Edit This Page

Concepts

The Concepts section helps you learn about the parts of the Kubernetes system and the abstractions Kubernetes uses to represent your clusterA set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. , and helps you obtain a deeper understanding of how Kubernetes works.

- Overview
- Kubernetes objects
- Kubernetes Control Plane
- What's next

Overview

To work with Kubernetes, you use *Kubernetes API objects* to describe your cluster's *desired state*: what applications or other workloads you want to run, what container images they use, the number of replicas, what network and disk resources you want to make available, and more. You set your desired state by creating objects using the Kubernetes API, typically via the command-line interface, `kubectl`. You can also use the Kubernetes API directly to interact with the cluster and set or modify your desired state.

Once you've set your desired state, the *Kubernetes Control Plane* makes the cluster's current state match the desired state via the Pod Lifecycle Event Generator (PLEG). To do so, Kubernetes performs a variety of tasks automatically-such as starting or restarting containers, scaling the number of replicas of a given application, and more. The Kubernetes Control Plane consists of a collection of processes running on your cluster:

- The **Kubernetes Master** is a collection of three processes that run on a single node in your cluster, which is designated as the master node. Those processes are: kube-apiserver, kube-controller-manager and kube-scheduler.
- Each individual non-master node in your cluster runs two processes:
    - **kubelet**, which communicates with the Kubernetes Master.
    - **kube-proxy**, a network proxy which reflects Kubernetes networking services on each node.

Kubernetes objects

Kubernetes contains a number of abstractions that represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is doing. These abstractions are represented by objects in the Kubernetes API. See Understanding Kubernetes objects for more details.

The basic Kubernetes objects include:

- Pod
- Service
- Volume
- Namespace

Kubernetes also contains higher-level abstractions that rely on controllers to build upon the basic objects, and provide additional functionality and convenience features. These include:

- Deployment
- DaemonSet
- StatefulSet
- ReplicaSet
- Job

Kubernetes Control Plane

The various parts of the Kubernetes Control Plane, such as the Kubernetes Master and kubelet processes, govern how Kubernetes communicates with your cluster. The Control Plane maintains a record of all of the Kubernetes Objects in the system, and runs continuous control loops to manage those objects' state. At any given time, the Control Plane's control loops will respond to changes in the cluster and work to make the actual state of all the objects in the system match the desired state that you provided.

For example, when you use the Kubernetes API to create a Deployment, you provide a new desired state for the system. The Kubernetes Control Plane records that object creation, and carries out your instructions by starting the required applications and scheduling them to cluster nodes-thus making the cluster's actual state match the desired state.

Kubernetes Master

The Kubernetes master is responsible for maintaining the desired state for your cluster. When you interact with Kubernetes, such as by using the `kubectl` command-line interface, you're communicating with your cluster's Kubernetes master.

> The "master" refers to a collection of processes managing the cluster state. Typically all these processes run on a single node in the cluster, and this node is also referred to as the master. The master can also be replicated for availability and redundancy.

Kubernetes Nodes

The nodes in a cluster are the machines (VMs, physical servers, etc) that run your applications and cloud workflows. The Kubernetes master controls each node; you'll rarely interact with nodes directly.

What's next

If you would like to write a concept page, see Using Page Templates for information about the concept page type and the concept template.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 11, 2020 at 3:24 AM PST by Tweak linking for Kubernetes object concept (#18122) (Page History)

Edit This Page

What is Kubernetes

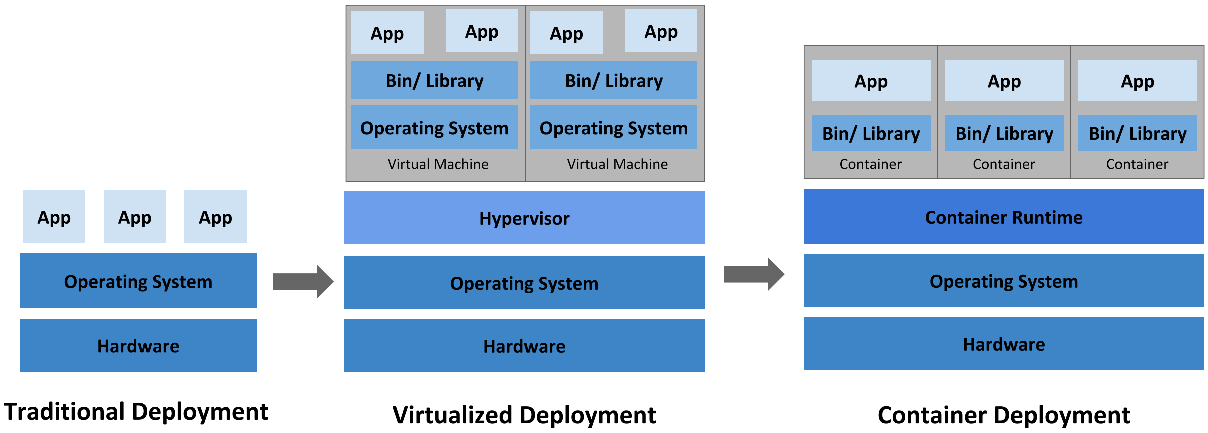
This page is an overview of Kubernetes.

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. Google open-sourced the Kubernetes project in 2014. Kubernetes builds upon a decade and a half of experience that Google has with running production workloads at scale, combined with best-of-breed ideas and practices from the community.

Going back in time

Let's take a look at why Kubernetes is so useful by going back in time.



Traditional deployment era: Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

Virtualized deployment era: As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

Container deployment era: Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
- Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- Observability not only surfaces OS-level information and metrics, but also application health and other signals.
- Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-prem, Google Kubernetes Engine, and anywhere else.
- Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically - not a monolithic stack running on one big single-purpose machine.
- Resource isolation: predictable application performance.
- Resource utilization: high efficiency and density.

Why you need Kubernetes and what can it do

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

- **Service discovery and load balancing**
  Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration**
  Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks**
  You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing**
  You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing**
  Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management**
  Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

What Kubernetes is not

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, and monitoring. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:

- Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
- Does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
- Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.
- Does not dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
- Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
- Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

What's next

- Take a look at the Kubernetes Components
- Ready to Get Started?

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on November 25, 2019 at 11:41 PM PST by Minor heading update according to the guideline (#17747) (Page History)

Edit This Page

Kubernetes Components

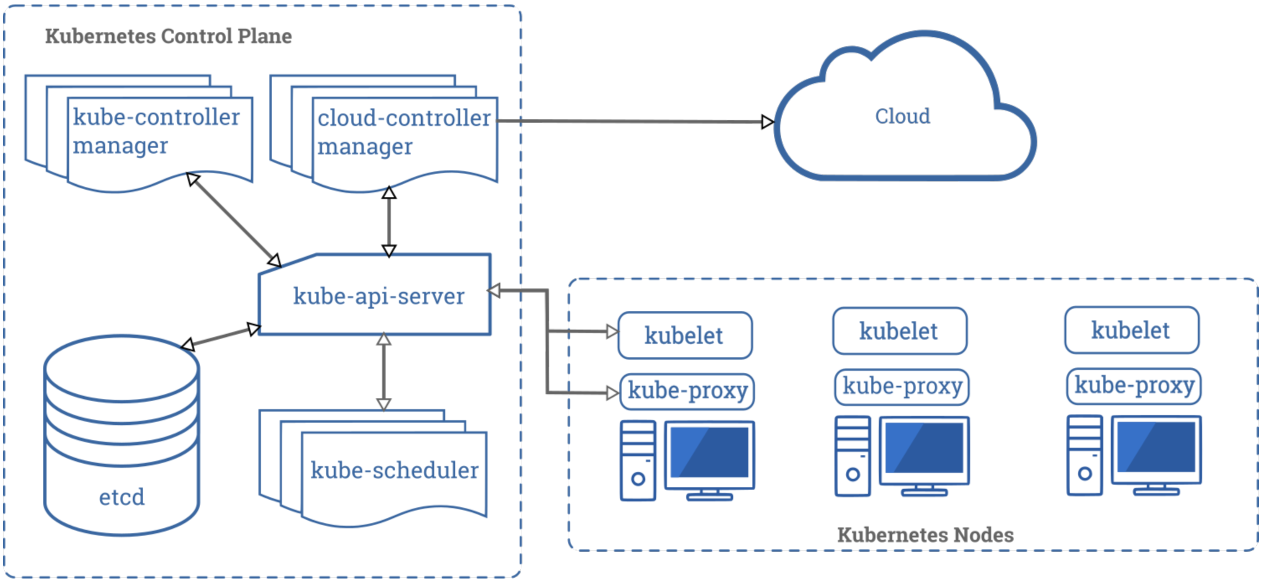
When you deploy Kubernetes, you get a cluster.

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the pods that are the components of the application. The Control Plane manages the worker nodes and the pods in the cluster. In production environments, the Control Plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

This document outlines the various components you need to have a complete and working Kubernetes cluster.

Here's the diagram of a Kubernetes cluster with all the components tied together.



Control Plane Components

The Control Plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new podThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. when a deployment's `replicas` field is unsatisfied).

Control Plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all Control Plane components on the same machine, and do not run user containers on this machine. See Building High-Availability Clusters for an example multi-master-VM setup.

kube-apiserver

The API server is a component of the Kubernetes control planeThe container orchestration layer that exposes the API and interfaces to define, deploy, and manage the lifecycle of containers. that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontallyâ€"that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for those data.

You can find in-depth information about etcd in the official documentation.

kube-scheduler

Control Plane component that watches for newly created pods with no assigned node, and selects a node for them to run on.

Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

kube-controller-manager

Control Plane component that runs controllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. processes.

Logically, each controllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

- Node Controller: Responsible for noticing and responding when nodes go down.
- Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).

- Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.

cloud-controller-manager

cloud-controller-manager runs controllers that interact with the underlying cloud providers. The cloud-controller-manager binary is an alpha feature introduced in Kubernetes release 1.6.

cloud-controller-manager runs cloud-provider-specific controller loops only. You must disable these controller loops in the kube-controller-manager. You can disable the controller loops by setting the `--cloud-provider` flag to `external` when starting the kube-controller-manager.

cloud-controller-manager allows the cloud vendor's code and the Kubernetes code to evolve independently of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to cloud-controller-manager while running Kubernetes.

The following controllers have cloud provider dependencies:

- Node Controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- Route Controller: For setting up routes in the underlying cloud infrastructure
- Service Controller: For creating, updating and deleting cloud provider load balancers
- Volume Controller: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes ServiceA way to expose an application running on a set of Pods as a network service. concept.

kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container Runtime

The container runtime is the software that is responsible for running containers.

Kubernetes supports several container runtimes: DockerDocker is a software technology providing operating-system-level virtualization also known as containers. , containerdA container runtime with an emphasis on simplicity, robustness and portability , CRI-OA lightweight container runtime specifically for Kubernetes , and any implementation of the Kubernetes CRI (Container Runtime Interface).

Addons

Addons use Kubernetes resources (DaemonSetEnsures a copy of a Pod is running across a set of nodes in a cluster. , DeploymentAn API object that manages a replicated application. , etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the `kube-system` namespace.

Selected addons are described below; for an extended list of available addons, please see Addons.

DNS

While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it.

Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard)

Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

Container Resource Monitoring

Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

Cluster-level Logging

A cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

What's next

- Learn about Nodes
- Learn about Controllers
- Learn about kube-scheduler
- Read etcd's official documentation

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 16, 2020 at 2:24 PM PST by Use Control Plane instead of Master (#18438) (Page History)

Edit This Page

The Kubernetes API

Overall API conventions are described in the API conventions doc.

API endpoints, resource types and samples are described in API Reference.

Remote access to the API is discussed in the Controlling API Access doc.

The Kubernetes API also serves as the foundation for the declarative configuration schema for the system. The kubectl command-line tool can be used to create, update, delete, and get API objects.

Kubernetes also stores its serialized state (currently in etcd) in terms of the API resources.

Kubernetes itself is decomposed into multiple components, which interact through its API.

API changes

In our experience, any system that is successful needs to grow and change as new use cases emerge or existing ones change. Therefore, we expect the Kubernetes API to continuously change and grow. However, we intend to not break compatibility with existing clients, for an extended period of time. In general, new API resources and new resource fields can be expected to be added frequently. Elimination of resources or fields will require following the API deprecation policy.

What constitutes a compatible change and how to change the API are detailed by the API change document.

OpenAPI and Swagger definitions

Complete API details are documented using OpenAPI.

Starting with Kubernetes 1.10, the Kubernetes API server serves an OpenAPI spec via the `/openapi/v2` endpoint. The requested format is specified by setting HTTP headers:

Header	**Possible Values**
Accept	`application/json`, `application/com.github.proto-openapi.spec.v2@v1.0+protobuf` (the default content-type is `application/json` for `*/*` or not passing this header)
Accept-Encoding	`gzip` (not passing this header is acceptable)

Prior to 1.14, format-separated endpoints (`/swagger.json`, `/swagger-2.0.0.json`, `/swagger-2.0.0.pb-v1`, `/swagger-2.0.0.pb-v1.gz`) serve the OpenAPI spec in different formats. These endpoints are deprecated, and are removed in Kubernetes 1.14.

Examples of getting OpenAPI spec:

Before 1.10	**Starting with Kubernetes 1.10**
GET /swagger.json	GET /openapi/v2 **Accept**: application/json
GET /swagger-2.0.0.pb-v1	GET /openapi/v2 **Accept**: application/com.github.proto-openapi.spec.v2@v1.0+protobuf
GET /swagger-2.0.0.pb-v1.gz	GET /openapi/v2 **Accept**: application/com.github.proto-openapi.spec.v2@v1.0+protobuf **Accept-Encoding**: gzip

Kubernetes implements an alternative Protobuf based serialization format for the API that is primarily intended for intra-cluster communication, documented in the design proposal and the IDL files for each schema are located in the Go packages that define the API objects.

Prior to 1.14, the Kubernetes apiserver also exposes an API that can be used to retrieve the Swagger v1.2 Kubernetes API spec at `/swaggerapi`. This endpoint is deprecated, and was removed in Kubernetes 1.14.

API versioning

To make it easier to eliminate fields or restructure resource representations, Kubernetes supports multiple API versions, each at a different API path, such as `/api/v1` or `/apis/extensions/v1beta1`.

We chose to version at the API level rather than at the resource or field level to ensure that the API presents a clear, consistent view of system resources and behavior, and to enable controlling access to end-of-life and/or experimental APIs. The JSON and Protobuf serialization schemas follow the same guidelines for schema changes - all descriptions below cover both formats.

Note that API versioning and Software versioning are only indirectly related. The API and release versioning proposal describes the relationship between API versioning and software versioning.

Different API versions imply different levels of stability and support. The criteria for each level are described in more detail in the API Changes documentation. They are summarized here:

- Alpha level:
  - The version names contain `alpha` (e.g. `v1alpha1`).
  - May be buggy. Enabling the feature may expose bugs. Disabled by default.
  - Support for feature may be dropped at any time without notice.
  - The API may change in incompatible ways in a later software release without notice.
  - Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.
- Beta level:
  - The version names contain `beta` (e.g. `v2beta3`).

  - Code is well tested. Enabling the feature is considered safe. Enabled by default.
  - Support for the overall feature will not be dropped, though details may change.
  - The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
  - Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters which can be upgraded independently, you may be able to relax this restriction.
  - **Please do try our beta features and give feedback on them! Once they exit beta, it may not be practical for us to make more changes.**
- Stable level:
  - The version name is `vX` where `X` is an integer.
  - Stable versions of features will appear in released software for many subsequent versions.

API groups

To make it easier to extend the Kubernetes API, we implemented *API groups*. The API group is specified in a REST path and in the `apiVersion` field of a serialized object.

Currently there are several API groups in use:

1. The *core* group, often referred to as the *legacy group*, is at the REST path `/api/v1` and uses `apiVersion: v1`.

2. The named groups are at REST path `/apis/$GROUP_NAME/$VERSION`, and use `apiVersion: $GROUP_NAME/$VERSION` (e.g. `apiVersion: batch/v1`). Full list of supported API groups can be seen in Kubernetes API reference.

There are two supported paths to extending the API with custom resources:

1. CustomResourceDefinition is for users with very basic CRUD needs.
2. Users needing the full set of Kubernetes API semantics can implement their own apiserver and use the aggregator to make it seamless for clients.

Enabling API groups

Certain resources and API groups are enabled by default. They can be enabled or disabled by setting `--runtime-config` on apiserver. `--runtime-config` accepts comma separated values. For ex: to disable batch/v1, set `--runtime-config=batch/v1=false`, to enable batch/v2alpha1, set `--`

`runtime-config=batch/v2alpha1`. The flag accepts comma separated set of key=value pairs describing runtime configuration of the apiserver.

IMPORTANT: Enabling or disabling groups or resources requires restarting apiserver and controller-manager to pick up the `--runtime-config` changes.

Enabling resources in the groups

DaemonSets, Deployments, HorizontalPodAutoscalers, Ingresses, Jobs and ReplicaSets are enabled by default. Other extensions resources can be enabled by setting `--runtime-config` on apiserver. `--runtime-config` accepts comma separated values. For example: to disable deployments and ingress, set `--runtime-config=extensions/v1beta1/deployments=false,extensions/v1beta1/ingresses=false`

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 15, 2020 at 6:11 AM PST by Update kubernetes-api.md (#17406) (Page History)

Edit This Page

Understanding Kubernetes Objects

This page explains how Kubernetes objects are represented in the Kubernetes API, and how you can express them in `.yaml` format.

Understanding Kubernetes objects

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- What containerized applications are running (and on which nodes)
- The resources available to those applications
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object is a "record of intent"-once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's *desired state*.

To work with Kubernetes objects-whether to create, modify, or delete them-you'll need to use the Kubernetes API. When you use the `kubectl` command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

Object Spec and Status

Every Kubernetes object includes two nested object fields that govern the object's configuration: the object *spec* and the object *status*. The *spec*, which you must provide, describes your desired state for the object-the characteristics that you want the object to have. The *status* describes the *actual state* of the object, and is supplied and updated by the Kubernetes system. At any given time, the Kubernetes Control Plane actively manages an object's actual state to match the desired state you supplied.

For example, a Kubernetes Deployment is an object that can represent an application running on your cluster. When you create the Deployment, you might set the Deployment spec to specify that you want three replicas of the application to be running. The Kubernetes system reads the Deployment spec and starts three instances of your desired application-updating the status to match your spec. If any of those instances should fail (a status change), the Kubernetes system responds to the difference between spec and status by making a correction-in this case, starting a replacement instance.

For more information on the object spec, status, and metadata, see the Kubernetes API Conventions.

Describing a Kubernetes object

When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name). When you use the Kubernetes API to create the object (either directly or via `kubectl`), that API request must include that information as JSON in the request body. **Most often, you provide the information to `kubectl` in a .yaml file.** `kubectl` converts the information to JSON when making the API request.

Here's an example `.yaml` file that shows the required fields and object spec for a Kubernetes Deployment:

application/deployment.yaml

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

One way to create a Deployment using a `.yaml` file like the one above is to use the `kubectl apply` command in the `kubectl` command-line interface, passing the `.yaml` file as an argument. Here's an example:

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml --record
```

The output is similar to this:

```
deployment.apps/nginx-deployment created
```

Required Fields

In the `.yaml` file for the Kubernetes object you want to create, you'll need to set values for the following fields:

- `apiVersion` - Which version of the Kubernetes API you're using to create this object
- `kind` - What kind of object you want to create
- `metadata` - Data that helps uniquely identify the object, including a `name` string, `UID`, and optional `namespace`

- `spec` - What state you desire for the object

The precise format of the object `spec` is different for every Kubernetes object, and contains nested fields specific to that object. The Kubernetes API Reference can help you find the spec format for all of the objects you can create using Kubernetes. For example, the `spec` format for a Pod can be found in PodSpec v1 core, and the `spec` format for a Deployment can be found in DeploymentSpec v1 apps.

What's next

- Kubernetes API overview explains some more API concepts
- Learn about the most important basic Kubernetes objects, such as Pod.
- Learn about controllers in Kubernetes

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 16, 2020 at 2:21 PM PST by Add missing "in" for DeploymentSpec link (#19135) (Page History)

Edit This Page

Kubernetes Object Management

The `kubectl` command-line tool supports several different ways to create and manage Kubernetes objects. This document provides an overview of the different approaches. Read the Kubectl book for details of managing objects by Kubectl.

Management techniques

> **Warning:** A Kubernetes object should be managed using only one technique. Mixing and matching techniques for the same object results in undefined behavior.

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Imperative commands	Live objects	Development projects	1+	Lowest
Imperative object configuration	Individual files	Production projects	1	Moderate
Declarative object configuration	Directories of files	Production projects	1+	Highest

Imperative commands

When using imperative commands, a user operates directly on live objects in a cluster. The user provides operations to the `kubectl` command as arguments or flags.

This is the simplest way to get started or to run a one-off task in a cluster. Because this technique operates directly on live objects, it provides no history of previous configurations.

Examples

Run an instance of the nginx container by creating a Deployment object:

```
kubectl run nginx --image nginx
```

Do the same thing using a different syntax:

```
kubectl create deployment nginx --image nginx
```

Trade-offs

Advantages compared to object configuration:

- Commands are simple, easy to learn and easy to remember.
- Commands require only a single step to make changes to the cluster.

Disadvantages compared to object configuration:

- Commands do not integrate with change review processes.
- Commands do not provide an audit trail associated with changes.
- Commands do not provide a source of records except for what is live.

- Commands do not provide a template for creating new objects.

Imperative object configuration

In imperative object configuration, the kubectl command specifies the operation (create, replace, etc.), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

See the API reference for more details on object definitions.

> **Warning:** The imperative `replace` command replaces the existing spec with the newly provided one, dropping all changes to the object missing from the configuration file. This approach should not be used with resource types whose specs are updated independently of the configuration file. Services of type `LoadBalancer`, for example, have their `externalIPs` field updated independently from the configuration by the cluster.

Examples

Create the objects defined in a configuration file:

```
kubectl create -f nginx.yaml
```

Delete the objects defined in two configuration files:

```
kubectl delete -f nginx.yaml -f redis.yaml
```

Update the objects defined in a configuration file by overwriting the live configuration:

```
kubectl replace -f nginx.yaml
```

Trade-offs

Advantages compared to imperative commands:

- Object configuration can be stored in a source control system such as Git.
- Object configuration can integrate with processes such as reviewing changes before push and audit trails.
- Object configuration provides a template for creating new objects.

Disadvantages compared to imperative commands:

- Object configuration requires basic understanding of the object schema.
- Object configuration requires the additional step of writing a YAML file.

Advantages compared to declarative object configuration:

- Imperative object configuration behavior is simpler and easier to understand.
- As of Kubernetes version 1.5, imperative object configuration is more mature.

Disadvantages compared to declarative object configuration:

- Imperative object configuration works best on files, not directories.
- Updates to live objects must be reflected in configuration files, or they will be lost during the next replacement.

Declarative object configuration

When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files. Create, update, and delete operations are automatically detected per-object by `kubectl`. This enables working on directories, where different operations might be needed for different objects.

> **Note:** Declarative object configuration retains changes made by other writers, even if the changes are not merged back to the object configuration file. This is possible by using the `patch` API operation to write only observed differences, instead of using the `replace` API operation to replace the entire object configuration.

Examples

Process all object configuration files in the `configs` directory, and create or patch the live objects. You can first `diff` to see what changes are going to be made, and then apply:

```
kubectl diff -f configs/
kubectl apply -f configs/
```

Recursively process directories:

```
kubectl diff -R -f configs/
kubectl apply -R -f configs/
```

Trade-offs

Advantages compared to imperative object configuration:

- Changes made directly to live objects are retained, even if they are not merged back into the configuration files.
- Declarative object configuration has better support for operating on directories and automatically detecting operation types (create, patch, delete) per-object.

Disadvantages compared to imperative object configuration:

- Declarative object configuration is harder to debug and understand results when they are unexpected.
- Partial updates using diffs create complex merge and patch operations.

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Managing Kubernetes Objects Using Object Configuration (Imperative)
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Managing Kubernetes Objects Using Kustomize (Declarative)
- Kubectl Command Reference
- Kubectl Book
- Kubernetes API Reference

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Object Names and IDs

Each object in your cluster has a *Name* that is unique for that type of resource. Every Kubernetes object also has a *UID* that is unique across your whole cluster.

For example, you can only have one Pod named `myapp-1234` within the same namespace, but you can have one Pod and one Deployment that are each named `myapp-1234`.

For non-unique user-provided attributes, Kubernetes provides labels and annotations.

- Names
- UIDs
- What's next

Names

A client-provided string that refers to an object in a resource URL, such as `/api/v1/pods/some-name`.

Only one object of a given kind can have a given name at a time. However, if you delete the object, you can make a new object with the same name.

Below are three types of commonly used name constraints for resources.

DNS Subdomain Names

Most resource types require a name that can be used as a DNS subdomain name as defined in RFC 1123. This means the name must:

- contain no more than 253 characters
- contain only lowercase alphanumeric characters, â€˜-' or â€˜.'
- start with an alphanumeric character
- end with an alphanumeric character

DNS Label Names

Some resource types require their names to follow the DNS label standard as defined in RFC 1123. This means the name must:

- contain at most 63 characters
- contain only lowercase alphanumeric characters or â€˜-'
- start with an alphanumeric character
- end with an alphanumeric character

Path Segment Names

Some resource types require their names to be able to be safely encoded as a path segment. In other words, the name may not be "." or ".." and the name may not contain "/" or "%".

Here's an example manifest for a Pod named `nginx-demo`.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
spec:
  containers:
```

```
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

> **Note:** Some resource types have additional restrictions on their names.

UIDs

A Kubernetes systems-generated string to uniquely identify objects.

Every object created over the whole lifetime of a Kubernetes cluster has a distinct UID. It is intended to distinguish between historical occurrences of similar entities.

Kubernetes UIDs are universally unique identifiers (also known as UUIDs). UUIDs are standardized as ISO/IEC 9834-8 and as ITU-T X.667.

What's next

- Read about labels in Kubernetes.
- See the Identifiers and Names in Kubernetes design document.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page

Page last modified on February 20, 2020 at 7:44 AM PST by Resource name constraints (1) (#19106) (Page History)

Edit This Page

Namespaces

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

When to Use Multiple Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces can not be nested inside one another and each Kubernetes resource can only be in one namespace.

Namespaces are a way to divide cluster resources between multiple users (via resource quota).

In future versions of Kubernetes, objects in the same namespace will have the same access control policies by default.

It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software: use labels to distinguish resources within the same namespace.

Working with Namespaces

Creation and deletion of namespaces are described in the Admin Guide documentation for namespaces.

Viewing namespaces

You can list the current namespaces in a cluster using:

```
kubectl get namespace
```

```
NAME          STATUS    AGE
default       Active    1d
kube-system   Active    1d
kube-public   Active    1d
```

Kubernetes starts with three initial namespaces:

- `default` The default namespace for objects with no other namespace
- `kube-system` The namespace for objects created by the Kubernetes system
- `kube-public` This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

Setting the namespace for a request

To set the namespace for a current request, use the `--namespace` flag.

For example:

```
kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
kubectl get pods --namespace=<insert-namespace-name-here>
```

Setting the namespace preference

You can permanently save the namespace for all subsequent kubectl commands in that context.

```
kubectl config set-context --current --namespace=<insert-namespace-name-here>
# Validate it
kubectl config view --minify | grep namespace:
```

Namespaces and DNS

When you create a Service, it creates a corresponding DNS entry. This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container just uses `<service-name>`, it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

Not All Objects are in a Namespace

Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. However namespace resources are not themselves in a namespace. And low-level resources, such as nodes and persistentVolumes, are not in any namespace.

To see which Kubernetes resources are and aren't in a namespace:

```
# In a namespace
kubectl api-resources --namespaced=true

# Not in a namespace
kubectl api-resources --namespaced=false
```

What's next

- Learn more about creating a new namespace.
- Learn more about deleting a namespace.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Labels and Selectors

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object.

```
"metadata": {
  "labels": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

Labels allow for efficient queries and watches and are ideal for use in UIs and CLIs. Non-identifying information should be recorded using annotations.

Motivation

Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings.

Service deployments and batch processing pipelines are often multi-dimensional entities (e.g., multiple partitions or deployments, multiple release tracks, multiple tiers, multiple micro-services per tier). Management often requires cross-cutting operations, which breaks encapsulation of strictly hierarchical representations, especially rigid hierarchies determined by the infrastructure rather than by users.

Example labels:

- `"release" : "stable"`, `"release" : "canary"`
- `"environment" : "dev"`, `"environment" : "qa"`, `"environment" : "production"`
- `"tier" : "frontend"`, `"tier" : "backend"`, `"tier" : "cache"`
- `"partition" : "customerA"`, `"partition" : "customerB"`
- `"track" : "daily"`, `"track" : "weekly"`

These are just examples of commonly used labels; you are free to develop your own conventions. Keep in mind that label Key must be unique for a given object.

Syntax and character set

Labels are key/value pairs. Valid label keys have two segments: an optional prefix and name, separated by a slash (`/`). The name segment is required and must be 63 characters or less, beginning and ending with an alphanumeric character (`[a-z0-9A-Z]`) with dashes (`-`), underscores (`_`), dots (`.`), and alphanumerics between. The prefix is optional. If specified, the prefix must be a DNS subdomain: a series of DNS labels separated by dots (`.`), not longer than 253 characters in total, followed by a slash (`/`).

If the prefix is omitted, the label Key is presumed to be private to the user. Automated system components (e.g. `kube-scheduler`, `kube-controller-manager`, `kube-apiserver`, `kubectl`, or other third-party automation) which add labels to end-user objects must specify a prefix.

The `kubernetes.io/` and `k8s.io/` prefixes are reserved for Kubernetes core components.

Valid label values must be 63 characters or less and must be empty or begin and end with an alphanumeric character (`[a-z0-9A-Z]`) with dashes (`-`), underscores (`_`), dots (`.`), and alphanumerics between.

For example, here's the configuration file for a Pod that has two labels `environment: production` and `app: nginx` :

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

Label selectors

Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

The API currently supports two types of selectors: *equality-based* and *set-based*. A label selector can be made of multiple *requirements* which are comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as a logical *AND* (&&) operator.

The semantics of empty or non-specified selectors are dependent on the context, and API types that use selectors should document the validity and meaning of them.

> **Note:** For some API types, such as ReplicaSets, the label selectors of two instances must not overlap within a namespace, or the controller can see that as conflicting instructions and fail to determine how many replicas should be present.

> **Caution:** For both equality-based and set-based conditions there is no logical *OR* (`||`) operator. Ensure your filter statements are structured accordingly.

Equality-based requirement

Equality- or *inequality-based* requirements allow filtering by label keys and values. Matching objects must satisfy all of the specified label constraints, though they may have additional labels as well. Three kinds of operators are admitted `=`,`==`,`!=`. The first two represent *equality* (and are simply synonyms), while the latter represents *inequality*. For example:

```
environment = production
tier != frontend
```

The former selects all resources with key equal to `environment` and value equal to `production`. The latter selects all resources with key equal to `tier` and value distinct from `frontend`, and all resources with no labels with the `tier` key. One could filter for resources in `production` excluding `frontend` using the comma operator: `environment=production,tier!=frontend`

One usage scenario for equality-based label requirement is for Pods to specify node selection criteria. For example, the sample Pod below selects nodes with the label "`accelerator=nvidia-tesla-p100`".

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-test
spec:
  containers:
    - name: cuda-test
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100
```

Set-based requirement

Set-based label requirements allow filtering keys according to a set of values. Three kinds of operators are supported: `in`,`notin` and `exists` (only the key identifier). For example:

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

The first example selects all resources with key equal to `environment` and value equal to `production` or `qa`. The second example selects all resources with key equal to `tier` and values other than `frontend` and `backend`, and all resources with no labels with the `tier` key. The third example selects all resources including a label with key `partition`; no values are checked. The fourth example selects all resources without a label with key `partition`; no values are checked. Similarly the comma separator acts as an *AND* operator. So filtering resources with a `partition` key (no matter the value) and with `environment` different thanÂ `qa` can be achieved using `partition,environment notin (qa)`. The *set-based* label selector is a general form of equality since `environment=production` is equivalent to `environment in (production)`; similarly for `!=` and `notin`.

Set-based requirements can be mixed with *equality-based* requirements. For example: `partition in (customerA, customerB),environment!=qa`.

API

LIST and WATCH filtering

LIST and WATCH operations may specify label selectors to filter the sets of objects returned using a query parameter. Both requirements are permitted (presented here as they would appear in a URL query string):

- *equality-based* requirements: `?labelSelector=environment%3Dproduction,tier%3Dfrontend`
- *set-based* requirements: `?labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in+%28frontend%29`

Both label selector styles can be used to list or watch resources via a REST client. For example, targeting `apiserver` with `kubectl` and using *equality-based* one may write:

```
kubectl get pods -l environment=production,tier=frontend
```

or using *set-based* requirements:

```
kubectl get pods -l 'environment in (production),tier in (frontend)'
```

As already mentioned *set-based* requirements are more expressive.Â For instance, they can implement the *OR* operator on values:

```
kubectl get pods -l 'environment in (production, qa)'
```

or restricting negative matching via *exists* operator:

```
kubectl get pods -l 'environment,environment notin (frontend)'
```

Set references in API objects

Some Kubernetes objects, such as `services` and `replicationcontrollers`, also use label selectors to specify sets of other resources, such as `pods`.

Service and ReplicationController

The set of pods that a `service` targets is defined with a label selector. Similarly, the population of pods that a `replicationcontroller` should manage is also defined with a label selector.

Labels selectors for both objects are defined in `json` or `yaml` files using maps, and only *equality-based* requirement selectors are supported:

```
"selector": {
    "component" : "redis",
}
```

or

```
selector:
    component: redis
```

this selector (respectively in `json` or `yaml` format) is equivalent to `component=redis` or `component in (redis)`.

Resources that support set-based requirements

Newer resources, such as `Job`, `Deployment`, `ReplicaSet`, and `DaemonSet`, support *set-based* requirements as well.

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

`matchLabels` 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". `matchExpressions` is a list of pod selector requirements. Valid operators include In, NotIn, Exists, and DoesNotExist. The values set must be non-empty in the case of In and NotIn. All of the requirements, from both `matchLabels` and `matchExpressions` are ANDed together - they must all be satisfied in order to match.

Selecting sets of nodes

One use case for selecting over labels is to constrain the set of nodes onto which a pod can schedule. See the documentation on node selection for more information.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 11, 2020 at 11:18 AM PST by Replace links with redirect destination (#19038) (Page History)

Edit This Page

Annotations

You can use Kubernetes annotations to attach arbitrary non-identifying metadata to objects. Clients such as tools and libraries can retrieve this metadata.

Attaching metadata to objects

You can use either labels or annotations to attach metadata to Kubernetes objects. Labels can be used to select objects and to find collections of objects that satisfy certain conditions. In contrast, annotations are not used to identify and select objects. The metadata in an annotation can be small or large, structured or unstructured, and can include characters not permitted by labels.

Annotations, like labels, are key/value maps:

```
"metadata": {
  "annotations": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

Here are some examples of information that could be recorded in annotations:

- Fields managed by a declarative configuration layer. Attaching these fields as annotations distinguishes them from default values set by clients or servers, and from auto-generated fields and fields set by auto-sizing or auto-scaling systems.

- Build, release, or image information like timestamps, release IDs, git branch, PR numbers, image hashes, and registry address.

- Pointers to logging, monitoring, analytics, or audit repositories.

- Client library or tool information that can be used for debugging purposes: for example, name, version, and build information.

- User or tool/system provenance information, such as URLs of related objects from other ecosystem components.

- Lightweight rollout tool metadata: for example, config or checkpoints.

- Phone or pager numbers of persons responsible, or directory entries that specify where that information can be found, such as a team web site.

- Directives from the end-user to the implementations to modify behavior or engage non-standard features.

Instead of using annotations, you could store this type of information in an external database or directory, but that would make it much harder to produce shared client libraries and tools for deployment, management, introspection, and the like.

Syntax and character set

Annotations are key/value pairs. Valid annotation keys have two segments: an optional prefix and name, separated by a slash (`/`). The name segment is required and must be 63 characters or less, beginning and ending with an alphanumeric character (`[a-z0-9A-Z]`) with dashes (`-`), underscores (`_`), dots (`.`), and alphanumerics between. The prefix is optional. If specified, the prefix must be a DNS subdomain: a series of DNS labels separated by dots (`.`), not longer than 253 characters in total, followed by a slash (`/`).

If the prefix is omitted, the annotation Key is presumed to be private to the user. Automated system components (e.g. `kube-scheduler`, `kube-controller-manager`, `kube-apiserver`, `kubectl`, or other third-party automation) which add annotations to end-user objects must specify a prefix.

The `kubernetes.io/` and `k8s.io/` prefixes are reserved for Kubernetes core components.

For example, here's the configuration file for a Pod that has the annotation `imageregistry: https://hub.docker.com/` :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

What's next

Learn more about Labels and Selectors.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on June 06, 2019 at 11:24 AM PST by Adding annotation example (#14440) (Page History)

Edit This Page

Field Selectors

Field selectors let you select Kubernetes resources based on the value of one or more resource fields. Here are some examples of field selector queries:

- `metadata.name=my-service`
- `metadata.namespace!=default`
- `status.phase=Pending`

This `kubectl` command selects all Pods for which the value of the `status.phase` field is `Running`:

```
kubectl get pods --field-selector status.phase=Running
```

> **Note:**
>
> Field selectors are essentially resource *filters*. By default, no selectors/filters are applied, meaning that all resources of the specified type are selected. This makes the following `kubectl` queries equivalent:
>
> ```
> kubectl get pods
> kubectl get pods --field-selector ""
> ```

Supported fields

Supported field selectors vary by Kubernetes resource type. All resource types support the `metadata.name` and `metadata.namespace` fields. Using unsupported field selectors produces an error. For example:

```
kubectl get ingress --field-selector foo.bar=baz
```

```
Error from server (BadRequest): Unable to find "ingresses" that
match label selector "", field selector "foo.bar=baz": "foo.bar"
is not a known field selector: only "metadata.name",
"metadata.namespace"
```

Supported operators

You can use the `=`, `==`, and `!=` operators with field selectors (`=` and `==` mean the same thing). This `kubectl` command, for example, selects all Kubernetes Services that aren't in the `default` namespace:

```
kubectl get services  --all-namespaces --field-selector
metadata.namespace!=default
```

Chained selectors

As with label and other selectors, field selectors can be chained together as a comma-separated list. This `kubectl` command selects all Pods for which the `status.phase` does not equal `Running` and the `spec.restartPolicy` field equals `Always`:

```
kubectl get pods --field-selector=status.phase!=Running,spec.restartPolicy=Always
```

Multiple resource types

You use field selectors across multiple resource types. This `kubectl` command selects all Statefulsets and Services that are not in the `default` namespace:

```
kubectl get statefulsets,services --all-namespaces --field-selector metadata.namespace!=default
```

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 18, 2020 at 10:11 PM PST by Update field-selectors.md (#18641) (Page History)

Edit This Page

Recommended Labels

You can visualize and manage Kubernetes objects with more tools than kubectl and the dashboard. A common set of labels allows tools to work interoperably, describing objects in a common manner that all tools can understand.

In addition to supporting tooling, the recommended labels describe applications in a way that can be queried.

- Labels
- Applications And Instances Of Applications
- Examples

The metadata is organized around the concept of an *application*. Kubernetes is not a platform as a service (PaaS) and doesn't have or enforce a formal notion of an application. Instead, applications are informal and described with metadata. The definition of what an application contains is loose.

> **Note:** These are recommended labels. They make it easier to manage applications but aren't required for any core tooling.

Shared labels and annotations share a common prefix: `app.kubernetes.io`. Labels without a prefix are private to users. The shared prefix ensures that shared labels do not interfere with custom user labels.

Labels

In order to take full advantage of using these labels, they should be applied on every resource object.

Key	**Description**	**Example**	**Type**
`app.kubernetes.io/name`	The name of the application	`mysql`	string
`app.kubernetes.io/instance`	A unique name identifying the instance of an application	`wordpress-abcxzy`	string
`app.kubernetes.io/version`	The current version of the application (e.g., a semantic version, revision hash, etc.)	`5.7.21`	string
`app.kubernetes.io/component`	The component within the architecture	`database`	string
`app.kubernetes.io/part-of`	The name of a higher level application this one is part of	`wordpress`	string
`app.kubernetes.io/managed-by`	The tool being used to manage the operation of an application	`helm`	string

To illustrate these labels in action, consider the following StatefulSet object:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
    app.kubernetes.io/managed-by: helm
```

Applications And Instances Of Applications

An application can be installed one or more times into a Kubernetes cluster and, in some cases, the same namespace. For example, wordpress can be installed more than once where different websites are different installations of wordpress.

The name of an application and the instance name are recorded separately. For example, WordPress has a `app.kubernetes.io/name` of `wordpress` while it has an instance name, represented as `app.kubernetes.io/instance` with a value of `wordpress-abcxzy`. This enables the application and instance of the application to be identifiable. Every instance of an application must have a unique name.

Examples

To illustrate different ways to use these labels the following examples have varying complexity.

A Simple Stateless Service

Consider the case for a simple stateless service deployed using `Deployment` and `Service` objects. The following two snippets represent how the labels could be used in their simplest form.

The `Deployment` is used to oversee the pods running the application itself.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy
...
```

The `Service` is used to expose the application.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy
...
```

Web Application With A Database

Consider a slightly more complicated application: a web application (WordPress) using a database (MySQL), installed using Helm. The following snippets illustrate the start of objects used to deploy this application.

The start to the following `Deployment` is used for WordPress:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
```

The `Service` is used to expose WordPress:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
```

MySQL is exposed as a `StatefulSet` with metadata for both it and the larger application it belongs to:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxzy
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
...
```

The `Service` is used to expose MySQL as part of WordPress:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxzy
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
...
```

With the MySQL `StatefulSet` and `Service` you'll notice information about both MySQL and Wordpress, the broader application, are included.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on June 23, 2019 at 11:55 PM PST by Update common-labels.md (#15058) (Page History)

Edit This Page

Nodes

A node is a worker machine in Kubernetes, previously known as a `minion`. A node may be a VM or physical machine, depending on the cluster. Each node contains the services necessary to run pods and is managed by the master components. The services on a node include the container runtime, kubelet and kube-proxy. See The Kubernetes Node section in the architecture design doc for more details.

Node Status

A node's status contains the following information:

- Addresses
- Conditions
- Capacity and Allocatable
- Info

Node status and other details about a node can be displayed using the following command:

```
kubectl describe node <insert-node-name-here>
```

Each section is described in detail below.

Addresses

The usage of these fields varies depending on your cloud provider or bare metal configuration.

- HostName: The hostname as reported by the node's kernel. Can be overridden via the kubelet `--hostname-override` parameter.
- ExternalIP: Typically the IP address of the node that is externally routable (available from outside the cluster).
- InternalIP: Typically the IP address of the node that is routable only within the cluster.

Conditions

The `conditions` field describes the status of all `Running` nodes. Examples of conditions include:

Node Condition	Description
`Ready`	`True` if the node is healthy and ready to accept pods, `False` if the node is not healthy and is not accepting pods, and `Unknown` if the node controller has not heard from the node in the last `node-monitor-grace-period` (default is 40 seconds)
`MemoryPressure`	`True` if pressure exists on the node memory - that is, if the node memory is low; otherwise `False`
`PIDPressure`	`True` if pressure exists on the processes - that is, if there are too many processes on the node; otherwise `False`
`DiskPressure`	`True` if pressure exists on the disk size - that is, if the disk capacity is low; otherwise `False`
`NetworkUnavailable`	`True` if the network for the node is not correctly configured, otherwise `False`

The node condition is represented as a JSON object. For example, the following response describes a healthy node.

```
"conditions": [
  {
    "type": "Ready",
    "status": "True",
    "reason": "KubeletReady",
    "message": "kubelet is posting ready status",
    "lastHeartbeatTime": "2019-06-05T18:38:35Z",
    "lastTransitionTime": "2019-06-05T11:41:27Z"
  }
]
```

If the Status of the Ready condition remains `Unknown` or `False` for longer than the `pod-eviction-timeout`, an argument is passed to the kube-controller-manager and all the Pods on the node are scheduled for deletion by the Node Controller. The default eviction timeout duration is **five minutes**. In some cases when the node is unreachable, the apiserver is unable to communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the apiserver is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.

In versions of Kubernetes prior to 1.5, the node controller would force delete these unreachable pods from the apiserver. However, in 1.5 and higher, the node controller does not force delete pods until it is confirmed that they have stopped running in the cluster. You can see the pods that might be running on an unreachable node as being in the `Terminating` or `Unknown` state. In cases where Kubernetes cannot deduce from the underlying infrastructure if a node has permanently left a cluster, the cluster administrator may need to delete the node object by hand. Deleting the node object from Kubernetes causes all the Pod objects running on the node to be deleted from the apiserver, and frees up their names.

The node lifecycle controller automatically creates taints that represent conditions. When the scheduler is assigning a Pod to a Node, the scheduler takes the Node's taints into account, except for any taints that the Pod tolerates.

Capacity and Allocatable

Describes the resources available on the node: CPU, memory and the maximum number of pods that can be scheduled onto the node.

The fields in the capacity block indicate the total amount of resources that a Node has. The allocatable block indicates the amount of resources on a Node that is available to be consumed by normal Pods.

You may read more about capacity and allocatable resources while learning how to reserve compute resources on a Node.

Info

Describes general information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), Docker version (if used), and OS name. This information is gathered by Kubelet from the node.

Management

Unlike pods and services, a node is not inherently created by Kubernetes: it is created externally by cloud providers like Google Compute Engine, or it exists in your pool of physical or virtual machines. So when Kubernetes creates a node, it creates an object that represents the node. After creation, Kubernetes checks whether the node is valid or not. For example, if you try to create a node from the following content:

```json
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes creates a node object internally (the representation), and validates the node by health checking based on the `metadata.name` field. If the node is valid - that is, if all necessary services are running - it is eligible to run a pod. Otherwise, it is ignored for any cluster activity until it becomes valid.

> **Note:** Kubernetes keeps the object for the invalid node and keeps checking to see whether it becomes valid. You must explicitly delete the Node object to stop this process.

Currently, there are three components that interact with the Kubernetes node interface: node controller, kubelet, and kubectl.

Node Controller

The node controller is a Kubernetes master component which manages various aspects of nodes.

The node controller has multiple roles in a node's life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

The second is keeping the node controller's internal list of nodes up to date with the cloud provider's list of available machines. When running in a cloud environment, whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.

The third is monitoring the nodes' health. The node controller is responsible for updating the NodeReady condition of NodeStatus to ConditionUnknown when a node becomes unreachable (i.e. the node controller stops receiving heartbeats for some reason, e.g. due to the node being down), and then later evicting all the pods from the node (using graceful termination) if the node continues to be unreachable. (The default timeouts are 40s to start reporting ConditionUnknown and 5m after that to start evicting pods.) The node controller checks the state of each node every `--node-monitor-period` seconds.

Heartbeats

Heartbeats, sent by Kubernetes nodes, help determine the availability of a node. There are two forms of heartbeats: updates of `NodeStatus` and the Lease object. Each Node has an associated Lease object in the `kube-node-lease` namespaceAn abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster. . Lease is a lightweight resource, which improves the performance of the node heartbeats as the cluster scales.

The kubelet is responsible for creating and updating the `NodeStatus` and a Lease object.

- The kubelet updates the `NodeStatus` either when there is change in status, or if there has been no update for a configured interval. The default interval for `NodeStatus` updates is 5 minutes (much longer than the 40 second default timeout for unreachable nodes).

- The kubelet creates and then updates its Lease object every 10 seconds (the default update interval). Lease updates occur independently from the `NodeStatus` updates.

Reliability

In Kubernetes 1.4, we updated the logic of the node controller to better handle cases when a large number of nodes have problems with reaching the master (e.g. because the master has networking problems). Starting with 1.4, the node controller looks at the state of all nodes in the cluster when making a decision about pod eviction.

In most cases, node controller limits the eviction rate to `--node-eviction-rate` (default 0.1) per second, meaning it won't evict pods from more than 1 node per 10 seconds.

The node eviction behavior changes when a node in a given availability zone becomes unhealthy. The node controller checks what percentage of nodes in the zone are unhealthy (NodeReady condition is ConditionUnknown or ConditionFalse) at the same time. If the fraction of unhealthy nodes is at least `--unhealthy-zone-threshold` (default 0.55) then the eviction rate is reduced: if the cluster is small (i.e. has less than or equal to `--large-cluster-size-threshold` nodes - default 50) then evictions are stopped, otherwise the eviction rate is reduced to `--secondary-node-eviction-rate` (default 0.01) per second. The reason these policies are implemented per availability zone is because one availability zone might become partitioned from the master while the others remain connected. If your cluster does not span multiple cloud provider availability zones, then there is only one availability zone (the whole cluster).

A key reason for spreading your nodes across availability zones is so that the workload can be shifted to healthy zones when one entire zone goes down. Therefore, if all nodes in a zone are unhealthy then the node controller evicts at the normal rate of `--node-eviction-rate`. The corner case is when all zones are completely unhealthy (i.e. there are no healthy nodes in the cluster). In such a case, the node controller assumes that there's some problem with master connectivity and stops all evictions until some connectivity is restored.

Starting in Kubernetes 1.6, the NodeController is also responsible for evicting pods that are running on nodes with `NoExecute` taints, when the pods do not tolerate the taints. Additionally, as an alpha feature that is disabled by default, the NodeController is responsible for adding taints corresponding to node problems like node unreachable or not ready. See this documentation for details about `NoExecute` taints and the alpha feature.

Starting in version 1.8, the node controller can be made responsible for creating taints that represent Node conditions. This is an alpha feature of version 1.8.

Self-Registration of Nodes

When the kubelet flag `--register-node` is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

For self-registration, the kubelet is started with the following options:

- `--kubeconfig` - Path to credentials to authenticate itself to the apiserver.
- `--cloud-provider` - How to talk to a cloud provider to read metadata about itself.
- `--register-node` - Automatically register with the API server.
- `--register-with-taints` - Register the node with the given list of taints (comma separated `<key>=<value>:<effect>`). No-op if `register-node` is false.
- `--node-ip` - IP address of the node.
- `--node-labels` - Labels to add when registering the node in the cluster (see label restrictions enforced by the NodeRestriction admission plugin in 1.13+).
- `--node-status-update-frequency` - Specifies how often kubelet posts node status to master.

When the Node authorization mode and NodeRestriction admission plugin are enabled, kubelets are only authorized to create/modify their own Node resource.

Manual Node Administration

A cluster administrator can create and modify node objects.

If the administrator wishes to create node objects manually, set the kubelet flag `--register-node=false`.

The administrator can modify node resources (regardless of the setting of `--register-node`). Modifications include setting labels on the node and marking it unschedulable.

Labels on nodes can be used in conjunction with node selectors on pods to control scheduling, e.g. to constrain a pod to only be eligible to run on a subset of the nodes.

Marking a node as unschedulable prevents new pods from being scheduled to that node, but does not affect any existing pods on the node. This is useful as a preparatory step before a node reboot, etc. For example, to mark a node unschedulable, run this command:

```
kubectl cordon $NODENAME
```

> **Note:** Pods created by a DaemonSet controller bypass the Kubernetes scheduler and do not respect the unschedulable attribute on a node. This assumes that daemons belong on the

machine even if it is being drained of applications while it prepares for a reboot.

Caution: `kubectl cordon` marks a node as â€˜unschedulable', which has the side effect of the service controller removing the node from any LoadBalancer node target lists it was previously eligible for, effectively removing incoming load balancer traffic from the cordoned node(s).

Node capacity

The capacity of the node (number of cpus and amount of memory) is part of the node object. Normally, nodes register themselves and report their capacity when creating the node object. If you are doing manual node administration, then you need to set node capacity when adding a node.

The Kubernetes scheduler ensures that there are enough resources for all the pods on a node. It checks that the sum of the requests of containers on the node is no greater than the node capacity. It includes all containers started by the kubelet, but not containers started directly by the container runtime nor any process running outside of the containers.

If you want to explicitly reserve resources for non-Pod processes, follow this tutorial to reserve resources for system daemons.

Node topology

FEATURE STATE: `Kubernetes v1.17` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

Master-Node Communication

This document catalogs the communication paths between the master (really the apiserver) and the Kubernetes cluster. The intent is to allow users to customize their installation to harden the network configuration such that the cluster can be run on an untrusted network (or on fully public IPs on a cloud provider).

- Cluster to Master
- Master to Cluster

Cluster to Master

All communication paths from the cluster to the master terminate at the apiserver (none of the other master components are designed to expose remote services). In a typical deployment, the apiserver is configured to listen for remote connections on a secure HTTPS port (443) with one or more forms of client authentication enabled. One or more forms of authorization should be enabled, especially if anonymous requests or service account tokens are allowed.

Nodes should be provisioned with the public root certificate for the cluster such that they can connect securely to the apiserver along with valid client credentials. For example, on a default GKE deployment, the client credentials provided to the kubelet are in the form of a client certificate. See kubelet TLS bootstrapping for automated provisioning of kubelet client certificates.

Pods that wish to connect to the apiserver can do so securely by leveraging a service account so that Kubernetes will automatically inject the public root certificate and a valid bearer token into the pod when it is instantiated. The `kubernetes` service (in all namespaces) is configured with a virtual IP address that is redirected (via kube-proxy) to the HTTPS endpoint on the apiserver.

The master components also communicate with the cluster apiserver over the secure port.

As a result, the default operating mode for connections from the cluster (nodes and pods running on the nodes) to the master is secured by default and can run over untrusted and/or public networks.

Master to Cluster

There are two primary communication paths from the master (apiserver) to the cluster. The first is from the apiserver to the kubelet process which runs on each node in the cluster. The second is from the apiserver to any node, pod, or service through the apiserver's proxy functionality.

apiserver to kubelet

The connections from the apiserver to the kubelet are used for:

- Fetching logs for pods.
- Attaching (through kubectl) to running pods.
- Providing the kubelet's port-forwarding functionality.

These connections terminate at the kubelet's HTTPS endpoint. By default, the apiserver does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks, and **unsafe** to run over untrusted and/or public networks.

To verify this connection, use the `--kubelet-certificate-authority` flag to provide the apiserver with a root certificate bundle to use to verify the kubelet's serving certificate.

If that is not possible, use SSH tunneling between the apiserver and kubelet if required to avoid connecting over an untrusted or public network.

