent.

Cluster-scoped dependents can only specify cluster-scoped owners. In v1.20+, if a cluster-scoped dependent specifies a namespaced kind as an owner, it is treated as having an unresolveable owner reference, and is not able to be garbage collected.

In v1.20+, if the garbage collector detects an invalid cross-namespace `ownerReference`, or a cluster-scoped dependent with an `ownerReference` referencing a namespaced kind, a warning Event with a reason of `OwnerRefInvalidNamespace` and an `involvedObject` of the invalid dependent is reported. You can check for that kind of Event by running `kubectl get events -A --field-selector=reason=OwnerRefInvalidNamespace`.

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 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".

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 in the foreground using kubectl, set `--cascade=foreground`. To orphan dependents, set `--cascade=orphan`.

The default behavior is to delete the dependents in the background which is the behavior when `--cascade` is omitted or explicitly set to `background`.

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

```
kubectl delete replicaset my-repset --cascade=orphan
```

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 for more information.

Known issues

Tracked at #26120

What's next

Design Doc 1

Design Doc 2

2.7 - TTL Controller for Finished Resources

FEATURE STATE: `Kubernetes v1.12 [alpha]`

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 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 `TTLAfterFinished`.

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. 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, that is to say it will 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 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 to set this field dynamically after the resource has finished, and choose different TTL values based on resource status, labels, etc.

Caveat

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) 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.

What's next

- Clean up Jobs automatically
- Design doc

2.8 - CronJob

FEATURE STATE: `Kubernetes v1.8 [beta]`

A *CronJob* creates Jobs on a repeating 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 format.

> **Caution:**
> All **CronJob** `schedule:` times are based on the timezone of the kube-controller-manager.
>
> If your control plane runs the kube-controller-manager in Pods or bare containers, the timezone set for the kube-controller-manager container determines the timezone that the cron job controller uses.

When creating the manifest for a CronJob resource, make sure the name you provide is a valid DNS subdomain name. The name must be 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.

CronJob

CronJobs are useful for creating periodic and recurring tasks, like running backups or sending emails. CronJobs can also schedule individual tasks for a specific time, such as scheduling a Job for when your cluster is likely to be idle.

Example

This example CronJob manifest prints the current time and a hello message every minute:

`application/job/cronjob.yaml`

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

(Running Automated Tasks with a CronJob takes you through this example in more detail).

Cron schedule syntax

```
# ┌───────────── minute (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of the month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                   7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * *
```

Entry	Description	Equivalent to
@yearly (or @annually)	Run once a year at midnight of 1 January	0 0 1 1 *
@monthly	Run once a month at midnight of the first day of the month	0 0 1 * *
@weekly	Run once a week at midnight on Sunday morning	0 0 * * 0
@daily (or @midnight)	Run once a day at midnight	0 0 * * *
@hourly	Run once an hour at the beginning of the hour	0 * * * *

For example, the line below states that the task must be started every Friday at midnight, as well as on the 13th of each month at midnight:

`0 0 13 * 5`

To generate CronJob schedule expressions, you can also use web tools like crontab.guru.

CronJob 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.

Caution: If `startingDeadlineSeconds` is set to a value less than 10 seconds, the CronJob may not be scheduled. This is because the CronJob controller checks things every 10 seconds.

For every CronJob, the CronJob 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
```

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.

New controller

There's an alternative implementation of the CronJob controller, available as an alpha feature since Kubernetes 1.20. To select version 2 of the CronJob controller, pass the following feature gate flag to the kube-controller-manager.

```
--feature-gates="CronJobControllerV2=true"
```

What's next

Cron expression format documents the format of CronJob `schedule` fields.

For instructions on creating and working with cron jobs, and for an example of CronJob manifest, see Running automated tasks with cron jobs.

2.9 - ReplicationController

Note: A `Deployment` that configures a `ReplicaSet` is now the recommended way to set up replication.

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

How a ReplicationController Works

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

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

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

Running an example ReplicationController

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

`controllers/replication.yaml`

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

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

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

The output is similar to this:

```
replicationcontroller/nginx created
```

Check on the status of the ReplicationController using this command:

```
kubectl describe replicationcontrollers/nginx
```

The output is similar to this:

```
Name:        nginx
Namespace:   default
Selector:    app=nginx
Labels:      app=nginx
Annotations:    <none>
Replicas:    3 current / 3 desired
Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=nginx
  Containers:
   nginx:
    Image:              nginx
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen       LastSeen     Count    From                         SubobjectPath
  ---------       --------     -----    ----                         -------------
  20s             20s          1        {replication-controller }
  20s             20s          1        {replication-controller }
  20s             20s          1        {replication-controller }
```

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.nar
echo $pods
```

The output is similar to this:

```
nginx-3ntk0 nginx-4ok8v nginx-qrm3m
```

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

Writing a ReplicationController Spec

As with all other Kubernetes config, a ReplicationController needs `apiVersion`, `kind`, and `metadata` fields. The name of a ReplicationController object must be a valid DNS subdomain name. For general information about working with config files, see object management .

A ReplicationController also needs a `.spec` section.

Pod Template

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

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a ReplicationController must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See pod selector.

Only a `.spec.template.spec.restartPolicy` equal to `Always` is allowed, which is the default if not specified.

For local container restarts, ReplicationControllers delegate to an agent on the node, for example the Kubelet 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. A ReplicationController manages all the pods with labels that match the selector. It does not distinguish between pods that it created or deleted and pods that another person or process created or deleted. This allows the ReplicationController to be replaced without affecting the running pods.

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

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

If you do end up with multiple controllers that have overlapping selectors, you will have to manage the deletion yourself (see below).

Multiple Replicas

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

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

Working with ReplicationControllers

Deleting a ReplicationController and its Pods

To delete a ReplicationController and all its pods, use `kubectl delete`. Kubectl will scale the ReplicationController to zero and wait for it to delete each pod before deleting the ReplicationController itself. If this kubectl command is interrupted, it can be restarted.

When using the REST API or Go client library, you need to do the steps explicitly (scale replicas to 0, wait for pod deletions, then delete the ReplicationController).

Deleting only a ReplicationController

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

Using kubectl, specify the `--cascade=false` option to `kubectl delete`.

When using the REST API or Go client library, you can delete the ReplicationController object.

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

Isolating pods from a ReplicationController

Pods may be removed from a ReplicationController's target set by changing their labels. This technique may be used to remove pods from service for debugging, 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 scales the number of replicas up or down by setting the `replicas` field. You can configure the ReplicationController to manage the replicas manually or by an auto-scaling control agent.

Rolling updates

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

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

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

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

Multiple release tracks

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

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

Using ReplicationControllers with Services

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

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

Writing programs for Replication

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

Responsibilities of the ReplicationController

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

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

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

API Object

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

Alternatives to ReplicationController

ReplicaSet

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

Deployment (Recommended)

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

Bare Pods

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

Job

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

DaemonSet

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

For more information

Read Run Stateless Application Deployment.