Finally, Kubelet authentication and/or authorization should be enabled to secure the kubelet API.

apiserver to nodes, pods, and services

The connections from the apiserver to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. They can be run over a secure HTTPS connection by prefixing `https:` to the node, pod, or service name in the API URL, but they will not validate the certificate provided by the HTTPS endpoint nor provide client credentials so while the connection will be encrypted, it will not provide any guarantees of integrity. These connections **are not currently safe** to run over untrusted and/or public networks.

SSH Tunnels

Kubernetes supports SSH tunnels to protect the Master -> Cluster communication paths. In this configuration, the apiserver initiates an SSH tunnel to each node in the cluster (connecting to the ssh server listening on port 22) and passes all traffic destined for a kubelet, node, pod, or service through the tunnel. This tunnel ensures that the traffic is not exposed outside of the network in which the nodes are running.

SSH tunnels are currently deprecated so you shouldn't opt to use them unless you know what you are doing. A replacement for this communication channel is being designed.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on August 12, 2019 at 5:37 PM PST by Applied title case to page title (#15824) (Page History)

Edit This Page

Controllers

In robotics and automation, a *control loop* is a non-terminating loop that regulates the state of a system.

Here is one example of a control loop: a thermostat in a room.

When you set the temperature, that's telling the thermostat about your *desired state*. The actual room temperature is the *current state*. The thermostat acts to bring the current state closer to the desired state, by turning equipment on or off.

In Kubernetes, controllers are control loops that watch the state of your clusterA set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. , then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

- Controller pattern
- Desired versus current state
- Design
- Ways of running controllers
- What's next

Controller pattern

A controller tracks at least one Kubernetes resource type. These objects have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state.

The controller might carry the action out itself; more commonly, in Kubernetes, a controller will send messages to the API serverControl plane component that serves the Kubernetes API. that have useful side effects. You'll see examples of this below.

Control via API server

The JobA finite or batch task that runs to completion. controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Job is a Kubernetes resource that runs a PodThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. , or perhaps several Pods, to carry out a task and then stop.

(Once scheduled, Pod objects become part of the desired state for a kubelet).

When the Job controller sees a new task it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done. The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods. Other components in the control planeThe container orchestration layer that exposes the API and interfaces to define, deploy, and manage the lifecycle of containers. act on the new information (there are new Pods to schedule and run), and eventually the work is done.

After you create a new Job, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to your desired state: creating Pods that do the work you wanted for that Job, so that the Job is closer to completion.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it `Finished`.

(This is a bit like how some thermostats turn a light off to indicate that your room is now at the temperature you set).

Direct control

By contrast with Job, some controllers need to make changes to things outside of your cluster.

For example, if you use a control loop to make sure there are enough NodesA node is a worker machine in Kubernetes. in your cluster, then that controller needs something outside the current cluster to set up new Nodes when needed.

Controllers that interact with external state find their desired state from the API server, then communicate directly with an external system to bring the current state closer in line.

(There actually is a controller that horizontally scales the nodes in your cluster. See Cluster autoscaling).

Desired versus current state

Kubernetes takes a cloud-native view of systems, and is able to handle constant change.

Your cluster could be changing at any point as work happens and control loops automatically fix failures. This means that, potentially, your cluster never reaches a stable state.

As long as the controllers for your cluster are running and able to make useful changes, it doesn't matter if the overall state is or is not stable.

Design

As a tenet of its design, Kubernetes uses lots of controllers that each manage a particular aspect of cluster state. Most commonly, a particular control loop (controller) uses one kind of resource as its desired state, and has a different kind of resource that it manages to make that desired state happen.

It's useful to have simple controllers rather than one, monolithic set of control loops that are interlinked. Controllers can fail, so Kubernetes is designed to allow for that.

For example: a controller for Jobs tracks Job objects (to discover new work) and Pod object (to run the Jobs, and then to see when the work is finished). In this case something else creates the Jobs, whereas the Job controller creates Pods.

> **Note:**
>
> There can be several controllers that create or update the same kind of object. Behind the scenes, Kubernetes controllers make sure that they only pay attention to the resources linked to their controlling resource.
>
> For example, you can have Deployments and Jobs; these both create Pods. The Job controller does not delete the Pods that your Deployment created, because there is information (labelsTags objects with identifying attributes that are meaningful and relevant to users.) the controllers can use to tell those Pods apart.

Ways of running controllers

Kubernetes comes with a set of built-in controllers that run inside the kube-controller-managerControl Plane component that runs controller processes. . These built-in controllers provide important core behaviors.

The Deployment controller and Job controller are examples of controllers that come as part of Kubernetes itself ("built-in" controllers). Kubernetes lets you run a resilient control plane, so that if any of the built-in controllers were to fail, another part of the control plane will take over the work.

You can find controllers that run outside the control plane, to extend Kubernetes. Or, if you want, you can write a new controller yourself. You can run your own controller as a set of Pods, or externally to Kubernetes. What fits best will depend on what that particular controller does.

What's next

- Read about the Kubernetes control plane
- Discover some of the basic Kubernetes objects
- Learn more about the Kubernetes API
- If you want to write your own controller, see Extension Patterns in Extending Kubernetes.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 11, 2020 at 3:24 AM PST by Tweak linking for Kubernetes object concept (#18122) (Page History)

Edit This Page

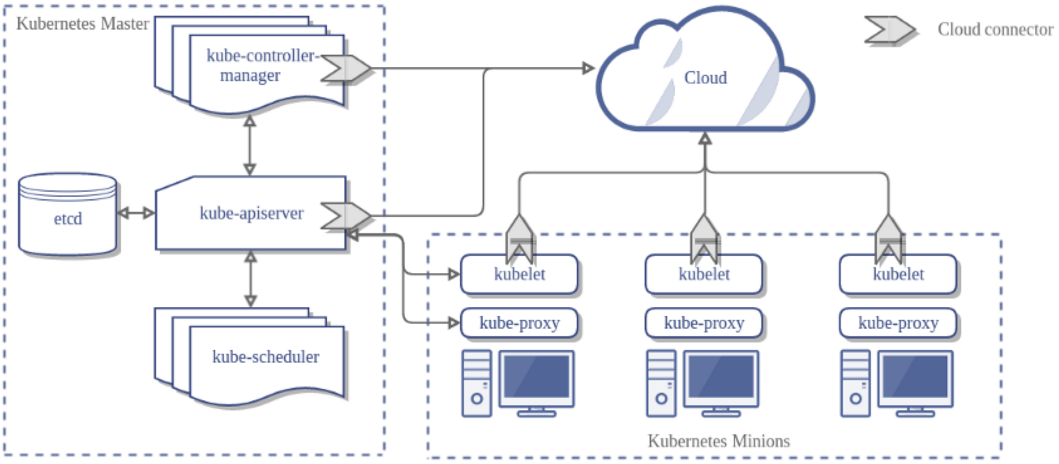
Concepts Underlying the Cloud Controller Manager

The cloud controller manager (CCM) concept (not to be confused with the binary) was originally created to allow cloud specific vendor code and the Kubernetes core to evolve independent of one another. The cloud controller manager runs alongside other master components such as the Kubernetes controller manager, the API server, and scheduler. It can also be started as a Kubernetes addon, in which case it runs on top of Kubernetes.

The cloud controller manager's design is based on a plugin mechanism that allows new cloud providers to integrate with Kubernetes easily by using plugins. There are plans in place for on-boarding new cloud providers on Kubernetes and for migrating cloud providers from the old model to the new CCM model.

This document discusses the concepts behind the cloud controller manager and gives details about its associated functions.

Here's the architecture of a Kubernetes cluster without the cloud controller manager:

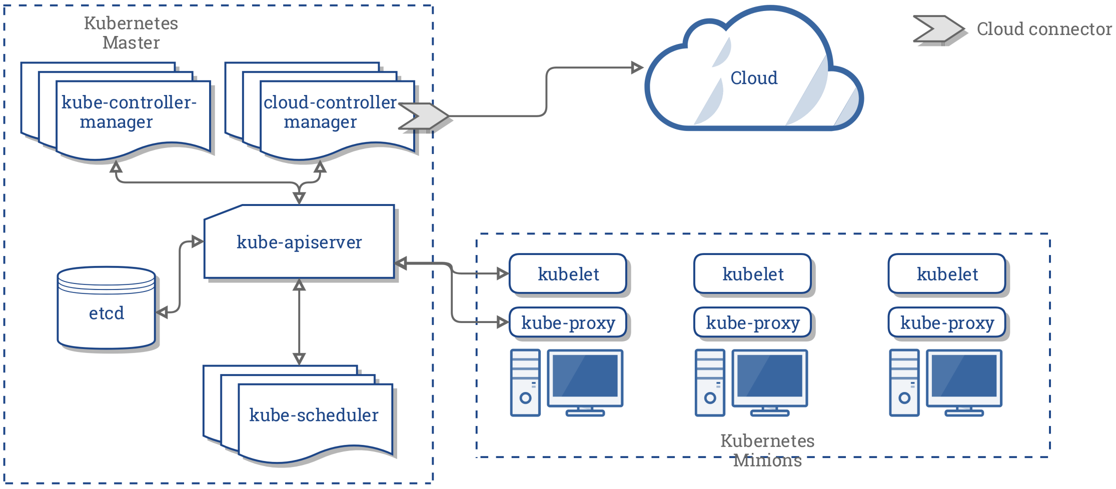


Design

In the preceding diagram, Kubernetes and the cloud provider are integrated through several different components:

- Kubelet
- Kubernetes controller manager
- Kubernetes API server

The CCM consolidates all of the cloud-dependent logic from the preceding three components to create a single point of integration with the cloud. The new architecture with the CCM looks like this:



Components of the CCM

The CCM breaks away some of the functionality of Kubernetes controller manager (KCM) and runs it as a separate process. Specifically, it breaks away those controllers in the KCM that are cloud dependent. The KCM has the following cloud dependent controller loops:

- Node controller
- Volume controller
- Route controller
- Service controller

In version 1.9, the CCM runs the following controllers from the preceding list:

- Node controller
- Route controller
- Service controller

  Note: Volume controller was deliberately chosen to not be a part of CCM. Due to the complexity involved and due to the existing efforts to abstract away vendor specific volume logic, it was decided that volume controller will not be moved to CCM.

The original plan to support volumes using CCM was to use Flex volumes to support pluggable volumes. However, a competing effort known as CSI is being planned to replace Flex.

Considering these dynamics, we decided to have an intermediate stop gap measure until CSI becomes ready.

Functions of the CCM

The CCM inherits its functions from components of Kubernetes that are dependent on a cloud provider. This section is structured based on those components.

1. Kubernetes controller manager

The majority of the CCM's functions are derived from the KCM. As mentioned in the previous section, the CCM runs the following control loops:

- Node controller
- Route controller
- Service controller

Node controller

The Node controller is responsible for initializing a node by obtaining information about the nodes running in the cluster from the cloud provider. The node controller performs the following functions:

1. Initialize a node with cloud specific zone/region labels.
2. Initialize a node with cloud specific instance details, for example, type and size.
3. Obtain the node's network addresses and hostname.
4. In case a node becomes unresponsive, check the cloud to see if the node has been deleted from the cloud. If the node has been deleted from the cloud, delete the Kubernetes Node object.

Route controller

The Route controller is responsible for configuring routes in the cloud appropriately so that containers on different nodes in the Kubernetes cluster can communicate with each other. The route controller is only applicable for Google Compute Engine clusters.

Service Controller

The Service controller is responsible for listening to service create, update, and delete events. Based on the current state of the services in Kubernetes, it configures cloud load balancers (such as ELB , Google LB, or Oracle Cloud Infrastructure LB) to reflect the state of the services in Kubernetes. Additionally, it ensures that service backends for cloud load balancers are up to date.

2. Kubelet

The Node controller contains the cloud-dependent functionality of the kubelet. Prior to the introduction of the CCM, the kubelet was responsible for initializing a node with cloud-specific details such as IP addresses, region/zone labels and instance type information. The introduction of the CCM has moved this initialization operation from the kubelet into the CCM.

In this new model, the kubelet initializes a node without cloud-specific information. However, it adds a taint to the newly created node that makes the node unschedulable until the CCM initializes the node with cloud-specific information. It then removes this taint.

Plugin mechanism

The cloud controller manager uses Go interfaces to allow implementations from any cloud to be plugged in. Specifically, it uses the CloudProvider Interface defined here.

The implementation of the four shared controllers highlighted above, and some scaffolding along with the shared cloudprovider interface, will stay in the Kubernetes core. Implementations specific to cloud providers will be built outside of the core and implement interfaces defined in the core.

For more information about developing plugins, see Developing Cloud Controller Manager.

Authorization

This section breaks down the access required on various API objects by the CCM to perform its operations.

Node Controller

The Node controller only works with Node objects. It requires full access to get, list, create, update, patch, watch, and delete Node objects.

v1/Node:

- Get
- List
- Create
- Update
- Patch
- Watch
- Delete

Route controller

The route controller listens to Node object creation and configures routes appropriately. It requires get access to Node objects.

v1/Node:

- Get

Service controller

The service controller listens to Service object create, update and delete events and then configures endpoints for those Services appropriately.

To access Services, it requires list, and watch access. To update Services, it requires patch and update access.

To set up endpoints for the Services, it requires access to create, list, get, watch, and update.

v1/Service:

- List
- Get
- Watch
- Patch
- Update

Others

The implementation of the core of CCM requires access to create events, and to ensure secure operation, it requires access to create ServiceAccounts.

v1/Event:

- Create
- Patch
- Update

v1/ServiceAccount:

- Create

The RBAC ClusterRole for the CCM looks like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cloud-controller-manager
rules:
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - create
  - patch
  - update
- apiGroups:
  - ""
  resources:
```

```
  - nodes
  verbs:
  - '*'
- apiGroups:
  - ""
  resources:
  - nodes/status
  verbs:
  - patch
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - list
  - patch
  - update
  - watch
- apiGroups:
  - ""
  resources:
  - serviceaccounts
  verbs:
  - create
- apiGroups:
  - ""
  resources:
  - persistentvolumes
  verbs:
  - get
  - list
  - update
  - watch
- apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - create
  - get
  - list
  - watch
  - update
```

Vendor Implementations

The following cloud providers have implemented CCMs:

- Alibaba Cloud
- AWS
- Azure

- BaiduCloud
- DigitalOcean
- GCP
- Hetzner
- Linode
- OpenStack
- Oracle
- TencentCloud

Cluster Administration

Complete instructions for configuring and running the CCM are provided here.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 16, 2020 at 4:58 PM PST by Add Alibaba Cloud cloud-controller-manager (#18255) (Page History)

Edit This Page

Images

You create your Docker image and push it to a registry before referring to it in a Kubernetes pod.

The `image` property of a container supports the same syntax as the `docker` command does, including private registries and tags.

- Updating Images
- Building Multi-architecture Images with Manifests

- Using a Private Registry

Updating Images

The default pull policy is `IfNotPresent` which causes the Kubelet to skip pulling an image if it already exists. If you would like to always force a pull, you can do one of the following:

- set the `imagePullPolicy` of the container to `Always`.
- omit the `imagePullPolicy` and use `:latest` as the tag for the image to use.
- omit the `imagePullPolicy` and the tag for the image to use.
- enable the AlwaysPullImages admission controller.

Note that you should avoid using `:latest` tag, see Best Practices for Configuration for more information.

Building Multi-architecture Images with Manifests

Docker CLI now supports the following command `docker manifest` with sub commands like `create`, `annotate` and `push`. These commands can be used to build and push the manifests. You can use `docker manifest inspect` to view the manifest.

Please see docker documentation here: https://docs.docker.com/edge/engine/reference/commandline/manifest/

See examples on how we use this in our build harness: https://cs.k8s.io/?q=docker%20manifest%20(create%7Cpush%7Cannotate)&i=nope&files=&repos=

These commands rely on and are implemented purely on the Docker CLI. You will need to either edit the `$HOME/.docker/config.json` and set `experimental` key to `enabled` or you can just set `DOCKER_CLI_EXPERIMENTAL` environment variable to `enabled` when you call the CLI commands.

> **Note:** Please use Docker *18.06 or above*, versions below that either have bugs or do not support the experimental command line option. Example https://github.com/docker/cli/issues/1135 causes problems under containerd.

If you run into trouble with uploading stale manifests, just clean up the older manifests in `$HOME/.docker/manifests` to start fresh.

For Kubernetes, we have typically used images with suffix `-$(ARCH)`. For backward compatibility, please generate the older images with suffixes. The idea is to generate say `pause` image which has the manifest for all the arch(es) and say `pause-amd64` which is backwards compatible for older configurations or YAML files which may have hard coded the images with suffixes.

Using a Private Registry

Private registries may require keys to read images from them. Credentials can be provided in several ways:

- Using Google Container Registry
  - Per-cluster
  - automatically configured on Google Compute Engine or Google Kubernetes Engine
  - all pods can read the project's private registry
- Using Amazon Elastic Container Registry (ECR)
  - use IAM roles and policies to control access to ECR repositories
  - automatically refreshes ECR login credentials
- Using Oracle Cloud Infrastructure Registry (OCIR)
  - use IAM roles and policies to control access to OCIR repositories
- Using Azure Container Registry (ACR)
- Using IBM Cloud Container Registry
- Configuring Nodes to Authenticate to a Private Registry
  - all pods can read any configured private registries
  - requires node configuration by cluster administrator
- Pre-pulled Images
  - all pods can use any images cached on a node
  - requires root access to all nodes to setup
- Specifying ImagePullSecrets on a Pod
  - only pods which provide own keys can access the private registry

Each option is described in more detail below.

Using Google Container Registry

Kubernetes has native support for the Google Container Registry (GCR), when running on Google Compute Engine (GCE). If you are running your cluster on GCE or Google Kubernetes Engine, simply use the full image name (e.g. gcr.io/my_project/image:tag).

All pods in a cluster will have read access to images in this registry.

The kubelet will authenticate to GCR using the instance's Google service account. The service account on the instance will have a `https://www.googleapis.com/auth/devstorage.read_only`, so it can pull from the project's GCR, but not push.

Using Amazon Elastic Container Registry

Kubernetes has native support for the Amazon Elastic Container Registry, when nodes are AWS EC2 instances.

Simply use the full image name (e.g. `ACCOUNT.dkr.ecr.REGION.amazonaws.com/imagename:tag`) in the Pod definition.

All users of the cluster who can create pods will be able to run pods that use any of the images in the ECR registry.

The kubelet will fetch and periodically refresh ECR credentials. It needs the following permissions to do this:

- `ecr:GetAuthorizationToken`
- `ecr:BatchCheckLayerAvailability`
- `ecr:GetDownloadUrlForLayer`
- `ecr:GetRepositoryPolicy`
- `ecr:DescribeRepositories`
- `ecr:ListImages`
- `ecr:BatchGetImage`

Requirements:

- You must be using kubelet version `v1.2.0` or newer. (e.g. run `/usr/bin/kubelet --version=true`).
- If your nodes are in region A and your registry is in a different region B, you need version `v1.3.0` or newer.
- ECR must be offered in your region

Troubleshooting:

- Verify all requirements above.
- Get $REGION (e.g. `us-west-2`) credentials on your workstation. SSH into the host and run Docker manually with those creds. Does it work?
- Verify kubelet is running with `--cloud-provider=aws`.
- Increase kubelet log level verbosity to at least 3 and check kubelet logs (e.g. `journalctl -u kubelet`) for log lines like:
  - `aws_credentials.go:109] unable to get ECR credentials from cache, checking ECR API`
  - `aws_credentials.go:116] Got ECR credentials from ECR API for <AWS account ID for ECR>.dkr.ecr.<AWS region>.amazonaws.com`

Using Azure Container Registry (ACR)

When using Azure Container Registry you can authenticate using either an admin user or a service principal. In either case, authentication is done via standard Docker authentication. These instructions assume the azure-cli command line tool.

You first need to create a registry and generate credentials, complete documentation for this can be found in the Azure container registry documentation.

Once you have created your container registry, you will use the following credentials to login:

- `DOCKER_USER` : service principal, or admin username
- `DOCKER_PASSWORD`: service principal password, or admin user password
- `DOCKER_REGISTRY_SERVER`: `${some-registry-name}.azurecr.io`
- `DOCKER_EMAIL`: `${some-email-address}`

Once you have those variables filled in you can configure a Kubernetes Secret and use it to deploy a Pod.

Using IBM Cloud Container Registry

IBM Cloud Container Registry provides a multi-tenant private image registry that you can use to safely store and share your Docker images. By default, images in your private registry are scanned by the integrated Vulnerability Advisor to detect security issues and potential vulnerabilities. Users in your IBM Cloud account can access your images, or you can create a token to grant access to registry namespaces.

To install the IBM Cloud Container Registry CLI plug-in and create a namespace for your images, see Getting started with IBM Cloud Container Registry.

You can use the IBM Cloud Container Registry to deploy containers from IBM Cloud public images and your private images into the `default` namespace of your IBM Cloud Kubernetes Service cluster. To deploy a container into other namespaces, or to use an image from a different IBM Cloud Container Registry region or IBM Cloud account, create a Kubernetes `imagePullSecret`. For more information, see Building containers from images.

Configuring Nodes to Authenticate to a Private Registry

> **Note:** If you are running on Google Kubernetes Engine, there will already be a `.dockercfg` on each node with credentials for Google Container Registry. You cannot use this approach.
>
> **Note:** If you are running on AWS EC2 and are using the EC2 Container Registry (ECR), the kubelet on each node will manage and update the ECR login credentials. You cannot use this approach.
>
> **Note:** This approach is suitable if you can control node configuration. It will not work reliably on GCE, and any other cloud provider that does automatic node replacement.
>
> **Note:** Kubernetes as of now only supports the `auths` and `HttpHeaders` section of docker config. This means credential helpers (`credHelpers` or `credsStore`) are not supported.

Docker stores keys for private registries in the `$HOME/.dockercfg` or `$HOME/.docker/config.json` file. If you put the same file in the search paths list below, kubelet uses it as the credential provider when pulling images.

- `{--root-dir:-/var/lib/kubelet}/config.json`
- `{cwd of kubelet}/config.json`
- `${HOME}/.docker/config.json`
- `/.docker/config.json`
- `{--root-dir:-/var/lib/kubelet}/.dockercfg`

- `{cwd of kubelet}/.dockercfg`
- `${HOME}/.dockercfg`
- `/.dockercfg`

  Note: You may have to set `HOME=/root` explicitly in your environment file for kubelet.

Here are the recommended steps to configuring your nodes to use a private registry. In this example, run these on your desktop/laptop:

1. Run `docker login [server]` for each set of credentials you want to use. This updates `$HOME/.docker/config.json`.
2. View `$HOME/.docker/config.json` in an editor to ensure it contains just the credentials you want to use.
3. Get a list of your nodes, for example:
   - if you want the names: `nodes=$(kubectl get nodes -o jsonpath='{range.items[*].metadata}{.name} {end}')`
   - if you want to get the IPs: `nodes=$(kubectl get nodes -o jsonpath='{range .items[*].status.addresses[?(@.type=="ExternalIP")]}{.address} {end}')`
4. Copy your local `.docker/config.json` to one of the search paths list above.
   - for example: `for n in $nodes; do scp ~/.docker/config.json root@$n:/var/lib/kubelet/config.json; done`

Verify by creating a pod that uses a private image, e.g.:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: private-image-test-1
spec:
  containers:
    - name: uses-private-image
      image: $PRIVATE_IMAGE_NAME
      imagePullPolicy: Always
      command: [ "echo", "SUCCESS" ]
EOF
```

```
pod/private-image-test-1 created
```

If everything is working, then, after a few moments, you can run:

```
kubectl logs private-image-test-1
```

and see that the command outputs:

```
SUCCESS
```

If you suspect that the command failed, you can run:

```
kubectl describe pods/private-image-test-1 | grep 'Failed'
```

In case of failure, the output is similar to:

```
  Fri, 26 Jun 2015 15:36:13 -0700    Fri, 26 Jun 2015 15:39:13
-0700    19    {kubelet node-i2hq}    spec.containers{uses-
private-image}    failed        Failed to pull image "user/
privaterepo:v1": Error: image user/privaterepo:v1 not found
```

You must ensure all nodes in the cluster have the same `.docker/config.json`. Otherwise, pods will run on some nodes and fail to run on others. For example, if you use node autoscaling, then each instance template needs to include the `.docker/config.json` or mount a drive that contains it.

All pods will have read access to images in any private registry once private registry keys are added to the `.docker/config.json`.

Pre-pulled Images

> **Note:** If you are running on Google Kubernetes Engine, there will already be a `.dockercfg` on each node with credentials for Google Container Registry. You cannot use this approach.

> **Note:** This approach is suitable if you can control node configuration. It will not work reliably on GCE, and any other cloud provider that does automatic node replacement.

By default, the kubelet will try to pull each image from the specified registry. However, if the `imagePullPolicy` property of the container is set to `IfNotPresent` or `Never`, then a local image is used (preferentially or exclusively, respectively).

If you want to rely on pre-pulled images as a substitute for registry authentication, you must ensure all nodes in the cluster have the same pre-pulled images.

This can be used to preload certain images for speed or as an alternative to authenticating to a private registry.

All pods will have read access to any pre-pulled images.

Specifying ImagePullSecrets on a Pod

> **Note:** This approach is currently the recommended approach for Google Kubernetes Engine, GCE, and any cloud-providers where node creation is automated.

Kubernetes supports specifying registry keys on a pod.

Creating a Secret with a Docker Config

Run the following command, substituting the appropriate uppercase values:

```
kubectl create secret docker-registry <name> --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
```

If you already have a Docker credentials file then, rather than using the above command, you can import the credentials file as a Kubernetes secret. Create a Secret based on existing Docker credentials explains how to set this up. This is particularly useful if you are using multiple private container registries, as `kubectl create secret docker-registry` creates a Secret that will only work with a single private registry.

> **Note:** Pods can only reference image pull secrets in their own namespace, so this process needs to be done one time per namespace.

Referring to an imagePullSecrets on a Pod

Now, you can create pods which reference that secret by adding an `imagePullSecrets` section to a pod definition.

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: foo
  namespace: awesomeapps
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
EOF

cat <<EOF >> ./kustomization.yaml
resources:
- pod.yaml
EOF
```

This needs to be done for each pod that is using a private registry.

However, setting of this field can be automated by setting the imagePullSecrets in a serviceAccount resource. Check Add ImagePullSecrets to a Service Account for detailed instructions.

You can use this in conjunction with a per-node `.docker/config.json`. The credentials will be merged. This approach will work on Google Kubernetes Engine.

Use Cases

There are a number of solutions for configuring private registries. Here are some common use cases and suggested solutions.

1. Cluster running only non-proprietary (e.g. open-source) images. No need to hide images.
    - Use public images on the Docker hub.
        - No configuration required.
        - On GCE/Google Kubernetes Engine, a local mirror is automatically used for improved speed and availability.
2. Cluster running some proprietary images which should be hidden to those outside the company, but visible to all cluster users.
    - Use a hosted private Docker registry.
        - It may be hosted on the Docker Hub, or elsewhere.
        - Manually configure .docker/config.json on each node as described above.
    - Or, run an internal private registry behind your firewall with open read access.
        - No Kubernetes configuration is required.
    - Or, when on GCE/Google Kubernetes Engine, use the project's Google Container Registry.
        - It will work better with cluster autoscaling than manual node configuration.
    - Or, on a cluster where changing the node configuration is inconvenient, use `imagePullSecrets`.
3. Cluster with proprietary images, a few of which require stricter access control.
    - Ensure AlwaysPullImages admission controller is active. Otherwise, all Pods potentially have access to all images.
    - Move sensitive data into a "Secret" resource, instead of packaging it in an image.
4. A multi-tenant cluster where each tenant needs own private registry.
    - Ensure AlwaysPullImages admission controller is active. Otherwise, all Pods of all tenants potentially have access to all images.
    - Run a private registry with authorization required.
    - Generate registry credential for each tenant, put into secret, and populate secret to each tenant namespace.
    - The tenant adds that secret to imagePullSecrets of each namespace.

If you need access to multiple registries, you can create one secret for each registry. Kubelet will merge any `imagePullSecrets` into a single virtual `.docker/config.json`

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 20, 2020 at 2:33 AM PST by Fixed outdated ECR credential debug message (#18631) (Page History)

Edit This Page

Container Environment Variables

This page describes the resources available to Containers in the Container environment.

- Container environment
- What's next

Container environment

The Kubernetes Container environment provides several important resources to Containers:

- A filesystem, which is a combination of an image and one or more volumes.
- Information about the Container itself.
- Information about other objects in the cluster.

Container information

The *hostname* of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the `gethostname` function call in libc.

The Pod name and namespace are available as environment variables through the downward API.

User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the Docker image.

Cluster information

A list of all services that were running when a Container was created is available to that Container as environment variables. Those environment variables match the syntax of Docker links.

For a service named *foo* that maps to a Container named *bar*, the following variables are defined:

```
FOO_SERVICE_HOST=<the host the service is running on>
FOO_SERVICE_PORT=<the port the service is running on>
```

Services have dedicated IP addresses and are available to the Container via DNS, if DNS addon is enabled.Â

What's next

- Learn more about Container lifecycle hooks.

- Get hands-on experience attaching handlers to Container lifecycle events.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Edit This Page

Runtime Class

FEATURE STATE: `Kubernetes v1.14` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Container Lifecycle Hooks

This page describes how kubelet managed Containers can use the Container lifecycle hook framework to run code triggered by events during their management lifecycle.

- Overview
- Container hooks
- What's next

Overview

Analogous to many programming language frameworks that have component lifecycle hooks, such as Angular, Kubernetes provides Containers with lifecycle hooks. The hooks enable Containers to be aware of events in their management lifecycle and run code implemented in a handler when the corresponding lifecycle hook is executed.

Container hooks

There are two hooks that are exposed to Containers:

`PostStart`

This hook executes immediately after a container is created. However, there is no guarantee that the hook will execute before the container ENTRYPOINT. No parameters are passed to the handler.

`PreStop`

This hook is called immediately before a container is terminated due to an API request or management event such as liveness probe failure, preemption, resource contention and others. A call to the preStop hook fails if the container is already in terminated or completed state. It is blocking, meaning it is synchronous, so it must complete before the call to delete the container can be sent. No parameters are passed to the handler.

A more detailed description of the termination behavior can be found in Termination of Pods.

Hook handler implementations

Containers can access a hook by implementing and registering a handler for that hook. There are two types of hook handlers that can be implemented for Containers:

- Exec - Executes a specific command, such as `pre-stop.sh`, inside the cgroups and namespaces of the Container. Resources consumed by the command are counted against the Container.
- HTTP - Executes an HTTP request against a specific endpoint on the Container.

Hook handler execution

When a Container lifecycle management hook is called, the Kubernetes management system executes the handler in the Container registered for that hook.Â

Hook handler calls are synchronous within the context of the Pod containing the Container. This means that for a `PostStart` hook, the Container ENTRYPOINT and hook fire asynchronously. However, if the hook takes too long to run or hangs, the Container cannot reach a `running` state.

The behavior is similar for a `PreStop` hook. If the hook hangs during execution, the Pod phase stays in a `Terminating` state and is killed after `terminationGracePeriodSeconds` of pod ends. If a `PostStart` or `PreStop` hook fails, it kills the Container.

Users should make their hook handlers as lightweight as possible. There are cases, however, when long running commands make sense, such as when saving state prior to stopping a Container.

Hook delivery guarantees

Hook delivery is intended to be *at least once*, which means that a hook may be called multiple times for any given event, such as for `PostStart` or `PreStop`. It is up to the hook implementation to handle this correctly.

Generally, only single deliveries are made. If, for example, an HTTP hook receiver is down and is unable to take traffic, there is no attempt to resend. In some rare cases, however, double delivery may occur. For instance, if a kubelet restarts in the middle of sending a hook, the hook might be resent after the kubelet comes back up.

Debugging Hook handlers

The logs for a Hook handler are not exposed in Pod events. If a handler fails for some reason, it broadcasts an event. For `PostStart`, this is the `FailedPostStartHook` event, and for `PreStop`, this is the `FailedPreStopHook` event. You can see these events by running `kubectl describe pod <pod_name>`. Here is some example output of events from running this command:

```
Events:
  FirstSeen  LastSeen  Count
From
SubObjectPath          Type      Reason                 Message
  ---------  --------  -----
----
-------------          --------  ------                 -------
  1m         1m        1      {default-
scheduler }
    Normal    Scheduled            Successfully assigned
test-1730497541-cq1d2 to gke-test-cluster-default-pool-a07e5d30-
siqd
  1m         1m        1      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}  spec.containers{main}  Normal
Pulling              pulling image "test:1.0"
  1m         1m        1      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}  spec.containers{main}  Normal
Created              Created container with docker id
5c6a256a2567; Security:[seccomp=unconfined]
  1m         1m        1      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}  spec.containers{main}  Normal
Pulled               Successfully pulled image "test:1.0"
  1m         1m        1      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}  spec.containers{main}  Normal
Started              Started container with docker id
5c6a256a2567
  38s        38s       1      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}  spec.containers{main}  Normal
Killing              Killing container with docker id
5c6a256a2567: PostStart handler: Error executing in Docker
Container: 1
  37s        37s       1      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}  spec.containers{main}  Normal
Killing              Killing container with docker id
8df9fdfd7054: PostStart handler: Error executing in Docker
Container: 1
  38s        37s       2      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}                         Warning
FailedSync           Error syncing pod, skipping: failed to
"StartContainer" for "main" with RunContainerError: "PostStart
handler: Error executing in Docker Container: 1"
  1m         22s       2      {kubelet gke-test-cluster-default-
pool-a07e5d30-siqd}  spec.containers{main}  Warning
FailedPostStartHook
```

What's next

- Learn more about the Container environment.
- Get hands-on experience attaching handlers to Container lifecycle events.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on October 28, 2019 at 4:49 AM PST by Improve Concepts section (#17013) (Page History)

Edit This Page

Pod Overview

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

Understanding Pods

A *Pod* is the basic execution unit of a Kubernetes application-the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents processes running on your ClusterA set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. .

A Pod encapsulates an application's container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: *a single instance of an application in Kubernetes*, which might consist of either a single containerA lightweight and portable executable image that contains software and all of its dependencies. or a small number of containers that are tightly coupled and that share resources.

Docker is the most common container runtime used in a Kubernetes Pod, but Pods support other container runtimes as well.

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

- **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.

- **Pods that run multiple containers that need to work together**. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers might form a single cohesive unit of service-one container serving files from a shared volume to the public, while a separate "sidecar" container refreshes or updates those files. The Pod wraps these containers and storage resources together as a single manageable entity. The Kubernetes Blog has some additional information on Pod use cases. For more information, see:

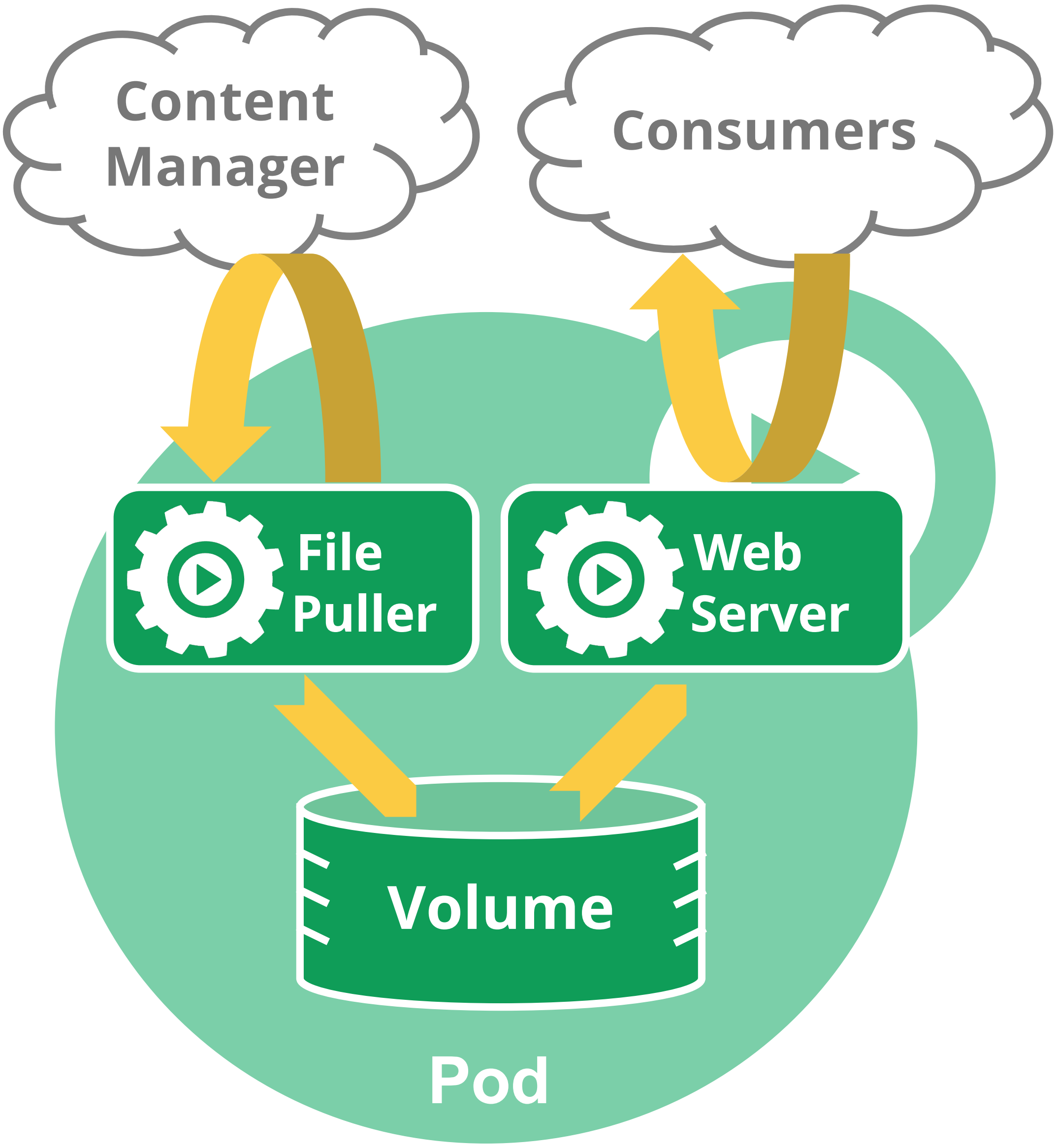
    - The Distributed System Toolkit: Patterns for Composite Containers
    - Container Design Patterns

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (e.g., run multiple instances), you should use multiple Pods, one for each instance. In Kubernetes, this is generally referred to as *replication*. Replicated Pods are usually created and managed as a group by an abstraction called a Controller. See Pods and Controllers for more information.

How Pods manage multiple Containers

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

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



Some Pods have init containersOne or more initialization containers that must run to completion before any app containers run. as well as app containersA container used to run part of a workload. Compare with init container. . Init containers run and complete before the app containers are started.

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

Networking

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

Storage

A Pod can specify a set of shared storage VolumesA directory containing data, accessible to the containers in a pod. . All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See Volumes for more information on how Kubernetes implements shared storage in a Pod.

Working with Pods

You'll rarely create individual Pods directly in Kubernetes-even singleton Pods. This is because Pods are designed as relatively ephemeral, disposable entities. When a Pod gets created (directly by you, or indirectly by a Controller), it is scheduled to run on a NodeA node is a worker machine in Kubernetes. in your cluster. The Pod remains on that Node until the process is terminated, the pod object is deleted, the Pod is *evicted* for lack of resources, or the Node fails.

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

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

Pods and Controllers

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

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

- Deployment
- StatefulSet
- DaemonSet

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

Pod Templates

Pod templates are pod specifications which are included in other objects, such as Replication Controllers, Jobs, and DaemonSets. Controllers use Pod Templates to make actual pods. The sample below is a simple manifest for a Pod which contains a container that prints a message.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

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

What's next

- Learn more about Pods
- Learn more about Pod behavior:
  - Pod Termination
  - Pod Lifecycle

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on December 26, 2019 at 6:13 PM PST by use https instead of http for a link to kubernetes.io/blog (#18287) (Page History)

Edit This Page

Pods

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

What is a Pod?

A *Pod* (as in a pod of whales or pea pod) is a group of one or more containersA lightweight and portable executable image that contains software and all of its dependencies. (such as Docker containers), with shared storage/network, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host" - it contains one or more application containers which are relatively tightly coupled â€” in a pre-container world, being executed on the same physical or virtual machine would mean being executed on the same logical host.

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

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

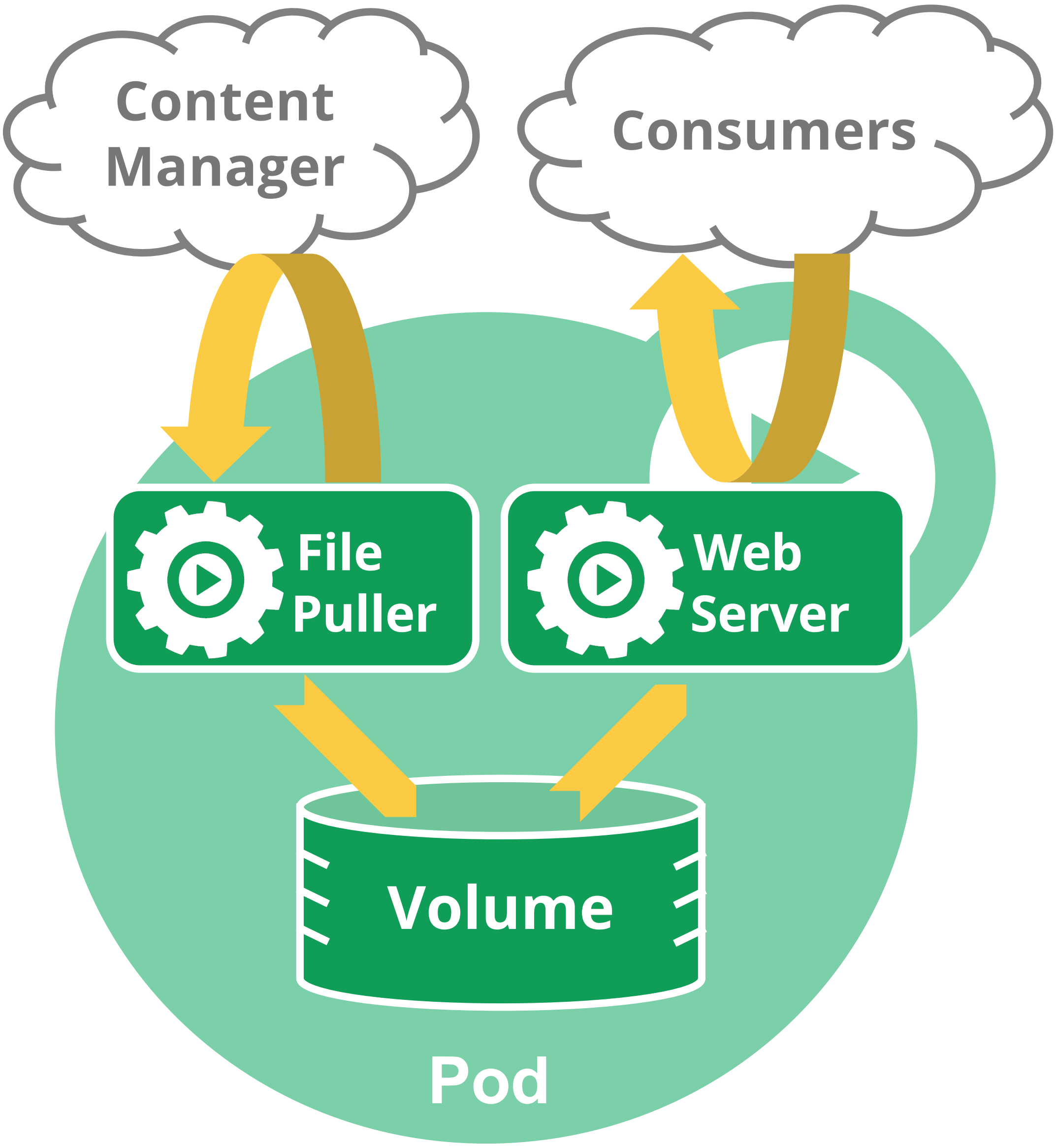
Containers within a Pod share an IP address and port space, and can find each other via `localhost`. They can also communicate with each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. Containers in different Pods have distinct IP addresses and can not communicate by IPC without special configuration. These containers usually communicate with each other via Pod IP addresses.

Applications within a Pod also have access to shared volumesA directory containing data, accessible to the containers in a pod. , which are defined as part of a Pod and are made available to be mounted into each application's filesystem.

In terms of Docker constructs, a Pod is modelled as a group of Docker containers with shared namespaces and shared filesystem volumes.

Like individual application containers, Pods are considered to be relatively ephemeral (rather than durable) entities. As discussed in pod lifecycle, 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 NodeA node is a worker machine in Kubernetes. dies, the Pods scheduled to that node are scheduled for deletion, after a timeout period. A given Pod (as defined by a UID) is not "rescheduled" to a new node; instead, it can be replaced by an identical Pod, with even the same name if desired, but with a new UID (see replication controller for more details).

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



Pod diagram

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

Motivation for Pods

Management

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

termination), coordinated replication, resource sharing, and dependency management are handled automatically for containers in a Pod.

Resource sharing and communication

Pods enable data sharing and communication among their constituents.

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

Containers within the Pod see the system hostname as being the same as the configured `name` for the Pod. There's more about this in the networking section.

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

Uses of pods

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

- content management systems, file and data loaders, local cache managers, etc.
- log and checkpoint backup, compression, rotation, snapshotting, etc.
- data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
- proxies, bridges, and adapters
- controllers, managers, configurators, and updaters

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

For a longer explanation, see The Distributed System ToolKit: Patterns for Composite Containers.

Alternatives considered

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

1. Transparency. Making the containers within the Pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of conveniences for users.

2. Decoupling software dependencies. The individual containers may be versioned, rebuilt and redeployed independently. Kubernetes may even support live updates of individual containers someday.
3. Ease of use. Users don't need to run their own process managers, worry about signal and exit-code propagation, etc.
4. Efficiency. Because the infrastructure takes on more responsibility, containers can be lighter weight.

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

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

Durability of pods (or lack thereof)

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

In general, users shouldn't need to create Pods directly. They should almost always use controllers even for singletons, for example, Deployments. Controllers provide self-healing with a cluster scope, as well as replication and rollout management. Controllers like StatefulSet can also provide support to stateful Pods.

The use of collective APIs as the primary user-facing primitive is relatively common among cluster scheduling systems, including Borg, Marathon, Aurora, and Tupperware.

Pod is exposed as a primitive in order to facilitate:

- scheduler and controller pluggability
- support for pod-level operations without the need to "proxy" them via controller APIs
- decoupling of Pod lifetime from controller lifetime, such as for bootstrapping
- decoupling of controllers and services â€” the endpoint controller just watches Pods
- clean composition of Kubelet-level functionality with cluster-level functionality â€” Kubelet is effectively the "pod controller"
- high-availability applications, which will expect Pods to be replaced in advance of their termination and certainly in advance of deletion, such as in the case of planned evictions or image prefetching.

Termination of Pods

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

An example flow:

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

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows a user to override the default and specify their own value. The value `0` force deletes the Pod. You must specify an additional flag `--force` along with `--grace-period=0` in order to perform force deletions.

Force deletion of pods

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

Force deletions can be potentially dangerous for some Pods and should be performed with caution. In case of StatefulSet Pods, please refer to the task documentation for deleting Pods from a StatefulSet.

Privileged mode for pod 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 Linux capabilities like manipulating the network stack and accessing devices. Processes within the container get almost the same privileges that are available to processes outside a container. With privileged mode, it should be easier to write network and volume plugins as separate Pods that don't need to be compiled into the kubelet.

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

API Object

Pod is a top-level resource in the Kubernetes REST API. The Pod API object definition describes the object in detail.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on August 06, 2019 at 4:41 AM PST by Update pod.md (#15301) (Page History)

Edit This Page

Pod Lifecycle

This page describes the lifecycle of a Pod.

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

Pod conditions

A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed. Each element of the PodCondition array has six possible fields:

- The `lastProbeTime` field provides a timestamp for when the Pod condition was last probed.

- The `lastTransitionTime` field provides a timestamp for when the Pod last transitioned from one status to another.

- The `message` field is a human-readable message indicating details about the transition.

- The `reason` field is a unique, one-word, CamelCase reason for the condition's last transition.

- The `status` field is a string, with possible values "`True`", "`False`", and "`Unknown`".

- The `type` field is a string with the following possible values:

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

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 Container'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 Container's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

Each probe has one of three results:

- Success: The Container passed the diagnostic.
- Failure: The Container failed the diagnostic.
- Unknown: The diagnostic failed, so no action should be taken.

The kubelet can optionally perform and react to three kinds of probes on running Containers:

- `livenessProbe`: Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a liveness probe, the default state is `Success`.

- `readinessProbe`: Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the

Pod. The default state of readiness before the initial delay is `Failure`. If a Container does not provide a readiness probe, the default state is `Success`.

- `startupProbe`: Indicates whether the application within the Container is started. All other probes are disabled if a startup probe is provided, until it succeeds. If the startup probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a startup probe, the default state is `Success`.

When should you use a liveness probe?

FEATURE STATE: `Kubernetes v1.0` stable
This feature is *stable*, meaning:

Edit This Page

Init Containers

This page provides an overview of init containers: specialized containers that run before app containers in a PodThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. . 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
- Using init containers
- Detailed behavior
- What's next

Understanding init containers

A PodThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. can have multiple containers running apps within it, but it can also have one or more init containers, which are run before the app containers are started.

Init containers are exactly like regular containers, except:

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

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

To specify an init container for a Pod, add the `initContainers` field into the Pod specification, as an array of objects of type Container, 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 readiness probes because they must run to completion before the Pod can be ready.

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

Using init containers

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

- Init containers can contain 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 SecretsStores sensitive information, such as passwords, OAuth tokens, and ssh keys. 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 ServiceA way to expose an application running on a set of Pods as a network 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_SERV
  ICE_PORT/register -d 'instance=$(<POD_NAME>)&ip=$(<POD_IP>)'
  ```

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

  ```
  sleep 60
  ```

- Clone a Git repository into a VolumeA directory containing data, accessible to the containers in a pod.

- Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app container. For example, place the `POD_IP` value in a configuration and generate the main app configuration file using Jinja.

Init containers in use

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

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep
3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup myservice; do echo
```

```
waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting
for mydb; sleep 2; done;']
```

You can start this Pod by running:

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

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

And check on its status with:

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

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

or for more details:

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

```
Name:          myapp-pod
Namespace:     default
[...]
Labels:        app=myapp
Status:        Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:         Running
[...]
  init-mydb:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Containers:
  myapp-container:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Events:
  FirstSeen    LastSeen    Count    From
SubObjectPath                           Type
Reason        Message
  ---------    --------    -----    ----
-------------                           --------
```

```
------        -------
  16s          16s         1        {default-
scheduler }
Normal        Scheduled     Successfully assigned myapp-pod to
172.17.4.201
  16s          16s         1        {kubelet 172.17.4.201}
spec.initContainers{init-myservice}     Normal
Pulling       pulling image "busybox"
  13s          13s         1        {kubelet 172.17.4.201}
spec.initContainers{init-myservice}     Normal
Pulled        Successfully pulled image "busybox"
  13s          13s         1        {kubelet 172.17.4.201}
spec.initContainers{init-myservice}     Normal
Created       Created container with docker id 5ced34a04634;
Security:[seccomp=unconfined]
  13s          13s         1        {kubelet 172.17.4.201}
spec.initContainers{init-myservice}     Normal
Started       Started container with docker id 5ced34a04634
```

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

```
kubectl logs myapp-pod -c init-myservice # Inspect the first
init container
kubectl logs myapp-pod -c init-mydb      # Inspect the second
init container
```

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

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

```
---
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
---
apiVersion: v1
kind: Service
metadata:
  name: mydb
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
```

To create the `mydb` and `myservice` services:

```
kubectl apply -f services.yaml
```

```
service/myservice created
service/mydb created
```

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

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

```
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

This simple example should provide some inspiration for you to create your own init containers. What's next contains a link to a more detailed example.

Detailed behavior

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

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

If the Pod restarts, 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

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on November 06, 2019 at 8:42 PM PST by Clarify init container use case for security (#17404) (Page History)

Edit This Page

Pod Preset

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

Understanding Pod Presets

A `Pod Preset` is an API resource for injecting additional runtime requirements into a Pod at creation time. You use label selectors to specify the Pods to which a given Pod Preset applies.

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

For more information about the background, see the design proposal for PodPreset.

How It Works

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

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

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

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

Disable Pod Preset for a Specific Pod

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

Enable Pod Preset

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

1. You have enabled the API type `settings.k8s.io/v1alpha1/podpreset`. For example, this can be done by including `settings.k8s.io/v1alpha1=true` in the `--runtime-config` option for the API server. In minikube add this flag `--extra-config=apiserver.runtime-config=settings.k8s.io/v1alpha1=true` while starting the cluster.

2. You have enabled the admission controller `PodPreset`. One way to doing this is to include `PodPreset` in the `--enable-admission-plugins` option value specified for the API server. In minikube add this flag

   ```
   --extra-config=apiserver.enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,MutatingAdmissionWebhook,ValidatingAdmissionWebhook,ResourceQuota,PodPreset
   ```

   while starting the cluster.

3. You have defined your Pod Presets by creating `PodPreset` objects in the namespace you will use.

What's next

- Injecting data into a Pod using PodPreset

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on July 21, 2019 at 2:22 PM PST by Improved PodPreset documentation (#15409) (Page History)

Edit This Page

Pod Topology Spread Constraints

FEATURE STATE: `Kubernetes v1.16` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

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.

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

How Disruption Budgets Work

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

Cluster managers and hosting providers should use tools which respect Pod Disruption Budgets by calling the Eviction API instead of directly deleting pods or deployments. Examples are the `kubectl drain` command and the Kubernetes-on-GCE cluster upgrade script (`cluster/gce/upgrade.sh`).

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

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

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

The "intended" number of pods is computed from the `.spec.replicas` of the pods controller. The controller is discovered from the pods using the `.metadata.ownerReferences` of the object.

PDBs cannot prevent involuntary disruptions from occurring, but they do count against the budget.

Pods which are deleted or unavailable due to a rolling upgrade to an application do count against the disruption budget, but controllers (like deployment and stateful-set) are not limited by PDBs when doing rolling upgrades - the handling of failures during application updates is configured in the controller spec. (Learn about updating a deployment.)

When a pod is evicted using the eviction API, it is gracefully terminated (see `terminationGracePeriodSeconds` in PodSpec.)

PDB Example

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

node-1	node-2	node-3
pod-a *available*	pod-b *available*	pod-c *available*
pod-x *available*		

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

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

node-1 *draining*	**node-2**	**node-3**
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*		

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

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

Now the cluster is in this state:

node-1 *draining*	**node-2**	**node-3**
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*	pod-d *starting*	pod-y

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

node-1 *drained*	**node-2**	**node-3**
	pod-b *available*	pod-c *available*
	pod-d *starting*	pod-y

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

The cluster state now looks like this:

node-1 *drained*	**node-2**	**node-3**
	pod-b *available*	pod-c *available*
	pod-d *available*	pod-y

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

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

node-1 *drained*	**node-2**	**node-3**	***no node***
	pod-b *available*	pod-c *available*	pod-e *pending*
	pod-d *available*	pod-y	

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

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

- how many replicas an application needs
- how long it takes to gracefully shutdown an instance
- how long it takes a new instance to start up
- the type of controller
- the cluster's resource capacity

Separating Cluster Owner and Application Owner Roles

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

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

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

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

How to perform Disruptive Actions on your Cluster

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

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

What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.
- Learn more about draining nodes

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on August 12, 2019 at 8:19 PM PST by Update disruptions.md (#15811) (Page History)

Edit This Page

Ephemeral Containers

FEATURE STATE: `Kubernetes v1.16` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

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.

The link a ReplicaSet has to its Pods is 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 ControllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. 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

```
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","meta
data":{"annotations":{},"labels":{"app":"guestbook","tier":"front
end"},"name":"frontend",...
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 just ReplicaSet. In Kubernetes 1.9 the API version `apps/v1` on the ReplicaSet kind is the current version and is enabled by default. The API version `apps/v1beta2` is deprecated. Refer to the first lines of the `frontend.yaml` example for guidance.

A ReplicaSet also needs a `.spec` section.

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/frontend' \
> -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' \
> -H "Content-Type: application/json"
```

Deleting just a ReplicaSet

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

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

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

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

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 20, 2020 at 9:37 PM PST by Update output of creating replicaset in controllers/replicaset (#19088) (Page History)

Edit This Page

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

```
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
```

```
replicationcontroller/nginx created
```

Check on the status of the ReplicationController using this command:

```
kubectl describe replicationcontrollers/nginx
```

```
Name:        nginx
Namespace:   default
Selector:    app=nginx
Labels:      app=nginx
Annotations:    <none>
Replicas:    3 current / 3 desired
Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=nginx
  Containers:
   nginx:
    Image:              nginx
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen       LastSeen     Count
From                        SubobjectPath    Type
Reason                Message
  ---------       --------     -----
----                        -------------    ----
------                -------
  20s             20s          1        {replication-
controller }                 Normal    SuccessfulCreate
Created pod: nginx-qrm3m
  20s             20s          1        {replication-
controller }                 Normal    SuccessfulCreate
Created pod: nginx-3ntk0
  20s             20s          1        {replication-
controller }                 Normal    SuccessfulCreate
Created pod: nginx-4ok8v
```

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

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

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

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

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

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

Writing a ReplicationController Spec

As with all other Kubernetes config, a ReplicationController needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see object management .

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 just 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, simply delete the ReplicationController object.

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

Isolating pods from a ReplicationController

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

Common usage patterns

Rescheduling

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

Scaling

The ReplicationController makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply updating the `replicas` field.

Rolling updates

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

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

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

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

Rolling update is implemented in the client tool `kubectl rolling-update`. Visit `kubectl rolling-update` task for more concrete examples.

Multiple release tracks

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

For instance, a service might target all pods with `tier in (frontend), environment in (prod)`. Now say you have 10 replicated pods that make up this tier. But you want to be able to â€˜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 simply ensures that the desired number of pods matches its label selector and are operational. Currently, only terminated pods are excluded from its count. In the future, readiness 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, rolling-update) 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 in a similar fashion as `kubectl rolling-update`. Deployments are recommended if you want this rolling update functionality, because unlike `kubectl rolling-update`, they are declarative, server-side, and have additional features.

Bare Pods

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

Job

Use a `Job` 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 AP Replication Controller.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page

Page last modified on November 06, 2019 at 2:21 PM PST by rcs can not be a shortcut in kubectl command (#17443) (Page History)

Edit This Page

Deployments

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

You describe a *desired state* in a Deployment, and the Deployment ControllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. 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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

In this example:

- A Deployment named `nginx-deployment` is created, indicated by the `.metadata.name` field.
- The Deployment creates three replicated Pods, indicated by the `replicas` field.

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

> **Note:** The `matchLabels` field is a map of {key,value} pairs. A single {key,value} in the `matchLabels` map is equivalent to an element of `matchExpressions`, whose key field is "key" the

operator is "In", and the values array contains only "value". All of the requirements, from both `matchLabels` and `matchExpressions`, must be satisfied in order to match.

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

Follow the steps given below to create the above Deployment:

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

1. Create the Deployment by running the following command:

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

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

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 cluster.
    - `DESIRED` displays the desired number of *replicas* of the application, which you define when you create the Deployment. This is the *desired state*.
    - `CURRENT` displays how many replicas are currently running.
    - `UP-TO-DATE` displays the number of replicas that have been updated to achieve the desired state.
    - `AVAILABLE` displays how many replicas of the application are available to your users.
    - `AGE` displays the amount of time that the application has been running.

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

3. To see the Deployment rollout status, run `kubectl rollout status deployment.v1.apps/nginx-deployment`. The output is similar to this:

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

   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 following output is returned:

   ```
   NAME                                READY     STATUS
   RESTARTS   AGE       LABELS
   nginx-deployment-75675f5897-7ci7o   1/1       Running   0
         18s       app=nginx,pod-template-hash=3123191453
   nginx-deployment-75675f5897-kzszj   1/1       Running   0
         18s       app=nginx,pod-template-hash=3123191453
   nginx-deployment-75675f5897-qqcnn   1/1       Running   0
         18s       app=nginx,pod-template-hash=3123191453
   ```

   The created ReplicaSet ensures that there are three `nginx` Pods.

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

Pod-template-hash label

Note: 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:1.9.1` image instead of the `nginx:1.7.9` image.

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

   or simply use the following command:

   ```
   kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1 --record
   ```

   The output is similar to this:

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

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

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

   The output is similar to this:

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

2. To see the rollout status, run:

   ```
   kubectl rollout status deployment.v1.apps/nginx-deployment
   ```

   The output is similar to this:

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

   or

   ```
   deployment.apps/nginx-deployment successfully rolled out
   ```

Get more details on your updated Deployment:

- After the rollout succeeds, you can view the Deployment by running `kubectl get deployments`. The output is similar to this:

  ```
  NAME               READY   UP-TO-DATE   AVAILABLE   AGE
  nginx-deployment   3/3     3            3           36s
  ```

- Run `kubectl get rs` to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas.

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME                          DESIRED   CURRENT   READY   AGE
  nginx-deployment-1564180365   3         3         3       6s
  nginx-deployment-2035384211   0         0         0       36s
  ```

- Running `get pods` should now show only the new Pods:

  ```
  kubectl get pods
  ```

  The output is similar to this:

  ```
  NAME                                READY     STATUS    
  RESTARTS   AGE
  nginx-deployment-1564180365-khku8   1/1       Running   
  0          14s
  nginx-deployment-1564180365-nacti   1/1       Running   
  0          14s
  nginx-deployment-1564180365-z9gth   1/1       Running   
  0          14s
  ```

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

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

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

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

- Get details of your Deployment:

```
kubectl describe deployments
```

The output is similar to this:

```
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Thu, 30 Nov 2017 10:56:25 +0000
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision=2
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3
available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
Labels:  app=nginx
 Containers:
  nginx:
    Image:        nginx:1.9.1
    Port:         80/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-1564180365 (3/3 replicas
created)
Events:
  Type    Reason             Age   From
Message
  ----    ------             ----  ----
-------
  Normal  ScalingReplicaSet  2m    deployment-controller
Scaled up replica set nginx-deployment-2035384211 to 3
  Normal  ScalingReplicaSet  24s   deployment-controller
Scaled up replica set nginx-deployment-1564180365 to 1
  Normal  ScalingReplicaSet  22s   deployment-controller
Scaled down replica set nginx-deployment-2035384211 to 2
  Normal  ScalingReplicaSet  22s   deployment-controller
Scaled up replica set nginx-deployment-1564180365 to 2
  Normal  ScalingReplicaSet  19s   deployment-controller
Scaled down replica set nginx-deployment-2035384211 to 1
  Normal  ScalingReplicaSet  19s   deployment-controller
Scaled up replica set nginx-deployment-1564180365 to 3
  Normal  ScalingReplicaSet  14s   deployment-controller
Scaled down replica set nginx-deployment-2035384211 to 0
```

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

Rollover (aka multiple updates in-flight)

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

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

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

Label selector updates

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

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

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

created, but note that the removed label still exists in any existing Pods and ReplicaSets.

Rolling Back a Deployment

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

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

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

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

  The output is similar to this:

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

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

  ```
  kubectl rollout status deployment.v1.apps/nginx-deployment
  ```

  The output is similar to this:

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

- Press Ctrl-C to stop the above rollout status watch. For more information on stuck rollouts, read more here.

- You see that the number of old replicas (`nginx-deployment-1564180365` and `nginx-deployment-2035384211`) is 2, and new replicas (nginx-deployment-3066724191) is 1.

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME                          DESIRED   CURRENT   READY   AGE
  nginx-deployment-1564180365   3         3         3       25s
  ```

```
nginx-deployment-2035384211   0         0         0       36s
nginx-deployment-3066724191   1         1         0       6s
```

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

  ```
  kubectl get pods
  ```

  The output is similar to this:

  ```
  NAME                                READY
  STATUS             RESTARTS   AGE
  nginx-deployment-1564180365-70iae   1/1
  Running            0          25s
  nginx-deployment-1564180365-jbqqo   1/1
  Running            0          25s
  nginx-deployment-1564180365-hysrc   1/1
  Running            0          25s
  nginx-deployment-3066724191-08mng   0/1
  ImagePullBackOff   0          6s
  ```

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

- Get the description of the Deployment:

  ```
  kubectl describe deployment
  ```

  The output is similar to this:

  ```
  Name:           nginx-deployment
  Namespace:      default
  CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
  Labels:         app=nginx
  Selector:       app=nginx
  Replicas:       3 desired | 1 updated | 4 total | 3
  available | 1 unavailable
  StrategyType:       RollingUpdate
  MinReadySeconds:    0
  RollingUpdateStrategy:  25% max unavailable, 25% max surge
  Pod Template:
    Labels:  app=nginx
    Containers:
     nginx:
      Image:        nginx:1.91
      Port:         80/TCP
      Host Port:    0/TCP
      Environment:  <none>
      Mounts:       <none>
    Volumes:        <none>
  ```

```
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    ReplicaSetUpdated
OldReplicaSets:     nginx-deployment-1564180365 (3/3
replicas created)
NewReplicaSet:      nginx-deployment-3066724191 (1/1
replicas created)
Events:
  FirstSeen LastSeen    Count   From
SubObjectPath   Type        Reason              Message
  --------- --------    -----   ----
-------------   --------    ------              -------
  1m        1m          1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled up replica set nginx-deployment-2035384211 to 3
  22s       22s         1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled up replica set nginx-deployment-1564180365 to 1
  22s       22s         1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled down replica set nginx-deployment-2035384211 to 2
  22s       22s         1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled up replica set nginx-deployment-1564180365 to 2
  21s       21s         1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled down replica set nginx-deployment-2035384211 to 1
  21s       21s         1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled up replica set nginx-deployment-1564180365 to 3
  13s       13s         1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled down replica set nginx-deployment-2035384211 to 0
  13s       13s         1       {deployment-
controller }                Normal      ScalingReplicaSet
Scaled up replica set nginx-deployment-3066724191 to 1
```

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

Checking Rollout History of a Deployment

Follow the steps given below to check the rollout history:

1. First, check the revisions of this Deployment:

   ```
   kubectl rollout history deployment.v1.apps/nginx-deployment
   ```

   The output is similar to this:

```
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           kubectl apply --filename=https://k8s.io/examples/
controllers/nginx-deployment.yaml --record=true
2           kubectl set image deployment.v1.apps/nginx-
deployment nginx=nginx:1.9.1 --record=true
3           kubectl set image deployment.v1.apps/nginx-
deployment nginx=nginx:1.91 --record=true
```

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

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

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/nginx-deployment nginx=nginx:1.9.1 --
record=true
  Containers:
   nginx:
    Image:      nginx:1.9.1
    Port:       80/TCP
     QoS Tier:
        cpu:      BestEffort
        memory:   BestEffort
    Environment Variables:      <none>
  No volumes.
```

Rolling Back to a Previous Revision

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

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

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

The output is similar to this:

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

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

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

The output is similar to this:

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

For more details about rollout related commands, read [kubectl rollout]().

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.v1.apps/nginx-deployment nginx=nginx:
1.9.1 --record=true
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3
available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.9.1
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-c4747d96c (3/3 replicas
created)
Events:
  Type    Reason              Age   From
Message
  ----    ------              ----  ----
-------
  Normal  ScalingReplicaSet   12m   deployment-controller
Scaled up replica set nginx-deployment-75675f5897 to 3
  Normal  ScalingReplicaSet   11m   deployment-controller
Scaled up replica set nginx-deployment-c4747d96c to 1
  Normal  ScalingReplicaSet   11m   deployment-controller
Scaled down replica set nginx-deployment-75675f5897 to 2
  Normal  ScalingReplicaSet   11m   deployment-controller
Scaled up replica set nginx-deployment-c4747d96c to 2
  Normal  ScalingReplicaSet   11m   deployment-controller
Scaled down replica set nginx-deployment-75675f5897 to 1
  Normal  ScalingReplicaSet   11m   deployment-controller
Scaled up replica set nginx-deployment-c4747d96c to 3
  Normal  ScalingReplicaSet   11m   deployment-controller
Scaled down replica set nginx-deployment-75675f5897 to 0
  Normal  ScalingReplicaSet   11m   deployment-controller
Scaled up replica set nginx-deployment-595696685f to 1
  Normal  DeploymentRollback  15s   deployment-controller
Rolled back deployment "nginx-deployment" to revision 2
  Normal  ScalingReplicaSet   15s   deployment-controller
Scaled down replica set nginx-deployment-595696685f to 0
```

Scaling a Deployment

You can scale a Deployment by using the following command:

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
```

The output is similar to this:

```
deployment.apps/nginx-deployment scaled
```

Assuming horizontal Pod autoscaling is enabled in your cluster, you can setup an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.

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

The output is similar to this:

```
deployment.apps/nginx-deployment scaled
```

Proportional scaling

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

For example, you are running a Deployment with 10 replicas, maxSurge=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=n
  ginx:sometag
  ```

  The output is similar to this:

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

- The image update starts a new rollout with ReplicaSet nginx-deployment-1989198191, but it's blocked due to the `maxUnavailable` requirement that you mentioned above. Check out the rollout status:

  ```
  kubectl get rs
  ```

  The output is similar to this:

```
NAME                          DESIRED   CURRENT   READY
AGE
nginx-deployment-1989198191   5         5         0
9s
nginx-deployment-618515232    8         8         8
1m
```

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

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

```
kubectl get deploy
```

The output is similar to this:

```
NAME                 DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
nginx-deployment     15        18        7
8           7m
```

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

```
kubectl get rs
```

The output is similar to this:

```
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191   7         7         0         7m
nginx-deployment-618515232    11        11        11        7m
```

Pausing and Resuming a Deployment

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

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

  ```
  kubectl get deploy
  ```

The output is similar to this:

```
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     3         3         3            3           1m
```

Get the rollout status:

```
kubectl get rs
```

The output is similar to this:

```
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         1m
```

- Pause by running the following command:

```
kubectl rollout pause deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
deployment.apps/nginx-deployment paused
```

- Then update the image of the Deployment:

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=n
ginx:1.9.1
```

The output is similar to this:

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

- Notice that no new rollout started:

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
deployments "nginx"
REVISION  CHANGE-CAUSE
1   <none>
```

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

```
kubectl get rs
```

The output is similar to this:

```
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         2m
```

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

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

The output is similar to this:

```
deployment.apps/nginx-deployment resource requirements
updated
```

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

- Eventually, resume the Deployment and observe a new ReplicaSet coming up with all the new updates:

  ```
  kubectl rollout resume deployment.v1.apps/nginx-deployment
  ```

  The output is similar to this:

  ```
  deployment.apps/nginx-deployment resumed
  ```

- Watch the status of the rollout until it's done.

  ```
  kubectl get rs -w
  ```

  The output is similar to this:

  ```
  NAME               DESIRED   CURRENT   READY     AGE
  nginx-2142116321   2         2         2         2m
  nginx-3926361531   2         2         0         6s
  nginx-3926361531   2         2         1         18s
  nginx-2142116321   1         2         2         2m
  nginx-2142116321   1         2         2         2m
  nginx-3926361531   3         2         1         18s
  nginx-3926361531   3         2         1         18s
  nginx-2142116321   1         1         1         2m
  nginx-3926361531   3         3         1         18s
  nginx-3926361531   3         3         2         19s
  nginx-2142116321   0         1         1         2m
  nginx-2142116321   0         1         1         2m
  nginx-2142116321   0         0         0         2m
  nginx-3926361531   3         3         3         20s
  ```

- Get the status of the latest rollout:

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME               DESIRED   CURRENT   READY     AGE
  nginx-2142116321   0         0         0         2m
  nginx-3926361531   3         3         3         28s
  ```

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

Deployment status

A Deployment enters various states during its lifecycle. It can be progressing 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.v1.apps/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 of 3 updated replicas are
available...
deployment.apps/nginx-deployment successfully rolled out
$ echo $?
0
```

Failed Deployment

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

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

One way you can detect this condition is to specify a deadline parameter in your Deployment spec: ([`.spec.progressDeadlineSeconds`]). `.spec.progressDeadlineSeconds` denotes the number of seconds the Deployment controller waits before indicating (in the Deployment status) that the Deployment progress has stalled.

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

```
kubectl patch deployment.v1.apps/nginx-deployment -p '{"spec":
{"progressDeadlineSeconds":600}}'
```

The output is similar to this:

```
deployment.apps/nginx-deployment patched
```

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

- Type=Progressing
- Status=False
- Reason=ProgressDeadlineExceeded

See the Kubernetes API conventions for more information on status conditions.

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

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

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

```
kubectl describe deployment nginx-deployment
```

The output is similar to this:

```
<...>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     True    ReplicaSetUpdated
  ReplicaFailure  True    FailedCreate
<...>
```

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

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: Replica set "nginx-deployment-4262182780" is
progressing.
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods "nginx-
deployment-4262182780-" is forbidden: exceeded quota:
      object-counts, requested: pods=1, used: pods=3, limited:
pods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2
```

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

```
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     False   ProgressDeadlineExceeded
  ReplicaFailure  True    FailedCreate
```

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

```
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
```

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

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

```
kubectl rollout status deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been
updated...
error: deployment "nginx" exceeded its progress deadline
$ echo $?
1
```

Operating on a failed deployment

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

Clean up Policy

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

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

Canary Deployment

If you want to roll out releases to a subset of users or servers using the Deployment, you can create multiple Deployments, one for each release, following the canary pattern described in managing resources.

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.

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 an 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`.

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. 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.

Rollback To

Field `.spec.rollbackTo` has been deprecated in API versions `extensions/v1beta1` and `apps/v1beta1`, and is no longer supported in API versions starting `apps/v1beta2`. Instead, `kubectl rollout undo` as introduced in Rolling Back to a Previous Revision should be used.

Revision History Limit

A Deployment's revision history is stored in the ReplicaSets it controls.

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

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

Paused

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

Alternative to Deployments

kubectl rolling-update

`kubectl rolling-update` updates Pods and ReplicationControllers in a similar fashion. But Deployments are recommended, since they are declarative, server side, and have additional features, such as rolling back to any previous revision even after the rolling update is done.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on November 22, 2019 at 6:43 AM PST by Fix invalid command (kubectl rolling-update) (#17607) (Page History)

Edit This Page

StatefulSets

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

Manages the deployment and scaling of a set of PodsThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. *, and provides guarantees about the ordering and uniqueness* of these Pods.

Like a DeploymentAn API object that manages a replicated application. , a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

Using StatefulSets

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.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

In the above example:

- A Headless Service, named `nginx`, is used to control the network domain.
- The StatefulSet, named `web`, has a Spec that indicates that 3 replicas of the nginx container will be launched in unique Pods.
- The `volumeClaimTemplates` will provide stable storage using PersistentVolumes provisioned by a PersistentVolume Provisioner.

Pod Selector

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

Pod Identity

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

Ordinal Index

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

Stable Network ID

Each Pod in a StatefulSet derives its hostname from the name of the StatefulSet and the ordinal of the Pod. The pattern for the constructed hostname is `$(statefulset name)-$(ordinal)`. The example above will create three Pods named `web-0,web-1,web-2`. A StatefulSet can use a Headless Service to control the domain of its Pods. The domain managed by this Service takes the form: `$(service name).$(namespace).svc.cluster.local`, where "cluster.local" is the cluster domain. As each Pod is created, it gets a matching DNS subdomain, taking the form: `$(podname).$(governing service domain)`, where the governing service is defined by the `serviceName` field on the StatefulSet.

As mentioned in the limitations 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
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	web-{0..N-1}.nginx.default.s
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	web-{0..N-1}.nginx.foo.svc.c
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	web-{0..N-1}.nginx.foo.

> **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 ControllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. 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.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on December 01, 2019 at 1:17 PM PST by Tweak StatefulSet concept page for YAML rendering (#17891) (Page History)

Edit This Page

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, such as `glusterd`, `ceph`, on each node.
- running a logs collection daemon on every node, such as `fluentd` or `filebeat`.
- running a node monitoring daemon on every node, such as Prometheus Node Exporter, Flowmill, Sysdig Agent, `collectd`, Dynatrace OneAgent, AppDynamics Agent, Datadog agent, New Relic agent, Ganglia `gmond`, Instana Agent or Elastic Metricbeat.

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

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

- Create a DaemonSet based on the YAML file:

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

Required Fields

As with all other Kubernetes config, a DaemonSet needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see deploying applications, configuring containers, and object management using kubectl documents.

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.

Also you should not normally create any Pods whose labels match this selector, either directly, via another DaemonSet, or via another workload resource such as ReplicaSet. Otherwise, the DaemonSet ControllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. will think that those Pods were created by it. Kubernetes will not stop you from doing this. One case where you might want to do this is manually create a Pod with a different value on a node for testing.

Running Pods on Only Some Nodes

If you specify a `.spec.template.spec.nodeSelector`, then the DaemonSet controller will create Pods on nodes which match that node selector. 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 v1.17` stable
This feature is *stable*, meaning:

Edit This Page

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
- Controlling how the garbage collector deletes dependents
- Known issues
- What's next

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. This means: 1) Namespace-scoped dependents can only specify owners in the same namespace, and owners that are cluster-scoped. 2) Cluster-scoped dependents can only specify cluster-scoped owners, but not namespace-scoped owners.

Controlling how the garbage collector deletes dependents

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

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

Foreground cascading deletion

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

- The object is still visible via the REST API
- The object's `deletionTimestamp` is set
- The object's `metadata.finalizers` contains the value "foregroundDeletion".

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

Note that in the "foregroundDeletion", only dependents with `ownerReference.blockOwnerDeletion=true` block the deletion of the owner object. Kubernetes version 1.7 added an admission controller that controls user access to set `blockOwnerDeletion` to true based on delete permissions on the owner object, so that unauthorized dependents cannot delay deletion of an owner object.

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

Background cascading deletion

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

Setting the cascading deletion policy

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

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

Here's an example that deletes dependents in background:

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

Here's an example that deletes dependents in foreground:

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

Here's an example that orphans dependents:

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

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

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

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

Additional note on Deployments

Prior to 1.7, When using cascading deletes with Deployments you *must* use `propagationPolicy: Foreground` to delete not only the ReplicaSets created, but also their Pods. If this type of *propagationPolicy* is not used, only the ReplicaSets will be deleted, and the Pods will be orphaned. See kubeadm/#149 for more information.

Known issues

Tracked at #26120

What's next

Design Doc 1

Design Doc 2

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on July 24, 2019 at 4:36 AM PST by Update garbage-collection.md (#15495) (Page History)

Edit This Page

TTL Controller for Finished Resources

FEATURE STATE: `Kubernetes v1.12` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

Jobs - Run to Completion

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

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

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

- Running an example Job
- Writing a Job Spec
- Handling Pod and Container Failures
- Job Termination and Cleanup
- Clean Up Finished Jobs Automatically
- Job Patterns
- Advanced Usage
- Alternatives
- Cron Jobs

Running an example Job

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

controllers/job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

You can run the example with this command:

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

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

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

```
kubectl describe jobs/pi
```

```
Name:           pi
Namespace:      default
Selector:       controller-uid=c9948307-e56d-4b5d-8302-
ae2d7b7da67c
Labels:         controller-uid=c9948307-e56d-4b5d-8302-
ae2d7b7da67c
                job-name=pi
Annotations:    kubectl.kubernetes.io/last-applied-configuration:
                  {"apiVersion":"batch/
v1","kind":"Job","metadata":{"annotations":
{},"name":"pi","namespace":"default"},"spec":{"backoffLimit":
4,"template":...
Parallelism:    1
Completions:    1
Start Time:     Mon, 02 Dec 2019 15:20:11 +0200
Completed At:   Mon, 02 Dec 2019 15:21:16 +0200
Duration:       65s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
           job-name=pi
  Containers:
   pi:
    Image:      perl
    Port:       <none>
    Host Port:  <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  14m   job-controller  Created pod:
pi-5rwd7
```

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

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

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

```
pi-5rwd7
```

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

View the standard output of one of the pods:

```
kubectl logs $pods
```

The output is similar to this:

```
3.
14159265358979323846264338327950288419716939937510582097494459230
78164062862089986280348253421170679821480865132823066470938446095
50582231725359408128481117450284102701938521105559644622948954930
38196442881097566593344612847564823378678316527120190914564856692
34603486104543266482133936072602491412737245870066063155881748815
20920962829254091715364367892590360011330530548820466521384146951
94151160943305727036575959195309218611738193261179310511854807446
23799627495673518857527248912279381830119491298336733624406566430
86021394946395224737190702179860943702770539217176293176752384674
81846766940513200056812714526356082778577134275778960917363717872
14684409012249534301465495853710507922796892589235420199561121290
21960864034418159813629774771309960518707211349999998372978049951
05973173281609631859502445945534690830264252230825334468503526193
11881710100031378387528865875332083814206171776691473035982534904
28755468731159562863882353787593751957781857780532171226806613001
92787661119590921642019893809525720106548586327886593615338182796
82303019520353018529689957736225994138912497217752834791315155748
57242454150695950829533116861727855889075098381754637464939319255
06040092770167113900984882401285836160356370766010471018194295559
61989467678374494482553797747268471040475346462080466842590694912
93313677028989152104752162056966024058038150193511253382430035587
64024749647326391419927260426992279678235478163600934172164121992
45863150302861829745557067498385054945885869269956909272107975093
02955321165344987202755960236480665499119881834797753566369807426
54252786255181841757467289097777279380008164706001614524919217321
72147723501414419735685481613611573525521334757418494684385233239
07394143334547762416862518983569485562099219222184272550254256887
67179049460165346680498862723279178608578438382796797668145410095
38837863609506800642251252051173929848960841284886269456042419652
85022210661186306744278622039194945047123713786960956364371917287
46776465757396241389086583264599581339047802759010
```

Writing a Job Spec

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

A Job also needs a [`.spec` section].

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 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 ControllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. 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 â€¦) capped at six minutes. The back-off count is reset if no new failed Pods appear before the Job's next status check.

> **Note:** Issue #54870 still exists for versions of Kubernetes prior to version 1.12
>
> **Note:** If your job has `restartPolicy = "OnFailure"`, keep in mind that your container running the Job will be terminated once the job backoff limit has been reached. This can make debugging the Job's executable more difficult. We suggest setting `restartPolicy = "Never"` when debugging the Job or using a logging system to ensure output from failed Jobs is not lost inadvertently.

Job Termination and Cleanup

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

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

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

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

Example:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 100
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

Note that both the Job spec and the Pod template spec 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
This feature is currently in a *alpha* state, meaning:

Edit This Page

CronJob

FEATURE STATE: `Kubernetes v1.8` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

EndpointSlices

FEATURE STATE: `Kubernetes v1.17` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Service

An abstract way to expose an application running on a set of PodsThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. as a network service.

With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

Motivation

Kubernetes PodsThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. are mortal. They are born and when they die, they are not resurrected. If you use a

DeploymentAn API object that manages a replicated application. to run your app, it can create and destroy Pods dynamically.

Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Enter *Services*.

Service resources

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a selectorAllows users to filter a list of resources based on labels. (see below for why you might want a Service *without* a selector).

For example, consider a stateless image-processing backend which is running with 3 replicas. Those replicas are fungibleâ€”frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves.

The Service abstraction enables this decoupling.

Cloud-native service discovery

If you're able to use Kubernetes APIs for service discovery in your application, you can query the API serverControl plane component that serves the Kubernetes API. for Endpoints, that get updated whenever the set of Pods in a Service changes.

For non-native applications, Kubernetes offers ways to place a network port or load balancer in between your application and the backend Pods.

Defining a Service

A Service in Kubernetes is a REST object, similar to a Pod. Like all of the REST objects, you can `POST` a Service definition to the API server to create a new instance.

For example, suppose you have a set of Pods that each listen on TCP port 9376 and carry a label `app=MyApp`:

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

This specification creates a new Service object named "my-service", which targets TCP port 9376 on any Pod with the `app=MyApp` label.

Kubernetes assigns this Service an IP address (sometimes called the "cluster IP"), which is used by the Service proxies (see Virtual IPs and service proxies below).

The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an Endpoint object also named "my-service".

> **Note:** A Service can map *any* incoming `port` to a `targetPort`. By default and for convenience, the `targetPort` is set to the same value as the `port` field.

Port definitions in Pods have names, and you can reference these names in the `targetPort` attribute of a Service. This works even if there is a mixture of Pods in the Service using a single configured name, with the same network protocol available via different port numbers. This offers a lot of flexibility for deploying and evolving your Services. For example, you can change the port numbers that Pods expose in the next version of your backend software, without breaking clients.

The default protocol for Services is TCP; you can also use any other supported protocol.

As many Services need to expose more than one port, Kubernetes supports multiple port definitions on a Service object. Each port definition can have the same `protocol`, or a different one.

Services without selectors

Services most commonly abstract access to Kubernetes Pods, but they can also abstract other kinds of backends. For example:

- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different NamespaceAn abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster. or on another cluster.
- You are migrating a workload to Kubernetes. Whilst evaluating the approach, you run only a proportion of your backends in Kubernetes.

In any of these scenarios you can define a Service *without* a Pod selector. For example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Because this Service has no selector, the corresponding Endpoint object is *not* created automatically. You can manually map the Service to the network address and port where it's running, by adding an Endpoint object manually:

```
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9376
```

> **Note:**
>
> The endpoint IPs *must not* be: loopback (127.0.0.0/8 for IPv4, ::1/128 for IPv6), or link-local (169.254.0.0/16 and 224.0.0.0/24 for IPv4, fe80::/64 for IPv6).
>
> Endpoint IP addresses cannot be the cluster IPs of other Kubernetes Services, because kube-proxykube-proxy is a network proxy that runs on each node in the cluster. doesn't support virtual IPs as a destination.

Accessing a Service without a selector works the same as if it had a selector. In the example above, traffic is routed to the single endpoint defined in the YAML: `192.0.2.42:9376` (TCP).

An ExternalName Service is a special case of Service that does not have selectors and uses DNS names instead. For more information, see the ExternalName section later in this document.

EndpointSlices

FEATURE STATE: `Kubernetes v1.17` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Service Topology

FEATURE STATE: `Kubernetes v1.17` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

DNS for Services and Pods

This page provides an overview of DNS support by Kubernetes.

- Introduction
- Services
- Pods
- What's next

Introduction

Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names.

What things get DNS names?

Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list will include the Pod's own namespace and the cluster's default domain. This is best illustrated by example:

Assume a Service named `foo` in the Kubernetes namespace `bar`. A Pod running in namespace `bar` can look up this service by simply doing a DNS query for `foo`. A Pod running in namespace `quux` can look up this service by doing a DNS query for `foo.bar`.

The following sections detail the supported record types and layout that is supported. Any other layout or names or queries that happen to work are considered implementation details and are subject to change without warning. For more up-to-date specification, see Kubernetes DNS-Based Service Discovery.

Services

A/AAAA records

"Normal" (not headless) Services are assigned a DNS A or AAAA record, depending on the IP family of the service, for a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. This resolves to the cluster IP of the Service.

"Headless" (without a cluster IP) Services are also assigned a DNS A or AAAA record, depending on the IP family of the service, for a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. Unlike normal Services, this resolves to the set of IPs of the pods selected by the Service. Clients are expected to consume the set or else use standard round-robin selection from the set.

SRV records

SRV Records are created for named ports that are part of normal or Headless Services. For each named port, the SRV record would have the form `_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster-domain.example`. For a regular service, this resolves to the port number and the domain name: `my-svc.my-namespace.svc.cluster-domain.example`. For a headless service, this resolves to multiple answers, one for each pod that is backing the service, and contains the port number and the domain name of the pod of the form `auto-generated-name.my-svc.my-namespace.svc.cluster-domain.example`.

Pods

Pod's hostname and subdomain fields

Currently when a pod is created, its hostname is the Pod's `metadata.name` value.

The Pod spec has an optional `hostname` field, which can be used to specify the Pod's hostname. When specified, it takes precedence over the Pod's name to be the hostname of the pod. For example, given a Pod with `hostname` set to "`my-host`", the Pod will have its hostname set to "`my-host`".

The Pod spec also has an optional `subdomain` field which can be used to specify its subdomain. For example, a Pod with `hostname` set to "`foo`", and `subdomain` set to "`bar`", in namespace "`my-namespace`", will have the fully qualified domain name (FQDN) "`foo.bar.my-namespace.svc.cluster-domain.example`".

Example:

```
apiVersion: v1
kind: Service
metadata:
  name: default-subdomain
spec:
  selector:
    name: busybox
  clusterIP: None
  ports:
  - name: foo # Actually, no port is needed.
    port: 1234
    targetPort: 1234
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostname: busybox-1
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
```

If there exists a headless service in the same namespace as the pod and with the same name as the subdomain, the cluster's DNS Server also returns an A or AAAA record for the Pod's fully qualified hostname. For example, given a Pod with the hostname set to "`busybox-1`" and the subdomain set to "`default-subdomain`", and a headless Service named "`default-subdomain`" in the same namespace, the pod will see its own FQDN as "`busybox-1.default-subdomain.my-namespace.svc.cluster-domain.example`". DNS serves an A or AAAA record at that name, pointing to the Pod's IP. Both pods "`busybox1`" and "`busybox2`" can have their distinct A or AAAA records.

The Endpoints object can specify the `hostname` for any endpoint addresses, along with its IP.

> **Note:** Because A or AAAA records are not created for Pod names, `hostname` is required for the Pod's A or AAAA record to be created. A Pod with no `hostname` but with `subdomain` will only create the A or AAAA record for the headless service (`default-subdomain.my-namespace.svc.cluster-domain.example`), pointing to the Pod's

IP address. Also, Pod needs to become ready in order to have a record unless `publishNotReadyAddresses=True` is set on the Service.

Pod's DNS Policy

DNS policies can be set on a per-pod basis. Currently Kubernetes supports the following pod-specific DNS policies. These policies are specified in the `dnsPolicy` field of a Pod Spec.

- "`Default`": The Pod inherits the name resolution configuration from the node that the pods run on. See related discussion for more details.
- "`ClusterFirst`": Any DNS query that does not match the configured cluster domain suffix, such as "`www.kubernetes.io`", is forwarded to the upstream nameserver inherited from the node. Cluster administrators may have extra stub-domain and upstream DNS servers configured. See related discussion for details on how DNS queries are handled in those cases.
- "`ClusterFirstWithHostNet`": For Pods running with hostNetwork, you should explicitly set its DNS policy "`ClusterFirstWithHostNet`".
- "`None`": It allows a Pod to ignore DNS settings from the Kubernetes environment. All DNS settings are supposed to be provided using the `dnsConfig` field in the Pod Spec. See Pod's DNS config subsection below.

  Note: "Default" is not the default DNS policy. If `dnsPolicy` is not explicitly specified, then "ClusterFirst" is used.

The example below shows a Pod with its DNS policy set to "`ClusterFirstWithHostNet`" because it has `hostNetwork` set to `true`.

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

Pod's DNS Config

Pod's DNS Config allows users more control on the DNS settings for a Pod.

The `dnsConfig` field is optional and it can work with any `dnsPolicy` settings. However, when a Pod's `dnsPolicy` is set to "`None`", the `dnsConfig` field has to be specified.

Below are the properties a user can specify in the `dnsConfig` field:

- `nameservers`: a list of IP addresses that will be used as DNS servers for the Pod. There can be at most 3 IP addresses specified. When the Pod's `dnsPolicy` is set to "`None`", the list must contain at least one IP address, otherwise this property is optional. The servers listed will be combined to the base nameservers generated from the specified DNS policy with duplicate addresses removed.
- `searches`: a list of DNS search domains for hostname lookup in the Pod. This property is optional. When specified, the provided list will be merged into the base search domain names generated from the chosen DNS policy. Duplicate domain names are removed. Kubernetes allows for at most 6 search domains.
- `options`: an optional list of objects where each object may have a `name` property (required) and a `value` property (optional). The contents in this property will be merged to the options generated from the specified DNS policy. Duplicate entries are removed.

The following is an example Pod with custom DNS settings:

[service/networking/custom-dns.yaml]()

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

When the Pod above is created, the container `test` gets the following contents in its `/etc/resolv.conf` file:

```
nameserver 1.2.3.4
search ns1.svc.cluster-domain.example my.dns.search.suffix
options ndots:2 edns0
```

For IPv6 setup, search path and name server should be setup like this:

```
kubectl exec -it dns-example -- cat /etc/resolv.conf
```

The output is similar to this:

```
nameserver fd00:79:30::a
search default.svc.cluster-domain.example svc.cluster-
domain.example cluster-domain.example
options ndots:5
```

Feature availability

The availability of Pod DNS Config and DNS Policy "`None`"" is shown as below.

k8s version	Feature support
1.14	Stable
1.10	Beta (on by default)
1.9	Alpha

What's next

For guidance on administering DNS configurations, check Configure DNS Service

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 12, 2020 at 8:48 PM PST by Add IPv6 DNS records to the dns pod service docs (#19079) (Page History)

Edit This Page

Connecting Applications with Services

The Kubernetes model for connecting containers

Now that you have a continuously running, replicated application you can expose it on a network. Before discussing the Kubernetes approach to networking, it is worthwhile to contrast it with the "normal" way networking works with Docker.

By default, Docker uses host-private networking, so containers can talk to other containers only if they are on the same machine. In order for Docker containers to communicate across nodes, there must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or ports must be allocated dynamically.

Coordinating port allocations across multiple developers or teams that provide containers is very difficult to do at scale, and exposes users to cluster-level issues outside of their control. Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports. This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT. The rest of this document elaborates on how you can run reliable services on such a networking model.

This guide uses a simple nginx server to demonstrate proof of concept. The same principles are embodied in a more complete Jenkins CI application.

Exposing pods to the cluster

We did this in a previous example, but let's do it once again and focus on the networking perspective. Create an nginx Pod, and note that it has a container port specification:

service/networking/run-my-nginx.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

This makes it accessible from any node in your cluster. Check the nodes the Pod is running on:

```
kubectl apply -f ./run-my-nginx.yaml
kubectl get pods -l run=my-nginx -o wide
```

```
NAME                        READY     STATUS    RESTARTS
AGE       IP            NODE
my-nginx-3800858182-jr4a2   1/1       Running   0
13s       10.244.3.4    kubernetes-minion-905m
my-nginx-3800858182-kna2y   1/1       Running   0
13s       10.244.2.5    kubernetes-minion-ljyd
```

Check your pods' IPs:

```
kubectl get pods -l run=my-nginx -o yaml | grep podIP
    podIP: 10.244.3.4
    podIP: 10.244.2.5
```

You should be able to ssh into any node in your cluster and curl both IPs. Note that the containers are *not* using port 80 on the node, nor are there any special NAT rules to route traffic to the pod. This means you can run multiple nginx pods on the same node all using the same containerPort and access them from any other pod or node in your cluster using IP. Like Docker, ports can still be published to the host node's interfaces, but the need for this is radically diminished because of the networking model.

You can read more about how we achieve this if you're curious.

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

You can create a Service for your 2 nginx replicas with `kubectl expose`:

```
kubectl expose deployment/my-nginx
```

```
service/my-nginx exposed
```

This is equivalent to `kubectl apply -f` the following yaml:

service/networking/nginx-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

This specification will create a Service which targets TCP port 80 on any Pod with the `run: my-nginx` label, and expose it on an abstracted Service port (`targetPort`: is the port the container accepts traffic on, `port`: is the abstracted Service port, which can be any port other pods use to access the Service). View Service API object to see the list of supported fields in service definition. Check your Service:

```
kubectl get svc my-nginx
```

```
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
my-nginx   ClusterIP   10.0.162.149   <none>        80/TCP    21s
```

As mentioned previously, a Service is backed by a group of Pods. These Pods are exposed through `endpoints`. The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named `my-nginx`. When a Pod dies, it is automatically removed from the endpoints, and new Pods matching the Service's selector will automatically get added to the endpoints. Check the endpoints, and note that the IPs are the same as the Pods created in the first step:

```
kubectl describe svc my-nginx
```

```
Name:                my-nginx
Namespace:           default
Labels:              run=my-nginx
Annotations:         <none>
Selector:            run=my-nginx
Type:                ClusterIP
IP:                  10.0.162.149
Port:                <unset> 80/TCP
Endpoints:           10.244.2.5:80,10.244.3.4:80
Session Affinity:    None
Events:              <none>
```

```
kubectl get ep my-nginx
```

```
NAME       ENDPOINTS                     AGE
my-nginx   10.244.2.5:80,10.244.3.4:80   1m
```

You should now be able to curl the nginx Service on `<CLUSTER-IP>:<PORT>` from any node in your cluster. Note that the Service IP is completely virtual, it never hits the wire. If you're curious about how this works you can read more about the service proxy.

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the CoreDNS cluster addon.

> **Note:** If the service environment variables are not desired (because possible clashing with expected program ones, too many variables to process, only using DNS, etc) you can disable this mode by setting the `enableServiceLinks` flag to `false` on the pod spec.

Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx Pods (your Pod name will be different):

```
kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 Pods and waiting for the Deployment to recreate them. This time around the Service exists *before* the replicas. This will give you scheduler-level Service spreading of your Pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
kubectl scale deployment my-nginx --replicas=0; kubectl scale
deployment my-nginx --replicas=2;

kubectl get pods -l run=my-nginx -o wide
```

```
NAME                        READY     STATUS    RESTARTS
AGE     IP            NODE
my-nginx-3800858182-e9ihh   1/1       Running   0
5s      10.244.2.7    kubernetes-minion-ljyd
my-nginx-3800858182-j4rm4   1/1       Running   0
5s      10.244.3.8    kubernetes-minion-905m
```

You may notice that the pods have different names, since they are killed and recreated.

```
kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
```

DNS

Kubernetes offers a DNS cluster addon Service that automatically assigns dns names to other Services. You can check if it's running on your cluster:

```
kubectl get services kube-dns --namespace=kube-system
```

```
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP
PORT(S)         AGE
kube-dns   ClusterIP   10.0.0.10    <none>        53/UDP,53/
TCP   8m
```

The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a DNS server that has assigned a name to that IP. Here we use the CoreDNS cluster addon (application name `kube-dns`), so you can talk to the Service from any pod in your cluster using standard methods (e.g. `gethostbyname()`). If CoreDNS isn't running, you can enable it referring to the CoreDNS README or Installing CoreDNS. Let's run another curl application to test this:

```
kubectl run curl --image=radial/busyboxplus:curl -i --tty
```

```
Waiting for pod default/curl-131556218-9fnch to be running,
status is Pending, pod ready: false
Hit enter for command prompt
```

Then, hit enter and run `nslookup my-nginx`:

```
[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      my-nginx
Address 1: 10.0.162.149
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure. For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A secret that makes the certificates accessible to pods

You can acquire all these from the nginx https example. This requires having go and make tools installed. If you don't want to install those, then follow the manual steps later. In short:

```
make keys KEY=/tmp/nginx.key CERT=/tmp/nginx.crt
kubectl create secret tls nginxsecret --key /tmp/nginx.key --
cert /tmp/nginx.crt
```

```
secret/nginxsecret created
```

```
kubectl get secrets
```

```
NAME                  TYPE
DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token
1         1d
nginxsecret           kubernetes.io/tls
2         1m
```

And also the configmap:

```
kubectl create configmap nginxconfigmap --from-file=default.conf
```

```
configmap/nginxconfigmap created
```

```
kubectl get configmaps
```

```
NAME              DATA   AGE
nginxconfigmap    1      114s
```

Following are the manual steps to follow in case you run into problems running make (on windows for example):

```
# Create a public private key pair
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /d/
tmp/nginx.key -out /d/tmp/nginx.crt -subj "/CN=my-nginx/O=my-
nginx"
# Convert the keys to base64 encoding
cat /d/tmp/nginx.crt | base64
cat /d/tmp/nginx.key | base64
```

Use the output from the previous commands to create a yaml file as follows. The base64 encoded value should all be on a single line.

```
apiVersion: "v1"
kind: "Secret"
metadata:
  name: "nginxsecret"
  namespace: "default"
type: kubernetes.io/tls
data:
  tls.crt: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURIekNDQWdlZ
0F3SUJBZ0lKQUp5M3lQK0pzMlpJTUEwR0NTcUdTSWIzRFFFQkJRVUFNQ1l4RVRBUE
JnTlYKQkFNVENHNW5hVzU0YzNaak1SRXdEd1lEVlFRS0V3aHVaMmx1ZUhOMll6QWV
GdzB4TnpFd01qWXdOekEzTVRKYQpGdzB4T0RFd01qWXdOekEzTVRKYU1DWXhFVEFQ
QmdOVkJBTVRDRzVuYVc1NGMzWmpNUkV3RHdZRFZRUUtFd2h1CloybHVlSE4yWXpDQ
0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBRENDQVFvQ2dnRUJBSjFxSU1SOV
dWM0IKMlZIQlRMRmtobDRONXljMEJxYUhIQktMSnJMcy8vdzZhU3hRS29GbHlJSU9
4NGUrMlN5ajBFcndCLzlYTnBwbQppeW1CL3JkRldkOXg5UWhBQUxCZkVaTmNiV3Ns
TVFVcnhBZW50VWt1dk1vLzgvMHRpbGhjc3paenJEYVJ4NEo5Ci82UVRtVVI3a0ZTW
UpOWTVQZkR3cGc3dlVvaDZmZ1Voam92VG42eHNVR0M2QURVODBpNXFlZWhNeVI1N2
lmU2YKNHZpaXdIY3hnL3lZR1JBRS9mRTRqakxCdmdONjc2SU90S01rZXV3R0ljNDF
hd05tNnNTSzRqYUNGeGpYSnZaZQp2by9kTlEybHhHWCtKT2l3SEhXbXNhdGp4WTRa
NVk3R1ZoK0QrWnYvcW1mMFgvbVY0Rmo1NzV3ajFMWVBocWtsCmdhSXZYRyt4U1FVQ
0F3RUFBYU5RTUU0d0hRWURWUjBPQkJZRUZPNG9OWkI3YXc1OUlsYkROMzhIYkduYn
hFVjcKTUI4R0ExVWRJd1FZTUJhQUZPNG9OWkI3YXc1OUlsYkROMzhIYkduYnhFVjd
NQXdHQTFVZEV3UUZNQU1CQWY4dwpEUVlKS29aSWh2Y05BUUVGQlFBRGdnRUJBRVhT
MW9FU0lFaXdyMDhWcVA0K2NwTHI3TW5FMTducDBvMm14alFvCjRGb0RvRjdRZnZqe
E04Tzd2TjB0clcxb2pGSW0vWDE4ZnZaL3k4ZzVaWG40Vm8zc3hKVmRBcStNZC9jTS
tzUGEKNmJjTkNUekZqeFpUV0UrKzE5NS9zb2dmOUZ3VDVDK3U2Q3B5N0M3MTZvUXR
UakViV05VdEt4cXI0Nk1OZWNCMApwRFhWZmdWQTRadkR4NFo3S2RiZDY5eXM3OVFH
Ymg5ZW1PZ05NZFlsSUswSGt0ejF5WU4vbVpmK3FqTkJqbWZjCkNnMnlwbGQ0Wi8rU
UNQZjl3SkoybFIrY2FnT0R4elBWcGxNSEcybzgvTHFDdnh6elZPUDUxeXdLZEtxaU
MwSVEKQ0I5T2wwWW5scE9UNEh1b2hSUzBPOStlMm9KdFZsNUIyczRpbDlhZ3RTVXF
xUlU9Ci0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K"
  tls.key: "LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1JSUV2UUlCQURBT
kJna3Foa2lHOXcwQkFRRUZBQVNDQktjd2dnU2pBZ0VBQW9JQkFRQ2RhaURFZlZsZH
```

```
dkbFIKd1V5eFpJWmVEZWNuTkFhbWh4d1NpeWF5N1AvOE9ta3NVQ3FCWmNpQ0RzZUh
2dGtzbzlCSzhBZi9WemFhWm9zcApnZjYzUlZuZmNmVUlRQUN3WHhHVFhHMXJKVEVG
SzhRSHA3VkpMcnpLUC9QOUxZcFlYTE0yYzZ3MmtjZUNmZitrCkU1bEVlNUJVbUNUV
09UM3c4S1lPNzFLSWVuNEZJWTZMMDUrc2JGQmd1Z0ExUE5JdWFubm9UTWtlZTRuMG
4rTDQKb3NCM01ZUDhtQmtRQlAzeE9JNHl3YjREZXUraURyU2pKSHJzQmlIT05Xc0R
adXJFaXVJMmdoY1kxeWIyWHI2UAozVFVOcGNSbC9pVG9zQngxcHJHclk4V09HZVdP
eGxZZmcvbWIvNnBuOUYvNWxlQlkrZStjSTlTMkQ0YXBKWUdpCkwxeHZzVWtGQWdNQ
kFBRUNnZ0VBZFhCK0xkbk8ySElOTGo5bWRsb25IUGlHWWVzZ294RGQwci9hQ1Zkan
k4dlEKTjIwL3FQWkUxek1yall6Ry9kVGhTMmMwc0QxaTBXSjdwR1lGb0xtdXlWTjl
tY0FXUTM5SjM0VHZaU2FFSWZWNgo5TE1jUHhNTmFsNjRLMFRVbUFQZytGam9QSFlh
UUxLOERLOUtnNXNrSE5pOWNzMlY5ckd6VWlVZWtBL0RBUlBTClI3L2ZjUFBacDRuR
WVBZmI3WTk1R1llb1p5V21SU3VKdlNyblBESGtUdW1vVlVWdkxMRHRzaG9reUxiTW
VtN3oKMmJzVmpwSW1GTHJqbGtmQXlpNHg0WjJrV3YyMFRrdWtsZU1jaVlMbjk4QWx
iRi9DSmRLM3QraTRoMTVlR2ZQegpoTnh3bk9QdlVTaDR2Q0o3c2Q5TmtEUGJvS2Jn
eVVHOXBYamZhRGR2UVFLQmdRRFFLM01nUkhkQ1pKNVFqZWFKClFGdXF4cHdnNzhZT
jQyL1NwenlUYmtGcVFoQWtyczJxWGx1MDZBRzhrZzIzQkswaHkzaE9zSGgxcXRVK3
NHZVAKOWRERHBsUWV0ODZsY2FlR3hoc0V0L1R6cEdtNGFKSm5oNzVVaTVGZk9QTDh
PTm1FZ3MxMVRhUldhNzZxelRyMgphRlpjQ2pWV1g0YnRSTHVwSkgrMjZnY0FhUUtC
Z1FEQmxVSUUzTnNVOFBBZEYvL25sQVB5VWs1T3lDdWc3dmVyClUycXlrdXFzYnBkS
i9hODViT1JhM05IVmpVM25uRGpHVHBWaE9JeXg5TEFrc2RwZEFjVmxvcG9HODhXYk
9lMTAKMUdqbnkySmdDK3JVWUZiRGtpUGx1K09IYnRnOXFYcGJMSHBzUVpsMGhucDB
YSFNYVm9CMUliQndnMGEyOFVadApCbFBtWmc2d1BRS0JnRHVIUVV2SDZHYTNDVUsx
NFdmOFhIcFFnMU16M2VvWTBPQm5iSDRvZUZKZmcraEppSXlnCm9RN3hqWldVR3BIc
3AyblRtcHErQWlSNzdyRVhsdlhtOElVU2FsbkNiRGlKY01Pc29RdFBZNS9NczJMRm
5LQTQKaENmL0pWb2FtZm1nZEN0ZGtFMXNINE9MR2lJVHdEbTRpb0dWZGIwMllnbzF
yb2htNUpLMUI3MkpBb0dBUW01UQpHNDhXOTVhL0w1eSt5dCsyZ3YvUHM2VnBvMjZl
TzRNQ3lJazJVem9ZWE9IYnNkODJkaC8xT2sybGdHZlI2K3VuCnc1YytZUXRSTHlhQ
md3MUtpbGhFZDBKTWU3cGpUSVpnQWJ0LzVPbnlDak9OVXN2aDJjS2lrQ1Z2dTZsZl
BjNkQKckliT2ZIaHhxV0RZK2Q1TGN1YSt2NzJ0RkxhenJsSlBsRzlOZHhrQ2dZRUF
5elIzT3UyMDNRVVV6bUlCRkwzZAp4Wm5XZ0JLSEo3TnNxcGFWb2RjL0d5aGVycjFD
ZzE2MmJaSjJDV2RsZkI0VEdtUjZZdmxTZEFOOFRwUWhFbUtKCnFBLzVzdHdxNWd0W
GVLOVJmMWxXK29xNThRNTBxMmk1NVdUTThoSDZhTjlaMTltZ0FGdE5VdGNqQUx2dF
YxdEYKWSs4WFJkSHJaRnBIWll2NWkwVW1VbGc9Ci0tLS0tRU5EIFBSSVZBVEUgS0V
ZLS0tLS0K"
```

Now create the secrets using the file:

```
kubectl apply -f nginxsecrets.yaml
kubectl get secrets
```

```
NAME                  TYPE
DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token
1         1d
nginxsecret           kubernetes.io/tls
2         1m
```

Now modify your nginx replicas to start an https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    run: my-nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      - name: configmap-volume
        configMap:
          name: nginxconfigmap
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
        - mountPath: /etc/nginx/conf.d
          name: configmap-volume
```

Noteworthy points about the nginx-secure-app manifest:

- It contains both Deployment and Service specification in the same file.
- The nginx server serves HTTP traffic on port 80 and HTTPS traffic on 443, and nginx Service exposes both ports.

- Each container has access to the keys through a volume mounted at `/etc/nginx/ssl`. This is setup *before* the nginx server is started.

  ```
  kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.yaml
  ```

At this point you can reach the nginx server from any node.

```
kubectl get pods -o yaml | grep -i podip
    podIP: 10.244.3.5
node $ curl -k https://10.244.3.5
...
<h1>Welcome to nginx!</h1>
```

Note how we supplied the `-k` parameter to curl in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell curl to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup. Let's test this from a pod (the same secret is being reused for simplicity, the pod only needs nginx.crt to access the Service):

service/networking/curlpod.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curl-deployment
spec:
  selector:
    matchLabels:
      app: curlpod
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: curlpod
        command:
        - sh
        - -c
        - while true; do sleep 1; done
        image: radial/busyboxplus:curl
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

```
kubectl apply -f ./curlpod.yaml
kubectl get pods -l app=curlpod
```

```
NAME                                   READY     STATUS
RESTARTS   AGE
curl-deployment-1515033274-1410r   1/1       Running
0          1m
```

```
kubectl exec curl-deployment-1515033274-1410r -- curl https://my-
nginx --cacert /etc/nginx/ssl/tls.crt
...
<title>Welcome to nginx!</title>
...
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The Service created in the last section already used `NodePort`, so your nginx HTTPS replica is ready to serve traffic on the internet if your node has a public IP.

```
kubectl get svc my-nginx -o yaml | grep nodePort -C 5
  uid: 07191fb3-f61a-11e5-8ae5-42010af00002
spec:
  clusterIP: 10.0.162.149
  ports:
  - name: http
    nodePort: 31704
    port: 8080
    protocol: TCP
    targetPort: 80
  - name: https
    nodePort: 32453
    port: 443
    protocol: TCP
    targetPort: 443
  selector:
    run: my-nginx
```

```
kubectl get nodes -o yaml | grep ExternalIP -C 1
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...

$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
...
<h1>Welcome to nginx!</h1>
```

Let's now recreate the Service to use a cloud load balancer, just change the `Type` of `my-nginx` Service from `NodePort` to `LoadBalancer`:

```
kubectl edit svc my-nginx
kubectl get svc my-nginx
```

```
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)               AGE
my-nginx   ClusterIP   10.0.162.149   162.222.184.144    80/TCP,
81/TCP,82/TCP  21s
```

```
curl https://<EXTERNAL-IP> -k
...
<title>Welcome to nginx!</title>
```

The IP address in the `EXTERNAL-IP` column is the one that is available on the public internet. The `CLUSTER-IP` is only available inside your cluster/private cloud network.

Note that on AWS, type `LoadBalancer` creates an ELB, which uses a (long) hostname, not an IP. It's too long to fit in the standard `kubectl get svc` output, in fact, so you'll need to do `kubectl describe service my-nginx` to see it. You'll see something like this:

```
kubectl describe service my-nginx
...
LoadBalancer Ingress:
a320587ffd19711e5a37606cf4a74574-1142138393.us-
east-1.elb.amazonaws.com
...
```

What's next

Kubernetes also supports Federated Services, which can span multiple clusters and cloud providers, to provide increased availability, better fault tolerance and greater scalability for your services. See the Federated Services User Guide for further information.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 12, 2020 at 5:23 PM PST by Improve wording of "Connecting Applications with Services" (#18076) (Page History)

Edit This Page

Ingress

FEATURE STATE: `Kubernetes v1.1` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Ingress Controllers

In order for the Ingress resource to work, the cluster must have an ingress controller running.

Unlike other types of controllers which run as part of the `kube-controller-manager` binary, Ingress controllers are not started automatically with a cluster. Use this page to choose the ingress controller implementation that best fits your cluster.

Kubernetes as a project currently supports and maintains GCE and nginx controllers.

- Additional controllers
- Using multiple Ingress controllers
- What's next

Additional controllers

- AKS Application Gateway Ingress Controller is an ingress controller that enables ingress to AKS clusters using the Azure Application Gateway.
- Ambassador API Gateway is an Envoy based ingress controller with community or commercial support from Datawire.
- AppsCode Inc. offers support and maintenance for the most widely used HAProxy based ingress controller Voyager.
- AWS ALB Ingress Controller enables ingress using the AWS Application Load Balancer.
- Contour is an Envoy based ingress controller provided and supported by VMware.
- Citrix provides an Ingress Controller for its hardware (MPX), virtualized (VPX) and free containerized (CPX) ADC for baremetal and cloud deployments.
- F5 Networks provides support and maintenance for the F5 BIG-IP Controller for Kubernetes.
- Gloo is an open-source ingress controller based on Envoy which offers API Gateway functionality with enterprise support from solo.io.
- HAProxy Ingress is a highly customizable community-driven ingress controller for HAProxy.
- HAProxy Technologies offers support and maintenance for the HAProxy Ingress Controller for Kubernetes. See the official documentation.
- Istio based ingress controller Control Ingress Traffic.
- Kong offers community or commercial support and maintenance for the Kong Ingress Controller for Kubernetes.
- NGINX, Inc. offers support and maintenance for the NGINX Ingress Controller for Kubernetes.
- Skipper HTTP router and reverse proxy for service composition, including use cases like Kubernetes Ingress, designed as a library to build your custom proxy

- Traefik is a fully featured ingress controller (Let's Encrypt, secrets, http2, websocket), and it also comes with commercial support by Containous.

Using multiple Ingress controllers

You may deploy any number of ingress controllers within a cluster. When you create an ingress, you should annotate each ingress with the appropriate `ingress.class` to indicate which ingress controller should be used if more than one exists within your cluster.

If you do not define a class, your cloud provider may use a default ingress controller.

Ideally, all ingress controllers should fulfill this specification, but the various ingress controllers operate slightly differently.

> **Note:** Make sure you review your ingress controller's documentation to understand the caveats of choosing it.

What's next

- Learn more about Ingress.
- Set up Ingress on Minikube with the NGINX Controller.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 09, 2020 at 2:06 PM PST by Added Azure Application Gateway Ingress Controller (#17832) (Page History)

Edit This Page

Network Policies

A network policy is a specification of how groups of podsThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. are allowed to communicate with each other and other network endpoints.

NetworkPolicy resources use labelsTags objects with identifying attributes that are meaningful and relevant to users. to select pods and define rules which specify what traffic is allowed to the selected pods.

Prerequisites

Network policies are implemented by the network plugin. To use network policies, you must be using a networking solution which supports NetworkPolicy. Creating a NetworkPolicy resource without a controller that implements it will have no effect.

Isolated and Non-isolated Pods

By default, pods are non-isolated; they accept traffic from any source.

Pods become isolated by having a NetworkPolicy that selects them. Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. (Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.)

Network policies do not conflict, they are additive. If any policy or policies select a pod, the pod is restricted to what is allowed by the union of those policies' ingress/egress rules. Thus, order of evaluation does not affect the policy result.

The NetworkPolicy resource

See the NetworkPolicy reference for a full definition of the resource.

An example NetworkPolicy might look like this:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
```

```
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

> **Note:** POSTing this to the API server for your cluster will have no effect unless your chosen networking solution supports network policy.

Mandatory Fields: As with all other Kubernetes config, a NetworkPolicy needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see Configure Containers Using a ConfigMap, and Object Management.

spec: NetworkPolicy spec has all the information needed to define a particular network policy in the given namespace.

podSelector: Each NetworkPolicy includes a `podSelector` which selects the grouping of pods to which the policy applies. The example policy selects pods with the label "role=db". An empty `podSelector` selects all pods in the namespace.

policyTypes: Each NetworkPolicy includes a `policyTypes` list which may include either `Ingress`, `Egress`, or both. The `policyTypes` field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no `policyTypes` are specified on a NetworkPolicy then by default `Ingress` will always be set and `Egress` will be set if the NetworkPolicy has any egress rules.

ingress: Each NetworkPolicy may include a list of whitelist `ingress` rules. Each rule allows traffic which matches both the `from` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port, from one of three sources, the first specified via an `ipBlock`, the second via a `namespaceSelector` and the third via a `podSelector`.

egress: Each NetworkPolicy may include a list of whitelist `egress` rules. Each rule allows traffic which matches both the `to` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port to any destination in `10.0.0.0/24`.

So, the example NetworkPolicy:

1. isolates "role=db" pods in the "default" namespace for both ingress and egress traffic (if they weren't already isolated)

2. (Ingress rules) allows connections to all pods in the "default" namespace with the label "role=db" on TCP port 6379 from:

    - any pod in the "default" namespace with the label "role=frontend"
    - any pod in a namespace with the label "project=myproject"
    - IP addresses in the ranges 172.17.0.0-172.17.0.255 and 172.17.2.0-172.17.255.255 (ie, all of 172.17.0.0/16 except 172.17.1.0/24)

3. (Egress rules) allows connections from any pod in the "default" namespace with the label "role=db" to CIDR 10.0.0.0/24 on TCP port 5978

See the Declare Network Policy walkthrough for further examples.

Behavior of `to` and `from` selectors

There are four kinds of selectors that can be specified in an `ingress` `from` section or `egress` `to` section:

podSelector: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.

namespaceSelector: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations.

namespaceSelector *and* **podSelector**: A single `to`/`from` entry that specifies both `namespaceSelector` and `podSelector` selects particular Pods within particular namespaces. Be careful to use correct YAML syntax; this policy:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
    podSelector:
      matchLabels:
        role: client
  ...
```

contains a single `from` element allowing connections from Pods with the label `role=client` in namespaces with the label `user=alice`. But *this* policy:

```
  ...
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          user: alice
    - podSelector:
        matchLabels:
          role: client
  ...
```

contains two elements in the `from` array, and allows connections from Pods in the local Namespace with the label `role=client`, *or* from any Pod in any namespace with the label `user=alice`.

When in doubt, use `kubectl describe` to see how Kubernetes has interpreted the policy.

ipBlock: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. These should be cluster-external IPs, since Pod IPs are ephemeral and unpredictable.

Cluster ingress and egress mechanisms often require rewriting the source or destination IP of packets. In cases where this happens, it is not defined whether this happens before or after NetworkPolicy processing, and the behavior may be different for different combinations of network plugin, cloud provider, `Service` implementation, etc.

In the case of ingress, this means that in some cases you may be able to filter incoming packets based on the actual original source IP, while in other cases, the "source IP" that the NetworkPolicy acts on may be the IP of a `LoadBalancer` or of the Pod's node, etc.

For egress, this means that connections from pods to `Service` IPs that get rewritten to cluster-external IPs may or may not be subject to `ipBlock`-based policies.

Default policies

By default, if no policies exist in a namespace, then all ingress and egress traffic is allowed to and from pods in that namespace. The following examples let you change the default behavior in that namespace.

Default deny all ingress traffic

You can create a "default" isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any ingress traffic to those pods.

service/networking/network-policy-default-deny-ingress.yaml

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will still be isolated. This policy does not change the default egress isolation behavior.

Default allow all ingress traffic

If you want to allow all traffic to all pods in a namespace (even if policies are added that cause some pods to be treated as "isolated"), you can create a policy that explicitly allows all traffic in that namespace.

service/networking/network-policy-allow-all-ingress.yaml

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

Default deny all egress traffic

You can create a "default" egress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any egress traffic from those pods.

service/networking/network-policy-default-deny-egress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed egress traffic. This policy does not change the default ingress isolation behavior.

Default allow all egress traffic

If you want to allow all traffic from all pods in a namespace (even if policies are added that cause some pods to be treated as "isolated"), you can create a policy that explicitly allows all egress traffic in that namespace.

service/networking/network-policy-allow-all-egress.yaml

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-egress
spec:
  podSelector: {}
  egress:
  - {}
  policyTypes:
  - Egress
```

Default deny all ingress and all egress traffic

You can create a "default" policy for a namespace which prevents all ingress AND egress traffic by creating the following NetworkPolicy in that namespace.

service/networking/network-policy-default-deny-all.yaml

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed ingress or egress traffic.

SCTP support

FEATURE STATE: `Kubernetes v1.12` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

Adding entries to Pod /etc/hosts with HostAliases

Adding entries to a Pod's /etc/hosts file provides Pod-level override of hostname resolution when DNS and other options are not applicable. In 1.7, users can add these custom entries with the HostAliases field in PodSpec.

Modification not using HostAliases is not suggested because the file is managed by Kubelet and can be overwritten on during Pod creation/restart.

- Default Hosts File Content
- Adding Additional Entries with HostAliases
- Why Does Kubelet Manage the Hosts File?

Default Hosts File Content

Let's start an Nginx Pod which is assigned a Pod IP:

```
kubectl run nginx --image nginx --generator=run-pod/v1
```

```
pod/nginx created
```

Examine a Pod IP:

```
kubectl get pods --output=wide
```

```
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4
worker0
```

The hosts file content would look like this:

```
kubectl exec nginx -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.200.0.4  nginx
```

By default, the `hosts` file only includes IPv4 and IPv6 boilerplates like `localhost` and its own hostname.

Adding Additional Entries with HostAliases

In addition to the default boilerplate, we can add additional entries to the `hosts` file to resolve `foo.local`, `bar.local` to `127.0.0.1` and `foo.remote`, `bar.remote` to `10.1.2.3`, we can by adding HostAliases to the Pod under `.spec.hostAliases`:

service/networking/hostaliases-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  restartPolicy: Never
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "foo.local"
    - "bar.local"
  - ip: "10.1.2.3"
    hostnames:
    - "foo.remote"
    - "bar.remote"
  containers:
  - name: cat-hosts
    image: busybox
    command:
    - cat
    args:
    - "/etc/hosts"
```

This Pod can be started with the following commands:

```
kubectl apply -f hostaliases-pod.yaml
```

```
pod/hostaliases-pod created
```

Examine a Pod IP and status:

```
kubectl get pod --output=wide
```

```
NAME                           READY     STATUS      RESTARTS
AGE       IP              NODE
hostaliases-pod                0/1       Completed   0
6s        10.200.0.5      worker0
```

The `hosts` file content would look like this:

```
kubectl logs hostaliases-pod
```

```
# Kubernetes-managed hosts file.
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
```

```
10.200.0.5  hostaliases-pod

# Entries added by HostAliases.
127.0.0.1   foo.local   bar.local
10.1.2.3foo.remote  bar.remote
```

With the additional entries specified at the bottom.

Why Does Kubelet Manage the Hosts File?

Kubelet manages the `hosts` file for each container of the Pod to prevent Docker from modifying the file after the containers have already been started.

Because of the managed-nature of the file, any user-written content will be overwritten whenever the `hosts` file is remounted by Kubelet in the event of a container restart or a Pod reschedule. Thus, it is not suggested to modify the contents of the file.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on May 21, 2019 at 9:42 AM PST by Switch to long command option (#14422) (Page History)

Edit This Page

IPv4/IPv6 dual-stack

FEATURE STATE: `Kubernetes v1.16` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

Volumes

On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers. First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state. Second, when running Containers together in a `Pod` it is often necessary to share files between those Containers. The Kubernetes `Volume` abstraction solves both of these problems.

Familiarity with Pods is suggested.

Background

Docker also has a concept of volumes, though it is somewhat looser and less managed. In Docker, a volume is simply a directory on disk or in another Container. Lifetimes are not managed and until very recently there were only local-disk-backed volumes. Docker now provides volume drivers, but the functionality is very limited for now (e.g. as of Docker 1.7 only one volume driver is allowed per Container and there is no way to pass parameters to volumes).

A Kubernetes volume, on the other hand, has an explicit lifetime - the same as the Pod that encloses it. Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.

At its core, a volume is just a directory, possibly with some data in it, which is accessible to the Containers in a Pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

To use a volume, a Pod specifies what volumes to provide for the Pod (the `.spec.volumes` field) and where to mount those into Containers (the `.spec.containers[*].volumeMounts` field).

A process in a container sees a filesystem view composed from their Docker image and volumes. The Docker image is at the root of the filesystem hierarchy, and any volumes are mounted at the specified paths within the image. Volumes can not mount onto other volumes or have hard links to other volumes. Each Container in the Pod must independently specify where to mount each volume.

Types of Volumes

Kubernetes supports several types of Volumes:

- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs
- cinder
- configMap
- csi
- downwardAPI
- emptyDir
- fc (fibre channel)
- flexVolume
- flocker
- gcePersistentDisk
- gitRepo (deprecated)
- glusterfs
- hostPath
- iscsi
- local
- nfs
- persistentVolumeClaim
- projected
- portworxVolume
- quobyte
- rbd
- scaleIO
- secret
- storageos
- vsphereVolume

We welcome additional contributions.

awsElasticBlockStore

An `awsElasticBlockStore` volume mounts an Amazon Web Services (AWS) EBS Volume into your Pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an EBS volume are preserved and the volume is merely unmounted. This means that an EBS volume can be pre-populated with data, and that data can be "handed off" between Pods.

> **Caution:** You must create an EBS volume using `aws ec2 create-volume` or the AWS API before you can use it.

There are some restrictions when using an `awsElasticBlockStore` volume:

- the nodes on which Pods are running must be AWS EC2 instances

- those instances need to be in the same region and availability-zone as the EBS volume
- EBS only supports a single EC2 instance mounting a volume

Creating an EBS volume

Before you can use an EBS volume with a Pod, you need to create it.

```
aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --
volume-type=gp2
```

Make sure the zone matches the zone you brought up your cluster in. (And also check that the size and EBS volume type are suitable for your use!)

AWS EBS Example configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

CSI Migration

FEATURE STATE: `Kubernetes v1.17` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Persistent Volumes

This document describes the current state of `PersistentVolumes` in Kubernetes. Familiarity with volumes is suggested.

Introduction

Managing storage is a distinct problem from managing compute instances. The `PersistentVolume` subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, we introduce two new API resources: `PersistentVolume` and `PersistentVolumeClaim`.

A `PersistentVolume` (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A `PersistentVolumeClaim` (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted once read/write or many times read-only).

While `PersistentVolumeClaims` allow a user to consume abstract storage resources, it is common that users need `PersistentVolumes` with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of `PersistentVolumes` that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the `StorageClass` resource.

See the detailed walkthrough with working examples.

Lifecycle of a volume and claim

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle:

Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

Static

A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

When none of the static PVs the administrator created match a user's `PersistentVolumeClaim`, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on `StorageClasses`: the PVC must request a [storage class]() and the administrator must have created and configured that class for dynamic provisioning to occur. Claims that request the class `""` effectively disable dynamic provisioning for themselves.

To enable dynamic storage provisioning based on storage class, the cluster administrator needs to enable the `DefaultStorageClass` [admission controller]() on the API server. This can be done, for example, by ensuring that `DefaultStorageClass` is among the comma-delimited, ordered list of values for the `--enable-admission-plugins` flag of the API server component. For more information on API server command-line flags, check [kube-apiserver]() documentation.

Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC. Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested. Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping, using a ClaimRef which is a bi-directional binding between the PersistentVolume and the PersistentVolumeClaim.

Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod.

Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule Pods and access their claimed PVs by including a `persistentVolumeClaim` in their Pod's volumes block. See below for syntax details.

Storage Object in Use Protection

The purpose of the Storage Object in Use Protection feature is to ensure that Persistent Volume Claims (PVCs) in active use by a Pod and Persistent Volume (PVs) that are bound to PVCs are not removed from the system, as this may result in data loss.

> **Note:** PVC is in active use by a Pod when a Pod object exists that is using the PVC.

If a user deletes a PVC in active use by a Pod, the PVC is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any Pods. Also, if an admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC.

You can see that a PVC is protected when the PVC's status is `Terminating` and the `Finalizers` list includes `kubernetes.io/pvc-protection`:

```
kubectl describe pvc hostpath
Name:          hostpath
Namespace:     default
StorageClass:  example-hostpath
Status:        Terminating
Volume:
Labels:        <none>
Annotations:   volume.beta.kubernetes.io/storage-class=example-
hostpath
               volume.beta.kubernetes.io/storage-provisioner=exam
ple.com/hostpath
Finalizers:    [kubernetes.io/pvc-protection]
...
```

You can see that a PV is protected when the PV's status is `Terminating` and the `Finalizers` list includes `kubernetes.io/pv-protection` too:

```
kubectl describe pv task-pv-volume
Name:            task-pv-volume
Labels:          type=local
Annotations:     <none>
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    standard
Status:          Terminating
Claim:
Reclaim Policy:  Delete
Access Modes:    RWO
Capacity:        1Gi
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /tmp/data
    HostPathType:
Events:            <none>
```

Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource. The reclaim policy for a `PersistentVolume` tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be Retained, Recycled, or Deleted.

Retain

The `Retain` reclaim policy allows for manual reclamation of the resource. When the `PersistentVolumeClaim` is deleted, the `PersistentVolume` still exists and the volume is considered "released". But it is not yet available for another claim because the previous claimant's data remains on the volume. An administrator can manually reclaim the volume with the following steps.

1. Delete the `PersistentVolume`. The associated storage asset in external infrastructure (such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume) still exists after the PV is deleted.
2. Manually clean up the data on the associated storage asset accordingly.
3. Manually delete the associated storage asset, or if you want to reuse the same storage asset, create a new `PersistentVolume` with the storage asset definition.

Delete

For volume plugins that support the `Delete` reclaim policy, deletion removes both the `PersistentVolume` object from Kubernetes, as well as the associated storage asset in the external infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume. Volumes that were dynamically provisioned inherit the reclaim policy of their `StorageClass`, which defaults to `Delete`. The administrator should configure the `StorageClass` according to users' expectations; otherwise, the PV must be edited or patched after it is created. See Change the Reclaim Policy of a PersistentVolume.

Recycle

> **Warning:** The `Recycle` reclaim policy is deprecated. Instead, the recommended approach is to use dynamic provisioning.

If supported by the underlying volume plugin, the `Recycle` reclaim policy performs a basic scrub (`rm -rf /thevolume/*`) on the volume and makes it available again for a new claim.

However, an administrator can configure a custom recycler Pod template using the Kubernetes controller manager command line arguments as described here. The custom recycler Pod template must contain a `volumes` specification, as shown in the example below:

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-recycler
  namespace: default
spec:
  restartPolicy: Never
  volumes:
  - name: vol
    hostPath:
      path: /any/path/it/will/be/replaced
  containers:
  - name: pv-recycler
    image: "k8s.gcr.io/busybox"
    command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?* /scrub/.[!.]* /scrub/*  && test -z \"$(ls -A /scrub)\" || exit 1"]
    volumeMounts:
    - name: vol
      mountPath: /scrub
```

However, the particular path specified in the custom recycler Pod template in the `volumes` part is replaced with the particular path of the volume that is being recycled.

Expanding Persistent Volumes Claims

FEATURE STATE: `Kubernetes v1.11` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Volume Snapshots

FEATURE STATE: `Kubernetes 1.17` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

CSI Volume Cloning

FEATURE STATE: `Kubernetes v1.16` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Storage Classes

This document describes the concept of a StorageClass in Kubernetes. Familiarity with volumes and persistent volumes is suggested.

Introduction

A `StorageClass` provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent. This concept is sometimes called "profiles" in other storage systems.

The StorageClass Resource

Each `StorageClass` contains the fields `provisioner`, `parameters`, and `reclaimPolicy`, which are used when a `PersistentVolume` belonging to the class needs to be dynamically provisioned.

The name of a `StorageClass` object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating `StorageClass` objects, and the objects cannot be updated once they are created.

Administrators can specify a default `StorageClass` just for PVCs that don't request any particular class to bind to: see the `PersistentVolumeClaim` section for details.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

Provisioner

Storage classes have a provisioner that determines what volume plugin is used for provisioning PVs. This field must be specified.

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	âœ“	AWS EBS
AzureFile	âœ“	Azure File
AzureDisk	âœ“	Azure Disk
CephFS	-	-
Cinder	âœ“	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	âœ“	-
GCEPersistentDisk	âœ“	GCE PD
Glusterfs	âœ“	Glusterfs
iSCSI	-	-
Quobyte	âœ“	Quobyte
NFS	-	-
RBD	âœ“	Ceph RBD
VsphereVolume	âœ“	vSphere
PortworxVolume	âœ“	Portworx Volume
ScaleIO	âœ“	ScaleIO
StorageOS	âœ“	StorageOS
Local	-	Local

You are not restricted to specifying the "internal" provisioners listed here (whose names are prefixed with "kubernetes.io" and shipped alongside Kubernetes). You can also run and specify external provisioners, which are independent programs that follow a specification defined by Kubernetes. Authors of external provisioners have full discretion over where their code lives, how the provisioner is shipped, how it needs to be run, what volume plugin it uses (including Flex), etc. The repository kubernetes-sigs/sig-storage-lib-external-provisioner houses a library for writing external provisioners that implements the bulk of the specification. Some external provisioners are listed under the repository kubernetes-incubator/external-storage.

For example, NFS doesn't provide an internal provisioner, but an external provisioner can be used. There are also cases when 3rd party storage vendors provide their own external provisioner.

Reclaim Policy

Persistent Volumes that are dynamically created by a storage class will have the reclaim policy specified in the `reclaimPolicy` field of the class, which can be either `Delete` or `Retain`. If no `reclaimPolicy` is specified when a `StorageClass` object is created, it will default to `Delete`.

Persistent Volumes that are created manually and managed via a storage class will have whatever reclaim policy they were assigned at creation.

Allow Volume Expansion

FEATURE STATE: `Kubernetes v1.11` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Volume Snapshot Classes

This document describes the concept of `VolumeSnapshotClass` in Kubernetes. Familiarity with volume snapshots and storage classes is suggested.

- Introduction
- The VolumeSnapshotClass Resource
- Parameters

Introduction

Just like `StorageClass` provides a way for administrators to describe the "classes" of storage they offer when provisioning a volume, `VolumeSnapshotClass` provides a way to describe the "classes" of storage when provisioning a volume snapshot.

The VolumeSnapshotClass Resource

Each `VolumeSnapshotClass` contains the fields `driver`, `deletionPolicy`, and `parameters`, which are used when a `VolumeSnapshot` belonging to the class needs to be dynamically provisioned.

The name of a `VolumeSnapshotClass` object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating `VolumeSnapshotClass` objects, and the objects cannot be updated once they are created.

Administrators can specify a default `VolumeSnapshotClass` just for VolumeSnapshots that don't request any particular class to bind to.

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snapclass
driver: hostpath.csi.k8s.io
deletionPolicy: Delete
parameters:
```

Driver

Volume snapshot classes have a driver that determines what CSI volume plugin is used for provisioning VolumeSnapshots. This field must be specified.

DeletionPolicy

Volume snapshot classes have a deletionPolicy. It enables you to configure what happens to a `VolumeSnapshotContent` when the `VolumeSnapshot` object it is bound to is to be deleted. The deletionPolicy of a volume snapshot can either be `Retain` or `Delete`. This field must be specified.

If the deletionPolicy is `Delete`, then the underlying storage snapshot will be deleted along with the `VolumeSnapshotContent` object. If the deletionPolicy is `Retain`, then both the underlying snapshot and `VolumeSnapshotContent` remain.

Parameters

Volume snapshot classes have parameters that describe volume snapshots belonging to the volume snapshot class. Different parameters may be accepted depending on the `driver`.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on December 10, 2019 at 5:41 AM PST by Official 1.17 Release Docs (#18011) (Page History)

Edit This Page

Dynamic Volume Provisioning

Dynamic volume provisioning allows storage volumes to be created on-demand. Without dynamic provisioning, cluster administrators have to manually make calls to their cloud or storage provider to create new storage volumes, and then create `PersistentVolume` objects to represent them in Kubernetes. The dynamic provisioning feature eliminates the need for cluster administrators to pre-provision storage. Instead, it automatically provisions storage when it is requested by users.

- Background
- Enabling Dynamic Provisioning
- Using Dynamic Provisioning
- Defaulting Behavior
- Topology Awareness

Background

The implementation of dynamic volume provisioning is based on the API object `StorageClass` from the API group `storage.k8s.io`. A cluster administrator can define as many `StorageClass` objects as needed, each specifying a *volume plugin* (aka *provisioner*) that provisions a volume and the set of parameters to pass to that provisioner when provisioning. A cluster administrator can define and expose multiple flavors of storage (from the same or different storage systems) within a cluster, each with a custom set of parameters. This design also ensures that end users don't have to worry about the complexity and nuances of how storage is provisioned, but still have the ability to select from multiple storage options.

More information on storage classes can be found here.

Enabling Dynamic Provisioning

To enable dynamic provisioning, a cluster administrator needs to pre-create one or more StorageClass objects for users. StorageClass objects define which provisioner should be used and what parameters should be passed to that provisioner when dynamic provisioning is invoked. The following manifest creates a storage class "slow" which provisions standard disk-like persistent disks.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

The following manifest creates a storage class "fast" which provisions SSD-like persistent disks.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

Using Dynamic Provisioning

Users request dynamically provisioned storage by including a storage class in their `PersistentVolumeClaim`. Before Kubernetes v1.6, this was done via the `volume.beta.kubernetes.io/storage-class` annotation. However, this annotation is deprecated since v1.6. Users now can and should instead use the `storageClassName` field of the `PersistentVolumeClaim` object. The value of this field must match the name of a `StorageClass` configured by the administrator (see below).

To select the "fast" storage class, for example, a user would create the following `PersistentVolumeClaim`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim1
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
  resources:
    requests:
      storage: 30Gi
```

This claim results in an SSD-like Persistent Disk being automatically provisioned. When the claim is deleted, the volume is destroyed.

Defaulting Behavior

Dynamic provisioning can be enabled on a cluster such that all claims are dynamically provisioned if no storage class is specified. A cluster administrator can enable this behavior by:

- Marking one `StorageClass` object as *default*;
- Making sure that the `DefaultStorageClass` admission controller is enabled on the API server.

An administrator can mark a specific `StorageClass` as default by adding the `storageclass.kubernetes.io/is-default-class` annotation to it. When a default `StorageClass` exists in a cluster and a user creates a `PersistentVolumeClaim` with `storageClassName` unspecified, the `DefaultStorageClass` admission controller automatically adds the `storageClassName` field pointing to the default storage class.

Note that there can be at most one *default* storage class on a cluster, or a `PersistentVolumeClaim` without `storageClassName` explicitly specified cannot be created.

Topology Awareness

In Multi-Zone clusters, Pods can be spread across Zones in a Region. Single-Zone storage backends should be provisioned in the Zones where Pods are scheduled. This can be accomplished by setting the Volume Binding Mode.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on May 30, 2019 at 7:34 AM PST by fix link to Storage Classes doc (#14551) (Page History)

Edit This Page

Node-specific Volume Limits

This page describes the maximum number of volumes that can be attached to a Node for various cloud providers.

Cloud providers like Google, Amazon, and Microsoft typically have a limit on how many volumes can be attached to a Node. It is important for Kubernetes to respect those limits. Otherwise, Pods scheduled on a Node could get stuck waiting for volumes to attach.

Kubernetes default limits

The Kubernetes scheduler has default limits on the number of volumes that can be attached to a Node:

Cloud service	Maximum volumes per Node
Amazon Elastic Block Store (EBS)	39
Google Persistent Disk	16
Microsoft Azure Disk Storage	16

Custom limits

You can change these limits by setting the value of the `KUBE_MAX_PD_VOLS` environment variable, and then starting the scheduler. CSI drivers might have a different procedure, see their documentation on how to customize their limits.

Use caution if you set a limit that is higher than the default limit. Consult the cloud provider's documentation to make sure that Nodes can actually support the limit you set.

The limit applies to the entire cluster, so it affects all Nodes.

Dynamic volume limits

FEATURE STATE: `Kubernetes v1.17` stable
This feature is *stable*, meaning:

Edit This Page

Configuration Best Practices

This document highlights and consolidates configuration best practices that are introduced throughout the user guide, Getting Started documentation, and examples.

This is a living document. If you think of something that is not on this list but might be useful to others, please don't hesitate to file an issue or submit a PR.

General Configuration Tips

- When defining configurations, specify the latest stable API version.

- Configuration files should be stored in version control before being pushed to the cluster. This allows you to quickly roll back a configuration change if necessary. It also aids cluster re-creation and restoration.

- Write your configuration files using YAML rather than JSON. Though these formats can be used interchangeably in almost all scenarios, YAML tends to be more user-friendly.

- Group related objects into a single file whenever it makes sense. One file is often easier to manage than several. See the guestbook-all-in-one.yaml file as an example of this syntax.

- Note also that many `kubectl` commands can be called on a directory. For example, you can call `kubectl apply` on a directory of config files.

- Don't specify default values unnecessarily: simple, minimal configuration will make errors less likely.

- Put object descriptions in annotations, to allow better introspection.

"Naked" Pods versus ReplicaSets, Deployments, and Jobs

- Don't use naked Pods (that is, Pods not bound to a ReplicaSet or Deployment) if you can avoid it. Naked Pods will not be rescheduled in the event of a node failure.

A Deployment, which both creates a ReplicaSet to ensure that the desired number of Pods is always available, and specifies a strategy to replace Pods (such as RollingUpdate), is almost always preferable to creating Pods directly, except for some explicit `restartPolicy: Never` scenarios. A Job may also be appropriate.

Services

- Create a Service before its corresponding backend workloads (Deployments or ReplicaSets), and before any workloads that need to access it. When Kubernetes starts a container, it provides environment variables pointing to all the Services which were running when the container was started. For example, if a Service named `foo` exists, all containers will get the following variables in their initial environment:

  ```
  FOO_SERVICE_HOST=<the host the Service is running on>
  FOO_SERVICE_PORT=<the port the Service is running on>
  ```

This does imply an ordering requirement - any `Service` that a `Pod` wants to access must be created before the `Pod` itself, or else the environment variables will not be populated. DNS does not have this restriction.

- An optional (though strongly recommended) cluster add-on is a DNS server. The DNS server watches the Kubernetes API for new `Services` and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all `Pods` should be able to do name resolution of `Services` automatically.

- Don't specify a `hostPort` for a Pod unless it is absolutely necessary. When you bind a Pod to a `hostPort`, it limits the number of places the Pod can be scheduled, because each <`hostIP`, `hostPort`, `protocol`> combination must be unique. If you don't specify the `hostIP` and `protocol` explicitly, Kubernetes will use `0.0.0.0` as the default `hostIP` and `TCP` as the default `protocol`.

If you only need access to the port for debugging purposes, you can use the apiserver proxy or `kubectl port-forward`.

If you explicitly need to expose a Pod's port on the node, consider using a NodePort Service before resorting to `hostPort`.

- Avoid using `hostNetwork`, for the same reasons as `hostPort`.

- Use headless Services (which have a `ClusterIP` of `None`) for easy service discovery when you don't need `kube-proxy` load balancing.

Using Labels

- Define and use labels that identify **semantic attributes** of your application or Deployment, such as `{ app: myapp, tier: frontend, phase: test, deployment: v3 }`. You can use these labels to select the appropriate Pods for other resources; for example, a Service that selects all `tier: frontend` Pods, or all `phase: test` components of `app: myapp`. See the guestbook app for examples of this approach.

A Service can be made to span multiple Deployments by omitting release-specific labels from its selector. Deployments make it easy to update a running service without downtime.

A desired state of an object is described by a Deployment, and if changes to that spec are *applied*, the deployment controller changes the actual state to the desired state at a controlled rate.

- You can manipulate labels for debugging. Because Kubernetes controllers (such as ReplicaSet) and Services match to Pods using selector labels, removing the relevant labels from a Pod will stop it from being considered by a controller or from being served traffic by a Service. If you remove the labels of an existing Pod, its controller will create a new Pod to take its place. This is a useful way to debug a previously "live" Pod in a "quarantine" environment. To interactively remove or add labels, use `kubectl label`.

Container Images

The imagePullPolicy and the tag of the image affect when the kubelet attempts to pull the specified image.

- `imagePullPolicy: IfNotPresent`: the image is pulled only if it is not already present locally.
- `imagePullPolicy: Always`: the image is pulled every time the pod is started.
- `imagePullPolicy` is omitted and either the image tag is `:latest` or it is omitted: `Always` is applied.
- `imagePullPolicy` is omitted and the image tag is present but not `:latest`: `IfNotPresent` is applied.
- `imagePullPolicy: Never`: the image is assumed to exist locally. No attempt is made to pull the image.

  Note: To make sure the container always uses the same version of the image, you can specify its digest, for example `sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2`. The digest uniquely identifies a specific version of the image, so it is never updated by Kubernetes unless you change the digest value.

  Note: You should avoid using the `:latest` tag when deploying containers in production as it is harder to track which version of the image is running and more difficult to roll back properly.

Note: The caching semantics of the underlying image provider make even `imagePullPolicy: Always` efficient. With Docker, for example, if the image already exists, the pull attempt is fast because all image layers are cached and no image download is needed.

Using kubectl

- Use `kubectl apply -f <directory>`. This looks for Kubernetes configuration in all `.yaml`, `.yml`, and `.json` files in `<directory>` and passes it to `apply`.
- Use label selectors for `get` and `delete` operations instead of specific object names. See the sections on label selectors and using labels effectively.
- Use `kubectl run` and `kubectl expose` to quickly create single-container Deployments and Services. See Use a Service to Access an Application in a Cluster for an example.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 19, 2020 at 3:14 AM PST by Latin Abbreviations "vs" Updated to "versus" (#19181) (Page History)

Edit This Page

Resource Bin Packing for Extended Resources

FEATURE STATE: `Kubernetes 1.16` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

Managing Compute Resources for Containers

When you specify a Pod, you can optionally specify how much CPU and memory (RAM) each Container needs. When Containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on. And when Containers have their limits specified, contention for resources on a node can be handled in a specified manner. For more details about the difference between requests and limits, see Resource QoS.

- Resource types
- Resource requests and limits of Pod and Container
- Meaning of CPU
- Meaning of memory
- How Pods with resource requests are scheduled
- How Pods with resource limits are run
- Monitoring compute resource usage
- Troubleshooting
- Local ephemeral storage
- Extended resources
- What's next

Resource types

CPU and *memory* are each a *resource type*. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes. If you're using Kubernetes v1.14 or newer, you can specify *huge page* resources. Huge pages are a Linux-specific feature where the node kernel allocates blocks of memory that are much larger than the default page size.

For example, on a system where the default page size is 4KiB, you could specify a limit, `hugepages-2Mi: 80Mi`. If the container tries allocating over 40 2MiB huge pages (a total of 80 MiB), that allocation fails.

> **Note:** You cannot overcommit `hugepages-*` resources. This is different from the `memory` and `cpu` resources.

CPU and memory are collectively referred to as *compute resources*, or just *resources*. Compute resources are measurable quantities that can be requested, allocated, and consumed. They are distinct from API resources. API resources, such as Pods and Services are objects that can be read and modified through the Kubernetes API server.

Resource requests and limits of Pod and Container

Each Container of a Pod can specify one or more of the following:

- `spec.containers[].resources.limits.cpu`
- `spec.containers[].resources.limits.memory`
- `spec.containers[].resources.limits.hugepages-<size>`
- `spec.containers[].resources.requests.cpu`
- `spec.containers[].resources.requests.memory`
- `spec.containers[].resources.requests.hugepages-<size>`

Although requests and limits can only be specified on individual Containers, it is convenient to talk about Pod resource requests and limits. A *Pod resource request/limit* for a particular resource type is the sum of the resource requests/limits of that type for each Container in the Pod.

Meaning of CPU

Limits and requests for CPU resources are measured in *cpu* units. One cpu, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 IBM vCPU
- 1 *Hyperthread* on a bare-metal Intel processor with Hyperthreading

Fractional requests are allowed. A Container with `spec.containers[].resources.requests.cpu` of `0.5` is guaranteed half as much CPU as one that asks for 1 CPU. The expression `0.1` is equivalent to the expression `100m`, which can be read as "one hundred millicpu". Some people say "one hundred millicores", and this is understood to mean the same thing. A request with a decimal point, like `0.1`, is converted to `100m` by the API, and precision finer than `1m` is not allowed. For this reason, the form `100m` might be preferred.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine.

Meaning of memory

Limits and requests for `memory` are measured in bytes. You can express memory as a plain integer or as a fixed-point integer using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

```
128974848, 129e6, 129M, 123Mi
```

Here's an example. The following Pod has two Containers. Each Container has a request of 0.25 cpu and 64MiB (2^{26} bytes) of memory. Each Container has a limit of 0.5 cpu and 128MiB of memory. You can say the Pod has a request of 0.5 cpu and 128 MiB of memory, and a limit of 1 cpu and 256MiB of memory.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

How Pods with resource requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum capacity for each of the resource types: the amount of CPU and memory it can provide for Pods. The scheduler ensures that, for each resource type, the sum of the resource requests of the scheduled Containers is less than the capacity of the node. Note that although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails. This protects against a resource shortage on a node when resource usage later increases, for example, during a daily peak in request rate.

How Pods with resource limits are run

When the kubelet starts a Container of a Pod, it passes the CPU and memory limits to the container runtime.

When using Docker:

- The `spec.containers[].resources.requests.cpu` is converted to its core value, which is potentially fractional, and multiplied by 1024. The greater of this number or 2 is used as the value of the `--cpu-shares` flag in the `docker run` command.

- The `spec.containers[].resources.limits.cpu` is converted to its millicore value and multiplied by 100. The resulting value is the total amount of CPU time that a container can use every 100ms. A container cannot use more than its share of CPU time during this interval.

  Note: The default quota period is 100ms. The minimum resolution of CPU quota is 1ms.

- The `spec.containers[].resources.limits.memory` is converted to an integer, and used as the value of the `--memory` flag in the `docker run` command.

If a Container exceeds its memory limit, it might be terminated. If it is restartable, the kubelet will restart it, as with any other type of runtime failure.

If a Container exceeds its memory request, it is likely that its Pod will be evicted whenever the node runs out of memory.

A Container might or might not be allowed to exceed its CPU limit for extended periods of time. However, it will not be killed for excessive CPU usage.

To determine whether a Container cannot be scheduled or is being killed due to resource limits, see the Troubleshooting section.

Monitoring compute resource usage

The resource usage of a Pod is reported as part of the Pod status.

If optional monitoring is configured for your cluster, then Pod resource usage can be retrieved from the monitoring system.

Troubleshooting

My Pods are pending with event message failedScheduling

If the scheduler cannot find any node where a Pod can fit, the Pod remains unscheduled until a place can be found. An event is produced each time the scheduler fails to find a place for the Pod, like this:

```
kubectl describe pod frontend | grep -A 3 Events
```

```
Events:
  FirstSeen LastSeen   Count  From          Subobject 
PathReason      Message
  36s   5s     6      {scheduler }              
FailedScheduling  Failed for reason PodExceedsFreeCPU and 
possibly others
```

In the preceding example, the Pod named "frontend" fails to be scheduled due to insufficient CPU resource on the node. Similar error messages can also suggest failure due to insufficient memory (PodExceedsFreeMemory). In general, if a Pod is pending with a message of this type, there are several things to try:

- Add more nodes to the cluster.
- Terminate unneeded Pods to make room for pending Pods.
- Check that the Pod is not larger than all the nodes. For example, if all the nodes have a capacity of `cpu: 1`, then a Pod with a request of `cpu: 1.1` will never be scheduled.

You can check node capacities and amounts allocated with the `kubectl describe nodes` command. For example:

```
kubectl describe nodes e2e-test-node-pool-4lw4
```

```
Name:            e2e-test-node-pool-4lw4
[ ... lines removed for clarity ...]
Capacity:
 cpu:                               2
 memory:                            7679792Ki
 pods:                              110
Allocatable:
 cpu:                               1800m
 memory:                            7474992Ki
 pods:                              110
[ ... lines removed for clarity ...]
Non-terminated Pods:        (5 in total)
  Namespace    Name                                  CPU 
Requests  CPU Limits  Memory Requests  Memory Limits
  ---------    ----                                  
-----------  ----------  --------------  ------------
  kube-system  fluentd-gcp-v1.38-28bv1               100m 
(5%)     0 (0%)      200Mi (2%)       200Mi (2%)
```

```
  kube-system  kube-dns-3297075139-61lj3             260m
(13%)    0 (0%)      100Mi (1%)       170Mi (2%)
  kube-system  kube-proxy-e2e-test-...               100m
(5%)     0 (0%)      0 (0%)           0 (0%)
  kube-system  monitoring-influxdb-grafana-v4-z1m12  200m
(10%)    200m (10%)  600Mi (8%)       600Mi (8%)
  kube-system  node-problem-detector-v0.1-fj7m3      20m
(1%)      200m (10%)  20Mi (0%)        100Mi (1%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests    CPU Limits    Memory Requests    Memory Limits
  ------------    ----------    ---------------    -------------
  680m (34%)      400m (20%)    920Mi (12%)        1070Mi (14%)
```

In the preceding output, you can see that if a Pod requests more than 1120m CPUs or 6.23Gi of memory, it will not fit on the node.

By looking at the `Pods` section, you can see which Pods are taking up space on the node.

The amount of resources available to Pods is less than the node capacity, because system daemons use a portion of the available resources. The `allocatable` field NodeStatus gives the amount of resources that are available to Pods. For more information, see Node Allocatable Resources.

The resource quota feature can be configured to limit the total amount of resources that can be consumed. If used in conjunction with namespaces, it can prevent one team from hogging all the resources.

My Container is terminated

Your Container might get terminated because it is resource-starved. To check whether a Container is being killed because it is hitting a resource limit, call `kubectl describe pod` on the Pod of interest:

```
kubectl describe pod simmemleak-hra99
```

```
Name:                           simmemleak-hra99
Namespace:                      default
Image(s):                       saadali/simmemleak
Node:                           kubernetes-node-tf0f/
10.240.216.66
Labels:                         name=simmemleak
Status:                         Running
Reason:
Message:
IP:                             10.244.2.75
Replication Controllers:        simmemleak (1/1 replicas created)
Containers:
  simmemleak:
    Image:  saadali/simmemleak
    Limits:
      cpu:                         100m
      memory:                      50Mi
    State:                         Running
      Started:                     Tue, 07 Jul 2015 12:54:41 -0700
    Last Termination State:        Terminated
      Exit Code:                   1
      Started:                     Fri, 07 Jul 2015 12:54:30 -0700
      Finished:                    Fri, 07 Jul 2015 12:54:33 -0700
    Ready:                         False
    Restart Count:                 5
Conditions:
  Type      Status
  Ready     False
Events:
  FirstSeen
LastSeen                         Count
From
SubobjectPath                              Reason      Message
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700  1
{scheduler }
     scheduled   Successfully assigned simmemleak-hra99 to
kubernetes-node-tf0f
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700  1      {kubelet kubernetes-node-tf0f}    implicitly
required container POD   pulled      Pod container image
"k8s.gcr.io/pause:0.8.0" already present on machine
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700  1      {kubelet kubernetes-node-tf0f}    implicitly
required container POD   created     Created with docker id
6a41280f516d
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700  1      {kubelet kubernetes-node-tf0f}    implicitly
required container POD   started     Started with docker id
6a41280f516d
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700  1      {kubelet kubernetes-node-tf0f}
spec.containers{simmemleak}          created     Created with
docker id 87348f12526a
```

In the preceding example, the `Restart Count: 5` indicates that the `simmemleak` Container in the Pod was terminated and restarted five times.

You can call `kubectl get pod` with the `-o go-template=...` option to fetch the status of previously terminated Containers:

```
kubectl get pod -o go-template='{{range.status.containerStatuses}
}{{"Container Name: "}}{{.name}}{{"\r\nLastState: "}}
{{.lastState}}{{end}}'  simmemleak-hra99
```

```
Container Name: simmemleak
LastState: map[terminated:map[exitCode:137 reason:OOM Killed
startedAt:2015-07-07T20:58:43Z finishedAt:2015-07-07T20:58:43Z
containerID:docker://
0e4095bba1feccdfe7ef9fb6ebffe972b4b14285d5acdec6f0d3ae8a22fad8b2]
]
```

You can see that the Container was terminated because of `reason:OOM Killed`, where `OOM` stands for Out Of Memory.

Local ephemeral storage

FEATURE STATE: `Kubernetes v1.17` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Pod Overhead

FEATURE STATE: `Kubernetes v1.16` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

Assigning Pods to Nodes

You can constrain a PodThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. to only be able to run on particular Node(s)A node is a worker machine in Kubernetes. , or to prefer to run on particular nodes. There are several ways to do this, and the recommended approaches all use label selectors to make the selection. Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (e.g. spread your pods across nodes, not place the pod on a node with insufficient free resources, etc.) but there are some circumstances where you may want more control on a node where a pod lands, e.g. to ensure that a pod ends up on a machine with an SSD attached to it, or to co-locate pods from two different services that communicate a lot into the same availability zone.

nodeSelector

`nodeSelector` is the simplest recommended form of node selection constraint. `nodeSelector` is a field of PodSpec. It specifies a map of key-value pairs. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels (it can have additional labels as well). The most common usage is one key-value pair.

Let's walk through an example of how to use `nodeSelector`.

Step Zero: Prerequisites

This example assumes that you have a basic understanding of Kubernetes pods and that you have set up a Kubernetes cluster.

Step One: Attach label to the node

Run `kubectl get nodes` to get the names of your cluster's nodes. Pick out the one that you want to add a label to, and then run `kubectl label nodes <node-name> <label-key>=<label-value>` to add a label to the node you've chosen. For example, if my node name is â€˜kubernetes-foo-node-1.c.a-robinson.internal' and my desired label is â€˜disktype=ssd', then I can run `kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd`.

You can verify that it worked by re-running `kubectl get nodes --show-labels` and checking that the node now has a label. You can also use `kubectl describe node "nodename"` to see the full list of labels of the given node.

Step Two: Add a nodeSelector field to your pod configuration

Take whatever pod config file you want to run, and add a nodeSelector section to it, like this. For example, if this is my pod config:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
```

Then add a nodeSelector like so:

pods/pod-nginx.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

When you then run `kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml`, the Pod will get scheduled on the node that you attached the label to. You can verify that it worked by running `kubectl get pods -o wide` and looking at the "NODE" that the Pod was assigned to.

Interlude: built-in node labels

In addition to labels you attach, nodes come pre-populated with a standard set of labels. These labels are

- `kubernetes.io/hostname`
- `failure-domain.beta.kubernetes.io/zone`
- `failure-domain.beta.kubernetes.io/region`
- `topology.kubernetes.io/zone`
- `topology.kubernetes.io/region`
- `beta.kubernetes.io/instance-type`
- `node.kubernetes.io/instance-type`
- `kubernetes.io/os`
- `kubernetes.io/arch`

> **Note:** The value of these labels is cloud provider specific and is not guaranteed to be reliable. For example, the value of `kubernetes.io/hostname` may be the same as the Node name in some environments and a different value in other environments.

Node isolation/restriction

Adding labels to Node objects allows targeting pods to specific nodes or groups of nodes. This can be used to ensure specific pods only run on nodes with certain isolation, security, or regulatory properties. When using labels for this purpose, choosing label keys that cannot be modified by the kubelet process on the node is strongly recommended. This prevents a compromised node from using its kubelet credential to set those labels on its own Node object, and influencing the scheduler to schedule workloads to the compromised node.

The `NodeRestriction` admission plugin prevents kubelets from setting or modifying labels with a `node-restriction.kubernetes.io/` prefix. To make use of that label prefix for node isolation:

1. Ensure you are using the [Node authorizer]() and have *enabled* the [NodeRestriction admission plugin]().
2. Add labels under the `node-restriction.kubernetes.io/` prefix to your Node objects, and use those labels in your node selectors. For example, `example.com.node-restriction.kubernetes.io/fips=true` or `example.com.node-restriction.kubernetes.io/pci-dss=true`.

Affinity and anti-affinity

`nodeSelector` provides a very simple way to constrain pods to nodes with particular labels. The affinity/anti-affinity feature, greatly expands the types of constraints you can express. The key enhancements are

1. the language is more expressive (not just "AND or exact match")
2. you can indicate that the rule is "soft"/"preference" rather than a hard requirement, so if the scheduler can't satisfy it, the pod will still be scheduled
3. you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located

The affinity feature consists of two types of affinity, "node affinity" and "inter-pod affinity/anti-affinity". Node affinity is like the existing `nodeSelector` (but with the first two benefits listed above), while inter-pod affinity/anti-affinity constrains against pod labels rather than node labels, as described in the third item listed above, in addition to having the first and second properties listed above.

Node affinity

Node affinity is conceptually similar to `nodeSelector` - it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

There are currently two types of node affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that *must* be met for a pod to be scheduled onto a node (just like `nodeSelector` but using a more expressive syntax), while the latter specifies *preferences* that the scheduler will try to enforce but will not guarantee. The "IgnoredDuringExecution" part of the names means that, similar to how `nodeSelector` works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node. In the future we plan to offer `r` equiredDuringSchedulingRequiredDuringExecution which will be just like requiredDuringSchedulingIgnoredDuringExecution except that it will evict pods from nodes that cease to satisfy the pods' node affinity requirements.

Thus an example of requiredDuringSchedulingIgnoredDuringExecution would be "only run the pod on nodes with Intel CPUs" and an example preferredDuringSchedulingIgnoredDuringExecution would be "try to run this set of pods in failure zone XYZ, but if it's not possible, then allow some to run elsewhere".

Node affinity is specified as field nodeAffinity of field affinity in the PodSpec.

Here's an example of a pod that uses node affinity:

pods/pod-with-node-affinity.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

This node affinity rule says the pod can only be placed on a node with a label whose key is kubernetes.io/e2e-az-name and whose value is either e2e-az1 or e2e-az2. In addition, among nodes that meet that criteria, nodes with a label whose key is another-node-label-key and whose value is another-node-label-value should be preferred.

You can see the operator `In` being used in the example. The new node affinity syntax supports the following operators: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`. You can use `NotIn` and `DoesNotExist` to achieve node anti-affinity behavior, or use [node taints]() to repel pods from specific nodes.

If you specify both `nodeSelector` and `nodeAffinity`, *both* must be satisfied for the pod to be scheduled onto a candidate node.

If you specify multiple `nodeSelectorTerms` associated with `nodeAffinity` types, then the pod can be scheduled onto a node **if one of** the `nodeSelectorTerms` is satisfied.

If you specify multiple `matchExpressions` associated with `nodeSelectorTerms`, then the pod can be scheduled onto a node **only if all** `matchExpressions` can be satisfied.

If you remove or change the label of the node where the pod is scheduled, the pod won't be removed. In other words, the affinity selection works only at the time of scheduling the pod.

The `weight` field in `preferredDuringSchedulingIgnoredDuringExecution` is in the range 1-100. For each node that meets all of the scheduling requirements (resource request, RequiredDuringScheduling affinity expressions, etc.), the scheduler will compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node matches the corresponding MatchExpressions. This score is then combined with the scores of other priority functions for the node. The node(s) with the highest total score are the most preferred.

Inter-pod affinity and anti-affinity

Inter-pod affinity and anti-affinity allow you to constrain which nodes your pod is eligible to be scheduled *based on labels on pods that are already running on the node* rather than based on labels on nodes. The rules are of the form "this pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more pods that meet rule Y". Y is expressed as a LabelSelector with an optional associated list of namespaces; unlike nodes, because pods are namespaced (and therefore the labels on pods are implicitly namespaced), a label selector over pod labels must specify which namespaces the selector should apply to. Conceptually X is a topology domain like node, rack, cloud provider zone, cloud provider region, etc. You express it using a `topologyKey` which is the key for the node label that the system uses to denote such a topology domain, e.g. see the label keys listed above in the section [Interlude: built-in node labels]().

> **Note:** Inter-pod affinity and anti-affinity require substantial amount of processing which can slow down scheduling in large clusters significantly. We do not recommend using them in clusters larger than several hundred nodes.

> **Note:** Pod anti-affinity requires nodes to be consistently labelled, i.e. every node in the cluster must have an appropriate label

matching `topologyKey`. If some or all nodes are missing the specified `topologyKey` label, it can lead to unintended behavior.

As with node affinity, there are currently two types of pod affinity and anti-affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution` which denote "hard" vs. "soft" requirements. See the description in the node affinity section earlier. An example of `requiredDuringSchedulingIgnoredDuringExecution` affinity would be "co-locate the pods of service A and service B in the same zone, since they communicate a lot with each other" and an example `preferredDuringSchedulingIgnoredDuringExecution` anti-affinity would be "spread the pods from this service across zones" (a hard requirement wouldn't make sense, since you probably have more pods than zones).

Inter-pod affinity is specified as field `podAffinity` of field `affinity` in the PodSpec. And inter-pod anti-affinity is specified as field `podAntiAffinity` of field `affinity` in the PodSpec.

An example of a pod that uses pod affinity:

pods/pod-with-pod-affinity.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
  - name: with-pod-affinity
    image: k8s.gcr.io/pause:2.0
```

The affinity on this pod defines one pod affinity rule and one pod anti-affinity rule. In this example, the `podAffinity` is `requiredDuringSchedulingIgnoredDuringExecution` while the `podAntiAffinity` is `preferredDuringSchedulingIgnoredDuringExecution`. The pod affinity rule says that the pod can be scheduled onto a node only if that node is in the same zone as at least one already-running pod that has a label with key "security" and value "S1". (More precisely, the pod is eligible to run on node N if node N has a label with key `failure-domain.beta.kubernetes.io/zone` and some value V such that there is at least one node in the cluster with key `failure-domain.beta.kubernetes.io/zone` and value V that is running a pod that has a label with key "security" and value "S1".) The pod anti-affinity rule says that the pod prefers not to be scheduled onto a node if that node is already running a pod with label having key "security" and value "S2". (If the `topologyKey` were `failure-domain.beta.kubernetes.io/zone` then it would mean that the pod cannot be scheduled onto a node if that node is in the same zone as a pod with label having key "security" and value "S2".) See the design doc for many more examples of pod affinity and anti-affinity, both the `requiredDuringSchedulingIgnoredDuringExecution` flavor and the `preferredDuringSchedulingIgnoredDuringExecution` flavor.

The legal operators for pod affinity and anti-affinity are `In`, `NotIn`, `Exists`, `DoesNotExist`.

In principle, the `topologyKey` can be any legal label-key. However, for performance and security reasons, there are some constraints on topologyKey:

1. For affinity and for `requiredDuringSchedulingIgnoredDuringExecution` pod anti-affinity, empty `topologyKey` is not allowed.
2. For `requiredDuringSchedulingIgnoredDuringExecution` pod anti-affinity, the admission controller `LimitPodHardAntiAffinityTopology` was introduced to limit `topologyKey` to `kubernetes.io/hostname`. If you want to make it available for custom topologies, you may modify the admission controller, or simply disable it.
3. For `preferredDuringSchedulingIgnoredDuringExecution` pod anti-affinity, empty `topologyKey` is interpreted as "all topologies" ("all topologies" here is now limited to the combination of `kubernetes.io/hostname`, `failure-domain.beta.kubernetes.io/zone` and `failure-domain.beta.kubernetes.io/region`).
4. Except for the above cases, the `topologyKey` can be any legal label-key.

In addition to `labelSelector` and `topologyKey`, you can optionally specify a list `namespaces` of namespaces which the `labelSelector` should match against (this goes at the same level of the definition as `labelSelector` and `topologyKey`). If omitted or empty, it defaults to the namespace of the pod where the affinity/anti-affinity definition appears.

All `matchExpressions` associated with `requiredDuringSchedulingIgnoredDuringExecution` affinity and anti-affinity must be satisfied for the pod to be scheduled onto a node.

More Practical Use-cases

Interpod Affinity and AntiAffinity can be even more useful when they are used with higher level collections such as ReplicaSets, StatefulSets, Deployments, etc. One can easily configure that a set of workloads should be co-located in the same defined topology, eg., the same node.

Always co-located in the same node

In a three node cluster, a web application has in-memory cache such as redis. We want the web-servers to be co-located with the cache as much as possible.

Here is the yaml snippet of a simple redis deployment with three replicas and selector label `app=store`. The deployment has `PodAntiAffinity` configured to ensure the scheduler does not co-locate replicas on a single node.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: redis-server
        image: redis:3.2-alpine
```

The below yaml snippet of the webserver deployment has `podAntiAffinity` and `podAffinity` configured. This informs the scheduler that all its replicas are to be co-located with pods that have selector label `app=store`. This will also ensure that each web-server replica does not co-locate on a single node.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
```

```
              - key: app
                operator: In
                values:
                - web-store
            topologyKey: "kubernetes.io/hostname"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: web-app
        image: nginx:1.12-alpine
```

If we create the above two deployments, our three node cluster should look like below.

node-1	**node-2**	**node-3**
webserver-1	*webserver-2*	*webserver-3*
cache-1	*cache-2*	*cache-3*

As you can see, all the 3 replicas of the `web-server` are automatically co-located with the cache as expected.

```
kubectl get pods -o wide
```

The output is similar to this:

```
NAME                           READY     STATUS    RESTARTS
AGE       IP           NODE
redis-cache-1450370735-6dzlj   1/1       Running   0
8m        10.192.4.2   kube-node-3
redis-cache-1450370735-j2j96   1/1       Running   0
8m        10.192.2.2   kube-node-1
redis-cache-1450370735-z73mh   1/1       Running   0
8m        10.192.3.1   kube-node-2
web-server-1287567482-5d4dz    1/1       Running   0
7m        10.192.2.3   kube-node-1
web-server-1287567482-6f7v5    1/1       Running   0
7m        10.192.4.3   kube-node-3
web-server-1287567482-s330j    1/1       Running   0
7m        10.192.3.2   kube-node-2
```

Never co-located in the same node

The above example uses `PodAntiAffinity` rule with `topologyKey: "kubernetes.io/hostname"` to deploy the redis cluster so that no two instances are located on the same host. See ZooKeeper tutorial for an example of a StatefulSet configured with anti-affinity for high availability, using the same technique.

nodeName

`nodeName` is the simplest form of node selection constraint, but due to its limitations it is typically not used. `nodeName` is a field of PodSpec. If it is non-empty, the scheduler ignores the pod and the kubelet running on the named node tries to run the pod. Thus, if `nodeName` is provided in the PodSpec, it takes precedence over the above methods for node selection.

Some of the limitations of using `nodeName` to select nodes are:

- If the named node does not exist, the pod will not be run, and in some cases may be automatically deleted.
- If the named node does not have the resources to accommodate the pod, the pod will fail and its reason will indicate why, e.g. OutOfmemory or OutOfcpu.
- Node names in cloud environments are not always predictable or stable.

Here is an example of a pod config file using the `nodeName` field:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: kube-01
```

The above pod will run on the node kube-01.

What's next

Taints allow a Node to *repel* a set of Pods.

The design documents for node affinity and for inter-pod affinity/anti-affinity contain extra background information about these features.

Once a Pod is assigned to a Node, the kubelet runs the Pod and allocates node-local resources. The topology manager can take part in node-level resource allocation decisions.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 17, 2020 at 1:32 AM PST by Fix the typo (#18732) (Page History)

Edit This Page

Taints and Tolerations

Node affinity, described here, is a property of *pods* that *attracts* them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite - they allow a *node* to *repel* a set of pods.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints. Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

- Concepts
- Example Use Cases
- Taint based Evictions
- Taint Nodes by Condition

Concepts

You add a taint to a node using kubectl taint. For example,

```
kubectl taint nodes node1 key=value:NoSchedule
```

places a taint on node `node1`. The taint has key `key`, value `value`, and taint effect `NoSchedule`. This means that no pod will be able to schedule onto `node1` unless it has a matching toleration.

To remove the taint added by the command above, you can run:

```
kubectl taint nodes node1 key:NoSchedule-
```

You specify a toleration for a pod in the PodSpec. Both of the following tolerations "match" the taint created by the `kubectl taint` line above, and thus a pod with either toleration would be able to schedule onto `node1`:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

```
tolerations:
- key: "key"
  operator: "Exists"
  effect: "NoSchedule"
```

Here's an example of a pod that uses tolerations:

pods/pod-with-toleration.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "example-key"
    operator: "Exists"
    effect: "NoSchedule"
```

A toleration "matches" a taint if the keys are the same and the effects are the same, and:

- the `operator` is `Exists` (in which case no `value` should be specified), or
- the `operator` is `Equal` and the `value`s are equal

`Operator` defaults to `Equal` if not specified.

> **Note:**
>
> There are two special cases:
>
> - An empty `key` with operator `Exists` matches all keys, values and effects which means this will tolerate everything.
>
>   ```
>   tolerations:
>   - operator: "Exists"
>   ```
>
> - An empty `effect` matches all effects with key `key`.
>
>   ```
>   tolerations:
>   - key: "key"
>     operator: "Exists"
>   ```

The above example used `effect` of `NoSchedule`. Alternatively, you can use `effect` of `PreferNoSchedule`. This is a "preference" or "soft" version of `NoSchedule` - the system will *try* to avoid placing a pod that does not tolerate the taint on the node, but it is not required. The third kind of `effect` is `NoExecute`, described later.

You can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all of a node's taints, then ignore the ones for which the pod has a matching toleration; the remaining un-ignored taints have the indicated effects on the pod. In particular,

- if there is at least one un-ignored taint with effect `NoSchedule` then Kubernetes will not schedule the pod onto that node
- if there is no un-ignored taint with effect `NoSchedule` but there is at least one un-ignored taint with effect `PreferNoSchedule` then Kubernetes will *try* to not schedule the pod onto the node
- if there is at least one un-ignored taint with effect `NoExecute` then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

For example, imagine you taint a node like this

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

And a pod has two tolerations:

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

In this case, the pod will not be able to schedule onto the node, because there is no toleration matching the third taint. But it will be able to continue running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

Normally, if a taint with effect `NoExecute` is added to a node, then any pods that do not tolerate the taint will be evicted immediately, and pods that do tolerate the taint will never be evicted. However, a toleration with `NoExecute` effect can specify an optional `tolerationSeconds` field that dictates how long the pod will stay bound to the node after the taint is added. For example,

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

means that if this pod is running and a matching taint is added to the node, then the pod will stay bound to the node for 3600 seconds, and then be evicted. If the taint is removed before that time, the pod will not be evicted.

Example Use Cases

Taints and tolerations are a flexible way to steer pods *away* from nodes or evict pods that shouldn't be running. A few of the use cases are

- **Dedicated Nodes**: If you want to dedicate a set of nodes for exclusive use by a particular set of users, you can add a taint to those nodes (say, `kubectl taint nodes nodename dedicated=groupName:NoSchedule`) and then add a corresponding toleration to their pods (this would be done most easily by writing a custom admission controller). The pods with the tolerations will then be allowed to use the tainted (dedicated) nodes as well as any other nodes in the cluster. If you want to dedicate the nodes to them *and* ensure they *only* use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g. `dedicated=groupName`), and the admission controller should additionally add a node affinity to require that the pods can only schedule onto nodes labeled with `dedicated=groupName`.

- **Nodes with Special Hardware**: In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes, thus leaving room for later-arriving pods that do need the specialized hardware. This can be done by tainting the nodes that have the specialized hardware (e.g. `kubectl taint nodes nodename special=true:NoSchedule` or `kubectl taint nodes nodename special=true:PreferNoSchedule`) and adding a corresponding toleration to pods that use the special hardware. As in the dedicated nodes use case, it is probably easiest to apply the tolerations using a custom admission controller. For example, it is recommended to use Extended Resources to represent the special hardware, taint your special hardware nodes with the extended resource name and run the ExtendedResourceToleration admission controller. Now, because the nodes are tainted, no pods without the toleration will schedule on them. But when you submit a pod that requests the extended resource, the `ExtendedResourceToleration` admission controller will automatically add the correct toleration to the pod and that pod will schedule on the special hardware nodes. This will make sure that these special hardware nodes are dedicated for pods requesting such hardware and you don't have to manually add tolerations to your pods.

- **Taint based Evictions (beta feature)**: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

Taint based Evictions

Earlier we mentioned the `NoExecute` taint effect, which affects pods that are already running on the node as follows

- pods that do not tolerate the taint are evicted immediately

- pods that tolerate the taint without specifying `tolerationSeconds` in their toleration specification remain bound forever
- pods that tolerate the taint with a specified `tolerationSeconds` remain bound for the specified amount of time

In addition, Kubernetes 1.6 introduced alpha support for representing node problems. In other words, the node controller automatically taints a node when certain condition is true. The following taints are built in:

- `node.kubernetes.io/not-ready`: Node is not ready. This corresponds to the NodeCondition `Ready` being "`False`".
- `node.kubernetes.io/unreachable`: Node is unreachable from the node controller. This corresponds to the NodeCondition `Ready` being "`Unknown`".
- `node.kubernetes.io/out-of-disk`: Node becomes out of disk.
- `node.kubernetes.io/memory-pressure`: Node has memory pressure.
- `node.kubernetes.io/disk-pressure`: Node has disk pressure.
- `node.kubernetes.io/network-unavailable`: Node's network is unavailable.
- `node.kubernetes.io/unschedulable`: Node is unschedulable.
- `node.cloudprovider.kubernetes.io/uninitialized`: When the kubelet is started with "external" cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.

In version 1.13, the `TaintBasedEvictions` feature is promoted to beta and enabled by default, hence the taints are automatically added by the NodeController (or kubelet) and the normal logic for evicting pods from nodes based on the Ready NodeCondition is disabled.

> **Note:** To maintain the existing rate limiting behavior of pod evictions due to node problems, the system actually adds the taints in a rate-limited way. This prevents massive pod evictions in scenarios such as the master becoming partitioned from the nodes.

This beta feature, in combination with `tolerationSeconds`, allows a pod to specify how long it should stay bound to a node that has one or both of these problems.

For example, an application with a lot of local state might want to stay bound to node for a long time in the event of network partition, in the hope that the partition will recover and thus the pod eviction can be avoided. The toleration the pod would use in that case would look like

```
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Note that Kubernetes automatically adds a toleration for `node.kubernetes.io/not-ready` with `tolerationSeconds=300` unless the pod configuration provided by the user already has a toleration for `node.kubernetes.io/not-ready`. Likewise it adds a toleration for `node.kubernetes.io/unreachable` with `tolerationSeconds=300` unless the pod configuration provided by the user already has a toleration for `node.kubernetes.io/unreachable`.

These automatically-added tolerations ensure that the default pod behavior of remaining bound for 5 minutes after one of these problems is detected is maintained. The two default tolerations are added by the DefaultTolerationSeconds admission controller.

DaemonSet pods are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

- `node.kubernetes.io/unreachable`
- `node.kubernetes.io/not-ready`

This ensures that DaemonSet pods are never evicted due to these problems, which matches the behavior when this feature is disabled.

Taint Nodes by Condition

The node lifecycle controller automatically creates taints corresponding to Node conditions. Similarly the scheduler does not check Node conditions; instead the scheduler checks taints. This assures that Node conditions don't affect what's scheduled onto the Node. The user can choose to ignore some of the Node's problems (represented as Node conditions) by adding appropriate Pod tolerations. Note that `TaintNodesByCondition` only taints nodes with `NoSchedule` effect. `NoExecute` effect is controlled by `TaintBasedEviction` which is a beta feature and enabled by default since version 1.13.

Starting in Kubernetes 1.8, the DaemonSet controller automatically adds the following `NoSchedule` tolerations to all daemons, to prevent DaemonSets from breaking.

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/out-of-disk` (*only for critical pods*)
- `node.kubernetes.io/unschedulable` (1.10 or later)
- `node.kubernetes.io/network-unavailable` (*host network only*)

Adding these tolerations ensures backward compatibility. You can also add arbitrary tolerations to DaemonSets.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 01, 2020 at 7:55 PM PST by fix: fixed eating initial 2 spaces inside code. (#18914) (Page History)

Edit This Page

Secrets

Kubernetes Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Storing confidential information in a Secret is safer and more flexible than putting it verbatim in a PodThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. definition or in a container imageStored instance

of a container that holds a set of software needed to run an application. .
See Secrets design document for more information.

- Overview of Secrets
- Using Secrets
- Details
- Use cases
- Best practices
- Security properties

Overview of Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image. Users can create secrets and the system also creates some secrets.

To use a secret, a Pod needs to reference the secret. A secret can be used with a Pod in two ways:

- As files in a volumeA directory containing data, accessible to the containers in a pod. mounted on one or more of its containers.
- By the kubelet when pulling images for the Pod.

Built-in Secrets

Service accounts automatically create and attach Secrets with API credentials

Kubernetes automatically creates secrets which contain credentials for accessing the API and automatically modifies your Pods to use this type of secret.

The automatic creation and use of API credentials can be disabled or overridden if desired. However, if all you need to do is securely access the API server, this is the recommended workflow.

See the ServiceAccount documentation for more information on how service accounts work.

Creating your own Secrets

Creating a Secret Using `kubectl`

Secrets can contain user credentials required by Pods to access a database. For example, a database connection string consists of a username and password. You can store the username in a file `./username.txt` and the password in a file `./password.txt` on your local machine.

```
# Create files needed for the rest of the example.
echo -n 'admin' > ./username.txt
echo -n '1f2d1e2e67df' > ./password.txt
```

The `kubectl create secret` command packages these files into a Secret and creates the object on the API server.

```
kubectl create secret generic db-user-pass --from-file=./
username.txt --from-file=./password.txt
```

The output is similar to:

```
secret "db-user-pass" created
```

> **Note:**
>
> Special characters such as `$`, `\`, `*`, and `!` will be interpreted by your shell and require escaping. In most shells, the easiest way to escape the password is to surround it with single quotes (`'`). For example, if your actual password is `S!B\*d$zDsb`, you should execute the command this way:
>
> ```
> kubectl create secret generic dev-db-secret --from-
> literal=username=devuser --from-literal=password='S!
> B\*d$zDsb'
> ```
>
> You do not need to escape special characters in passwords from files (`--from-file`).

You can check that the secret was created:

```
kubectl get secrets
```

The output is similar to:

```
NAME                      TYPE
DATA      AGE
db-user-pass              Opaque
2         51s
```

You can view a description of the secret:

```
kubectl describe secrets/db-user-pass
```

The output is similar to:

```
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
```

```
====
password.txt:    12 bytes
username.txt:    5 bytes
```

> **Note:** The commands `kubectl get` and `kubectl describe` avoid showing the contents of a secret by default. This is to protect the secret from being exposed accidentally to an onlooker, or from being stored in a terminal log.

See decoding a secret to learn how to view the contents of a secret.

Creating a Secret manually

You can also create a Secret in a file first, in JSON or YAML format, and then create that object. The Secret contains two maps: `data` and `stringData`. The `data` field is used to store arbitrary data, encoded using base64. The `stringData` field is provided for convenience, and allows you to provide secret data as unencoded strings.

For example, to store two strings in a Secret using the `data` field, convert the strings to base64 as follows:

```
echo -n 'admin' | base64
```

The output is similar to:

```
YWRtaW4=
```

```
echo -n '1f2d1e2e67df' | base64
```

The output is similar to:

```
MWYyZDFlMmU2N2Rm
```

Write a Secret that looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Now create the Secret using `kubectl apply`:

```
kubectl apply -f ./secret.yaml
```

The output is similar to:

```
secret "mysecret" created
```

For certain scenarios, you may wish to use the `stringData` field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

For example, if your application uses the following configuration file:

```
apiUrl: "https://my.api.com/api/v1"
username: "user"
password: "password"
```

You could store this in a Secret using the following definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  config.yaml: |-
    apiUrl: "https://my.api.com/api/v1"
    username: {{username}}
    password: {{password}}
```

Your deployment tool could then replace the `{{username}}` and `{{password}}` template variables before running `kubectl apply`.

The `stringData` field is a write-only convenience field. It is never output when retrieving Secrets. For example, if you run the following command:

```
kubectl get secret mysecret -o yaml
```

The output is similar to:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:40:59Z
  name: mysecret
  namespace: default
  resourceVersion: "7225"
  uid: c280ad2e-e916-11e8-98f2-025000000001
type: Opaque
data:
  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdX
Nlcm5hbWU6IHt7dXNlcm5hbWV9fQpwYXNzd29yZDoge3twYXNzd29yZH19
```

If a field, such as `username`, is specified in both `data` and `stringData`, the value from `stringData` is used. For example, the following Secret definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
stringData:
  username: administrator
```

Results in the following Secret:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:46:46Z
  name: mysecret
  namespace: default
  resourceVersion: "7579"
  uid: 91460ecb-e917-11e8-98f2-025000000001
type: Opaque
data:
  username: YWRtaW5pc3RyYXRvcg==
```

Where `YWRtaW5pc3RyYXRvcg==` decodes to `administrator`.

The keys of `data` and `stringData` must consist of alphanumeric characters, â€˜-', â€˜_' or â€˜.'.

> **Note:** The serialized JSON and YAML values of secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the `base64` utility on Darwin/macOS, users should avoid using the `-b` option to split long lines. Conversely, Linux users *should* add the option `-w 0` to `base64` commands or the pipeline `base64 | tr -d '\n'` if the `-w` option is not available.

Creating a Secret from a generator

Since Kubernetes v1.14, `kubectl` supports managing objects using Kustomize. Kustomize provides resource Generators to create Secrets and ConfigMaps. The Kustomize generators should be specified in a `kustomization.yaml` file inside a directory. After generating the Secret, you can create the Secret on the API server with `kubectl apply`.

Generating a Secret from files

You can generate a Secret by defining a `secretGenerator` from the files ./username.txt and ./password.txt:

```
cat <<EOF >./kustomization.yaml
secretGenerator:
```

```
- name: db-user-pass
  files:
  - username.txt
  - password.txt
EOF
```

Apply the directory, containing the `kustomization.yaml`, to create the Secret.

```
kubectl apply -k .
```

The output is similar to:

```
secret/db-user-pass-96mffmfh4k created
```

You can check that the secret was created:

```
kubectl get secrets
```

The output is similar to:

```
NAME
TYPE                                  DATA      AGE
db-user-pass-96mffmfh4k
Opaque                                2         51s
```

```
kubectl describe secrets/db-user-pass-96mffmfh4k
```

The output is similar to:

```
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password.txt:    12 bytes
username.txt:    5 bytes
```

Generating a Secret from string literals

You can create a Secret by defining a `secretGenerator` from literals `username=admin` and `password=secret`:

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: db-user-pass
  literals:
  - username=admin
  - password=secret
EOF
```

Apply the directory, containing the `kustomization.yaml`, to create the Secret.

```
kubectl apply -k .
```

The output is similar to:

```
secret/db-user-pass-dddghtt9b5 created
```

> **Note:** When a Secret is generated, the Secret name is created by hashing the Secret data and appending this value to the name. This ensures that a new Secret is generated each time the data is modified.

Decoding a Secret

Secrets can be retrieved by running `kubectl get secret`. For example, you can view the Secret created in the previous section by running the following command:

```
kubectl get secret mysecret -o yaml
```

The output is similar to:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Decode the `password` field:

```
echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

The output is similar to:

```
1f2d1e2e67df
```

Editing a Secret

An existing Secret may be edited with the following command:

```
kubectl edit secrets mysecret
```

This will open the default configured editor and allow for updating the base64 encoded Secret values in the `data` field:

```
# Please edit the object below. Lines beginning with a '#' will
be ignored,
# and an empty file will abort the edit. If an error occurs
while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: { ... }
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
```

Using Secrets

Secrets can be mounted as data volumes or exposed as environment variablesContainer environment variables are name=value pairs that provide useful information into containers running in a Pod. to be used by a container in a Pod. Secrets can also be used by other parts of the system, without being directly exposed to the Pod. For example, Secrets can hold credentials that other parts of the system should use to interact with external systems on your behalf.

Using Secrets as files from a Pod

To consume a Secret in a volume in a Pod:

1. Create a secret or use an existing one. Multiple Pods can reference the same secret.
2. Modify your Pod definition to add a volume under `.spec.volumes[]`. Name the volume anything, and have a `.spec.volumes[].secret.secretName` field equal to the name of the Secret object.
3. Add a `.spec.containers[].volumeMounts[]` to each container that needs the secret. Specify `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the secrets to appear.
4. Modify your image or command line so that the program looks for files in that directory. Each key in the secret `data` map becomes the filename under `mountPath`.

This is an example of a Pod that mounts a Secret in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

Each Secret you want to use needs to be referred to in `.spec.volumes`.

If there are multiple containers in the Pod, then each container needs its own `volumeMounts` block, but only one `.spec.volumes` is needed per Secret.

You can package many files into one secret, or use many secrets, whichever is convenient.

Projection of Secret keys to specific paths

You can also control the paths within the volume where Secret keys are projected. You can use the `.spec.volumes[].secret.items` field to change the target path of each key:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
```

What will happen:

- `username` secret is stored under `/etc/foo/my-group/my-username` file instead of `/etc/foo/username`.
- `password` secret is not projected.

If `.spec.volumes[].secret.items` is used, only keys specified in `items` are projected. To consume all keys from the secret, all of them must be listed in the `items` field. All listed keys must exist in the corresponding secret. Otherwise, the volume is not created.

Secret files permissions

You can set the file access permission bits for a single Secret key. If you don't specify any permissions, `0644` is used by default. You can also set a default mode for the entire Secret volume and override per key if needed.

For example, you can specify a default mode like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      defaultMode: 256
```

Then, the secret will be mounted on `/etc/foo` and all the files created by the secret volume mount will have permission `0400`.

Note that the JSON spec doesn't support octal notation, so use the value 256 for 0400 permissions. If you use YAML instead of JSON for the Pod, you can use octal notation to specify permissions in a more natural way.

You can also use mapping, as in the previous example, and specify different permissions for different files like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
```

```
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
        mode: 511
```

In this case, the file resulting in `/etc/foo/my-group/my-username` will have permission value of `0777`. Owing to JSON limitations, you must specify the mode in decimal notation.

Note that this permission value might be displayed in decimal notation if you read it later.

Consuming Secret values from volumes

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base64 decoded and stored inside these files. This is the result of commands executed inside the container from the example above:

```
ls /etc/foo/
```

The output is similar to:

```
username
password
```

```
cat /etc/foo/username
```

The output is similar to:

```
admin
```

```
cat /etc/foo/password
```

The output is similar to:

```
1f2d1e2e67df
```

The program in a container is responsible for reading the secrets from the files.

Mounted Secrets are updated automatically

When a secret currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted secret is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the Secret. The type of the cache is configurable using the `ConfigMapAndSecretChangeDetectionStrategy` field in the KubeletConfiguration struct. A Secret can be either propagated by watch (default), ttl-based, or simply redirecting all requests directly to the API server. As a result, the total delay from the moment when the Secret is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (it equals to watch propagation delay, ttl of cache, or zero correspondingly).

> **Note:** A container using a Secret as a subPath volume mount will not receive Secret updates.

Using Secrets as environment variables

To use a secret in an environment variableContainer environment variables are name=value pairs that provide useful information into containers running in a Pod. in a Pod:

1. Create a secret or use an existing one. Multiple Pods can reference the same secret.
2. Modify your Pod definition in each container that you wish to consume the value of a secret key to add an environment variable for each secret key you wish to consume. The environment variable that consumes the secret key should populate the secret's name and key in `env[].valueFrom.secretKeyRef`.
3. Modify your image and/or command line so that the program looks for values in the specified environment variables.

This is an example of a Pod that uses secrets from environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

Consuming Secret Values from environment variables

Inside a container that consumes a secret in an environment variables, the secret keys appear as normal environment variables containing the base64 decoded values of the secret data. This is the result of commands executed inside the container from the example above:

```
echo $SECRET_USERNAME
```

The output is similar to:

```
admin
```

```
echo $SECRET_PASSWORD
```

The output is similar to:

```
1f2d1e2e67df
```

Using imagePullSecrets

The `imagePullSecrets` field is a list of references to secrets in the same namespace. You can use an `imagePullSecrets` to pass a secret that contains a Docker (or other) image registry password to the kubelet. The kubelet uses this information to pull a private image on behalf of your Pod. See the PodSpec API for more information about the `imagePullSecrets` field.

Manually specifying an imagePullSecret

You can learn how to specify `ImagePullSecrets` from the container images documentation.

Arranging for imagePullSecrets to be automatically attached

You can manually create `imagePullSecrets`, and reference it from a ServiceAccount. Any Pods created with that ServiceAccount or created with that ServiceAccount by default, will get their `imagePullSecrets` field set to that of the service account. See Add ImagePullSecrets to a service account for a detailed explanation of that process.

Automatic mounting of manually created Secrets

Manually created secrets (for example, one containing a token for accessing a GitHub account) can be automatically attached to pods based on their service account. See Injecting Information into Pods Using a PodPreset for a detailed explanation of that process.

Details

Restrictions

Secret volume sources are validated to ensure that the specified object reference actually points to an object of type Secret. Therefore, a secret needs to be created before any Pods that depend on it.

Secret resources reside in a namespaceAn abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster. . Secrets can only be referenced by Pods in that same namespace.

Individual secrets are limited to 1MiB in size. This is to discourage creation of very large secrets which would exhaust the API server and kubelet memory. However, creation of many smaller secrets could also exhaust memory. More comprehensive limits on memory usage due to secrets is a planned feature.

The kubelet only supports the use of secrets for Pods where the secrets are obtained from the API server. This includes any Pods created using `kubectl`, or indirectly via a replication controller. It does not include Pods created as a result of the kubelet `--manifest-url` flag, its `--config` flag, or its REST API (these are not common ways to create Pods.)

Secrets must be created before they are consumed in Pods as environment variables unless they are marked as optional. References to secrets that do not exist will prevent the Pod from starting.

References (`secretKeyRef` field) to keys that do not exist in a named Secret will prevent the Pod from starting.

Secrets used to populate environment variables by the `envFrom` field that have keys that are considered invalid environment variable names will have those keys skipped. The Pod will be allowed to start. There will be an event whose reason is `InvalidVariableNames` and the message will contain the list of invalid keys that were skipped. The example shows a pod which refers to the default/mysecret that contains 2 invalid keys: `1badkey` and `2alsobad`.

```
kubectl get events
```

The output is similar to:

```
LASTSEEN   FIRSTSEEN   COUNT     NAME            KIND
SUBOBJECT                         TYPE      REASON
0s         0s          1         dapi-test-pod
Pod                                         Warning
InvalidEnvironmentVariableNames   kubelet, 127.0.0.1      Keys
[1badkey, 2alsobad] from the EnvFrom secret default/mysecret
were skipped since they are considered invalid environment
variable names.
```

Secret and Pod lifetime interaction

When a Pod is created by calling the Kubernetes API, there is no check if a referenced secret exists. Once a Pod is scheduled, the kubelet will try to fetch the secret value. If the secret cannot be fetched because it does not exist or because of a temporary lack of connection to the API server, the kubelet will periodically retry. It will report an event about the Pod explaining the reason it is not started yet. Once the secret is fetched, the kubelet will create and mount a volume containing it. None of the Pod's containers will start until all the Pod's volumes are mounted.

Use cases

Use-Case: Pod with ssh keys

Create a secret containing some ssh keys:

```
kubectl create secret generic ssh-key-secret --from-file=ssh-privatekey=/path/to/.ssh/id_rsa --from-file=ssh-publickey=/path/to/.ssh/id_rsa.pub
```

The output is similar to:

```
secret "ssh-key-secret" created
```

You can also create a `kustomization.yaml` with a `secretGenerator` field containing ssh keys.

> **Caution:** Think carefully before sending your own ssh keys: other users of the cluster may have access to the secret. Use a service account which you want to be accessible to all the users with whom you share the Kubernetes cluster, and can revoke this account if the users are compromised.

Now you can create a Pod which references the secret with the ssh key and consumes it in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
  labels:
    name: secret-test
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: ssh-key-secret
  containers:
  - name: ssh-test-container
    image: mySshImage
    volumeMounts:
```

```
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

When the container's command runs, the pieces of the key will be available in:

```
/etc/secret-volume/ssh-publickey
/etc/secret-volume/ssh-privatekey
```

The container is then free to use the secret data to establish an ssh connection.

Use-Case: Pods with prod / test credentials

This example illustrates a Pod which consumes a secret containing production credentials and another Pod which consumes a secret with test environment credentials.

You can create a `kustomization.yaml` with a `secretGenerator` field or run `kubectl create secret`.

```
kubectl create secret generic prod-db-secret --from-literal=username=produser --from-literal=password=Y4nys7f11
```

The output is similar to:

```
secret "prod-db-secret" created
```

```
kubectl create secret generic test-db-secret --from-literal=username=testuser --from-literal=password=iluvtests
```

The output is similar to:

```
secret "test-db-secret" created
```

> **Note:**
>
> Special characters such as `$`, `\`, `*`, and `!` will be interpreted by your shell and require escaping. In most shells, the easiest way to escape the password is to surround it with single quotes (`'`). For example, if your actual password is `S!B\*d$zDsb`, you should execute the command this way:
>
> ```
> kubectl create secret generic dev-db-secret --from-literal=username=devuser --from-literal=password='S!B\*d$zDsb'
> ```
>
> You do not need to escape special characters in passwords from files (`--from-file`).

Now make the Pods:

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: List
items:
- kind: Pod
  apiVersion: v1
  metadata:
    name: prod-db-client-pod
    labels:
      name: prod-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: prod-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
- kind: Pod
  apiVersion: v1
  metadata:
    name: test-db-client-pod
    labels:
      name: test-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: test-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
EOF
```

Add the pods to the same kustomization.yaml:

```
cat <<EOF >> kustomization.yaml
resources:
- pod.yaml
EOF
```

Apply all those objects on the API server by running:

```
kubectl apply -k .
```

Both containers will have the following files present on their filesystems with the values for each container's environment:

```
/etc/secret-volume/username
/etc/secret-volume/password
```

Note how the specs for the two Pods differ only in one field; this facilitates creating Pods with different capabilities from a common Pod template.

You could further simplify the base Pod specification by using two service accounts:

1. `prod-user` with the `prod-db-secret`
2. `test-user` with the `test-db-secret`

The Pod specification is shortened to:

```
apiVersion: v1
kind: Pod
metadata:
  name: prod-db-client-pod
  labels:
    name: prod-db-client
spec:
  serviceAccount: prod-db-client
  containers:
  - name: db-client-container
    image: myClientImage
```

Use-case: dotfiles in a secret volume

You can make your data "hidden" by defining a key that begins with a dot. This key represents a dotfile or "hidden" file. For example, when the following secret is mounted into a volume, `secret-volume`:

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: dotfile-secret
  containers:
```

```
  - name: dotfile-test-container
    image: k8s.gcr.io/busybox
    command:
    - ls
    - "-l"
    - "/etc/secret-volume"
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

The volume will contain a single file, called `.secret-file`, and the `dotfile-test-container` will have this file present at the path `/etc/secret-volume/.secret-file`.

> **Note:** Files beginning with dot characters are hidden from the output of `ls -l`; you must use `ls -la` to see them when listing directory contents.

Use-case: Secret visible to one container in a Pod

Consider a program that needs to handle HTTP requests, do some complex business logic, and then sign some messages with an HMAC. Because it has complex application logic, there might be an unnoticed remote file reading exploit in the server, which could expose the private key to an attacker.

This could be divided into two processes in two containers: a frontend container which handles user interaction and business logic, but which cannot see the private key; and a signer container that can see the private key, and responds to simple signing requests from the frontend (for example, over localhost networking).

With this partitioned approach, an attacker now has to trick the application server into doing something rather arbitrary, which may be harder than getting it to read a file.

Best practices

Clients that use the Secret API

When deploying applications that interact with the Secret API, you should limit access using authorization policies such as RBAC.

Secrets often hold values that span a spectrum of importance, many of which can cause escalations within Kubernetes (e.g. service account tokens) and to external systems. Even if an individual app can reason about the power of the secrets it expects to interact with, other apps within the same namespace can render those assumptions invalid.

For these reasons `watch` and `list` requests for secrets within a namespace are extremely powerful capabilities and should be avoided, since listing secrets allows the clients to inspect the values of all secrets that are in that namespace. The ability to `watch` and `list` all secrets in a cluster should be reserved for only the most privileged, system-level components.

Applications that need to access the Secret API should perform `get` requests on the secrets they need. This lets administrators restrict access to all secrets while white-listing access to individual instances that the app needs.

For improved performance over a looping `get`, clients can design resources that reference a secret then `watch` the resource, re-requesting the secret when the reference changes. Additionally, a "bulk watch" API to let clients `watch` individual resources has also been proposed, and will likely be available in future releases of Kubernetes.

Security properties

Protections

Because secrets can be created independently of the Pods that use them, there is less risk of the secret being exposed during the workflow of creating, viewing, and editing Pods. The system can also take additional precautions with Secrets, such as avoiding writing them to disk where possible.

A secret is only sent to a node if a Pod on that node requires it. The kubelet stores the secret into a `tmpfs` so that the secret is not written to disk storage. Once the Pod that depends on the secret is deleted, the kubelet will delete its local copy of the secret data as well.

There may be secrets for several Pods on the same node. However, only the secrets that a Pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the secrets of another Pod.

There may be several containers in a Pod. However, each container in a Pod has to request the secret volume in its `volumeMounts` for it to be visible within the container. This can be used to construct useful security partitions at the Pod level.

On most Kubernetes distributions, communication between users and the API server, and from the API server to the kubelets, is protected by SSL/TLS. Secrets are protected when transmitted over these channels.

FEATURE STATE: `Kubernetes v1.13` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Organizing Cluster Access Using kubeconfig Files

Use kubeconfig files to organize information about clusters, users, namespaces, and authentication mechanisms. The `kubectl` command-line tool uses kubeconfig files to find the information it needs to choose a cluster and communicate with the API server of a cluster.

> **Note:** A file that is used to configure access to clusters is called a *kubeconfig file*. This is a generic way of referring to configuration files. It does not mean that there is a file named `kubeconfig`.

By default, `kubectl` looks for a file named `config` in the `$HOME/.kube` directory. You can specify other kubeconfig files by setting the `KUBECONFIG` environment variable or by setting the `--kubeconfig` flag.

For step-by-step instructions on creating and specifying kubeconfig files, see Configure Access to Multiple Clusters.

Supporting multiple clusters, users, and authentication mechanisms

Suppose you have several clusters, and your users and components authenticate in a variety of ways. For example:

- A running kubelet might authenticate using certificates.
- A user might authenticate using tokens.
- Administrators might have sets of certificates that they provide to individual users.

With kubeconfig files, you can organize your clusters, users, and namespaces. You can also define contexts to quickly and easily switch between clusters and namespaces.

Context

A *context* element in a kubeconfig file is used to group access parameters under a convenient name. Each context has three parameters: cluster, namespace, and user. By default, the `kubectl` command-line tool uses parameters from the *current context* to communicate with the cluster.

To choose the current context:

```
kubectl config use-context
```

The KUBECONFIG environment variable

The `KUBECONFIG` environment variable holds a list of kubeconfig files. For Linux and Mac, the list is colon-delimited. For Windows, the list is semicolon-delimited. The `KUBECONFIG` environment variable is not required. If the `KUBECONFIG` environment variable doesn't exist, `kubectl` uses the default kubeconfig file, `$HOME/.kube/config`.

If the `KUBECONFIG` environment variable does exist, `kubectl` uses an effective configuration that is the result of merging the files listed in the `KUBECONFIG` environment variable.

Merging kubeconfig files

To see your configuration, enter this command:

```
kubectl config view
```

As described previously, the output might be from a single kubeconfig file, or it might be the result of merging several kubeconfig files.

Here are the rules that `kubectl` uses when it merges kubeconfig files:

1. If the `--kubeconfig` flag is set, use only the specified file. Do not merge. Only one instance of this flag is allowed.

Otherwise, if the `KUBECONFIG` environment variable is set, use it as a list of files that should be merged. Merge the files listed in the `KUBECONFIG` environment variable according to these rules:

- Ignore empty filenames.
- Produce errors for files with content that cannot be deserialized.
- The first file to set a particular value or map key wins.
- Never change the value or map key. Example: Preserve the context of the first file to set `current-context`. Example: If two files specify a `red-user`, use only values from the first file's `red-user`. Even if the second file has non-conflicting entries under `red-user`, discard them.

For an example of setting the `KUBECONFIG` environment variable, see Setting the KUBECONFIG environment variable.

Otherwise, use the default kubeconfig file, `$HOME/.kube/config`, with no merging.

1. Determine the context to use based on the first hit in this chain:

    1. Use the `--context` command-line flag if it exists.
    2. Use the `current-context` from the merged kubeconfig files.

An empty context is allowed at this point.

1. Determine the cluster and user. At this point, there might or might not be a context. Determine the cluster and user based on the first hit in this chain, which is run twice: once for user and once for cluster:

    1. Use a command-line flag if it exists: `--user` or `--cluster`.
    2. If the context is non-empty, take the user or cluster from the context.

The user and cluster can be empty at this point.

1. Determine the actual cluster information to use. At this point, there might or might not be cluster information. Build each piece of the cluster information based on this chain; the first hit wins:

    1. Use command line flags if they exist: `--server`, `--certificate-authority`, `--insecure-skip-tls-verify`.
    2. If any cluster information attributes exist from the merged kubeconfig files, use them.
    3. If there is no server location, fail.

2. Determine the actual user information to use. Build user information using the same rules as cluster information, except allow only one authentication technique per user:

    1. Use command line flags if they exist: `--client-certificate`, `--client-key`, `--username`, `--password`, `--token`.
    2. Use the `user` fields from the merged kubeconfig files.
    3. If there are two conflicting techniques, fail.

3. For any information still missing, use default values and potentially prompt for authentication information.

File references

File and path references in a kubeconfig file are relative to the location of the kubeconfig file. File references on the command line are relative to the current working directory. In `$HOME/.kube/config`, relative paths are stored relatively, and absolute paths are stored absolutely.

What's next

- Configure Access to Multiple Clusters
- `kubectl config`

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 07, 2019 at 11:31 PM PST by fix spelling mistake (#12113) (Page History)

Edit This Page

Pod Priority and Preemption

FEATURE STATE: `Kubernetes 1.14` stable
This feature is *stable*, meaning:

Edit This Page

Scheduling Framework

FEATURE STATE: `Kubernetes 1.15` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

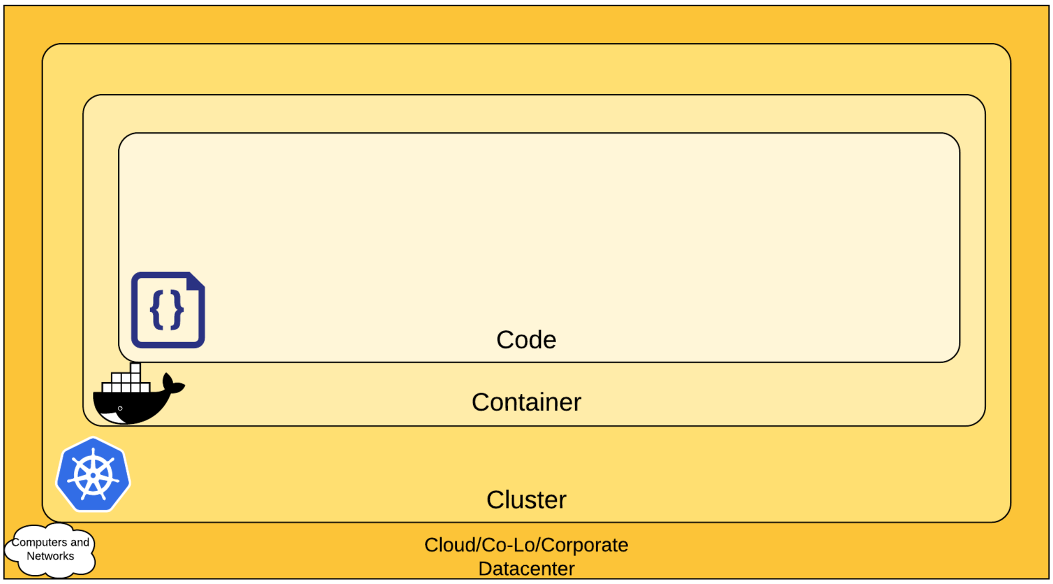
Overview of Cloud Native Security

Kubernetes Security (and security in general) is an immense topic that has many highly interrelated parts. In today's era where open source software is integrated into many of the systems that help web applications run, there are some overarching concepts that can help guide your intuition about how you can think about security holistically. This guide will define a mental model for some general concepts surrounding Cloud Native Security. The mental model is completely arbitrary and you should only use it if it helps you think about where to secure your software stack.

The 4C's of Cloud Native Security

Let's start with a diagram that may help you understand how you can think about security in layers.

> **Note:** This layered approach augments the defense in depth approach to security, which is widely regarded as a best practice for securing software systems. The 4C's are Cloud, Clusters, Containers, and Code.



The 4C's of Cloud Native Security

As you can see from the above figure, each one of the 4C's depend on the security of the squares in which they fit. It is nearly impossibly to safeguard against poor security standards in Cloud, Containers, and Code by only addressing security at the code level. However, when these areas are dealt with appropriately, then adding security to your code augments an already strong base. These areas of concern will now be described in more detail below.

Cloud

In many ways, the Cloud (or co-located servers, or the corporate datacenter) is the trusted computing base of a Kubernetes cluster. If these components themselves are vulnerable (or configured in a vulnerable way) then there's no real way to guarantee the security of any components built on top of this base. Each cloud provider has extensive security recommendations they make to their customers on how to run workloads securely in their environment. It is out of the scope of this guide to give recommendations on cloud security since every cloud provider and workload is different. Here are some links to some of the popular cloud providers' documentation for security as well as give general guidance for securing the infrastructure that makes up a Kubernetes cluster.

Cloud Provider Security Table

IaaS Provider	Link
Alibaba Cloud	https://www.alibabacloud.com/trust-center
Amazon Web Services	https://aws.amazon.com/security/
Google Cloud Platform	https://cloud.google.com/security/

IaaS Provider	Link
IBM Cloud	https://www.ibm.com/cloud/security
Microsoft Azure	https://docs.microsoft.com/en-us/azure/security/azure-security
VMWare VSphere	https://www.vmware.com/security/hardening-guides.html

If you are running on your own hardware or a different cloud provider you will need to consult your documentation for security best practices.

General Infrastructure Guidance Table

Area of Concern for Kubernetes Infrastructure	Recommendation
Network access to API Server (Masters)	Ideally all access to the Kubernetes Masters is not allowed publicly on the internet and is controlled by network access control lists restricted to the set of IP addresses needed to administer the cluster.
Network access to Nodes (Worker Servers)	Nodes should be configured to *only* accept connections (via network access control lists) from the masters on the specified ports, and accept connections for services in Kubernetes of type NodePort and LoadBalancer. If possible, these nodes should not be exposed on the public internet entirely.
Kubernetes access to Cloud Provider API	Each cloud provider will need to grant a different set of permissions to the Kubernetes Masters and Nodes, so this recommendation will be more generic. It is best to provide the cluster with cloud provider access that follows the principle of least privilege for the resources it needs to administer. An example for Kops in AWS can be found here: https://github.com/kubernetes/kops/blob/master/docs/iam_roles.md#iam-roles
Access to etcd	Access to etcd (the datastore of Kubernetes) should be limited to the masters only. Depending on your configuration, you should also attempt to use etcd over TLS. More info can be found here: https://github.com/etcd-io/etcd/tree/master/Documentation#security
etcd Encryption	Wherever possible it's a good practice to encrypt all drives at rest, but since etcd holds the state of the entire cluster (including Secrets) its disk should especially be encrypted at rest.

Cluster

This section will provide links for securing workloads in Kubernetes. There are two areas of concern for securing Kubernetes:

- Securing the components that are configurable which make up the cluster
- Securing the components which run in the cluster

Components *of* the Cluster

If you want to protect your cluster from accidental or malicious access, and adopt good information practices, read and follow the advice about securing your cluster.

Components *in* the Cluster (your application)

Depending on the attack surface of your application, you may want to focus on specific aspects of security. For example, if you are running a service (Service A) that is critical in a chain of other resources and a separate workload (Service B) which is vulnerable to a resource exhaustion attack, by not putting resource limits on Service B you run the risk of also compromising Service A. Below is a table of links of things to consider when securing workloads running in Kubernetes.

Area of Concern for Workload Security	Recommendation
RBAC Authorization (Access to the Kubernetes API)	https://kubernetes.io/docs/reference/access-authn-authz/rbac/
Authentication	https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/
Application secrets management (and encrypting them in etcd at rest)	https://kubernetes.io/docs/concepts/configuration/secret/ https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/
Pod Security Policies	https://kubernetes.io/docs/concepts/policy/pod-security-policy/
Quality of Service (and Cluster resource management)	https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/
Network Policies	https://kubernetes.io/docs/concepts/services-networking/network-policies/
TLS For Kubernetes Ingress	https://kubernetes.io/docs/concepts/services-networking/ingress/#tls

Container

In order to run software in Kubernetes, it must be in a container. Because of this, there are certain security considerations that must be taken into account in order to benefit from the workload security primitives of Kubernetes. Container security is also outside the scope of this guide, but here is a table of general recommendations and links for further exploration of this topic.

Area of Concern for Containers	**Recommendation**
Container Vulnerability Scanning and OS Dependency Security	As part of an image build step or on a regular basis you should scan your containers for known vulnerabilities with a tool such as CoreOS's Clair
Image Signing and Enforcement	Two other CNCF Projects (TUF and Notary) are useful tools for signing container images and maintaining a system of trust for the content of your containers. If you use Docker, it is built in to the Docker Engine as Docker Content Trust. On the enforcement piece, IBM's Portieris project is a tool that runs as a Kubernetes Dynamic Admission Controller to ensure that images are properly signed via Notary before being admitted to the Cluster.
Disallow privileged users	When constructing containers, consult your documentation for how to create users inside of the containers that have the least level of operating system privilege necessary in order to carry out the goal of the container.

Code

Finally moving down into the application code level, this is one of the primary attack surfaces over which you have the most control. This is also outside of the scope of Kubernetes but here are a few recommendations:

General Code Security Guidance Table

Area of Concern for Code	**Recommendation**
Access over TLS only	If your code needs to communicate via TCP, ideally it would be performing a TLS handshake with the client ahead of time. With the exception of a few cases, the default behavior should be to encrypt everything in transit. Going one step further, even "behind the firewall" in our VPC's it's still a good idea to encrypt network traffic between services. This can be done through a process known as mutual or mTLS which performs a two sided verification of communication between two certificate holding services. There are numerous tools that can be used to accomplish this in Kubernetes such as Linkerd and Istio.

Area of Concern for Code	**Recommendation**
Limiting port ranges of communication	This recommendation may be a bit self-explanatory, but wherever possible you should only expose the ports on your service that are absolutely essential for communication or metric gathering.
3rd Party Dependency Security	Since our applications tend to have dependencies outside of our own codebases, it is a good practice to regularly scan the code's dependencies to ensure that they are still secure with no vulnerabilities currently filed against them. Each language has a tool for performing this check automatically.
Static Code Analysis	Most languages provide a way for a snippet of code to be analyzed for any potentially unsafe coding practices. Whenever possible you should perform checks using automated tooling that can scan codebases for common security errors. Some of the tools can be found here: https://www.owasp.org/index.php/Source_Code_Analysis_Tools
Dynamic probing attacks	There are a few automated tools that are able to be run against your service to try some of the well known attacks that commonly befall services. These include SQL injection, CSRF, and XSS. One of the most popular dynamic analysis tools is the OWASP Zed Attack proxy https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

Robust automation

Most of the above mentioned suggestions can actually be automated in your code delivery pipeline as part of a series of checks in security. To learn about a more "Continuous Hacking" approach to software delivery, this article provides more detail.

What's next

- Read about network policies for Pods
- Read about securing your cluster
- Read about API access control
- Read about data encryption in transit for the control plane
- Read about data encryption at rest
- Read about Secrets in Kubernetes

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 07, 2020 at 7:09 PM PST by Refined unclear sentence on 3rd party dependencies (#18015) (Page History)

Edit This Page

Limit Ranges

By default, containers run with unbounded compute resources on a Kubernetes cluster. With Resource quotas, cluster administrators can restrict the resource consumption and creation on a namespace basis.

Within a namespace, a Pod or Container can consume as much CPU and memory as defined by the namespace's resource quota. There is a concern that one Pod or Container could monopolize all of the resources. Limit Range is a policy to constrain resource by Pod or Container in a namespace.

A limit range, defined by a `LimitRange` object, provides constraints that can:

- Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
- Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace.
- Enforce a ratio between request and limit for a resource in a namespace.
- Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

Enabling Limit Range

Limit Range support is enabled by default for many Kubernetes distributions. It is enabled when the apiserver `--enable-admission-plugins=` flag has `LimitRanger` admission controller as one of its arguments.

A limit range is enforced in a particular namespace when there is a `LimitRange` object in that namespace.

Overview of Limit Range:

- The administrator creates one `LimitRange` in one namespace.
- Users create resources like Pods, Containers, and PersistentVolumeClaims in the namespace.
- The `LimitRanger` admission controller enforces defaults limits for all Pods and Container that do not set compute resource requirements and tracks usage to ensure it does not exceed resource minimum , maximum and ratio defined in any `LimitRange` present in the namespace.
- If creating or updating a resource (Pod, Container, PersistentVolumeClaim) violates a limit range constraint, the request to the API server will fail with HTTP status code `403 FORBIDDEN` and a message explaining the constraint that would have been violated.
- If limit range is activated in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values; otherwise, the system may reject pod creation.

- LimitRange validations occurs only at Pod Admission stage, not on Running pods.

Examples of policies that could be created using limit range are:

- In a 2 node cluster with a capacity of 8 GiB RAM, and 16 cores, constrain Pods in a namespace to request 100m and not exceeds 500m for CPU , request 200Mi and not exceed 600Mi
- Define default CPU limits and request to 150m and Memory default request to 300Mi for containers started with no cpu and memory requests in their spec.

In the case where the total limits of the namespace is less than the sum of the limits of the Pods/Containers, there may be contention for resources; The Containers or Pods will not be created.

Neither contention nor changes to limitrange will affect already created resources.

Limiting Container compute resources

The following section discusses the creation of a LimitRange acting at Container Level. A Pod with 04 containers is first created; each container within the Pod has a specific `spec.resource` configuration each container within the pod is handled differently by the LimitRanger admission controller.

Create a namespace `limitrange-demo` using the following kubectl command:

```
kubectl create namespace limitrange-demo
```

To avoid passing the target limitrange-demo in your kubectl commands, change your context with the following command:

```
kubectl config set-context --current --namespace=limitrange-demo
```

Here is the configuration file for a LimitRange object:

admin/resource/limit-mem-cpu-container.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-mem-cpu-per-container
spec:
  limits:
  - max:
      cpu: "800m"
      memory: "1Gi"
    min:
      cpu: "100m"
      memory: "99Mi"
    default:
      cpu: "700m"
      memory: "900Mi"
    defaultRequest:
      cpu: "110m"
      memory: "111Mi"
    type: Container
```

This object defines minimum and maximum Memory/CPU limits, default cpu/Memory requests and default limits for CPU/Memory resources to be apply to containers.

Create the `limit-mem-cpu-per-container` LimitRange in the `limitrange-demo` namespace with the following kubectl command:

```
kubectl create -f https://k8s.io/examples/admin/resource/limit-
mem-cpu-container.yaml -n limitrange-demo
```

```
kubectl describe limitrange/limit-mem-cpu-per-container -n
limitrange-demo
```

```
Type        Resource  Min   Max   Default Request  Default
Limit  Max Limit/Request Ratio
----        --------  ---   ---   ---------------
-------------  -----------------------
Container   cpu       100m  800m  110m
700m           -
Container   memory    99Mi  1Gi   111Mi
900Mi          -
```

Here is the configuration file for a Pod with 04 containers to demonstrate LimitRange features :

admin/resource/limit-range-pod-1.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
spec:
  containers:
  - name: busybox-cnt01
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt01; sleep 10;done"]
    resources:
      requests:
        memory: "100Mi"
        cpu: "100m"
      limits:
        memory: "200Mi"
        cpu: "500m"
  - name: busybox-cnt02
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt02; sleep 10;done"]
    resources:
      requests:
        memory: "100Mi"
        cpu: "100m"
  - name: busybox-cnt03
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt03; sleep 10;done"]
    resources:
      limits:
        memory: "200Mi"
        cpu: "500m"
  - name: busybox-cnt04
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt04; sleep 10;done"]
```

Create the `busybox1` Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/limit-range-pod-1.yaml -n limitrange-demo
```

Container spec with valid CPU/Memory requests and limits

View the `busybox-cnt01` resource configuration:

```
kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[0].resources"
```

```
{
  "limits": {
    "cpu": "500m",
    "memory": "200Mi"
  },
  "requests": {
    "cpu": "100m",
    "memory": "100Mi"
  }
}
```

- The `busybox-cnt01` Container inside `busybox` Pod defined `requests.cpu=100m` and `requests.memory=100Mi`.
- `100m <= 500m <= 800m` , The container cpu limit (500m) falls inside the authorized CPU limit range.
- `99Mi <= 200Mi <= 1Gi` , The container memory limit (200Mi) falls inside the authorized Memory limit range.
- No request/limits ratio validation for CPU/Memory , thus the container is valid and created.

Container spec with a valid CPU/Memory requests but no limits

View the `busybox-cnt02` resource configuration

```
kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[1].resources"
```

```
{
  "limits": {
    "cpu": "700m",
    "memory": "900Mi"
  },
  "requests": {
    "cpu": "100m",
    "memory": "100Mi"
  }
}
```

- The `busybox-cnt02` Container inside `busybox1` Pod defined `requests.cpu=100m` and `requests.memory=100Mi` but not limits for cpu and memory.
- The container do not have a limits section, the default limits defined in the limit-mem-cpu-per-container LimitRange object are injected to this container `limits.cpu=700mi` and `limits.memory=900Mi`.

- `100m <= 700m <= 800m` , The container cpu limit (700m) falls inside the authorized CPU limit range.
- `99Mi <= 900Mi <= 1Gi` , The container memory limit (900Mi) falls inside the authorized Memory limit range.
- No request/limits ratio set , thus the container is valid and created.

Container spec with a valid CPU/Memory limits but no requests

View the `busybox-cnt03` resource configuration

```
kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[2].resources"
```

```
{
  "limits": {
    "cpu": "500m",
    "memory": "200Mi"
  },
  "requests": {
    "cpu": "500m",
    "memory": "200Mi"
  }
}
```

- The `busybox-cnt03` Container inside `busybox1` Pod defined `limits.cpu=500m` and `limits.memory=200Mi` but no `requests` for cpu and memory.
- The container do not define a request section, the defaultRequest defined in the limit-mem-cpu-per-container LimitRange is not used to fill its limits section but the limits defined by the container are set as requests `limits.cpu=500m` and `limits.memory=200Mi`.
- `100m <= 500m <= 800m` , The container cpu limit (500m) falls inside the authorized CPU limit range.
- `99Mi <= 200Mi <= 1Gi` , The container memory limit (200Mi) falls inside the authorized Memory limit range.
- No request/limits ratio set , thus the container is valid and created.

Container spec with no CPU/Memory requests/limits

View the `busybox-cnt04` resource configuration:

```
kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[3].resources"
```

```
{
  "limits": {
    "cpu": "700m",
    "memory": "900Mi"
  },
  "requests": {
    "cpu": "110m",
    "memory": "111Mi"
  }
}
```

- The busybox-cnt04 Container inside busybox1 define neither limits nor requests.
- The container do not define a limit section, the default limit defined in the limit-mem-cpu-per-container LimitRange is used to fill its request limits.cpu=700m and limits.memory=900Mi .
- The container do not define a request section, the defaultRequest defined in the limit-mem-cpu-per-container LimitRange is used to fill its request section requests.cpu=110m and requests.memory=111Mi
- 100m <= 700m <= 800m , The container cpu limit (700m) falls inside the authorized CPU limit range.
- 99Mi <= 900Mi <= 1Gi , The container memory limit (900Mi) falls inside the authorized Memory limitrange .
- No request/limits ratio set , thus the container is valid and created.

All containers defined in the busybox Pod passed LimitRange validations, this the Pod is valid and create in the namespace.

Limiting Pod compute resources

The following section discusses how to constrain resources at Pod level.

admin/resource/limit-mem-cpu-pod.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-mem-cpu-per-pod
spec:
  limits:
  - max:
      cpu: "2"
      memory: "2Gi"
    type: Pod
```

Without having to delete busybox1 Pod, create the limit-mem-cpu-pod LimitRange in the limitrange-demo namespace:

```
kubectl apply -f https://k8s.io/examples/admin/resource/limit-mem-cpu-pod.yaml -n limitrange-demo
```

The limitrange is created and limits CPU to 2 Core and Memory to 2Gi per Pod:

```
limitrange/limit-mem-cpu-per-pod created
```

Describe the limit-mem-cpu-per-pod limit object using the following kubectl command:

```
kubectl describe limitrange/limit-mem-cpu-per-pod
```

```
Name:       limit-mem-cpu-per-pod
Namespace:  limitrange-demo
Type        Resource  Min  Max  Default Request  Default Limit
Max Limit/Request Ratio
----        --------  ---  ---  ---------------  -------------
-----------------------
Pod         cpu       -    2    -                -              -
Pod         memory    -    2Gi  -                -              -
```

Now create the `busybox2` Pod:

admin/resource/limit-range-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
spec:
  containers:
  - name: busybox-cnt01
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt01; sleep
10;done"]
    resources:
      requests:
        memory: "100Mi"
        cpu: "100m"
      limits:
        memory: "200Mi"
        cpu: "500m"
  - name: busybox-cnt02
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt02; sleep
10;done"]
    resources:
      requests:
        memory: "100Mi"
        cpu: "100m"
  - name: busybox-cnt03
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt03; sleep
10;done"]
    resources:
      limits:
        memory: "200Mi"
        cpu: "500m"
  - name: busybox-cnt04
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello from cnt04; sleep
10;done"]
```

```
kubectl apply -f https://k8s.io/examples/admin/resource/limit-
range-pod-2.yaml -n limitrange-demo
```

The `busybox2` Pod definition is identical to `busybox1` but an error is reported since Pod's resources are now limited:

```
Error from server (Forbidden): error when creating "limit-range-pod-2.yaml": pods "busybox2" is forbidden: [maximum cpu usage per Pod is 2, but limit is 2400m., maximum memory usage per Pod is 2Gi, but limit is 2306867200.]
```

```
kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[].resources.limits.memory"
"200Mi"
"900Mi"
"200Mi"
"900Mi"
```

`busybox2` Pod will not be admitted on the cluster since the total memory limit of its container is greater than the limit defined in the LimitRange. `busybox1` will not be evicted since it was created and admitted on the cluster before the LimitRange creation.

Limiting Storage resources

You can enforce minimum and maximum size of storage resources that can be requested by each PersistentVolumeClaim in a namespace using a LimitRange:

admin/resource/storagelimits.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: storagelimits
spec:
  limits:
  - type: PersistentVolumeClaim
    max:
      storage: 2Gi
    min:
      storage: 1Gi
```

Apply the YAML using `kubectl create`:

```
kubectl create -f https://k8s.io/examples/admin/resource/storagelimits.yaml -n limitrange-demo
```

```
limitrange/storagelimits created
```

Describe the created object:

```
kubectl describe limits/storagelimits
```

The output should look like:

```
Name:                    storagelimits
Namespace:               limitrange-demo
Type                     Resource  Min   Max   Default Request
Default Limit  Max Limit/Request Ratio
----                     --------  ---   ---   ---------------
-------------  -----------------------
PersistentVolumeClaim  storage   1Gi   2Gi   -
-              -
```

admin/resource/pvc-limit-lower.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-limit-lower
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

```
kubectl create -f https://k8s.io/examples/admin/resource/pvc-
limit-lower.yaml -n limitrange-demo
```

While creating a PVC with `requests.storage` lower than the Min value in the LimitRange, an Error thrown by the server:

```
Error from server (Forbidden): error when creating "pvc-limit-
lower.yaml": persistentvolumeclaims "pvc-limit-lower" is
forbidden: minimum storage usage per PersistentVolumeClaim is
1Gi, but request is 500Mi.
```

Same behaviour is noted if the `requests.storage` is greater than the Max value in the LimitRange:

admin/resource/pvc-limit-greater.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-limit-greater
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

```
kubectl create -f https://k8s.io/examples/admin/resource/pvc-
limit-greater.yaml -n limitrange-demo
```

```
Error from server (Forbidden): error when creating "pvc-limit-
greater.yaml": persistentvolumeclaims "pvc-limit-greater" is
forbidden: maximum storage usage per PersistentVolumeClaim is
2Gi, but request is 5Gi.
```

Limits/Requests Ratio

If `LimitRangeItem.maxLimitRequestRatio` is specified in the `LimitRangeSpec`, the named resource must have a request and limit that are both non-zero where limit divided by request is less than or equal to the enumerated value

The following `LimitRange` enforces memory limit to be at most twice the amount of the memory request for any pod in the namespace.

admin/resource/limit-memory-ratio-pod.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-memory-ratio-pod
spec:
  limits:
  - maxLimitRequestRatio:
      memory: 2
    type: Pod
```

```
kubectl apply -f https://k8s.io/examples/admin/resource/limit-
memory-ratio-pod.yaml
```

Describe the LimitRange with the following kubectl command:

```
kubectl describe limitrange/limit-memory-ratio-pod
```

```
Name:        limit-memory-ratio-pod
Namespace:   limitrange-demo
Type         Resource  Min  Max  Default Request  Default Limit
Max Limit/Request Ratio
----         --------  ---  ---  ---------------  -------------
-----------------------
Pod          memory    -    -    -                -             2
```

Let's create a pod with `requests.memory=100Mi` and `limits.memory=300Mi`:

admin/resource/limit-range-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox3
spec:
  containers:
  - name: busybox-cnt01
    image: busybox
    resources:
      limits:
        memory: "300Mi"
      requests:
        memory: "100Mi"
```

```
kubectl apply -f https://k8s.io/examples/admin/resource/limit-range-pod-3.yaml
```

The pod creation failed as the ratio here (3) is greater than the enforced limit (2) in `limit-memory-ratio-pod` LimitRange

```
Error from server (Forbidden): error when creating "limit-range-pod-3.yaml": pods "busybox3" is forbidden: memory max limit to request ratio per Pod is 2, but provided ratio is 3.000000.
```

Clean up

Delete the `limitrange-demo` namespace to free all resources:

```
kubectl delete ns limitrange-demo
```

Examples

- See a tutorial on how to limit compute resources per namespace .
- Check how to limit storage consumption.
- See a detailed example on quota per namespace.

What's next

See LimitRanger design doc for more information.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 28, 2020 at 12:59 AM PST by remove $ following the style guide (#18855) (Page History)

Edit This Page

Resource Quotas

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Resource quotas are a tool for administrators to address this concern.

A resource quota, defined by a `ResourceQuota` object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that project.

Resource quotas work like this:

- Different teams work in different namespaces. Currently this is voluntary, but support for making this mandatory via ACLs is planned.
- The administrator creates one `ResourceQuota` for each namespace.
- Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a `ResourceQuota`.
- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code `403 FORBIDDEN` with a message explaining the constraint that would have been violated.
- If quota is enabled in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation. Hint: Use the `LimitRanger` admission controller to force defaults for pods that make no compute resource requirements. See the walkthrough for an example of how to avoid this problem.

Examples of policies that could be created using namespaces and quotas are:

- In a cluster with a capacity of 32 GiB RAM, and 16 cores, let team A use 20 GiB and 10 cores, let B use 10GiB and 4 cores, and hold 2GiB and 2 cores in reserve for future allocation.
- Limit the "testing" namespace to using 1 core and 1GiB RAM. Let the "production" namespace use any amount.

In the case where the total capacity of the cluster is less than the sum of the quotas of the namespaces, there may be contention for resources. This is handled on a first-come-first-served basis.

Neither contention nor changes to quota will affect already created resources.

Enabling Resource Quota

Resource Quota support is enabled by default for many Kubernetes distributions. It is enabled when the apiserver `--enable-admission-plugins=` flag has `ResourceQuota` as one of its arguments.

A resource quota is enforced in a particular namespace when there is a `ResourceQuota` in that namespace.

Compute Resource Quota

You can limit the total sum of compute resources that can be requested in a given namespace.

The following resource types are supported:

Resource Name	Description
`limits.cpu`	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
`limits.memory`	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
`requests.cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
`requests.memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.

Resource Quota For Extended Resources

In addition to the resources mentioned above, in release 1.10, quota support for extended resources is added.

As overcommit is not allowed for extended resources, it makes no sense to specify both `requests` and `limits` for the same extended resource in a quota. So for extended resources, only quota items with prefix `requests.` is allowed for now.

Take the GPU resource as an example, if the resource name is `nvidia.com/gpu`, and you want to limit the total number of GPUs requested in a namespace to 4, you can define a quota as follows:

- `requests.nvidia.com/gpu: 4`

See Viewing and Setting Quotas for more detail information.

Storage Resource Quota

You can limit the total sum of storage resources that can be requested in a given namespace.

In addition, you can limit consumption of storage resources based on associated storage-class.

Resource Name	Description
`requests.storage`	Across all persistent volume claims, the sum of storage requests cannot exceed this value.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the namespace.
`<storage-class-name>.storageclass.storage.k8s.io/requests.storage`	Across all persistent volume claims associated with the storage-class-name, the sum of storage requests cannot exceed this value.
`<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`	Across all persistent volume claims associated with the storage-class-name, the total number of persistent volume claims that can exist in the namespace.

For example, if an operator wants to quota storage with `gold` storage class separate from `bronze` storage class, the operator can define a quota as follows:

- `gold.storageclass.storage.k8s.io/requests.storage: 500Gi`
- `bronze.storageclass.storage.k8s.io/requests.storage: 100Gi`

In release 1.8, quota support for local ephemeral storage is added as an alpha feature:

Resource Name	Description
`requests.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage requests cannot exceed this value.
`limits.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage limits cannot exceed this value.

Object Count Quota

The 1.9 release added support to quota all standard namespaced resource types using the following syntax:

- `count/<resource>.<group>`

Here is an example set of resources users may want to put under object count quota:

- `count/persistentvolumeclaims`
- `count/services`
- `count/secrets`
- `count/configmaps`
- `count/replicationcontrollers`
- `count/deployments.apps`
- `count/replicasets.apps`
- `count/statefulsets.apps`
- `count/jobs.batch`
- `count/cronjobs.batch`
- `count/deployments.extensions`

The 1.15 release added support for custom resources using the same syntax. For example, to create a quota on a `widgets` custom resource in the `example.com` API group, use `count/widgets.example.com`.

When using `count/*` resource quota, an object is charged against the quota if it exists in server storage. These types of quotas are useful to protect against exhaustion of storage resources. For example, you may want to quota the number of secrets in a server given their large size. Too many secrets in a cluster can actually prevent servers and controllers from starting! You may choose to quota jobs to protect against a poorly configured cronjob creating too many jobs in a namespace causing a denial of service.

Prior to the 1.9 release, it was possible to do generic object count quota on a limited set of resources. In addition, it is possible to further constrain quota for particular resources by their type.

The following types are supported:

Resource Name	**Description**
`configmaps`	The total number of config maps that can exist in the namespace.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the namespace.
`pods`	The total number of pods in a non-terminal state that can exist in the namespace. A pod is in a terminal state if `.status.phase in (Failed, Succeeded)` is true.
`replicationcontrollers`	The total number of replication controllers that can exist in the namespace.
`resourcequotas`	The total number of resource quotas that can exist in the namespace.
`services`	The total number of services that can exist in the namespace.
`services.loadbalancers`	The total number of services of type load balancer that can exist in the namespace.

Resource Name	Description
`services.nodeports`	The total number of services of type node port that can exist in the namespace.
`secrets`	The total number of secrets that can exist in the namespace.

For example, `pods` quota counts and enforces a maximum on the number of `pods` created in a single namespace that are not terminal. You might want to set a `pods` quota on a namespace to avoid the case where a user creates many small pods and exhausts the cluster's supply of Pod IPs.

Quota Scopes

Each quota can have an associated set of scopes. A quota will only measure usage for a resource if it matches the intersection of enumerated scopes.

When a scope is added to the quota, it limits the number of resources it supports to those that pertain to the scope. Resources specified on the quota outside of the allowed set results in a validation error.

Scope	Description
`Terminating`	Match pods where `.spec.activeDeadlineSeconds >= 0`
`NotTerminating`	Match pods where `.spec.activeDeadlineSeconds is nil`
`BestEffort`	Match pods that have best effort quality of service.
`NotBestEffort`	Match pods that do not have best effort quality of service.

The `BestEffort` scope restricts a quota to tracking the following resource: `pods`

The `Terminating`, `NotTerminating`, and `NotBestEffort` scopes restrict a quota to tracking the following resources:

- `cpu`
- `limits.cpu`
- `limits.memory`
- `memory`
- `pods`
- `requests.cpu`
- `requests.memory`

Resource Quota Per PriorityClass

FEATURE STATE: `Kubernetes 1.12` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Pod Security Policies

FEATURE STATE: `Kubernetes v1.17` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Kubernetes Scheduler

In Kubernetes, *scheduling* refers to making sure that PodsThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. are matched to NodesA node is a worker machine in Kubernetes. so that KubeletAn agent that runs on each node in the cluster. It makes sure that containers are running in a pod. can run them.

- Scheduling overview
- kube-scheduler
- Scheduling with kube-scheduler
- What's next

Scheduling overview

A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.

If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling.

kube-scheduler

kube-scheduler is the default scheduler for Kubernetes and runs as part of the control planeThe container orchestration layer that exposes the API and interfaces to define, deploy, and manage the lifecycle of containers. . kube-scheduler is designed so that, if you want and need to, you can write your own scheduling component and use that instead.

For every newly created pod or other unscheduled pods, kube-scheduler selects an optimal node for them to run on. However, every container in pods has different requirements for resources and every pod also has different requirements. Therefore, existing nodes need to be filtered according to the specific scheduling requirements.

In a cluster, Nodes that meet the scheduling requirements for a Pod are called *feasible* nodes. If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.

The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called *binding*.

Factors that need taken into account for scheduling decisions include individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and so on.

Scheduling with kube-scheduler

kube-scheduler selects a node for the pod in a 2-step operation:

1. Filtering
2. Scoring

The *filtering* step finds the set of Nodes where it's feasible to schedule the Pod. For example, the PodFitsResources filter checks whether a candidate Node has enough available resource to meet a Pod's specific resource requests. After this step, the node list contains any suitable Nodes; often, there will be more than one. If the list is empty, that Pod isn't (yet) schedulable.

In the *scoring* step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules.

Finally, kube-scheduler assigns the Pod to the Node with the highest ranking. If there is more than one node with equal scores, kube-scheduler selects one of these at random.

Default policies

kube-scheduler has a default set of scheduling policies.

Filtering

- `PodFitsHostPorts`: Checks if a Node has free ports (the network protocol kind) for the Pod ports the Pod is requesting.
- `PodFitsHost`: Checks if a Pod specifies a specific Node by its hostname.
- `PodFitsResources`: Checks if the Node has free resources (eg, CPU and Memory) to meet the requirement of the Pod.
- `PodMatchNodeSelector`: Checks if a Pod's Node SelectorAllows users to filter a list of resources based on labels. matches the Node's label(s)Tags objects with identifying attributes that are meaningful and relevant to users. .

- `NoVolumeZoneConflict`: Evaluate if the VolumesA directory containing data, accessible to the containers in a pod. that a Pod requests are available on the Node, given the failure zone restrictions for that storage.

- `NoDiskConflict`: Evaluates if a Pod can fit on a Node due to the volumes it requests, and those that are already mounted.

- `MaxCSIVolumeCount`: Decides how many CSIThe Container Storage Interface (CSI) defines a standard interface to expose storage systems to containers. volumes should be attached, and whether that's over a configured limit.

- `CheckNodeMemoryPressure`: If a Node is reporting memory pressure, and there's no configured exception, the Pod won't be scheduled there.

- `CheckNodePIDPressure`: If a Node is reporting that process IDs are scarce, and there's no configured exception, the Pod won't be scheduled there.

- `CheckNodeDiskPressure`: If a Node is reporting storage pressure (a filesystem that is full or nearly full), and there's no configured exception, the Pod won't be scheduled there.

- `CheckNodeCondition`: Nodes can report that they have a completely full filesystem, that networking isn't available or that kubelet is otherwise not ready to run Pods. If such a condition is set for a Node, and there's no configured exception, the Pod won't be scheduled there.

- `PodToleratesNodeTaints`: checks if a Pod's tolerationsA core object consisting of three required properties: key, value, and effect. Tolerations enable the scheduling of pods on nodes or node groups that have a matching taint. can tolerate the Node's taintsA core object consisting of three required properties: key, value, and effect. Taints prevent the scheduling of pods on nodes or node groups. .

- `CheckVolumeBinding`: Evaluates if a Pod can fit due to the volumes it requests. This applies for both bound and unbound PVCsClaims storage resources defined in a PersistentVolume so that it can be mounted as a volume in a container. .

Scoring

- `SelectorSpreadPriority`: Spreads Pods across hosts, considering Pods that belong to the same ServiceA way to expose an application running on a set of Pods as a network service. , StatefulSetManages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods. or ReplicaSetReplicaSet ensures that a specified number of Pod replicas are running at one time .

- `InterPodAffinityPriority`: Computes a sum by iterating through the elements of weightedPodAffinityTerm and adding "weight" to the sum if the corresponding PodAffinityTerm is satisfied for that node; the node(s) with the highest sum are the most preferred.

- `LeastRequestedPriority`: Favors nodes with fewer requested resources. In other words, the more Pods that are placed on a Node, and the more resources those Pods use, the lower the ranking this policy will give.

- `MostRequestedPriority`: Favors nodes with most requested resources. This policy will fit the scheduled Pods onto the smallest number of Nodes needed to run your overall set of workloads.

- `RequestedToCapacityRatioPriority`: Creates a requestedToCapacity based ResourceAllocationPriority using default resource scoring function shape.

- `BalancedResourceAllocation`: Favors nodes with balanced resource usage.

- `NodePreferAvoidPodsPriority`: Prioritizes nodes according to the node annotation `scheduler.alpha.kubernetes.io/preferAvoidPods`. You can use this to hint that two different Pods shouldn't run on the same Node.

- `NodeAffinityPriority`: Prioritizes nodes according to node affinity scheduling preferences indicated in PreferredDuringSchedulingIgnoredDuringExecution. You can read more about this in Assigning Pods to Nodes.

- `TaintTolerationPriority`: Prepares the priority list for all the nodes, based on the number of intolerable taints on the node. This policy adjusts a node's rank taking that list into account.

- `ImageLocalityPriority`: Favors nodes that already have the container imagesStored instance of a container that holds a set of software needed to run an application. for that Pod cached locally.

- `ServiceSpreadingPriority`: For a given Service, this policy aims to make sure that the Pods for the Service run on different nodes. It favours scheduling onto nodes that don't have Pods for the service already assigned there. The overall outcome is that the Service becomes more resilient to a single Node failure.

- `CalculateAntiAffinityPriorityMap`: This policy helps implement pod anti-affinity.

- `EqualPriorityMap`: Gives an equal weight of one to all nodes.

What's next

- Read about scheduler performance tuning
- Read about Pod topology spread constraints
- Read the reference documentation for kube-scheduler

- Learn about configuring multiple schedulers
- Learn about topology management policies
- Learn about Pod Overhead

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Page last modified on January 15, 2020 at 2:15 AM PST by Update kube-scheduler.md (#18133) (Page History)

Edit This Page

Scheduler Performance Tuning

FEATURE STATE: `Kubernetes 1.14` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Cluster Administration Overview

The cluster administration overview is for anyone creating or administering a Kubernetes cluster. It assumes some familiarity with core Kubernetes concepts.

- Planning a cluster
- Managing a cluster
- Securing a cluster
- Optional Cluster Services

Planning a cluster

See the guides in Setup for examples of how to plan, set up, and configure Kubernetes clusters. The solutions listed in this article are called *distros*.

Before choosing a guide, here are some considerations:

- Do you just want to try out Kubernetes on your computer, or do you want to build a high-availability, multi-node cluster? Choose distros best suited for your needs.
- **If you are designing for high-availability**, learn about configuring clusters in multiple zones.
- Will you be using **a hosted Kubernetes cluster**, such as Google Kubernetes Engine, or **hosting your own cluster**?
- Will your cluster be **on-premises**, or **in the cloud (IaaS)**? Kubernetes does not directly support hybrid clusters. Instead, you can set up multiple clusters.
- **If you are configuring Kubernetes on-premises**, consider which networking model fits best.
- Will you be running Kubernetes on **"bare metal" hardware** or on **virtual machines (VMs)**?
- Do you **just want to run a cluster**, or do you expect to do **active development of Kubernetes project code**? If the latter, choose an actively-developed distro. Some distros only use binary releases, but offer a greater variety of choices.
- Familiarize yourself with the components needed to run a cluster.

Note: Not all distros are actively maintained. Choose distros which have been tested with a recent version of Kubernetes.

Managing a cluster

- Managing a cluster describes several topics related to the lifecycle of a cluster: creating a new cluster, upgrading your cluster's master and worker nodes, performing node maintenance (e.g. kernel upgrades), and upgrading the Kubernetes API version of a running cluster.
- Learn how to manage nodes.
- Learn how to set up and manage the resource quota for shared clusters.

Securing a cluster

- Certificates describes the steps to generate certificates using different tool chains.
- Kubernetes Container Environment describes the environment for Kubelet managed containers on a Kubernetes node.
- Controlling Access to the Kubernetes API describes how to set up permissions for users and service accounts.
- Authenticating explains authentication in Kubernetes, including the various authentication options.
- Authorization is separate from authentication, and controls how HTTP calls are handled.
- Using Admission Controllers explains plug-ins which intercepts requests to the Kubernetes API server after authentication and authorization.
- Using Sysctls in a Kubernetes Cluster describes to an administrator how to use the `sysctl` command-line tool to set kernel parameters .
- Auditing describes how to interact with Kubernetes' audit logs.

Securing the kubelet

- Master-Node communication
- TLS bootstrapping
- Kubelet authentication/authorization

Optional Cluster Services

- DNS Integration describes how to resolve a DNS name directly to a Kubernetes service.

- Logging and Monitoring Cluster Activity explains how logging in Kubernetes works and how to implement it.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on May 23, 2019 at 4:56 AM PST by Fix dangling "pick-right-solution" references - English (#14468) (Page History)

Edit This Page

Certificates

When using client certificate authentication, you can generate certificates manually through `easyrsa`, `openssl` or `cfssl`.

- Distributing Self-Signed CA Certificate
- Certificates API

easyrsa

easyrsa can manually generate certificates for your cluster.

1. Download, unpack, and initialize the patched version of easyrsa3.

   ```
   curl -LO https://storage.googleapis.com/kubernetes-release/easy-rsa/easy-rsa.tar.gz
   tar xzf easy-rsa.tar.gz
   cd easy-rsa-master/easyrsa3
   ./easyrsa init-pki
   ```

2. Generate a new certificate authority (CA). `--batch` sets automatic mode; `--req-cn` specifies the Common Name (CN) for the CA's new root certificate.

   ```
   ./easyrsa --batch "--req-cn=${MASTER_IP}@`date +%s`" build-ca nopass
   ```

3. Generate server certificate and key. The argument `--subject-alt-name` sets the possible IPs and DNS names the API server will be accessed with. The `MASTER_CLUSTER_IP` is usually the first IP from the service CIDR that is specified as the `--service-cluster-ip-range` argument for both the API server and the controller manager component. The argument `--days` is used to set the number of days after which the certificate expires. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

   ```
   ./easyrsa --subject-alt-name="IP:${MASTER_IP},"\
   "IP:${MASTER_CLUSTER_IP},"\
   "DNS:kubernetes,"\
   "DNS:kubernetes.default,"\
   "DNS:kubernetes.default.svc,"\
   "DNS:kubernetes.default.svc.cluster,"\
   "DNS:kubernetes.default.svc.cluster.local" \
   --days=10000 \
   build-server-full server nopass
   ```

4. Copy `pki/ca.crt`, `pki/issued/server.crt`, and `pki/private/server.key` to your directory.

5. Fill in and add the following parameters into the API server start parameters:

   ```
   --client-ca-file=/yourdirectory/ca.crt
   --tls-cert-file=/yourdirectory/server.crt
   --tls-private-key-file=/yourdirectory/server.key
   ```

openssl

openssl can manually generate certificates for your cluster.

1. Generate a ca.key with 2048bit:

   ```
   openssl genrsa -out ca.key 2048
   ```

2. According to the ca.key generate a ca.crt (use -days to set the certificate effective time):

   ```
   openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTER_IP}" -days 10000 -out ca.crt
   ```

3. Generate a server.key with 2048bit:

   ```
   openssl genrsa -out server.key 2048
   ```

4. Create a config file for generating a Certificate Signing Request (CSR). Be sure to substitute the values marked with angle brackets (e.g. `<MASTER_IP>`) with real values before saving this to a file (e.g. `csr.conf`). Note that the value for `MASTER_CLUSTER_IP` is the service cluster IP for the API server as described in previous subsection. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

   ```
   [ req ]
   default_bits = 2048
   prompt = no
   default_md = sha256
   req_extensions = req_ext
   distinguished_name = dn

   [ dn ]
   C = <country>
   ST = <state>
   L = <city>
   O = <organization>
   OU = <organization unit>
   CN = <MASTER_IP>

   [ req_ext ]
   subjectAltName = @alt_names

   [ alt_names ]
   DNS.1 = kubernetes
   ```

```
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster
DNS.5 = kubernetes.default.svc.cluster.local
IP.1 = <MASTER_IP>
IP.2 = <MASTER_CLUSTER_IP>

[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
subjectAltName=@alt_names
```

5. Generate the certificate signing request based on the config file:

   ```
   openssl req -new -key server.key -out server.csr -config csr.conf
   ```

6. Generate the server certificate using the ca.key, ca.crt and server.csr:

   ```
   openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
   -CAcreateserial -out server.crt -days 10000 \
   -extensions v3_ext -extfile csr.conf
   ```

7. View the certificate:

   ```
   openssl x509  -noout -text -in ./server.crt
   ```

Finally, add the same parameters into the API server start parameters.

cfssl

cfssl is another tool for certificate generation.

1. Download, unpack and prepare the command line tools as shown below. Note that you may need to adapt the sample commands based on the hardware architecture and cfssl version you are using.

   ```
   curl -L https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 -o cfssl
   chmod +x cfssl
   curl -L https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64 -o cfssljson
   chmod +x cfssljson
   curl -L https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64 -o cfssl-certinfo
   chmod +x cfssl-certinfo
   ```

2. Create a directory to hold the artifacts and initialize cfssl:

   ```
   mkdir cert
   cd cert
   ```

```
../cfssl print-defaults config > config.json
../cfssl print-defaults csr > csr.json
```

3. Create a JSON config file for generating the CA file, for example, `ca-config.json`:

   ```
   {
     "signing": {
       "default": {
         "expiry": "8760h"
       },
       "profiles": {
         "kubernetes": {
           "usages": [
             "signing",
             "key encipherment",
             "server auth",
             "client auth"
           ],
           "expiry": "8760h"
         }
       }
     }
   }
   ```

4. Create a JSON config file for CA certificate signing request (CSR), for example, `ca-csr.json`. Be sure to replace the values marked with angle brackets with real values you want to use.

   ```
   {
     "CN": "kubernetes",
     "key": {
       "algo": "rsa",
       "size": 2048
     },
     "names":[{
       "C": "<country>",
       "ST": "<state>",
       "L": "<city>",
       "O": "<organization>",
       "OU": "<organization unit>"
     }]
   }
   ```

5. Generate CA key (`ca-key.pem`) and certificate (`ca.pem`):

   ```
   ../cfssl gencert -initca ca-csr.json | ../cfssljson -bare ca
   ```

6. Create a JSON config file for generating keys and certificates for the API server, for example, `server-csr.json`. Be sure to replace the values in angle brackets with real values you want to use. The `MASTER_CLUSTER_IP` is the service cluster IP for the API server as described in

previous subsection. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

```
{
  "CN": "kubernetes",
  "hosts": [
    "127.0.0.1",
    "<MASTER_IP>",
    "<MASTER_CLUSTER_IP>",
    "kubernetes",
    "kubernetes.default",
    "kubernetes.default.svc",
    "kubernetes.default.svc.cluster",
    "kubernetes.default.svc.cluster.local"
  ],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [{
    "C": "<country>",
    "ST": "<state>",
    "L": "<city>",
    "O": "<organization>",
    "OU": "<organization unit>"
  }]
}
```

7. Generate the key and certificate for the API server, which are by default saved into file `server-key.pem` and `server.pem` respectively:

```
../cfssl gencert -ca=ca.pem -ca-key=ca-key.pem \
--config=ca-config.json -profile=kubernetes \
server-csr.json | ../cfssljson -bare server
```

Distributing Self-Signed CA Certificate

A client node may refuse to recognize a self-signed CA certificate as valid. For a non-production deployment, or for a deployment that runs behind a company firewall, you can distribute a self-signed CA certificate to all clients and refresh the local list for valid certificates.

On each client, perform the following operations:

```
sudo cp ca.crt /usr/local/share/ca-certificates/kubernetes.crt
sudo update-ca-certificates
```

```
Updating certificates in /etc/ssl/certs...
1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....
done.
```

Certificates API

You can use the `certificates.k8s.io` API to provision x509 certificates to use for authentication as documented here.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on October 10, 2019 at 5:21 AM PST by Add explanation for "CN" (#16714) (Page History)

Edit This Page

Cloud Providers

This page explains how to manage Kubernetes running on a specific cloud provider.

- AWS
- Azure
- CloudStack
- GCE
- OpenStack
- OVirt
- Photon
- vSphere
- IBM Cloud Kubernetes Service
- Baidu Cloud Container Engine
- Tencent Kubernetes Engine

kubeadm

kubeadm is a popular option for creating kubernetes clusters. kubeadm has configuration options to specify configuration information for cloud providers. For example a typical in-tree cloud provider can be configured using kubeadm as shown below:

```
apiVersion: kubeadm.k8s.io/v1beta2
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    cloud-provider: "openstack"
    cloud-config: "/etc/kubernetes/cloud.conf"
---
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
kubernetesVersion: v1.13.0
apiServer:
  extraArgs:
    cloud-provider: "openstack"
    cloud-config: "/etc/kubernetes/cloud.conf"
  extraVolumes:
  - name: cloud
    hostPath: "/etc/kubernetes/cloud.conf"
    mountPath: "/etc/kubernetes/cloud.conf"
controllerManager:
  extraArgs:
    cloud-provider: "openstack"
    cloud-config: "/etc/kubernetes/cloud.conf"
  extraVolumes:
  - name: cloud
    hostPath: "/etc/kubernetes/cloud.conf"
    mountPath: "/etc/kubernetes/cloud.conf"
```

The in-tree cloud providers typically need both `--cloud-provider` and `--cloud-config` specified in the command lines for the kube-apiserver, kube-controller-manager and the kubelet. The contents of the file specified in `--cloud-config` for each provider is documented below as well.

For all external cloud providers, please follow the instructions on the individual repositories, which are listed under their headings below, or one may view the list of all repositories

AWS

This section describes all the possible configurations which can be used when running Kubernetes on Amazon Web Services.

If you wish to use the external cloud provider, its repository is kubernetes/cloud-provider-aws

Node Name

The AWS cloud provider uses the private DNS name of the AWS instance as the name of the Kubernetes Node object.

Load Balancers

You can setup external load balancers to use specific features in AWS by configuring the annotations as shown below.

```
apiVersion: v1
kind: Service
metadata:
  name: example
  namespace: kube-system
  labels:
    run: example
  annotations:
     service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:xx-xxxx-x:xxxxxxxxx:xxxxxxx/xxxxx-xxxx-xxxx-xxxx-xxxxxxxxx #replace this value
     service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
spec:
  type: LoadBalancer
  ports:
  - port: 443
    targetPort: 5556
    protocol: TCP
  selector:
    app: example
```

Different settings can be applied to a load balancer service in AWS using *annotations*. The following describes the annotations supported on AWS ELBs:

- `service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval`: Used to specify access log emit interval.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-enabled`: Used on the service to enable or disable access logs.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name`: Used to specify access log s3 bucket name.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix`: Used to specify access log s3 bucket prefix.
- `service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags`: Used on the service to specify a comma-separated list of key-value pairs which will be recorded as additional tags in the ELB. For example: `"Key1=Val1,Key2=Val2,KeyNoVal1=,KeyNoVal2"`.
- `service.beta.kubernetes.io/aws-load-balancer-backend-protocol`: Used on the service to specify the protocol spoken by the backend (pod) behind a listener. If `http` (default) or `https`, an HTTPS listener that terminates the connection and parses headers is created. If set to `ssl` or `tcp`, a "raw" SSL listener is used. If set to `http` and `aws-load-balancer-ssl-cert` is not used then a HTTP listener is used.
- `service.beta.kubernetes.io/aws-load-balancer-ssl-cert`: Used on the service to request a secure listener. Value is a valid certificate ARN. For more, see ELB Listener Config CertARN is an IAM or CM certificate ARN, e.g. `arn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-123456789012`.
- `service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled`: Used on the service to enable or disable connection draining.
- `service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout`: Used on the service to specify a connection draining timeout.
- `service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout`: Used on the service to specify the idle connection timeout.
- `service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled`: Used on the service to enable or disable cross-zone load balancing.
- `service.beta.kubernetes.io/aws-load-balancer-security-groups`: Used to specify the security groups to be added to ELB created. This replaces all other security groups previously assigned to the ELB.
- `service.beta.kubernetes.io/aws-load-balancer-extra-security-groups`: Used on the service to specify additional security groups to be added to ELB created
- `service.beta.kubernetes.io/aws-load-balancer-internal`: Used on the service to indicate that we want an internal ELB.
- `service.beta.kubernetes.io/aws-load-balancer-proxy-protocol`: Used on the service to enable the proxy protocol on an ELB. Right now

we only accept the value `*` which means enabling the proxy protocol on all ELB backends. In the future we could adjust this to allow setting the proxy protocol only on certain backends.
- `service.beta.kubernetes.io/aws-load-balancer-ssl-ports`: Used on the service to specify a comma-separated list of ports that will use SSL/HTTPS listeners. Defaults to `*` (all)

The information for the annotations for AWS is taken from the comments on [aws.go]()

Azure

If you wish to use the external cloud provider, its repository is [kubernetes/cloud-provider-azure]()

Node Name

The Azure cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the Azure VM name.

CloudStack

If you wish to use the external cloud provider, its repository is [apache/cloudstack-kubernetes-provider]()

Node Name

The CloudStack cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the CloudStack VM name.

GCE

If you wish to use the external cloud provider, its repository is [kubernetes/cloud-provider-gcp]()

Node Name

The GCE cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the first segment of the Kubernetes Node name must match the GCE instance name (e.g. a Node named `kubernetes-node-2.c.my-proj.internal` must correspond to an instance named `kubernetes-node-2`).

OpenStack

This section describes all the possible configurations which can be used when using OpenStack with Kubernetes.

If you wish to use the external cloud provider, its repository is kubernetes/cloud-provider-openstack

Node Name

The OpenStack cloud provider uses the instance name (as determined from OpenStack metadata) as the name of the Kubernetes Node object. Note that the instance name must be a valid Kubernetes Node name in order for the kubelet to successfully register its Node object.

Services

The OpenStack cloud provider implementation for Kubernetes supports the use of these OpenStack services from the underlying cloud, where available:

Service	API Version(s)	Required
Block Storage (Cinder)	V1â€ , V2, V3	No
Compute (Nova)	V2	No
Identity (Keystone)	V2â€¡, V3	Yes
Load Balancing (Neutron)	V1Â§, V2	No
Load Balancing (Octavia)	V2	No

â€ Block Storage V1 API support is deprecated, Block Storage V3 API support was added in Kubernetes 1.9.

â€¡ Identity V2 API support is deprecated and will be removed from the provider in a future release. As of the "Queens" release, OpenStack will no longer expose the Identity V2 API.

Â§ Load Balancing V1 API support was removed in Kubernetes 1.9.

Service discovery is achieved by listing the service catalog managed by OpenStack Identity (Keystone) using the `auth-url` provided in the provider configuration. The provider will gracefully degrade in functionality when OpenStack services other than Keystone are not available and simply disclaim support for impacted features. Certain features are also enabled or disabled based on the list of extensions published by Neutron in the underlying cloud.

cloud.conf

Kubernetes knows how to interact with OpenStack via the file cloud.conf. It is the file that will provide Kubernetes with credentials and location for the OpenStack auth endpoint. You can create a cloud.conf file by specifying the following details in it

Typical configuration

This is an example of a typical configuration that touches the values that most often need to be set. It points the provider at the OpenStack cloud's Keystone endpoint, provides details for how to authenticate with it, and configures the load balancer:

```
[Global]
username=user
password=pass
auth-url=https://<keystone_ip>/identity/v3
tenant-id=c869168a828847f39f7f06edd7305637
domain-id=2a73b8f597c04551a0fdc8e95544be8a

[LoadBalancer]
subnet-id=6937f8fa-858d-4bc9-a3a5-18d2c957166a
```

Global

These configuration options for the OpenStack provider pertain to its global configuration and should appear in the `[Global]` section of the `cloud.conf` file:

- `auth-url` (Required): The URL of the keystone API used to authenticate. On OpenStack control panels, this can be found at Access and Security > API Access > Credentials.
- `username` (Required): Refers to the username of a valid user set in keystone.
- `password` (Required): Refers to the password of a valid user set in keystone.
- `tenant-id` (Required): Used to specify the id of the project where you want to create your resources.
- `tenant-name` (Optional): Used to specify the name of the project where you want to create your resources.
- `trust-id` (Optional): Used to specify the identifier of the trust to use for authorization. A trust represents a user's (the trustor) authorization to delegate roles to another user (the trustee), and optionally allow the trustee to impersonate the trustor. Available trusts are found under the `/v3/OS-TRUST/trusts` endpoint of the Keystone API.
- `domain-id` (Optional): Used to specify the id of the domain your user belongs to.
- `domain-name` (Optional): Used to specify the name of the domain your user belongs to.
- `region` (Optional): Used to specify the identifier of the region to use when running on a multi-region OpenStack cloud. A region is a general division of an OpenStack deployment. Although a region does not have a strict geographical connotation, a deployment can use a geographical name for a region identifier such as `us-east`. Available regions are found under the `/v3/regions` endpoint of the Keystone API.
- `ca-file` (Optional): Used to specify the path to your custom CA file.

When using Keystone V3 - which changes tenant to project - the `tenant-id` value is automatically mapped to the project construct in the API.

Load Balancer

These configuration options for the OpenStack provider pertain to the load balancer and should appear in the `[LoadBalancer]` section of the `cloud.conf` file:

- `lb-version` (Optional): Used to override automatic version detection. Valid values are `v1` or `v2`. Where no value is provided automatic detection will select the highest supported version exposed by the underlying OpenStack cloud.
- `use-octavia` (Optional): Used to determine whether to look for and use an Octavia LBaaS V2 service catalog endpoint. Valid values are `true` or `false`. Where `true` is specified and an Octaiva LBaaS V2 entry can not be found, the provider will fall back and attempt to find a Neutron LBaaS V2 endpoint instead. The default value is `false`.
- `subnet-id` (Optional): Used to specify the id of the subnet you want to create your loadbalancer on. Can be found at Network > Networks. Click on the respective network to get its subnets.
- `floating-network-id` (Optional): If specified, will create a floating IP for the load balancer.
- `lb-method` (Optional): Used to specify an algorithm by which load will be distributed amongst members of the load balancer pool. The value can be `ROUND_ROBIN`, `LEAST_CONNECTIONS`, or `SOURCE_IP`. The default behavior if none is specified is `ROUND_ROBIN`.
- `lb-provider` (Optional): Used to specify the provider of the load balancer. If not specified, the default provider service configured in neutron will be used.
- `create-monitor` (Optional): Indicates whether or not to create a health monitor for the Neutron load balancer. Valid values are `true` and `false`. The default is `false`. When `true` is specified then `monitor-delay`, `monitor-timeout`, and `monitor-max-retries` must also be set.
- `monitor-delay` (Optional): The time between sending probes to members of the load balancer. Ensure that you specify a valid time unit. The valid time units are "ns", "us" (or "Âµs"), "ms", "s", "m", "h"
- `monitor-timeout` (Optional): Maximum time for a monitor to wait for a ping reply before it times out. The value must be less than the delay value. Ensure that you specify a valid time unit. The valid time units are "ns", "us" (or "Âµs"), "ms", "s", "m", "h"
- `monitor-max-retries` (Optional): Number of permissible ping failures before changing the load balancer member's status to INACTIVE. Must be a number between 1 and 10.
- `manage-security-groups` (Optional): Determines whether or not the load balancer should automatically manage the security group rules. Valid values are `true` and `false`. The default is `false`. When `true` is specified `node-security-group` must also be supplied.
- `node-security-group` (Optional): ID of the security group to manage.

Block Storage

These configuration options for the OpenStack provider pertain to block storage and should appear in the `[BlockStorage]` section of the `cloud.conf` file:

- `bs-version` (Optional): Used to override automatic version detection. Valid values are `v1`, `v2`, `v3` and `auto`. When `auto` is specified automatic detection will select the highest supported version exposed by the underlying OpenStack cloud. The default value if none is provided is `auto`.
- `trust-device-path` (Optional): In most scenarios the block device names provided by Cinder (e.g. `/dev/vda`) can not be trusted. This boolean toggles this behavior. Setting it to `true` results in trusting the block device names provided by Cinder. The default value of `false` results in the discovery of the device path based on its serial number and `/dev/disk/by-id` mapping and is the recommended approach.
- `ignore-volume-az` (Optional): Used to influence availability zone use when attaching Cinder volumes. When Nova and Cinder have different availability zones, this should be set to `true`. This is most commonly the case where there are many Nova availability zones but only one Cinder availability zone. The default value is `false` to preserve the behavior used in earlier releases, but may change in the future.
- `node-volume-attach-limit` (Optional): Maximum number of Volumes that can be attached to the node, default is 256 for cinder.

If deploying Kubernetes versions <= 1.8 on an OpenStack deployment that uses paths rather than ports to differentiate between endpoints it may be necessary to explicitly set the `bs-version` parameter. A path based endpoint is of the form `http://foo.bar/volume` while a port based endpoint is of the form `http://foo.bar:xxx`.

In environments that use path based endpoints and Kubernetes is using the older auto-detection logic a `BS API version autodetection failed.` error will be returned on attempting volume detachment. To workaround this issue it is possible to force the use of Cinder API version 2 by adding this to the cloud provider configuration:

```
[BlockStorage]
bs-version=v2
```

Metadata

These configuration options for the OpenStack provider pertain to metadata and should appear in the `[Metadata]` section of the `cloud.conf` file:

- `search-order` (Optional): This configuration key influences the way that the provider retrieves metadata relating to the instance(s) in which it runs. The default value of `configDrive,metadataService` results in the provider retrieving metadata relating to the instance from the

config drive first if available and then the metadata service. Alternative values are:
    - `configDrive` - Only retrieve instance metadata from the configuration drive.
    - `metadataService` - Only retrieve instance metadata from the metadata service.
    - `metadataService,configDrive` - Retrieve instance metadata from the metadata service first if available, then the configuration drive.

Influencing this behavior may be desirable as the metadata on the configuration drive may grow stale over time, whereas the metadata service always provides the most up to date view. Not all OpenStack clouds provide both configuration drive and metadata service though and only one or the other may be available which is why the default is to check both.

Route

These configuration options for the OpenStack provider pertain to the kubenet Kubernetes network plugin and should appear in the `[Route]` section of the `cloud.conf` file:

- `router-id` (Optional): If the underlying cloud's Neutron deployment supports the `extraroutes` extension then use `router-id` to specify a router to add routes to. The router chosen must span the private networks containing your cluster nodes (typically there is only one node network, and this value should be the default router for the node network). This value is required to use kubenet on OpenStack.

OVirt

Node Name

The OVirt cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the VM FQDN (reported by OVirt under `<vm><guest_info><fqdn>...</fqdn></guest_info></vm>`)

Photon

Node Name

The Photon cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the Photon VM name (or if `overrideIP` is set to true in the `--cloud-config`, the Kubernetes Node name must match the Photon VM IP address).

vSphere

- vSphere >= 6.7U3
- vSphere < 6.7U3

For all vSphere deployments on vSphere >= 6.7U3, the external vSphere cloud provider, along with the vSphere CSI driver is recommended. See Deploying a Kubernetes Cluster on vSphere with CSI and CPI for a quick start guide.

If you are running vSphere < 6.7U3, the in-tree vSphere cloud provider is recommended. See Running a Kubernetes Cluster on vSphere with kubeadm for a quick start guide.

For in-depth documentation on the vSphere cloud provider, visit the vSphere cloud provider docs site.

IBM Cloud Kubernetes Service

Compute nodes

By using the IBM Cloud Kubernetes Service provider, you can create clusters with a mixture of virtual and physical (bare metal) nodes in a single zone or across multiple zones in a region. For more information, see Planning your cluster and worker node setup.

The name of the Kubernetes Node object is the private IP address of the IBM Cloud Kubernetes Service worker node instance.

Networking

The IBM Cloud Kubernetes Service provider provides VLANs for quality network performance and network isolation for nodes. You can set up custom firewalls and Calico network policies to add an extra layer of security for your cluster, or connect your cluster to your on-prem data center via VPN. For more information, see Planning in-cluster and private networking.

To expose apps to the public or within the cluster, you can leverage NodePort, LoadBalancer, or Ingress services. You can also customize the Ingress application load balancer with annotations. For more information, see Planning to expose your apps with external networking.

Storage

The IBM Cloud Kubernetes Service provider leverages Kubernetes-native persistent volumes to enable users to mount file, block, and cloud object storage to their apps. You can also use database-as-a-service and third-party add-ons for persistent storage of your data. For more information, see Planning highly available persistent storage.

Baidu Cloud Container Engine

Node Name

The Baidu cloud provider uses the private IP address of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the Baidu VM private IP.

Tencent Kubernetes Engine

If you wish to use the external cloud provider, its repository is [TencentCloud/tencentcloud-cloud-controller-manager]().

Node Name

The Tencent cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the Tencent VM private IP.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](). Open an issue in the GitHub repo if you want to [report a problem]() or [suggest an improvement]().

Create an Issue Edit This Page
Page last modified on December 02, 2019 at 4:27 PM PST by Fix broken link to AWS provider settings source (#17899) (Page History)

Edit This Page

Managing Resources

You've deployed your application and exposed it via a service. Now what? Kubernetes provides a number of tools to help you manage your application deployment, including scaling and updating. Among the features that we will discuss in more depth are configuration files and labels.

Organizing resource configurations

Many applications require multiple resources to be created, such as a Deployment and a Service. Management of multiple resources can be simplified by grouping them together in the same file (separated by `---` in YAML). For example:

application/nginx-app.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Multiple resources can be created the same way as a single resource:

```
kubectl apply -f https://k8s.io/examples/application/nginx-app.yaml
```

```
service/my-nginx-svc created
deployment.apps/my-nginx created
```

The resources will be created in the order they appear in the file. Therefore, it's best to specify the service first, since that will ensure the scheduler can spread the pods associated with the service as they are created by the controller(s), such as Deployment.

`kubectl apply` also accepts multiple `-f` arguments:

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

And a directory can be specified rather than or in addition to individual files:

```
kubectl apply -f https://k8s.io/examples/application/nginx/
```

`kubectl` will read any files with suffixes `.yaml`, `.yml`, or `.json`.

It is a recommended practice to put resources related to the same microservice or application tier into the same file, and to group all of the files associated with your application in the same directory. If the tiers of your application bind to each other using DNS, then you can then simply deploy all of the components of your stack en masse.

A URL can also be specified as a configuration source, which is handy for deploying directly from configuration files checked into github:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/application/nginx/nginx-deployment.yaml
```

```
deployment.apps/my-nginx created
```

Bulk operations in kubectl

Resource creation isn't the only operation that `kubectl` can perform in bulk. It can also extract resource names from configuration files in order to perform other operations, in particular to delete the same resources you created:

```
kubectl delete -f https://k8s.io/examples/application/nginx-app.yaml
```

```
deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted
```

In the case of just two resources, it's also easy to specify both on the command line using the resource/name syntax:

```
kubectl delete deployments/my-nginx services/my-nginx-svc
```

For larger numbers of resources, you'll find it easier to specify the selector (label query) specified using `-l` or `--selector`, to filter resources by their labels:

```
kubectl delete deployment,services -l app=nginx
```

```
deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted
```

Because `kubectl` outputs resource names in the same syntax it accepts, it's easy to chain operations using `$()` or `xargs`:

```
kubectl get $(kubectl create -f docs/concepts/cluster-
administration/nginx/ -o name | grep service)
```

```
NAME           TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
     AGE
my-nginx-svc   LoadBalancer   10.0.0.208   <pending>     80/
TCP       0s
```

With the above commands, we first create resources under `examples/application/nginx/` and print the resources created with `-o name` output format (print each resource as resource/name). Then we `grep` only the "service", and then print it with `kubectl get`.

If you happen to organize your resources across several subdirectories within a particular directory, you can recursively perform the operations on the subdirectories also, by specifying `--recursive` or `-R` alongside the `--filename,-f` flag.

For instance, assume there is a directory `project/k8s/development` that holds all of the manifestsA serialized specification of one or more Kubernetes API objects. needed for the development environment, organized by resource type:

```
project/k8s/development
â”œâ”€â”€ configmap
â”‚Â Â  â””â”€â”€ my-configmap.yaml
â”œâ”€â”€ deployment
â”‚Â Â  â””â”€â”€ my-deployment.yaml
â””â”€â”€ pvc
    â””â”€â”€ my-pvc.yaml
```

By default, performing a bulk operation on `project/k8s/development` will stop at the first level of the directory, not processing any subdirectories. If we had tried to create the resources in this directory using the following command, we would have encountered an error:

```
kubectl apply -f project/k8s/development
```

```
error: you must provide one or more resources by argument or
filename (.json|.yaml|.yml|stdin)
```

Instead, specify the `--recursive` or `-R` flag with the `--filename,-f` flag as such:

```
kubectl apply -f project/k8s/development --recursive
```

```
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

The `--recursive` flag works with any operation that accepts the `--filename,-f` flag such as: `kubectl {create,get,delete,describe,rollout} etc.`

The `--recursive` flag also works when multiple `-f` arguments are provided:

```
kubectl apply -f project/k8s/namespaces -f project/k8s/
development --recursive
```

```
namespace/development created
namespace/staging created
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

If you're interested in learning more about `kubectl`, go ahead and read kubectl Overview.

Using labels effectively

The examples we've used so far apply at most a single label to any resource. There are many scenarios where multiple labels should be used to distinguish sets from one another.

For instance, different applications would use different values for the `app` label, but a multi-tier application, such as the guestbook example, would additionally need to distinguish each tier. The frontend could carry the following labels:

```
     labels:
        app: guestbook
        tier: frontend
```

while the Redis master and slave would have different `tier` labels, and perhaps even an additional `role` label:

```
     labels:
        app: guestbook
        tier: backend
        role: master
```

and

```
     labels:
        app: guestbook
        tier: backend
        role: slave
```

The labels allow us to slice and dice our resources along any dimension specified by a label:

```
kubectl apply -f examples/guestbook/all-in-one/guestbook-all-in-
one.yaml
kubectl get pods -Lapp -Ltier -Lrole
```

```
NAME                           READY     STATUS    RESTARTS
AGE       APP         TIER       ROLE
guestbook-fe-4nlpb             1/1       Running   0
1m        guestbook   frontend   <none>
guestbook-fe-ght6d             1/1       Running   0
1m        guestbook   frontend   <none>
guestbook-fe-jpy62             1/1       Running   0
1m        guestbook   frontend   <none>
guestbook-redis-master-5pg3b   1/1       Running   0
1m        guestbook   backend    master
guestbook-redis-slave-2q2yf    1/1       Running   0
1m        guestbook   backend    slave
guestbook-redis-slave-qgazl    1/1       Running   0
1m        guestbook   backend    slave
my-nginx-divi2                 1/1       Running   0
29m       nginx       <none>     <none>
my-nginx-o0ef1                 1/1       Running   0
29m       nginx       <none>     <none>
```

```
kubectl get pods -lapp=guestbook,role=slave
```

```
NAME                          READY     STATUS    RESTARTS   AGE
guestbook-redis-slave-2q2yf   1/1       Running   0          3m
guestbook-redis-slave-qgazl   1/1       Running   0          3m
```

Canary deployments

Another scenario where multiple labels are needed is to distinguish deployments of different releases or configurations of the same component. It is common practice to deploy a *canary* of a new application release (specified via image tag in the pod template) side by side with the previous release so that the new release can receive live production traffic before fully rolling it out.

For instance, you can use a `track` label to differentiate different releases.

The primary, stable release would have a `track` label with value as `stable`:

```
     name: frontend
     replicas: 3
     ...
     labels:
        app: guestbook
        tier: frontend
        track: stable
     ...
     image: gb-frontend:v3
```

and then you can create a new release of the guestbook frontend that carries the `track` label with different value (i.e. `canary`), so that two sets of pods would not overlap:

```
     name: frontend-canary
     replicas: 1
     ...
     labels:
        app: guestbook
        tier: frontend
        track: canary
     ...
     image: gb-frontend:v4
```

The frontend service would span both sets of replicas by selecting the common subset of their labels (i.e. omitting the `track` label), so that the traffic will be redirected to both applications:

```
  selector:
     app: guestbook
     tier: frontend
```

You can tweak the number of replicas of the stable and canary releases to determine the ratio of each release that will receive live production traffic (in this case, 3:1). Once you're confident, you can update the stable track to the new application release and remove the canary one.

For a more concrete example, check the tutorial of deploying Ghost.

Updating labels

Sometimes existing pods and other resources need to be relabeled before creating new resources. This can be done with `kubectl label`. For example, if you want to label all your nginx pods as frontend tier, simply run:

```
kubectl label pods -l app=nginx tier=fe
```

```
pod/my-nginx-2035384211-j5fhi labeled
pod/my-nginx-2035384211-u2c7e labeled
pod/my-nginx-2035384211-u3t6x labeled
```

This first filters all pods with the label "app=nginx", and then labels them with the "tier=fe". To see the pods you just labeled, run:

```
kubectl get pods -l app=nginx -L tier
```

```
NAME                        READY     STATUS    RESTARTS
AGE       TIER
my-nginx-2035384211-j5fhi   1/1       Running   0
23m       fe
my-nginx-2035384211-u2c7e   1/1       Running   0
23m       fe
```

```
my-nginx-2035384211-u3t6x   1/1       Running   0
23m       fe
```

This outputs all "app=nginx" pods, with an additional label column of pods' tier (specified with `-L` or `--label-columns`).

For more information, please see labels and kubectl label.

Updating annotations

Sometimes you would want to attach annotations to resources. Annotations are arbitrary non-identifying metadata for retrieval by API clients such as tools, libraries, etc. This can be done with `kubectl annotate`. For example:

```
kubectl annotate pods my-nginx-v4-9gw19 description='my frontend
running nginx'
kubectl get pods my-nginx-v4-9gw19 -o yaml
```

```
apiVersion: v1
kind: pod
metadata:
  annotations:
    description: my frontend running nginx
...
```

For more information, please see annotations and kubectl annotate document.

Scaling your application

When load on your application grows or shrinks, it's easy to scale with `kubectl`. For instance, to decrease the number of nginx replicas from 3 to 1, do:

```
kubectl scale deployment/my-nginx --replicas=1
```

```
deployment.extensions/my-nginx scaled
```

Now you only have one pod managed by the deployment.

```
kubectl get pods -l app=nginx
```

```
NAME                        READY     STATUS    RESTARTS   AGE
my-nginx-2035384211-j5fhi   1/1       Running   0          30m
```

To have the system automatically choose the number of nginx replicas as needed, ranging from 1 to 3, do:

```
kubectl autoscale deployment/my-nginx --min=1 --max=3
```

```
horizontalpodautoscaler.autoscaling/my-nginx autoscaled
```

Now your nginx replicas will be scaled up and down as needed, automatically.

For more information, please see kubectl scale, kubectl autoscale and horizontal pod autoscaler document.

In-place updates of resources

Sometimes it's necessary to make narrow, non-disruptive updates to resources you've created.

kubectl apply

It is suggested to maintain a set of configuration files in source control (see configuration as code), so that they can be maintained and versioned along with the code for the resources they configure. Then, you can use `kubectl apply` to push your configuration changes to the cluster.

This command will compare the version of the configuration that you're pushing with the previous version and apply the changes you've made, without overwriting any automated changes to properties you haven't specified.

```
kubectl apply -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml
deployment.apps/my-nginx configured
```

Note that `kubectl apply` attaches an annotation to the resource in order to determine the changes to the configuration since the previous invocation. When it's invoked, `kubectl apply` does a three-way diff between the previous configuration, the provided input and the current configuration of the resource, in order to determine how to modify the resource.

Currently, resources are created without this annotation, so the first invocation of `kubectl apply` will fall back to a two-way diff between the provided input and the current configuration of the resource. During this first invocation, it cannot detect the deletion of properties set when the resource was created. For this reason, it will not remove them.

All subsequent calls to `kubectl apply`, and other commands that modify the configuration, such as `kubectl replace` and `kubectl edit`, will update the annotation, allowing subsequent calls to `kubectl apply` to detect and perform deletions using a three-way diff.

kubectl edit

Alternatively, you may also update resources with `kubectl edit`:

```
kubectl edit deployment/my-nginx
```

This is equivalent to first `get` the resource, edit it in text editor, and then `apply` the resource with the updated version:

```
kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml
vi /tmp/nginx.yaml
# do some edit, and then save the file

kubectl apply -f /tmp/nginx.yaml
deployment.apps/my-nginx configured

rm /tmp/nginx.yaml
```

This allows you to do more significant changes more easily. Note that you can specify the editor with your `EDITOR` or `KUBE_EDITOR` environment variables.

For more information, please see kubectl edit document.

kubectl patch

You can use `kubectl patch` to update API objects in place. This command supports JSON patch, JSON merge patch, and strategic merge patch. See Update API Objects in Place Using kubectl patch and kubectl patch.

Disruptive updates

In some cases, you may need to update resource fields that cannot be updated once initialized, or you may just want to make a recursive change immediately, such as to fix broken pods created by a Deployment. To change such fields, use `replace --force`, which deletes and re-creates the resource. In this case, you can simply modify your original configuration file:

```
kubectl replace -f https://k8s.io/examples/application/nginx/
nginx-deployment.yaml --force
```

```
deployment.apps/my-nginx deleted
deployment.apps/my-nginx replaced
```

Updating your application without a service outage

At some point, you'll eventually need to update your deployed application, typically by specifying a new image or image tag, as in the canary deployment scenario above. `kubectl` supports several update operations, each of which is applicable to different scenarios.

We'll guide you through how to create and update applications with Deployments.

Let's say you were running version 1.7.9 of nginx:

```
kubectl run my-nginx --image=nginx:1.7.9 --replicas=3
```

```
deployment.apps/my-nginx created
```

To update to version 1.9.1, simply change `.spec.template.spec.containers[0].image` from `nginx:1.7.9` to `nginx:1.9.1`, with the kubectl commands we learned above.

```
kubectl edit deployment/my-nginx
```

That's it! The Deployment will declaratively update the deployed nginx application progressively behind the scene. It ensures that only a certain number of old replicas may be down while they are being updated, and only a certain number of new replicas may be created above the desired number of pods. To learn more details about it, visit Deployment page.

What's next

- Learn about how to use `kubectl` for application introspection and debugging.
- Configuration Best Practices and Tips

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on October 14, 2019 at 6:51 PM PST by Missing "Manifest" from glossary k8s.io/docs/reference/glossary/ #13546 (#16193) (Page History)

Edit This Page

Cluster Networking

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

1. Highly-coupled container-to-container communications: this is solved by pods and `localhost` communications.
2. Pod-to-Pod communications: this is the primary focus of this document.

3. Pod-to-Service communications: this is covered by services.
4. External-to-Service communications: this is covered by services.

- The Kubernetes network model
- How to implement the Kubernetes networking model
- What's next

Kubernetes is all about sharing machines between applications. Typically, sharing machines requires ensuring that two applications do not try to use the same ports. Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control.

Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

The Kubernetes network model

Every `Pod` gets its own IP address. This means you do not need to explicitly create links between `Pods` and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where `Pods` can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- pods on a node can communicate with all pods on all nodes without NAT
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Note: For those platforms that support `Pods` running in the host network (e.g. Linux):

- pods in the host network of a node can communicate with all pods on all nodes without NAT

This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

Kubernetes IP addresses exist at the `Pod` scope - containers within a `Pod` share their network namespaces - including their IP address. This means that containers within a `Pod` can all reach each other's ports on `localhost`. This also means that containers within a `Pod` must coordinate port usage, but this is no different from processes in a VM. This is called the "IP-per-pod" model.

How this is implemented is a detail of the particular container runtime in use.

It is possible to request ports on the `Node` itself which forward to your `Pod` (called host ports), but this is a very niche operation. How that forwarding is implemented is also a detail of the container runtime. The `Pod` itself is blind to the existence or non-existence of host ports.

How to implement the Kubernetes networking model

There are a number of ways that this network model can be implemented. This document is not an exhaustive study of the various methods, but hopefully serves as an introduction to various technologies and serves as a jumping-off point.

The following networking options are sorted alphabetically - the order does not imply any preferential status.

ACI

Cisco Application Centric Infrastructure offers an integrated overlay and underlay SDN solution that supports containers, virtual machines, and bare metal servers. ACI provides container networking integration for ACI. An overview of the integration is provided here.

Antrea

Project Antrea is an opensource Kubernetes networking solution intended to be Kubernetes native. It leverages Open vSwitch as the networking data plane. Open vSwitch is a high-performance programmable virtual switch that supports both Linux and Windows. Open vSwitch enables Antrea to implement Kubernetes Network Policies in a high-performance and efficient manner. Thanks to the "programmable" characteristic of Open vSwitch, Antrea is able to implement an extensive set of networking and security features and services on top of Open vSwitch.

AOS from Apstra

AOS is an Intent-Based Networking system that creates and manages complex datacenter environments from a simple integrated platform. AOS leverages a highly scalable distributed design to eliminate network outages while minimizing costs.

The AOS Reference Design currently supports Layer-3 connected hosts that eliminate legacy Layer-2 switching problems. These Layer-3 hosts can be Linux servers (Debian, Ubuntu, CentOS) that create BGP neighbor relationships directly with the top of rack switches (TORs). AOS automates the routing adjacencies and then provides fine grained control over the route health injections (RHI) that are common in a Kubernetes deployment.

AOS has a rich set of REST API endpoints that enable Kubernetes to quickly change the network policy based on application requirements. Further enhancements will integrate the AOS Graph model used for the network design with the workload provisioning, enabling an end to end management system for both private and public clouds.

AOS supports the use of common vendor equipment from manufacturers including Cisco, Arista, Dell, Mellanox, HPE, and a large number of white-box systems and open network operating systems like Microsoft SONiC, Dell OPX, and Cumulus Linux.

Details on how the AOS system works can be accessed here: http://www.apstra.com/products/how-it-works/

AWS VPC CNI for Kubernetes

The AWS VPC CNI offers integrated AWS Virtual Private Cloud (VPC) networking for Kubernetes clusters. This CNI plugin offers high throughput and availability, low latency, and minimal network jitter. Additionally, users can apply existing AWS VPC networking and security best practices for building Kubernetes clusters. This includes the ability to use VPC flow logs, VPC routing policies, and security groups for network traffic isolation.

Using this CNI plugin allows Kubernetes pods to have the same IP address inside the pod as they do on the VPC network. The CNI allocates AWS Elastic Networking Interfaces (ENIs) to each Kubernetes node and using the secondary IP range from each ENI for pods on the node. The CNI includes controls for pre-allocation of ENIs and IP addresses for fast pod startup times and enables large clusters of up to 2,000 nodes.

Additionally, the CNI can be run alongside Calico for network policy enforcement. The AWS VPC CNI project is open source with documentation on GitHub.

Azure CNI for Kubernetes

Azure CNI is an open source plugin that integrates Kubernetes Pods with an Azure Virtual Network (also known as VNet) providing network performance at par with VMs. Pods can connect to peered VNet and to on-premises over Express Route or site-to-site VPN and are also directly reachable from these networks. Pods can access Azure services, such as storage and SQL, that are protected by Service Endpoints or Private Link. You can use VNet security policies and routing to filter Pod traffic. The plugin assigns VNet IPs to Pods by utilizing a pool of secondary IPs pre-configured on the Network Interface of a Kubernetes node.

Azure CNI is available natively in the Azure Kubernetes Service (AKS).

Big Cloud Fabric from Big Switch Networks

Big Cloud Fabric is a cloud native networking architecture, designed to run Kubernetes in private cloud/on-premises environments. Using unified physical & virtual SDN, Big Cloud Fabric tackles inherent container networking problems such as load balancing, visibility, troubleshooting, security policies & container traffic monitoring.

With the help of the Big Cloud Fabric's virtual pod multi-tenant architecture, container orchestration systems such as Kubernetes, RedHat OpenShift, Mesosphere DC/OS & Docker Swarm will be natively integrated alongside with VM orchestration systems such as VMware, OpenStack & Nutanix. Customers will be able to securely inter-connect any number of these clusters and enable inter-tenant communication between them if needed.

BCF was recognized by Gartner as a visionary in the latest Magic Quadrant. One of the BCF Kubernetes on-premises deployments (which includes Kubernetes, DC/OS & VMware running on multiple DCs across different geographic regions) is also referenced here.

Cilium

Cilium is open source software for providing and transparently securing network connectivity between application containers. Cilium is L7/HTTP aware and can enforce network policies on L3-L7 using an identity based security model that is decoupled from network addressing, and it can be used in combination with other CNI plugins.

CNI-Genie from Huawei

CNI-Genie is a CNI plugin that enables Kubernetes to simultaneously have access to different implementations of the Kubernetes network model in runtime. This includes any implementation that runs as a CNI plugin, such as Flannel, Calico, Romana, Weave-net.

CNI-Genie also supports assigning multiple IP addresses to a pod, each from a different CNI plugin.

cni-ipvlan-vpc-k8s

cni-ipvlan-vpc-k8s contains a set of CNI and IPAM plugins to provide a simple, host-local, low latency, high throughput, and compliant networking stack for Kubernetes within Amazon Virtual Private Cloud (VPC) environments by making use of Amazon Elastic Network Interfaces (ENI) and binding AWS-managed IPs into Pods using the Linux kernel's IPvlan driver in L2 mode.

The plugins are designed to be straightforward to configure and deploy within a VPC. Kubelets boot and then self-configure and scale their IP usage as needed without requiring the often recommended complexities of administering overlay networks, BGP, disabling source/destination checks, or adjusting VPC route tables to provide per-instance subnets to each host (which is limited to 50-100 entries per VPC). In short, cni-ipvlan-vpc-k8s significantly reduces the network complexity required to deploy Kubernetes at scale within AWS.

Contiv

Contiv provides configurable networking (native l3 using BGP, overlay using vxlan, classic l2, or Cisco-SDN/ACI) for various use cases. Contiv is all open sourced.

Contrail / Tungsten Fabric

Contrail, based on Tungsten Fabric, is a truly open, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with various orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide different isolation modes for virtual machines, containers/pods and bare metal workloads.

DANM

DANM is a networking solution for telco workloads running in a Kubernetes cluster. It's built up from the following components:

- A CNI plugin capable of provisioning IPVLAN interfaces with advanced features
- An in-built IPAM module with the capability of managing multiple, cluster-wide, discontinuous L3 networks and provide a dynamic, static, or no IP allocation scheme on-demand
- A CNI metaplugin capable of attaching multiple network interfaces to a container, either through its own CNI, or through delegating the job to any of the popular CNI solution like SRI-OV, or Flannel in parallel
- A Kubernetes controller capable of centrally managing both VxLAN and VLAN interfaces of all Kubernetes hosts
- Another Kubernetes controller extending Kubernetes' Service-based service discovery concept to work over all network interfaces of a Pod

With this toolset DANM is able to provide multiple separated network interfaces, the possibility to use different networking back ends and advanced IPAM features for the pods.

Flannel

Flannel is a very simple overlay network that satisfies the Kubernetes requirements. Many people have reported success with Flannel and Kubernetes.

Google Compute Engine (GCE)

For the Google Compute Engine cluster configuration scripts, advanced routing is used to assign each VM a subnet (default is `/24` - 254 IPs). Any traffic bound for that subnet will be routed directly to the VM by the GCE network fabric. This is in addition to the "main" IP address assigned to the VM, which is NAT'ed for outbound internet access. A linux bridge (called `cbr0`) is configured to exist on that subnet, and is passed to docker's `--bridge` flag.

Docker is started with:

```
DOCKER_OPTS="--bridge=cbr0 --iptables=false --ip-masq=false"
```

This bridge is created by Kubelet (controlled by the `--network-plugin=kubenet` flag) according to the `Node`'s `.spec.podCIDR`.

Docker will now allocate IPs from the `cbr-cidr` block. Containers can reach each other and `Nodes` over the `cbr0` bridge. Those IPs are all routable within the GCE project network.

GCE itself does not know anything about these IPs, though, so it will not NAT them for outbound internet traffic. To achieve that an iptables rule is used to masquerade (aka SNAT - to make it seem as if packets came from the `Node` itself) traffic that is bound for IPs outside the GCE project network (10.0.0.0/8).

```
iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o eth0 -j MASQUERADE
```

Lastly IP forwarding is enabled in the kernel (so the kernel will process packets for bridged containers):

```
sysctl net.ipv4.ip_forward=1
```

The result of all this is that all `Pods` can reach each other and can egress traffic to the internet.

Jaguar

Jaguar is an open source solution for Kubernetes's network based on OpenDaylight. Jaguar provides overlay network using vxlan and Jaguar CNIPlugin provides one IP address per pod.

k-vswitch

k-vswitch is a simple Kubernetes networking plugin based on Open vSwitch. It leverages existing functionality in Open vSwitch to provide a robust networking plugin that is easy-to-operate, performant and secure.

Knitter

Knitter is a network solution which supports multiple networking in Kubernetes. It provides the ability of tenant management and network management. Knitter includes a set of end-to-end NFV container networking solutions besides multiple network planes, such as keeping IP address for applications, IP address migration, etc.

Kube-OVN

Kube-OVN is an OVN-based kubernetes network fabric for enterprises. With the help of OVN/OVS, it provides some advanced overlay network features like subnet, QoS, static IP allocation, traffic mirroring, gateway, openflow-based network policy and service proxy.

Kube-router

Kube-router is a purpose-built networking solution for Kubernetes that aims to provide high performance and operational simplicity. Kube-router provides a Linux LVS/IPVS-based service proxy, a Linux kernel forwarding-based pod-to-pod networking solution with no overlays, and iptables/ipset-based network policy enforcer.

L2 networks and linux bridging

If you have a "dumb" L2 network, such as a simple switch in a "bare-metal" environment, you should be able to do something similar to the above GCE setup. Note that these instructions have only been tried very casually - it seems to work, but has not been thoroughly tested. If you use this technique and perfect the process, please let us know.

Follow the "With Linux Bridge devices" section of this very nice tutorial from Lars Kellogg-Stedman.

Multus (a Multi Network plugin)

Multus is a Multi CNI plugin to support the Multi Networking feature in Kubernetes using CRD based network objects in Kubernetes.

Multus supports all reference plugins (eg. Flannel, DHCP, Macvlan) that implement the CNI specification and 3rd party plugins (eg. Calico, Weave, Cilium, Contiv). In addition to it, Multus supports SRIOV, DPDK, OVS-DPDK & VPP workloads in Kubernetes with both cloud native and NFV based applications in Kubernetes.

NSX-T

VMware NSX-T is a network virtualization and security platform. NSX-T can provide network virtualization for a multi-cloud and multi-hypervisor environment and is focused on emerging application frameworks and architectures that have heterogeneous endpoints and technology stacks. In addition to vSphere hypervisors, these environments include other hypervisors such as KVM, containers, and bare metal.

NSX-T Container Plug-in (NCP) provides integration between NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.

Nuage Networks VCS (Virtualized Cloud Services)

Nuage provides a highly scalable policy-based Software-Defined Networking (SDN) platform. Nuage uses the open source Open vSwitch for the data plane along with a feature rich SDN Controller built on open standards.

The Nuage platform uses overlays to provide seamless policy-based networking between Kubernetes Pods and non-Kubernetes environments (VMs and bare metal servers). Nuage's policy abstraction model is designed with applications in mind and makes it easy to declare fine-grained policies for applications.The platform's real-time analytics engine enables visibility and security monitoring for Kubernetes applications.

OpenVSwitch

OpenVSwitch is a somewhat more mature but also complicated way to build an overlay network. This is endorsed by several of the "Big Shops" for networking.

OVN (Open Virtual Networking)

OVN is an opensource network virtualization solution developed by the Open vSwitch community. It lets one create logical switches, logical routers, stateful ACLs, load-balancers etc to build different virtual networking topologies. The project has a specific Kubernetes plugin and documentation at ovn-kubernetes.

Project Calico

Project Calico is an open source container networking provider and network policy engine.

Calico provides a highly scalable networking and network policy solution for connecting Kubernetes pods based on the same IP networking principles as the internet, for both Linux (open source) and Windows (proprietary - available from Tigera). Calico can be deployed without encapsulation or overlays to provide high-performance, high-scale data center networking. Calico also provides fine-grained, intent based network security policy for Kubernetes pods via its distributed firewall.

Calico can also be run in policy enforcement mode in conjunction with other networking solutions such as Flannel, aka canal, or native GCE, AWS or Azure networking.

Romana

Romana is an open source network and security automation solution that lets you deploy Kubernetes without an overlay network. Romana supports Kubernetes Network Policy to provide isolation across network namespaces.

Weave Net from Weaveworks

Weave Net is a resilient and simple to use network for Kubernetes and its hosted applications. Weave Net runs as a CNI plug-in or stand-alone. In either version, it doesn't require any configuration or extra code to run, and in both cases, the network provides one IP address per pod - as is standard for Kubernetes.

What's next

The early design of the networking model and its rationale, and some future plans are described in more detail in the networking design document.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on January 18, 2020 at 8:39 AM PST by Added a new section on Azure CNI (#17830) (Page History)

Edit This Page

Logging Architecture

Application and systems logs can help you understand what is happening inside your cluster. The logs are particularly useful for debugging problems and monitoring cluster activity. Most modern applications have some kind of logging mechanism; as such, most container engines are likewise designed to support some kind of logging. The easiest and most embraced logging method for containerized applications is to write to the standard output and standard error streams.

However, the native functionality provided by a container engine or runtime is usually not enough for a complete logging solution. For example, if a container crashes, a pod is evicted, or a node dies, you'll usually still want to access your application's logs. As such, logs should have a separate storage and lifecycle independent of nodes, pods, or containers. This concept is called *cluster-level-logging*. Cluster-level logging requires a separate backend to store, analyze, and query logs. Kubernetes provides no native storage solution for log data, but you can integrate many existing logging solutions into your Kubernetes cluster.

- Basic logging in Kubernetes
- Logging at the node level
- Cluster-level logging architectures

Cluster-level logging architectures are described in assumption that a logging backend is present inside or outside of your cluster. If you're not interested in having cluster-level logging, you might still find the description of how logs are stored and handled on the node to be useful.

Basic logging in Kubernetes

In this section, you can see an example of basic logging in Kubernetes that outputs data to the standard output stream. This demonstration uses a pod specification with a container that writes some text to standard output once per second.

debug/counter-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
            'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

To run this pod, use the following command:

```
kubectl apply -f https://k8s.io/examples/debug/counter-pod.yaml
```

The output is:

```
pod/counter created
```

To fetch the logs, use the `kubectl logs` command, as follows:

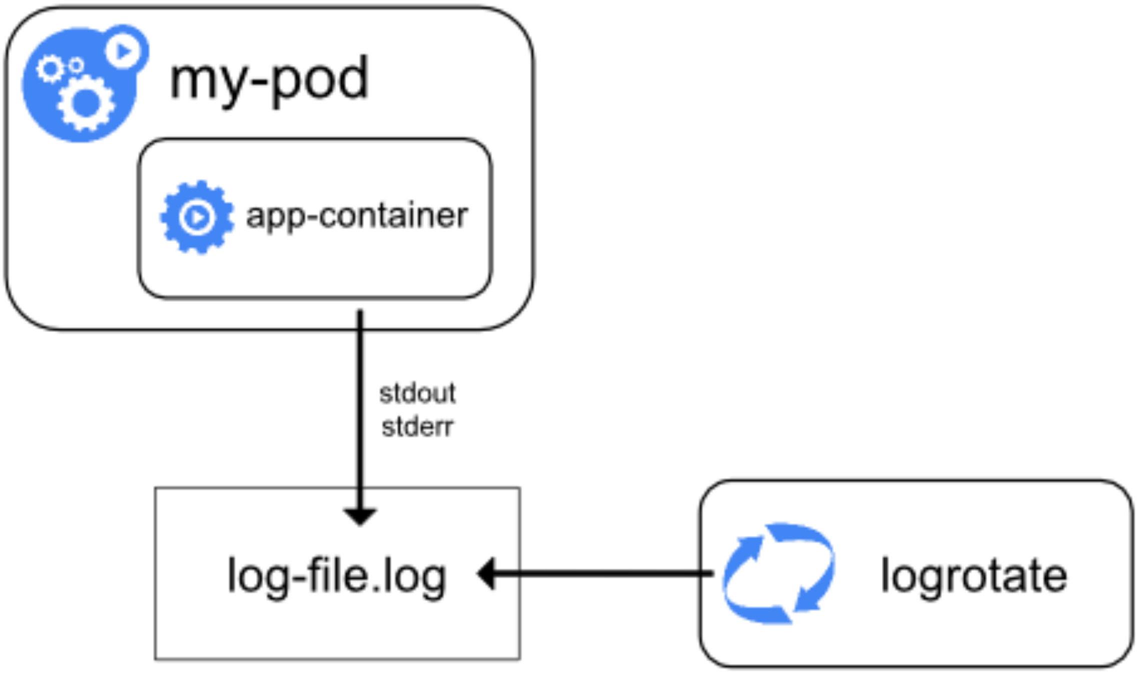
```
kubectl logs counter
```

The output is:

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

You can use `kubectl logs` to retrieve logs from a previous instantiation of a container with `--previous` flag, in case the container has crashed. If your pod has multiple containers, you should specify which container's logs you want to access by appending a container name to the command. See the [`kubectl logs` documentation] for more details.

Logging at the node level



Everything a containerized application writes to `stdout` and `stderr` is handled and redirected somewhere by a container engine. For example, the Docker container engine redirects those two streams to a logging driver, which is configured in Kubernetes to write to a file in json format.

> **Note:** The Docker json logging driver treats each line as a separate message. When using the Docker logging driver, there is no direct support for multi-line messages. You need to handle multi-line messages at the logging agent level or higher.

By default, if a container restarts, the kubelet keeps one terminated container with its logs. If a pod is evicted from the node, all corresponding containers are also evicted, along with their logs.

An important consideration in node-level logging is implementing log rotation, so that logs don't consume all available storage on the node. Kubernetes currently is not responsible for rotating logs, but rather a deployment tool should set up a solution to address that. For example, in Kubernetes clusters, deployed by the `kube-up.sh` script, there is a `logrotate` tool configured to run each hour. You can also set up a container runtime to rotate application's logs automatically, e.g. by using Docker's `log-opt`. In the `kube-up.sh` script, the latter approach is used for COS image on GCP, and the former approach is used in any other environment. In both cases, by default rotation is configured to take place when log file exceeds 10MB.

As an example, you can find detailed information about how `kube-up.sh` sets up logging for COS image on GCP in the corresponding script.

When you run `kubectl logs` as in the basic logging example, the kubelet on the node handles the request and reads directly from the log file, returning the contents in the response.

> **Note:** Currently, if some external system has performed the rotation, only the contents of the latest log file will be available through `kubectl logs`. E.g. if there's a 10MB file, `logrotate` performs the rotation and there are two files, one 10MB in size and one empty, `kubectl logs` will return an empty response.

System component logs

There are two types of system components: those that run in a container and those that do not run in a container. For example:

- The Kubernetes scheduler and kube-proxy run in a container.
- The kubelet and container runtime, for example Docker, do not run in containers.

On machines with systemd, the kubelet and container runtime write to journald. If systemd is not present, they write to `.log` files in the `/var/log` directory. System components inside containers always write to the `/var/log` directory, bypassing the default logging mechanism. They use the klog logging library. You can find the conventions for logging severity for those components in the development docs on logging.

Similarly to the container logs, system component logs in the `/var/log` directory should be rotated. In Kubernetes clusters brought up by the `kube-up.sh` script, those logs are configured to be rotated by the `logrotate` tool daily or once the size exceeds 100MB.

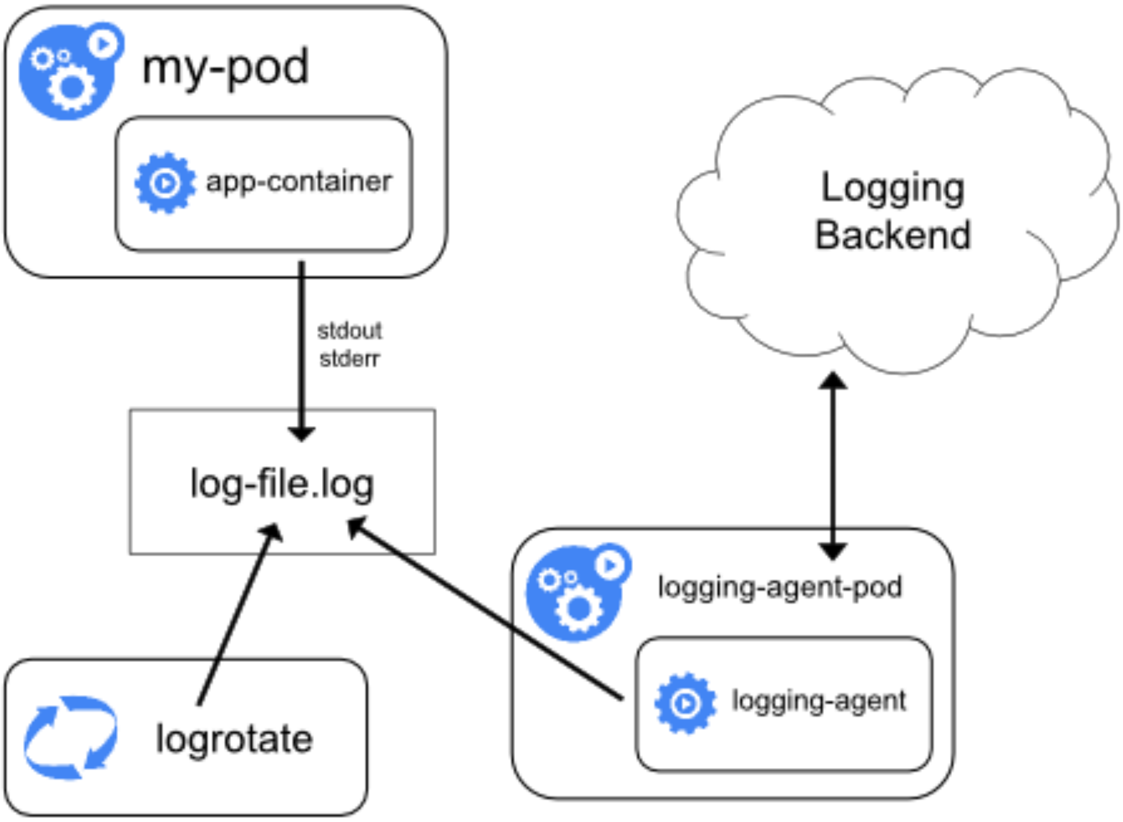
Cluster-level logging architectures

While Kubernetes does not provide a native solution for cluster-level logging, there are several common approaches you can consider. Here are some options:

- Use a node-level logging agent that runs on every node.

- Include a dedicated sidecar container for logging in an application pod.
- Push logs directly to a backend from within an application.

Using a node logging agent



You can implement cluster-level logging by including a *node-level logging agent* on each node. The logging agent is a dedicated tool that exposes logs or pushes logs to a backend. Commonly, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Because the logging agent must run on every node, it's common to implement it as either a DaemonSet replica, a manifest pod, or a dedicated native process on the node. However the latter two approaches are deprecated and highly discouraged.

Using a node-level logging agent is the most common and encouraged approach for a Kubernetes cluster, because it creates only one agent per node, and it doesn't require any changes to the applications running on the node. However, node-level logging *only works for applications' standard output and standard error*.

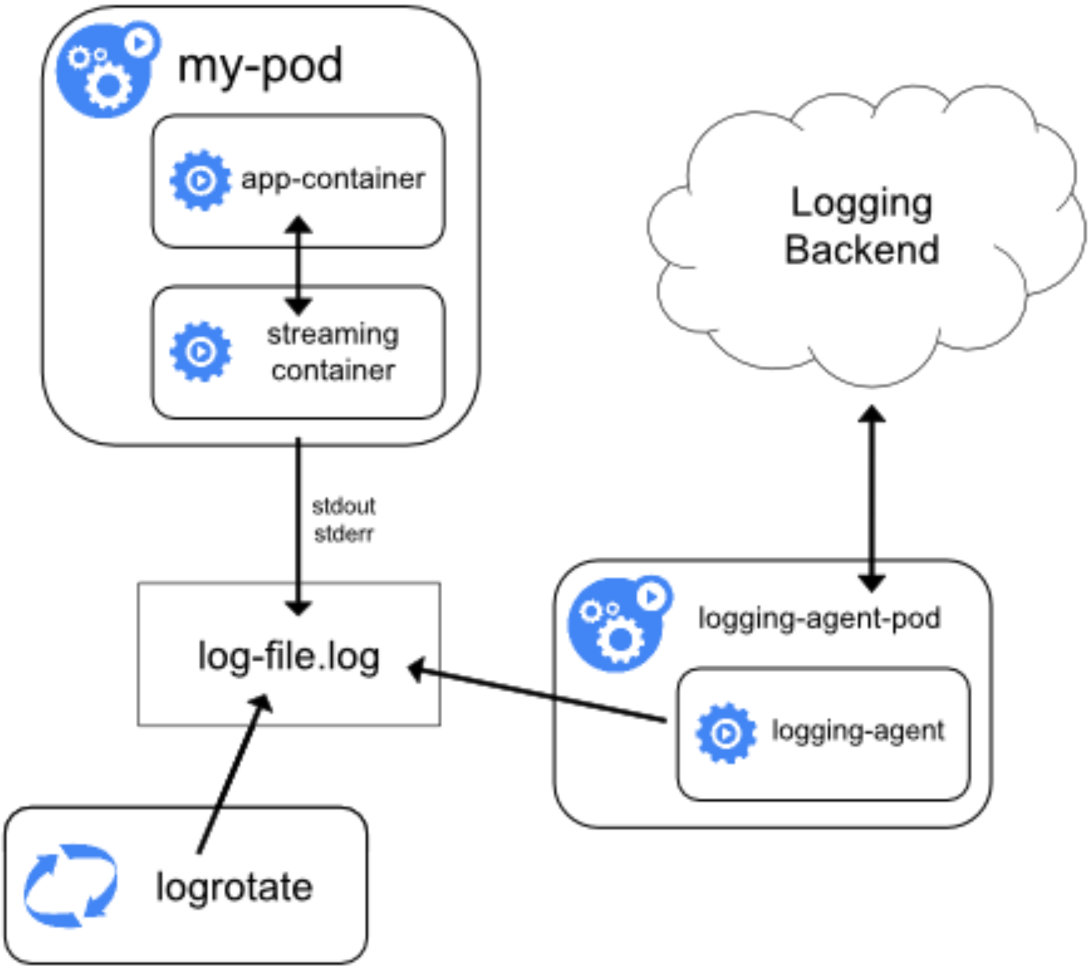
Kubernetes doesn't specify a logging agent, but two optional logging agents are packaged with the Kubernetes release: Stackdriver Logging for use with Google Cloud Platform, and Elasticsearch. You can find more information and instructions in the dedicated documents. Both use fluentd with custom configuration as an agent on the node.

Using a sidecar container with the logging agent

You can use a sidecar container in one of the following ways:

- The sidecar container streams application logs to its own `stdout`.
- The sidecar container runs a logging agent, which is configured to pick up logs from an application container.

Streaming sidecar container



By having your sidecar containers stream to their own `stdout` and `stderr` streams, you can take advantage of the kubelet and the logging agent that already run on each node. The sidecar containers read logs from a file, a socket, or the journald. Each individual sidecar container prints log to its own `stdout` or `stderr` stream.

This approach allows you to separate several log streams from different parts of your application, some of which can lack support for writing to `stdout` or `stderr`. The logic behind redirecting logs is minimal, so it's hardly a significant overhead. Additionally, because `stdout` and `stderr` are handled by the kubelet, you can use built-in tools like `kubectl logs`.

Consider the following example. A pod runs a single container, and the container writes to two different log files, using two different formats. Here's a configuration file for the Pod:

admin/logging/two-files-counter-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

It would be a mess to have log entries of different formats in the same log stream, even if you managed to redirect both components to the `stdout` stream of the container. Instead, you could introduce two sidecar containers. Each sidecar container could tail a particular log file from a shared volume and then redirect the logs to its own `stdout` stream.

Here's a configuration file for a pod that has two sidecar containers:

admin/logging/two-files-counter-pod-streaming-sidecar.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-1
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-2
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

Now when you run this pod, you can access each log stream separately by running the following commands:

```
kubectl logs counter count-log-1
```

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

```
kubectl logs counter count-log-2
```

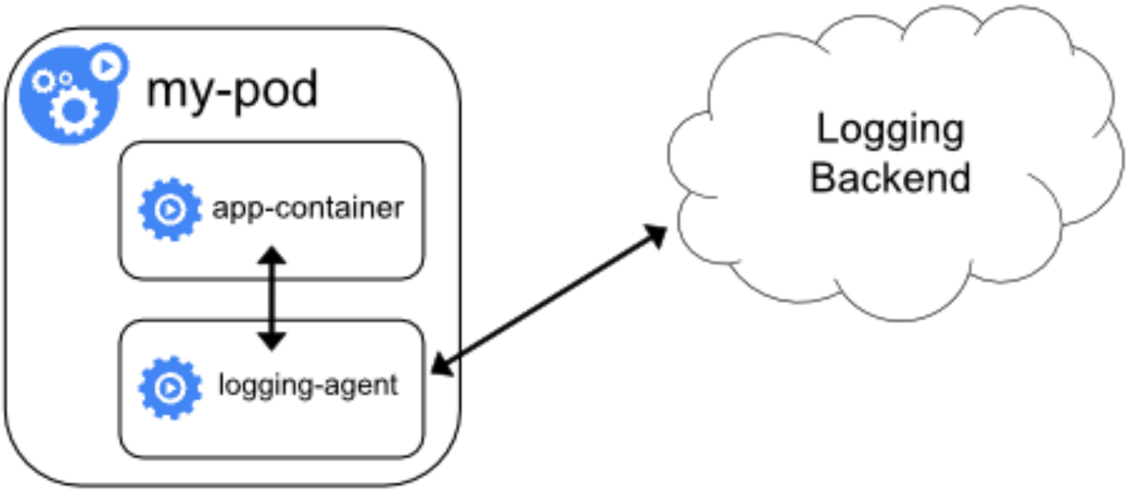
```
Mon Jan  1 00:00:00 UTC 2001 INFO 0
Mon Jan  1 00:00:01 UTC 2001 INFO 1
Mon Jan  1 00:00:02 UTC 2001 INFO 2
...
```

The node-level agent installed in your cluster picks up those log streams automatically without any further configuration. If you like, you can configure the agent to parse log lines depending on the source container.

Note, that despite low CPU and memory usage (order of couple of millicores for cpu and order of several megabytes for memory), writing logs to a file and then streaming them to `stdout` can double disk usage. If you have an application that writes to a single file, it's generally better to set `/dev/stdout` as destination rather than implementing the streaming sidecar container approach.

Sidecar containers can also be used to rotate log files that cannot be rotated by the application itself. An example of this approach is a small container running logrotate periodically. However, it's recommended to use `stdout` and `stderr` directly and leave rotation and retention policies to the kubelet.

Sidecar container with a logging agent



If the node-level logging agent is not flexible enough for your situation, you can create a sidecar container with a separate logging agent that you have configured specifically to run with your application.

> **Note:** Using a logging agent in a sidecar container can lead to significant resource consumption. Moreover, you won't be able to access those logs using `kubectl logs` command, because they are not controlled by the kubelet.

As an example, you could use Stackdriver, which uses fluentd as a logging agent. Here are two configuration files that you can use to implement this approach. The first file contains a ConfigMap to configure fluentd.

admin/logging/fluentd-sidecar-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluentd-config
data:
  fluentd.conf: |
    <source>
      type tail
      format none
      path /var/log/1.log
      pos_file /var/log/1.log.pos
      tag count.format1
    </source>

    <source>
      type tail
      format none
      path /var/log/2.log
      pos_file /var/log/2.log.pos
      tag count.format2
    </source>

    <match **>
      type google_cloud
    </match>
```

> **Note:** The configuration of fluentd is beyond the scope of this article. For information about configuring fluentd, see the official fluentd documentation.

The second file describes a pod that has a sidecar container running fluentd. The pod mounts a volume where fluentd can pick up its configuration data.

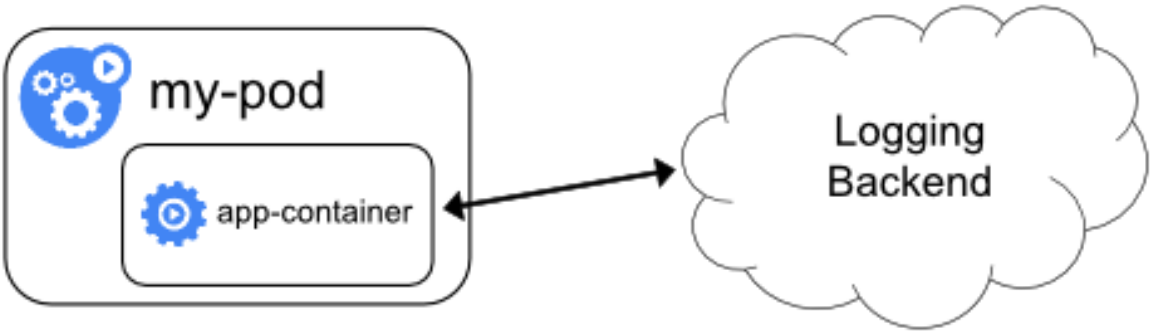
admin/logging/two-files-counter-pod-agent-sidecar.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-agent
    image: k8s.gcr.io/fluentd-gcp:1.30
    env:
    - name: FLUENTD_ARGS
      value: -c /etc/fluentd-config/fluentd.conf
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: config-volume
      mountPath: /etc/fluentd-config
  volumes:
  - name: varlog
    emptyDir: {}
  - name: config-volume
    configMap:
      name: fluentd-config
```

After some time you can find log messages in the Stackdriver interface.

Remember, that this is just an example and you can actually replace fluentd with any logging agent, reading from any source inside an application container.

Exposing logs directly from the application



You can implement cluster-level logging by exposing or pushing logs directly from every application; however, the implementation for such a logging mechanism is outside the scope of Kubernetes.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on July 09, 2019 at 1:08 PM PST by Separate commands from output (#15331) (Page History)

Edit This Page

Metrics For The Kubernetes Control Plane

System component metrics can give a better look into what is happening inside them. Metrics are particularly useful for building dashboards and alerts.

Metrics in Kubernetes control plane are emitted in prometheus format and are human readable.

Metrics in Kubernetes

In most cases metrics are available on `/metrics` endpoint of the HTTP server. For components that doesn't expose endpoint by default it can be enabled using `--bind-address` flag.

Examples of those components: * kube-controller-managerControl Plane component that runs controller processes. * kube-proxykube-proxy is a network proxy that runs on each node in the cluster. * kube-apiserverControl plane component that serves the Kubernetes API. * kube-schedulerControl Plane component that watches for newly created pods with no assigned node, and selects a node for them to run on. * kubeletAn agent that runs on each node in the cluster. It makes sure that containers are running in a pod.

In a production environment you may want to configure Prometheus Server or some other metrics scraper to periodically gather these metrics and make them available in some kind of time series database.

Note that kubeletAn agent that runs on each node in the cluster. It makes sure that containers are running in a pod. also exposes metrics in `/metrics/cadvisor`, `/metrics/resource` and `/metrics/probes` endpoints. Those metrics do not have same lifecycle.

If your cluster uses RBACManages authorization decisions, allowing admins to dynamically configure access policies through the Kubernetes API. , reading metrics requires authorization via a user, group or ServiceAccount with a ClusterRole that allows accessing `/metrics`. For example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - nonResourceURLs:
      - "/metrics"
    verbs:
      - get
```

Metric lifecycle

Alpha metric â†’ Stable metric â†’ Deprecated metric â†’ Hidden metric â†’ Deletion

Alpha metrics have no stability guarantees; as such they can be modified or deleted at any time.

Stable metrics can be guaranteed to not change; Specifically, stability means:

- the metric itself will not be deleted (or renamed)
- the type of metric will not be modified

Deprecated metric signal that the metric will eventually be deleted; to find which version, you need to check annotation, which includes from which kubernetes version that metric will be considered deprecated.

Before deprecation:

```
# HELP some_counter this counts things
# TYPE some_counter counter
some_counter 0
```

After deprecation:

```
# HELP some_counter (Deprecated since 1.15.0) this counts things
# TYPE some_counter counter
some_counter 0
```

Once a metric is hidden then by default the metrics is not published for scraping. To use a hidden metric, you need to override the configuration for the relevant cluster component.

Once a metric is deleted, the metric is not published. You cannot change this using an override.

Show Hidden Metrics

As described above, admins can enable hidden metrics through a command-line flag on a specific binary. This intends to be used as an escape hatch for admins if they missed the migration of the metrics deprecated in the last release.

The flag `show-hidden-metrics-for-version` takes a version for which you want to show metrics deprecated in that release. The version is expressed as x.y, where x is the major version, y is the minor version. The patch version is not needed even though a metrics can be deprecated in a patch release, the reason for that is the metrics deprecation policy runs against the minor release.

The flag can only take the previous minor version as it's value. All metrics hidden in previous will be emitted if admins set the previous version to `show-hidden-metrics-for-version`. The too old version is not allowed because this violates the metrics deprecated policy.

Take metric `A` as an example, here assumed that `A` is deprecated in 1.n. According to metrics deprecated policy, we can reach the following conclusion:

- In release `1.n`, the metric is deprecated, and it can be emitted by default.
- In release `1.n+1`, the metric is hidden by default and it can be emitted by command line `show-hidden-metrics-for-version=1.n`.
- In release `1.n+2`, the metric should be removed from the codebase. No escape hatch anymore.

If you're upgrading from release `1.12` to `1.13`, but still depend on a metric `A` deprecated in `1.12`, you should set hidden metrics via command line: `--show-hidden-metrics=1.12` and remember to remove this metric dependency before upgrading to `1.14`

Component metrics

kube-controller-manager metrics

Controller manager metrics provide important insight into the performance and health of the controller manager. These metrics include common Go language runtime metrics such as go_routine count and controller specific metrics such as etcd request latencies or Cloudprovider (AWS, GCE, OpenStack) API latencies that can be used to gauge the health of a cluster.

Starting from Kubernetes 1.7, detailed Cloudprovider metrics are available for storage operations for GCE, AWS, Vsphere and OpenStack. These metrics can be used to monitor health of persistent volume operations.

For example, for GCE these metrics are called:

```
cloudprovider_gce_api_request_duration_seconds { request =
"instance_list"}
cloudprovider_gce_api_request_duration_seconds { request =
"disk_insert"}
cloudprovider_gce_api_request_duration_seconds { request =
"disk_delete"}
cloudprovider_gce_api_request_duration_seconds { request =
"attach_disk"}
cloudprovider_gce_api_request_duration_seconds { request =
"detach_disk"}
cloudprovider_gce_api_request_duration_seconds { request =
"list_disk"}
```

What's next

- Read about the Prometheus text format for metrics
- See the list of stable Kubernetes metrics
- Read about the Kubernetes deprecation policy

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 07, 2020 at 7:01 PM PST by Document control plane monitoring (#17578) (Page History)

Edit This Page

Configuring kubelet Garbage Collection

Garbage collection is a helpful function of kubelet that will clean up unused images and unused containers. Kubelet will perform garbage collection for containers every minute and garbage collection for images every five minutes.

External garbage collection tools are not recommended as these tools can potentially break the behavior of kubelet by removing containers expected to exist.

Image Collection

Kubernetes manages lifecycle of all images through imageManager, with the cooperation of cadvisor.

The policy for garbage collecting images takes two factors into consideration: `HighThresholdPercent` and `LowThresholdPercent`. Disk usage above the high threshold will trigger garbage collection. The garbage collection will delete least recently used images until the low threshold has been met.

Container Collection

The policy for garbage collecting containers considers three user-defined variables. `MinAge` is the minimum age at which a container can be garbage collected. `MaxPerPodContainer` is the maximum number of dead containers every single pod (UID, container name) pair is allowed to have. `MaxContainers` is the maximum number of total dead containers. These variables can be individually disabled by setting `MinAge` to zero and setting `MaxPerPodContainer` and `MaxContainers` respectively to less than zero.

Kubelet will act on containers that are unidentified, deleted, or outside of the boundaries set by the previously mentioned flags. The oldest containers will generally be removed first. `MaxPerPodContainer` and `MaxContainer` may potentially conflict with each other in situations where retaining the maximum number of containers per pod (`MaxPerPodContainer`) would go outside the allowable range of global dead containers (`MaxContainers`). `MaxPerPodContainer` would be adjusted in this situation: A worst case scenario would be to downgrade `MaxPerPodContainer` to 1 and evict the oldest containers. Additionally, containers owned by pods that have been deleted are removed once they are older than `MinAge`.

Containers that are not managed by kubelet are not subject to container garbage collection.

User Configuration

Users can adjust the following thresholds to tune image garbage collection with the following kubelet flags :

1. `image-gc-high-threshold`, the percent of disk usage which triggers image garbage collection. Default is 85%.
2. `image-gc-low-threshold`, the percent of disk usage to which image garbage collection attempts to free. Default is 80%.

We also allow users to customize garbage collection policy through the following kubelet flags:

1. `minimum-container-ttl-duration`, minimum age for a finished container before it is garbage collected. Default is 0 minute, which means every finished container will be garbage collected.
2. `maximum-dead-containers-per-container`, maximum number of old instances to be retained per container. Default is 1.
3. `maximum-dead-containers`, maximum number of old instances of containers to retain globally. Default is -1, which means there is no global limit.

Containers can potentially be garbage collected before their usefulness has expired. These containers can contain logs and other data that can be useful for troubleshooting. A sufficiently large value for `maximum-dead-containers-per-container` is highly recommended to allow at least 1 dead container to be retained per expected container. A larger value for `maximum-dead-containers` is also recommended for a similar reason. See this issue for more details.

Deprecation

Some kubelet Garbage Collection features in this doc will be replaced by kubelet eviction in the future.

Including:

Existing Flag	**New Flag**	**Rationale**
`--image-gc-high-threshold`	`--eviction-hard` or `--eviction-soft`	existing eviction signals can trigger image garbage collection
`--image-gc-low-threshold`	`--eviction-minimum-reclaim`	eviction reclaims achieve the same behavior
`--maximum-dead-containers`		deprecated once old logs are stored outside of container's context

Existing Flag	New Flag	Rationale
`--maximum-dead-containers-per-container`		deprecated once old logs are stored outside of container's context
`--minimum-container-ttl-duration`		deprecated once old logs are stored outside of container's context
`--low-diskspace-threshold-mb`	`--eviction-hard` or `eviction-soft`	eviction generalizes disk thresholds to other resources
`--outofdisk-transition-frequency`	`--eviction-pressure-transition-period`	eviction generalizes disk pressure transition to other resources

What's next

See Configuring Out Of Resource Handling for more details.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on October 09, 2019 at 1:09 PM PST by remove myself from some reviewers lists (#16753) (Page History)

Edit This Page

Federation

> **Deprecated**
>
> Use of `Federation v1` is strongly discouraged. `Federation V1` never achieved GA status and is no longer under active development. Documentation is for historical purposes only.
>
> For more information, see the intended replacement, Kubernetes Federation v2.

This page explains why and how to manage multiple Kubernetes clusters using federation.

Why federation

Federation makes it easy to manage multiple clusters. It does so by providing 2 major building blocks:

- Sync resources across clusters: Federation provides the ability to keep resources in multiple clusters in sync. For example, you can ensure that the same deployment exists in multiple clusters.
- Cross cluster discovery: Federation provides the ability to auto-configure DNS servers and load balancers with backends from all clusters. For example, you can ensure that a global VIP or DNS record can be used to access backends from multiple clusters.

Some other use cases that federation enables are:

- High Availability: By spreading load across clusters and auto configuring DNS servers and load balancers, federation minimises the impact of cluster failure.
- Avoiding provider lock-in: By making it easier to migrate applications across clusters, federation prevents cluster provider lock-in.

Federation is not helpful unless you have multiple clusters. Some of the reasons why you might want multiple clusters are:

- Low latency: Having clusters in multiple regions minimises latency by serving users from the cluster that is closest to them.
- Fault isolation: It might be better to have multiple small clusters rather than a single large cluster for fault isolation (for example: multiple clusters in different availability zones of a cloud provider).
- Scalability: There are scalability limits to a single kubernetes cluster (this should not be the case for most users. For more details: Kubernetes Scaling and Performance Goals).
- Hybrid cloud: You can have multiple clusters on different cloud providers or on-premises data centers.

Caveats

While there are a lot of attractive use cases for federation, there are also some caveats:

- Increased network bandwidth and cost: The federation control plane watches all clusters to ensure that the current state is as expected. This can lead to significant network cost if the clusters are running in different regions on a cloud provider or on different cloud providers.
- Reduced cross cluster isolation: A bug in the federation control plane can impact all clusters. This is mitigated by keeping the logic in federation control plane to a minimum. It mostly delegates to the control plane in kubernetes clusters whenever it can. The design and implementation also errs on the side of safety and avoiding multi-cluster outage.
- Maturity: The federation project is relatively new and is not very mature. Not all resources are available and many are still alpha. Issue 88 enumerates known issues with the system that the team is busy solving.

Hybrid cloud capabilities

Federations of Kubernetes Clusters can include clusters running in different cloud providers (e.g. Google Cloud, AWS), and on-premises (e.g. on OpenStack). Kubefed is the recommended way to deploy federated clusters.

Thereafter, your API resources can span different clusters and cloud providers.

Setting up federation

To be able to federate multiple clusters, you first need to set up a federation control plane. Follow the setup guide to set up the federation control plane.

API resources

Once you have the control plane set up, you can start creating federation API resources. The following guides explain some of the resources in detail:

- Cluster
- ConfigMap
- DaemonSets
- Deployment
- Events
- Hpa
- Ingress
- Jobs
- Namespaces
- ReplicaSets
- Secrets

- Services

The API reference docs list all the resources supported by federation apiserver.

Cascading deletion

Kubernetes version 1.6 includes support for cascading deletion of federated resources. With cascading deletion, when you delete a resource from the federation control plane, you also delete the corresponding resources in all underlying clusters.

Cascading deletion is not enabled by default when using the REST API. To enable it, set the option `DeleteOptions.orphanDependents=false` when you delete a resource from the federation control plane using the REST API. Using `kubectl delete` enables cascading deletion by default. You can disable it by running `kubectl delete --cascade=false`

Note: Kubernetes version 1.5 included cascading deletion support for a subset of federation resources.

Scope of a single cluster

On IaaS providers such as Google Compute Engine or Amazon Web Services, a VM exists in a zone or availability zone. We suggest that all the VMs in a Kubernetes cluster should be in the same availability zone, because:

- compared to having a single global Kubernetes cluster, there are fewer single-points of failure.
- compared to a cluster that spans availability zones, it is easier to reason about the availability properties of a single-zone cluster.
- when the Kubernetes developers are designing the system (e.g. making assumptions about latency, bandwidth, or correlated failures) they are assuming all the machines are in a single data center, or otherwise closely connected.

It is recommended to run fewer clusters with more VMs per availability zone; but it is possible to run multiple clusters per availability zones.

Reasons to prefer fewer clusters per availability zone are:

- improved bin packing of Pods in some cases with more nodes in one cluster (less resource fragmentation).
- reduced operational overhead (though the advantage is diminished as ops tooling and processes mature).
- reduced costs for per-cluster fixed resource costs, e.g. apiserver VMs (but small as a percentage of overall cluster cost for medium to large clusters).

Reasons to have multiple clusters include:

- strict security policies requiring isolation of one class of work from another (but, see Partitioning Clusters below).
- test clusters to canary new Kubernetes releases or other cluster software.

Selecting the right number of clusters

The selection of the number of Kubernetes clusters may be a relatively static choice, only revisited occasionally. By contrast, the number of nodes in a cluster and the number of pods in a service may change frequently according to load and growth.

To pick the number of clusters, first, decide which regions you need to be in to have adequate latency to all your end users, for services that will run on Kubernetes (if you use a Content Distribution Network, the latency requirements for the CDN-hosted content need not be considered). Legal issues might influence this as well. For example, a company with a global customer base might decide to have clusters in US, EU, AP, and SA regions. Call the number of regions to be in `R`.

Second, decide how many clusters should be able to be unavailable at the same time, while still being available. Call the number that can be unavailable `U`. If you are not sure, then 1 is a fine choice.

If it is allowable for load-balancing to direct traffic to any region in the event of a cluster failure, then you need at least the larger of `R` or `U + 1` clusters. If it is not (e.g. you want to ensure low latency for all users in the event of a cluster failure), then you need to have `R * (U + 1)` clusters (`U + 1` in each of `R` regions). In any case, try to put each cluster in a different zone.

Finally, if any of your clusters would need more than the maximum recommended number of nodes for a Kubernetes cluster, then you may need even more clusters. Kubernetes v1.3 supports clusters up to 1000 nodes in size. Kubernetes v1.8 supports clusters up to 5000 nodes. See Building Large Clusters for more guidance.

What's next

- Learn more about the Federation proposal.
- See this setup guide for cluster federation.
- See this Kubecon2016 talk on federation
- See this Kubecon2017 Europe update on federation
- See this Kubecon2018 Europe update on sig-multicluster
- See this Kubecon2018 Europe Federation-v2 prototype presentation
- See this Federation-v2 Userguide

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on October 21, 2019 at 7:39 PM PST by Update broken links for API resources in federation.md (#17082) (Page History)

Edit This Page

Proxies in Kubernetes

This page explains proxies used with Kubernetes.

- Proxies
- Requesting redirects

Proxies

There are several different proxies you may encounter when using Kubernetes:

1. The kubectl proxy:

    - runs on a user's desktop or in a pod
    - proxies from a localhost address to the Kubernetes apiserver
    - client to proxy uses HTTP
    - proxy to apiserver uses HTTPS
    - locates apiserver
    - adds authentication headers

2. The apiserver proxy:

    - is a bastion built into the apiserver
    - connects a user outside of the cluster to cluster IPs which otherwise might not be reachable
    - runs in the apiserver processes
    - client to proxy uses HTTPS (or http if apiserver so configured)
    - proxy to target may use HTTP or HTTPS as chosen by proxy using available information
    - can be used to reach a Node, Pod, or Service
    - does load balancing when used to reach a Service

3. The kube proxy:

    - runs on each node
    - proxies UDP, TCP and SCTP
    - does not understand HTTP
    - provides load balancing
    - is just used to reach services

4. A Proxy/Load-balancer in front of apiserver(s):

    - existence and implementation varies from cluster to cluster (e.g. nginx)
    - sits between all clients and one or more apiservers
    - acts as load balancer if there are several apiservers.

5. Cloud Load Balancers on external services:
    - are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer)
    - are created automatically when the Kubernetes service has type `LoadBalancer`
    - usually supports UDP/TCP only
    - SCTP support is up to the load balancer implementation of the cloud provider
    - implementation varies by cloud provider.

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are setup correctly.

Requesting redirects

Proxies have replaced redirect capabilities. Redirects have been deprecated.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on September 27, 2018 at 6:41 PM PST by [Do Not Merge] Release 1.12 (#10292) (Page History)

Edit This Page

Installing Addons

Add-ons extend the functionality of Kubernetes.

This page lists some of the available add-ons and links to their respective installation instructions.

Add-ons in each section are sorted alphabetically - the ordering does not imply any preferential status.

- Networking and Network Policy
- Service Discovery
- Visualization & Control
- Infrastructure
- Legacy Add-ons

Networking and Network Policy

- ACI provides integrated container networking and network security with Cisco ACI.
- Calico is a networking and network policy provider. Calico supports a flexible set of networking options so you can choose the most efficient option for your situation, including non-overlay and overlay networks, with or without BGP. Calico uses the same engine to enforce network policy for hosts, pods, and (if using Istio & Envoy) applications at the service mesh layer.
- Canal unites Flannel and Calico, providing networking and network policy.
- Cilium is a L3 network and network policy plugin that can enforce HTTP/API/L7 policies transparently. Both routing and overlay/encapsulation mode are supported, and it can work on top of other CNI plugins.
- CNI-Genie enables Kubernetes to seamlessly connect to a choice of CNI plugins, such as Calico, Canal, Flannel, Romana, or Weave.
- Contiv provides configurable networking (native L3 using BGP, overlay using vxlan, classic L2, and Cisco-SDN/ACI) for various use cases and a rich policy framework. Contiv project is fully open sourced. The installer provides both kubeadm and non-kubeadm based installation options.
- Contrail, based on Tungsten Fabric, is an open source, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide isolation modes for virtual machines, containers/pods and bare metal workloads.
- Flannel is an overlay network provider that can be used with Kubernetes.
- Knitter is a plugin to support multiple network interfaces in a Kubernetes pod.
- Multus is a Multi plugin for multiple network support in Kubernetes to support all CNI plugins (e.g. Calico, Cilium, Contiv, Flannel), in addition to SRIOV, DPDK, OVS-DPDK and VPP based workloads in Kubernetes.
- NSX-T Container Plug-in (NCP) provides integration between VMware NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.
- Nuage is an SDN platform that provides policy-based networking between Kubernetes Pods and non-Kubernetes environments with visibility and security monitoring.

- Romana is a Layer 3 networking solution for pod networks that also supports the NetworkPolicy API. Kubeadm add-on installation details available here.
- Weave Net provides networking and network policy, will carry on working on both sides of a network partition, and does not require an external database.

Service Discovery

- CoreDNS is a flexible, extensible DNS server which can be installed as the in-cluster DNS for pods.

Visualization & Control

- Dashboard is a dashboard web interface for Kubernetes.
- Weave Scope is a tool for graphically visualizing your containers, pods, services etc. Use it in conjunction with a Weave Cloud account or host the UI yourself.

Infrastructure

- KubeVirt is an add-on to run virtual machines on Kubernetes. Usually run on bare-metal clusters.

Legacy Add-ons

There are several other add-ons documented in the deprecated cluster/addons directory.

Well-maintained ones should be linked to here. PRs welcome!

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 08, 2020 at 6:47 AM PST by Correct description of Knitter CNI plugin (#18983) (Page History)

Edit This Page

Extending your Kubernetes Cluster

Kubernetes is highly configurable and extensible. As a result, there is rarely a need to fork or submit patches to the Kubernetes project code.

This guide describes the options for customizing a Kubernetes cluster. It is aimed at cluster operatorsA person who configures, controls, and monitors clusters. who want to understand how to adapt their Kubernetes cluster to the needs of their work environment. Developers who are prospective Platform DevelopersA person who customizes the Kubernetes platform to fit the needs of their project. or Kubernetes Project ContributorsSomeone who donates code, documentation, or their time to help the Kubernetes project or community. will also find it useful as an introduction to what extension points and patterns exist, and their trade-offs and limitations.

Overview

Customization approaches can be broadly divided into *configuration*, which only involves changing flags, local configuration files, or API resources; and *extensions*, which involve running additional programs or services. This document is primarily about extensions.

Configuration

Configuration files and *flags* are documented in the Reference section of the online documentation, under each binary:

- kubelet
- kube-apiserver
- kube-controller-manager
- kube-scheduler.

Flags and configuration files may not always be changeable in a hosted Kubernetes service or a distribution with managed installation. When they are changeable, they are usually only changeable by the cluster administrator. Also, they are subject to change in future Kubernetes versions, and setting them may require restarting processes. For those reasons, they should be used only when there are no other options.

Built-in Policy APIs, such as ResourceQuota, PodSecurityPolicies, NetworkPolicy and Role-based Access Control (RBAC), are built-in Kubernetes APIs. APIs are typically used with hosted Kubernetes services and with managed Kubernetes installations. They are declarative and use the same conventions as other Kubernetes resources like pods, so new cluster configuration can be repeatable and be managed the same way as applications. And, where they are stable, they enjoy a defined support policy like other Kubernetes APIs. For these reasons, they are preferred over *configuration files* and *flags* where suitable.

Extensions

Extensions are software components that extend and deeply integrate with Kubernetes. They adapt it to support new types and new kinds of hardware.

Most cluster administrators will use a hosted or distribution instance of Kubernetes. As a result, most Kubernetes users will need to install extensions and fewer will need to author new ones.

Extension Patterns

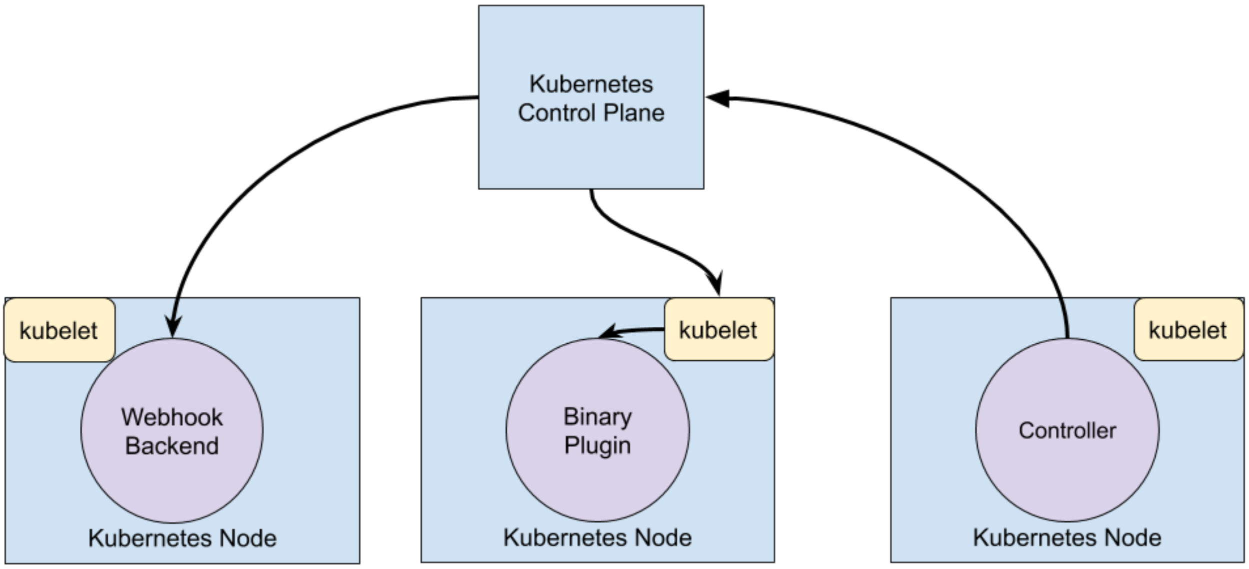
Kubernetes is designed to be automated by writing client programs. Any program that reads and/or writes to the Kubernetes API can provide useful automation. *Automation* can run on the cluster or off it. By following the guidance in this doc you can write highly available and robust automation. Automation generally works with any Kubernetes cluster, including hosted clusters and managed installations.

There is a specific pattern for writing client programs that work well with Kubernetes called the *Controller* pattern. Controllers typically read an object's `.spec`, possibly do things, and then update the object's `.status`.

A controller is a client of Kubernetes. When Kubernetes is the client and calls out to a remote service, it is called a *Webhook*. The remote service is called a *Webhook Backend*. Like Controllers, Webhooks do add a point of failure.

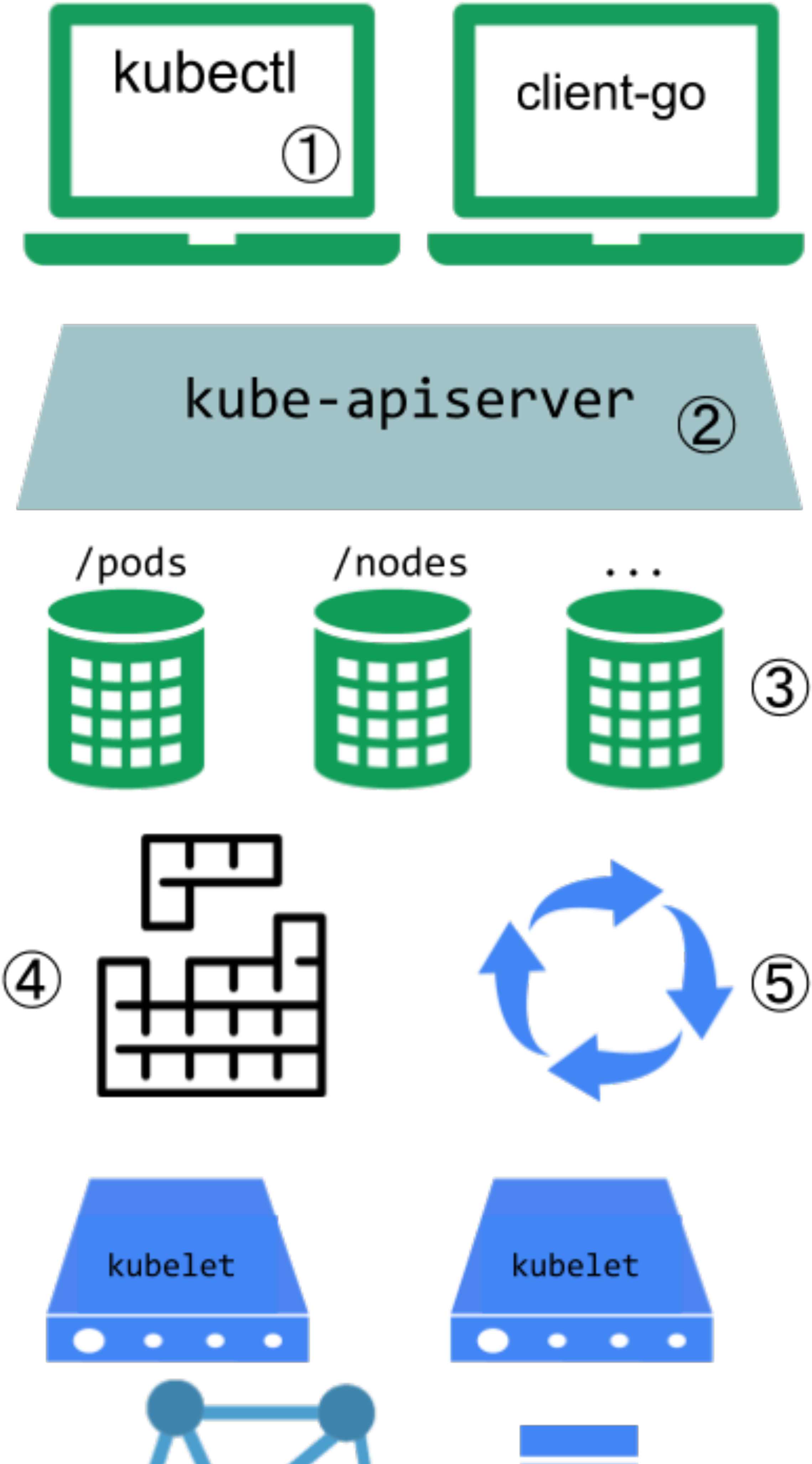
In the webhook model, Kubernetes makes a network request to a remote service. In the *Binary Plugin* model, Kubernetes executes a binary (program). Binary plugins are used by the kubelet (e.g. Flex Volume Plugins and Network Plugins) and by kubectl.

Below is a diagram showing how the extension points interact with the Kubernetes control plane.



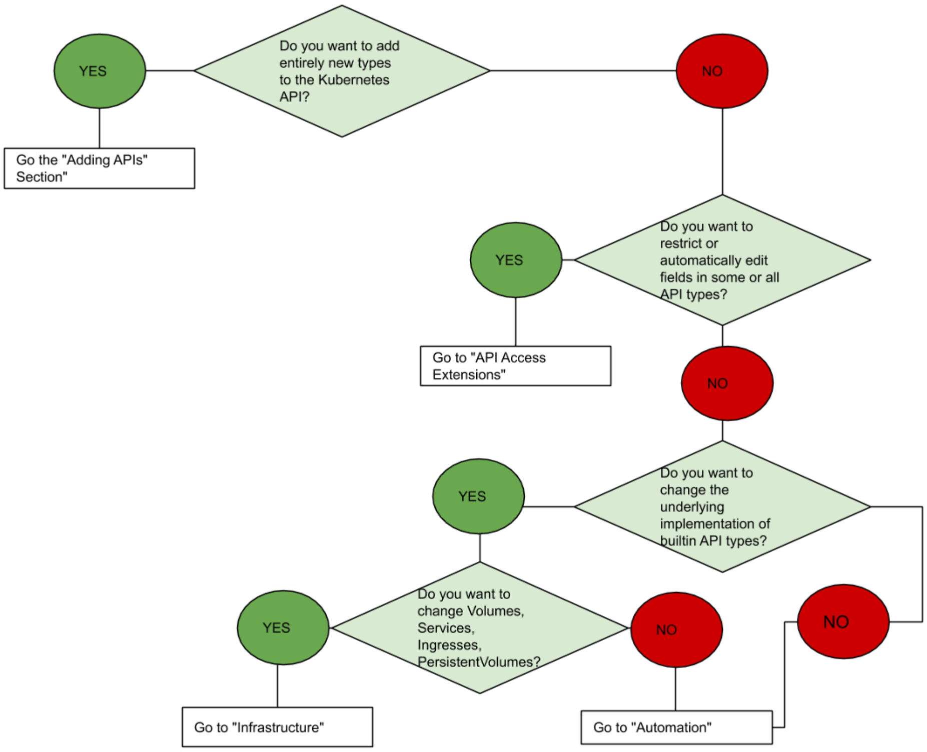
Extension Points

This diagram shows the extension points in a Kubernetes system.


kubectl
①
client-go
kube-apiserver
②
/pods
/nodes
...
③
④
⑤
kubelet
kubelet

1. Users often interact with the Kubernetes API using `kubectl`. Kubectl plugins extend the kubectl binary. They only affect the individual user's local environment, and so cannot enforce site-wide policies.
2. The apiserver handles all requests. Several types of extension points in the apiserver allow authenticating requests, or blocking them based on their content, editing content, and handling deletion. These are described in the API Access Extensions section.
3. The apiserver serves various kinds of *resources*. *Built-in resource kinds*, like `pods`, are defined by the Kubernetes project and can't be changed. You can also add resources that you define, or that other projects have defined, called *Custom Resources*, as explained in the Custom Resources section. Custom Resources are often used with API Access Extensions.
4. The Kubernetes scheduler decides which nodes to place pods on. There are several ways to extend scheduling. These are described in the Scheduler Extensions section.
5. Much of the behavior of Kubernetes is implemented by programs called Controllers which are clients of the API-Server. Controllers are often used in conjunction with Custom Resources.
6. The kubelet runs on servers, and helps pods appear like virtual servers with their own IPs on the cluster network. Network Plugins allow for different implementations of pod networking.
7. The kubelet also mounts and unmounts volumes for containers. New types of storage can be supported via Storage Plugins.

If you are unsure where to start, this flowchart can help. Note that some solutions may involve several types of extensions.


YES
Do you want to add entirely new types to the Kubernetes API?
NO
Go the "Adding APIs" Section"
YES
Do you want to restrict or automatically edit fields in some or all API types?
Go to "API Access Extensions"
NO
YES
Do you want to change the underlying implementation of builtin API types?
YES
Do you want to change Volumes, Services, Ingresses, PersistentVolumes?
NO
NO
Go to "Infrastructure"
Go to "Automation"

API Extensions

User-Defined Types

Consider adding a Custom Resource to Kubernetes if you want to define new controllers, application configuration objects or other declarative APIs, and to manage them using Kubernetes tools, such as `kubectl`.

Do not use a Custom Resource as data storage for application, user, or monitoring data.

For more about Custom Resources, see the Custom Resources concept guide.

Combining New APIs with Automation

The combination of a custom resource API and a control loop is called the Operator pattern. The Operator pattern is used to manage specific, usually stateful, applications. These custom APIs and control loops can also be used to control other resources, such as storage or policies.

Changing Built-in Resources

When you extend the Kubernetes API by adding custom resources, the added resources always fall into a new API Groups. You cannot replace or change existing API groups. Adding an API does not directly let you affect the behavior of existing APIs (e.g. Pods), but API Access Extensions do.

API Access Extensions

When a request reaches the Kubernetes API Server, it is first Authenticated, then Authorized, then subject to various types of Admission Control. See Controlling Access to the Kubernetes API for more on this flow.

Each of these steps offers extension points.

Kubernetes has several built-in authentication methods that it supports. It can also sit behind an authenticating proxy, and it can send a token from an Authorization header to a remote service for verification (a webhook). All of these methods are covered in the Authentication documentation.

Authentication

Authentication maps headers or certificates in all requests to a username for the client making the request.

Kubernetes provides several built-in authentication methods, and an Authentication webhook method if those don't meet your needs.

Authorization

Authorization determines whether specific users can read, write, and do other operations on API resources. It just works at the level of whole resources - it doesn't discriminate based on arbitrary object fields. If the built-in authorization options don't meet your needs, and Authorization webhook allows calling out to user-provided code to make an authorization decision.

Dynamic Admission Control

After a request is authorized, if it is a write operation, it also goes through Admission Control steps. In addition to the built-in steps, there are several extensions:

- The Image Policy webhook restricts what images can be run in containers.
- To make arbitrary admission control decisions, a general Admission webhook can be used. Admission Webhooks can reject creations or updates.

Infrastructure Extensions

Storage Plugins

Flex Volumes allow users to mount volume types without built-in support by having the Kubelet call a Binary Plugin to mount the volume.

Device Plugins

Device plugins allow a node to discover new Node resources (in addition to the builtin ones like cpu and memory) via a Device Plugin.

Network Plugins

Different networking fabrics can be supported via node-level Network Plugins.

Scheduler Extensions

The scheduler is a special type of controller that watches pods, and assigns pods to nodes. The default scheduler can be replaced entirely, while continuing to use other Kubernetes components, or multiple schedulers can run at the same time.

This is a significant undertaking, and almost all Kubernetes users find they do not need to modify the scheduler.

The scheduler also supports a webhook that permits a webhook backend (scheduler extension) to filter and prioritize the nodes chosen for a pod.

What's next

- Learn more about Custom Resources
- Learn about Dynamic admission control
- Learn more about Infrastructure extensions
  - Network Plugins
  - Device Plugins
- Learn about kubectl plugins
- Learn about the Operator pattern

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on November 09, 2019 at 5:55 AM PST by Typo, should be extension's instead of extensions (#17373) (Page History)

Edit This Page

Extending the Kubernetes API with the aggregation layer

The aggregation layer allows Kubernetes to be extended with additional APIs, beyond what is offered by the core Kubernetes APIs.

Overview

The aggregation layer enables installing additional Kubernetes-style APIs in your cluster. These can either be pre-built, existing 3rd party solutions, such as service-catalog, or user-created APIs like apiserver-builder, which can get you started.

The aggregation layer runs in-process with the kube-apiserver. Until an extension resource is registered, the aggregation layer will do nothing. To register an API, users must add an APIService object, which "claims" the URL path in the Kubernetes API. At that point, the aggregation layer will proxy anything sent to that API path (e.g. /apis/myextension.mycompany.io/v1/â€¦) to the registered APIService.

Ordinarily, the APIService will be implemented by an *extension-apiserver* in a pod running in the cluster. This extension-apiserver will normally need to be paired with one or more controllers if active management of the added resources is needed. As a result, the apiserver-builder will actually provide a skeleton for both. As another example, when the service-catalog is installed, it provides both the extension-apiserver and controller for the services it provides.

Extension-apiservers should have low latency connections to and from the kube-apiserver. In particular, discovery requests are required to round-trip from the kube-apiserver in five seconds or less. If your deployment cannot achieve this, you should consider how to change it. For now, setting the `EnableAggregatedDiscoveryTimeout=false` feature gate on the kube-apiserver will disable the timeout restriction. It will be removed in a future release.

What's next

- To get the aggregator working in your environment, configure the aggregation layer.
- Then, setup an extension api-server to work with the aggregation layer.
- Also, learn how to extend the Kubernetes API using Custom Resource Definitions.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on September 16, 2019 at 10:34 AM PST by add latency requirements to aggregated apiserver discovery (#16372) (Page History)

Edit This Page

Custom Resources

Custom resources are extensions of the Kubernetes API. This page discusses when to add a custom resource to your Kubernetes cluster and when to use a standalone service. It describes the two methods for adding custom resources and how to choose between them.

Custom resources

A *resource* is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind. For example, the built-in *pods* resource contains a collection of Pod objects.

A *custom resource* is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation. However, many core Kubernetes functions are now built using custom resources, making Kubernetes more modular.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects using kubectl, just as they do for built-in resources like *Pods*.

Custom controllers

On their own, custom resources simply let you store and retrieve structured data. When you combine a custom resource with a *custom controller*, custom resources provide a true *declarative API*.

A declarative API allows you to *declare* or specify the desired state of your resource and tries to keep the current state of Kubernetes objects in sync with the desired state. The controller interprets the structured data as a record of the user's desired state, and continually maintains this state.

You can deploy and update a custom controller on a running cluster, independently of the cluster's lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources. The Operator pattern combines custom resources and custom controllers. You can use custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Should I add a custom resource to my Kubernetes Cluster?

When creating a new API, consider whether to aggregate your API with the Kubernetes cluster APIs or let your API stand alone.

Consider API aggregation if:	**Prefer a stand-alone API if:**
Your API is Declarative.	Your API does not fit the Declarative model.
You want your new types to be readable and writable using `kubectl`.	`kubectl` support is not required
You want to view your new types in a Kubernetes UI, such as dashboard, alongside built-in types.	Kubernetes UI support is not required.
You are developing a new API.	You already have a program that serves your API and works well.
You are willing to accept the format restriction that Kubernetes puts on REST resource paths, such as API Groups and Namespaces. (See the API Overview.)	You need to have specific REST paths to be compatible with an already defined REST API.
Your resources are naturally scoped to a cluster or namespaces of a cluster.	Cluster or namespace scoped resources are a poor fit; you need control over the specifics of resource paths.
You want to reuse Kubernetes API support features.	You don't need those features.

Declarative APIs

In a Declarative API, typically:

- Your API consists of a relatively small number of relatively small objects (resources).
- The objects define configuration of applications or infrastructure.
- The objects are updated relatively infrequently.
- Humans often need to read and write the objects.
- The main operations on the objects are CRUD-y (creating, reading, updating and deleting).
- Transactions across objects are not required: the API represents a desired state, not an exact state.

Imperative APIs are not declarative. Signs that your API might not be declarative include:

- The client says "do this", and then gets a synchronous response back when it is done.
- The client says "do this", and then gets an operation ID back, and has to check a separate Operation object to determine completion of the request.
- You talk about Remote Procedure Calls (RPCs).
- Directly storing large amounts of data (e.g. > a few kB per object, or >1000s of objects).
- High bandwidth access (10s of requests per second sustained) needed.
- Store end-user data (such as images, PII, etc.) or other large-scale data processed by applications.
- The natural operations on the objects are not CRUD-y.

- The API is not easily modeled as objects.
- You chose to represent pending operations with an operation ID or an operation object.

Should I use a configMap or a custom resource?

Use a ConfigMap if any of the following apply:

- There is an existing, well-documented config file format, such as a `mysql.cnf` or `pom.xml`.
- You want to put the entire config file into one key of a configMap.
- The main use of the config file is for a program running in a Pod on your cluster to consume the file to configure itself.
- Consumers of the file prefer to consume via file in a Pod or environment variable in a pod, rather than the Kubernetes API.
- You want to perform rolling updates via Deployment, etc., when the file is updated.

  Note: Use a secret for sensitive data, which is similar to a configMap but more secure.

Use a custom resource (CRD or Aggregated API) if most of the following apply:

- You want to use Kubernetes client libraries and CLIs to create and update the new resource.
- You want top-level support from kubectl (for example: `kubectl get my-object object-name`).
- You want to build new automation that watches for updates on the new object, and then CRUD other objects, or vice versa.
- You want to write automation that handles updates to the object.
- You want to use Kubernetes API conventions like `.spec`, `.status`, and `.metadata`.
- You want the object to be an abstraction over a collection of controlled resources, or a summarization of other resources.

Adding custom resources

Kubernetes provides two ways to add custom resources to your cluster:

- CRDs are simple and can be created without any programming.
- API Aggregation requires programming, but allows more control over API behaviors like how data is stored and conversion between API versions.

Kubernetes provides these two options to meet the needs of different users, so that neither ease of use nor flexibility is compromised.

Aggregated APIs are subordinate APIServers that sit behind the primary API server, which acts as a proxy. This arrangement is called API Aggregation (AA). To users, it simply appears that the Kubernetes API is extended.

CRDs allow users to create new types of resources without adding another APIserver. You do not need to understand API Aggregation to use CRDs.

Regardless of how they are installed, the new resources are referred to as Custom Resources to distinguish them from built-in Kubernetes resources (like pods).

CustomResourceDefinitions

The CustomResourceDefinition API resource allows you to define custom resources. Defining a CRD object creates a new custom resource with a name and schema that you specify. The Kubernetes API serves and handles the storage of your custom resource.

This frees you from writing your own API server to handle the custom resource, but the generic nature of the implementation means you have less flexibility than with API server aggregation.

Refer to the custom controller example for an example of how to register a new custom resource, work with instances of your new resource type, and use a controller to handle events.

API server aggregation

Usually, each resource in the Kubernetes API requires code that handles REST requests and manages persistent storage of objects. The main Kubernetes API server handles built-in resources like *pods* and *services*, and can also generically handle custom resources through CRDs.

The aggregation layer allows you to provide specialized implementations for your custom resources by writing and deploying your own standalone API server. The main API server delegates requests to you for the custom resources that you handle, making them available to all of its clients.

Choosing a method for adding custom resources

CRDs are easier to use. Aggregated APIs are more flexible. Choose the method that best meets your needs.

Typically, CRDs are a good fit if:

- You have a handful of fields
- You are using the resource within your company, or as part of a small open-source project (as opposed to a commercial product)

Comparing ease of use

CRDs are easier to create than Aggregated APIs.

CRDs	Aggregated API
Do not require programming. Users can choose any language for a CRD controller.	Requires programming in Go and building binary and image. Users can choose any language for a CRD controller.
No additional service to run; CRs are handled by API Server.	An additional service to create and that could fail.
No ongoing support once the CRD is created. Any bug fixes are picked up as part of normal Kubernetes Master upgrades.	May need to periodically pickup bug fixes from upstream and rebuild and update the Aggregated APIserver.
No need to handle multiple versions of your API. For example: when you control the client for this resource, you can upgrade it in sync with the API.	You need to handle multiple versions of your API, for example: when developing an extension to share with the world.

Advanced features and flexibility

Aggregated APIs offer more advanced API features and customization of other features, for example: the storage layer.

Feature	Description	CRDs	Aggregated API
Validation	Help users prevent errors and allow you to evolve your API independently of your clients. These features are most useful when there are many clients who can't all update at the same time.	Yes. Most validation can be specified in the CRD using OpenAPI v3.0 validation. Any other validations supported by addition of a Validating Webhook.	Yes, arbitrary validation checks
Defaulting	See above	Yes, either via OpenAPI v3.0 validation `default` keyword (GA in 1.17), or via a Mutating Webhook (though this will not be run when reading from etcd for old objects)	Yes

Feature	**Description**	**CRDs**	**Aggregated API**
Multi-versioning	Allows serving the same object through two API versions. Can help ease API changes like renaming fields. Less important if you control your client versions.	Yes	Yes
Custom Storage	If you need storage with a different performance mode (for example, time-series database instead of key-value store) or isolation for security (for example, encryption secrets or different	No	Yes
Custom Business Logic	Perform arbitrary checks or actions when creating, reading, updating or deleting an object	Yes, using Webhooks.	Yes
Scale Subresource	Allows systems like HorizontalPodAutoscaler and PodDisruptionBudget interact with your new resource	Yes	Yes
Status Subresource	• Finer-grained access control: user writes spec section, controller writes status section. • Allows incrementing object Generation on custom resource data mutation (requires separate spec and status sections in the resource)	Yes	Yes
Other Subresources	Add operations other than CRUD, such as "logs" or "exec".	No	Yes
strategic-merge-patch	The new endpoints support PATCH with `Content-Type: application/strategic-merge-patch+json`. Useful for updating objects that may be modified both locally, and by the server. For more information, see "Update API Objects in Place Using kubectl patch"	No	Yes

Feature	Description	CRDs	Aggregated API
Protocol Buffers	The new resource supports clients that want to use Protocol Buffers	No	Yes
OpenAPI Schema	Is there an OpenAPI (swagger) schema for the types that can be dynamically fetched from the server? Is the user protected from misspelling field names by ensuring only allowed fields are set? Are types enforced (in other words, don't put an `int` in a `string` field?)	Yes, based on the OpenAPI v3.0 validation schema (GA in 1.16)	Yes

Common Features

When you create a custom resource, either via a CRDs or an AA, you get many features for your API, compared to implementing it outside the Kubernetes platform:

Feature	What it does
CRUD	The new endpoints support CRUD basic operations via HTTP and `kubectl`
Watch	The new endpoints support Kubernetes Watch operations via HTTP
Discovery	Clients like kubectl and dashboard automatically offer list, display, and field edit operations on your resources
json-patch	The new endpoints support PATCH with `Content-Type: application/json-patch+json`
merge-patch	The new endpoints support PATCH with `Content-Type: application/merge-patch+json`
HTTPS	The new endpoints uses HTTPS
Built-in Authentication	Access to the extension uses the core apiserver (aggregation layer) for authentication
Built-in Authorization	Access to the extension can reuse the authorization used by the core apiserver (e.g. RBAC)
Finalizers	Block deletion of extension resources until external cleanup happens.
Admission Webhooks	Set default values and validate extension resources during any create/update/delete operation.
UI/CLI Display	Kubectl, dashboard can display extension resources.
Unset versus Empty	Clients can distinguish unset fields from zero-valued fields.
Client Libraries Generation	Kubernetes provides generic client libraries, as well as tools to generate type-specific client libraries.

Feature	What it does
Labels and annotations	Common metadata across objects that tools know how to edit for core and custom resources.

Preparing to install a custom resource

There are several points to be aware of before adding a custom resource to your cluster.

Third party code and new points of failure

While creating a CRD does not automatically add any new points of failure (for example, by causing third party code to run on your API server), packages (for example, Charts) or other installation bundles often include CRDs as well as a Deployment of third-party code that implements the business logic for a new custom resource.

Installing an Aggregated APIserver always involves running a new Deployment.

Storage

Custom resources consume storage space in the same way that ConfigMaps do. Creating too many custom resources may overload your API server's storage space.

Aggregated API servers may use the same storage as the main API server, in which case the same warning applies.

Authentication, authorization, and auditing

CRDs always use the same authentication, authorization, and audit logging as the built-in resources of your API Server.

If you use RBAC for authorization, most RBAC roles will not grant access to the new resources (except the cluster-admin role or any role created with wildcard rules). You'll need to explicitly grant access to the new resources. CRDs and Aggregated APIs often come bundled with new role definitions for the types they add.

Aggregated API servers may or may not use the same authentication, authorization, and auditing as the primary API server.

Accessing a custom resource

Kubernetes client libraries can be used to access custom resources. Not all client libraries support custom resources. The go and python client libraries do.

When you add a custom resource, you can access it using:

- kubectl
- The kubernetes dynamic client.
- A REST client that you write.
- A client generated using Kubernetes client generation tools (generating one is an advanced undertaking, but some projects may provide a client along with the CRD or AA).

What's next

- Learn how to Extend the Kubernetes API with the aggregation layer.

- Learn how to Extend the Kubernetes API with CustomResourceDefinition.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on February 19, 2020 at 3:14 AM PST by Latin Abbreviations "vs" Updated to "versus" (#19181) (Page History)

Edit This Page

Network Plugins

FEATURE STATE: `Kubernetes v1.17` alpha
This feature is currently in a *alpha* state, meaning:

Edit This Page

Device Plugins

FEATURE STATE: `Kubernetes v1.10` beta
This feature is currently in a *beta* state, meaning:

Edit This Page

Operator pattern

Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.

Motivation

The Operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. Human operators who look after specific applications and services have deep knowledge of how the system ought to behave, how to deploy it, and how to react if there are problems.

People who run workloads on Kubernetes often like to use automation to take care of repeatable tasks. The Operator pattern captures how you can write code to automate a task beyond what Kubernetes itself provides.

Operators in Kubernetes

Kubernetes is designed for automation. Out of the box, you get lots of built-in automation from the core of Kubernetes. You can use Kubernetes to automate deploying and running workloads, *and* you can automate how Kubernetes does that.

Kubernetes' controllersA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. concept lets you extend the cluster's behaviour without modifying the code of Kubernetes itself. Operators are clients of the Kubernetes API that act as controllers for a Custom Resource.

An example Operator

Some of the things that you can use an operator to automate include:

- deploying an application on demand
- taking and restoring backups of that application's state
- handling upgrades of the application code alongside related changes such as database schemas or extra configuration settings
- publishing a Service to applications that don't support Kubernetes APIs to discover them
- simulating failure in all or part of your cluster to test its resilience
- choosing a leader for a distributed application without an internal member election process

What might an Operator look like in more detail? Here's an example in more detail:

1. A custom resource named SampleDB, that you can configure into the cluster.
2. A Deployment that makes sure a Pod is running that contains the controller part of the operator.
3. A container image of the operator code.
4. Controller code that queries the control plane to find out what SampleDB resources are configured.
5. The core of the Operator is code to tell the API server how to make reality match the configured resources.
    - If you add a new SampleDB, the operator sets up PersistentVolumeClaims to provide durable database storage, a StatefulSet to run SampleDB and a Job to handle initial configuration.
    - If you delete it, the Operator takes a snapshot, then makes sure that the StatefulSet and Volumes are also removed.
6. The operator also manages regular database backups. For each SampleDB resource, the operator determines when to create a Pod that can connect to the database and take backups. These Pods would rely on a ConfigMap and / or a Secret that has database connection details and credentials.
7. Because the Operator aims to provide robust automation for the resource it manages, there would be additional supporting code. For this example, code checks to see if the database is running an old version and, if so, creates Job objects that upgrade it for you.

Deploying Operators

The most common way to deploy an Operator is to add the Custom Resource Definition and its associated Controller to your cluster. The Controller will normally run outside of the control planeThe container orchestration layer that exposes the API and interfaces to define, deploy, and manage the lifecycle of containers. , much as you would run any containerized application. For example, you can run the controller in your cluster as a Deployment.

Using an Operator

Once you have an Operator deployed, you'd use it by adding, modifying or deleting the kind of resource that the Operator uses. Following the above example, you would set up a Deployment for the Operator itself, and then:

```
kubectl get SampleDB                   # find configured databases

kubectl edit SampleDB/example-database # manually change some settings
```

â€¦and that's it! The Operator will take care of applying the changes as well as keeping the existing service in good shape.

Writing your own Operator

If there isn't an Operator in the ecosystem that implements the behavior you want, you can code your own. In What's next you'll find a few links to libraries and tools you can use to write your own cloud native Operator.

You also implement an Operator (that is, a Controller) using any language / runtime that can act as a client for the Kubernetes API.

What's next

- Learn more about Custom Resources
- Find ready-made operators on OperatorHub.io to suit your use case
- Use existing tools to write your own operator, eg:
    - using KUDO (Kubernetes Universal Declarative Operator)
    - using kubebuilder
    - using Metacontroller along with WebHooks that you implement yourself
    - using the Operator Framework
- Publish your operator for other people to use
- Read CoreOS' original article that introduced the Operator pattern
- Read an article from Google Cloud about best practices for building Operators

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on September 08, 2019 at 10:53 PM PST by Fixed typo (#16248) (Page History)

Edit This Page

Service Catalog

Service Catalog is an extension API that enables applications running in Kubernetes clusters to easily use external managed software offerings, such as a datastore service offered by a cloud provider.

It provides a way to list, provision, and bind with external Managed ServicesA software offering maintained by a third-party provider. from Service BrokersAn endpoint for a set of Managed Services offered and maintained by a third-party. without needing detailed knowledge about how those services are created or managed.

A service broker, as defined by the Open service broker API spec, is an endpoint for a set of managed services offered and maintained by a third-party, which could be a cloud provider such as AWS, GCP, or Azure. Some examples of managed services are Microsoft Azure Cloud Queue, Amazon Simple Queue Service, and Google Cloud Pub/Sub, but they can be any software offering that can be used by an application.

Using Service Catalog, a cluster operatorA person who configures, controls, and monitors clusters. can browse the list of managed services offered by a service broker, provision an instance of a managed service, and bind with it to make it available to an application in the Kubernetes cluster.

Example use case

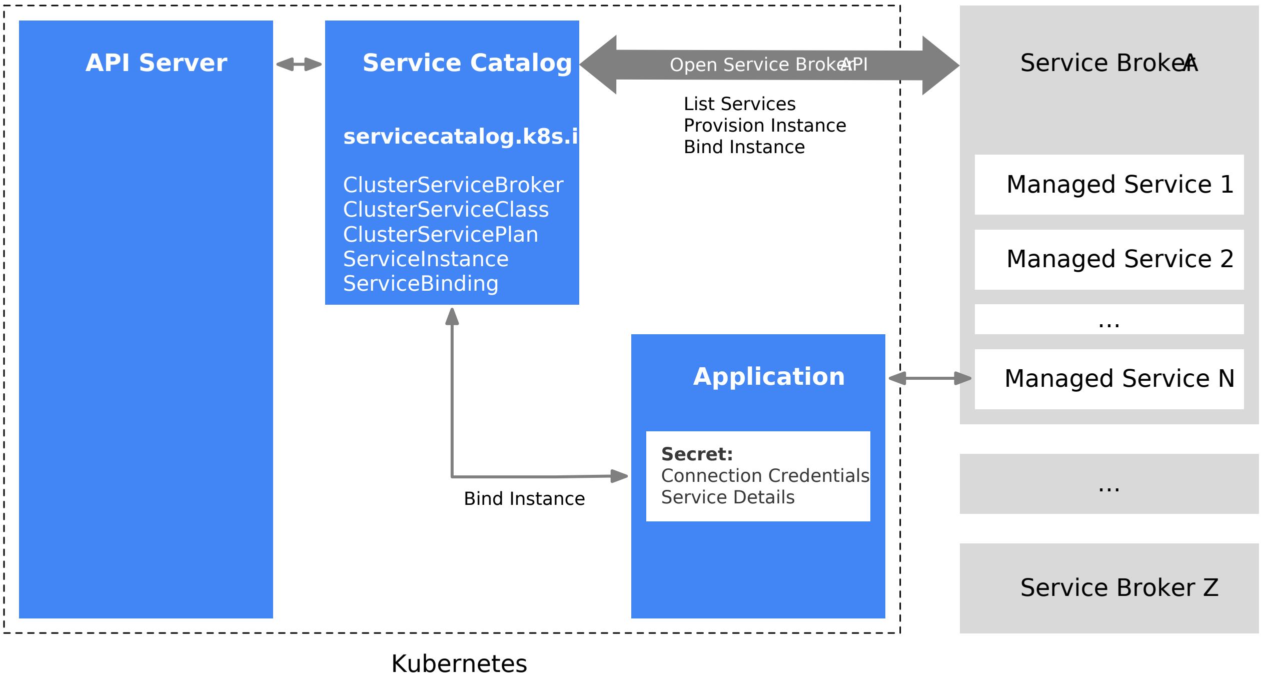
An application developerA person who writes an application that runs in a Kubernetes cluster. wants to use message queuing as part of their application running in a Kubernetes cluster. However, they do not want to deal with the overhead of setting such a service up and administering it themselves. Fortunately, there is a cloud provider that offers message queuing as a managed service through its service broker.

A cluster operator can setup Service Catalog and use it to communicate with the cloud provider's service broker to provision an instance of the message queuing service and make it available to the application within the Kubernetes cluster. The application developer therefore does not need to be concerned with the implementation details or management of the message queue. The application can simply use it as a service.

Architecture

Service Catalog uses the Open service broker API to communicate with service brokers, acting as an intermediary for the Kubernetes API Server to negotiate the initial provisioning and retrieve the credentials necessary for the application to use a managed service.

It is implemented as an extension API server and a controller, using etcd for storage. It also uses the aggregation layer available in Kubernetes 1.7+ to present its API.



API Resources

Service Catalog installs the `servicecatalog.k8s.io` API and provides the following Kubernetes resources:

- `ClusterServiceBroker`: An in-cluster representation of a service broker, encapsulating its server connection details. These are created and managed by cluster operators who wish to use that broker server to make new types of managed services available within their cluster.
- `ClusterServiceClass`: A managed service offered by a particular service broker. When a new `ClusterServiceBroker` resource is added to the cluster, the Service Catalog controller connects to the service broker to obtain a list of available managed services. It then creates a new `ClusterServiceClass` resource corresponding to each managed service.
- `ClusterServicePlan`: A specific offering of a managed service. For example, a managed service may have different plans available, such as a free tier or paid tier, or it may have different configuration options, such as using SSD storage or having more resources. Similar to `ClusterServiceClass`, when a new `ClusterServiceBroker` is added to the cluster, Service Catalog creates a new `ClusterServicePlan` resource corresponding to each Service Plan available for each managed service.
- `ServiceInstance`: A provisioned instance of a `ClusterServiceClass`. These are created by cluster operators to make a specific instance of a managed service available for use by one or more in-cluster applications. When a new `ServiceInstance` resource is created, the Service Catalog controller connects to the appropriate service broker and instruct it to provision the service instance.
- `ServiceBinding`: Access credentials to a `ServiceInstance`. These are created by cluster operators who want their applications to make use of a `ServiceInstance`. Upon creation, the Service Catalog controller

creates a Kubernetes `Secret` containing connection details and credentials for the Service Instance, which can be mounted into Pods.

Authentication

Service Catalog supports these methods of authentication:

- Basic (username/password)
- OAuth 2.0 Bearer Token

Usage

A cluster operator can use Service Catalog API Resources to provision managed services and make them available within a Kubernetes cluster. The steps involved are:

1. Listing the managed services and Service Plans available from a service broker.
2. Provisioning a new instance of the managed service.
3. Binding to the managed service, which returns the connection credentials.
4. Mapping the connection credentials into the application.

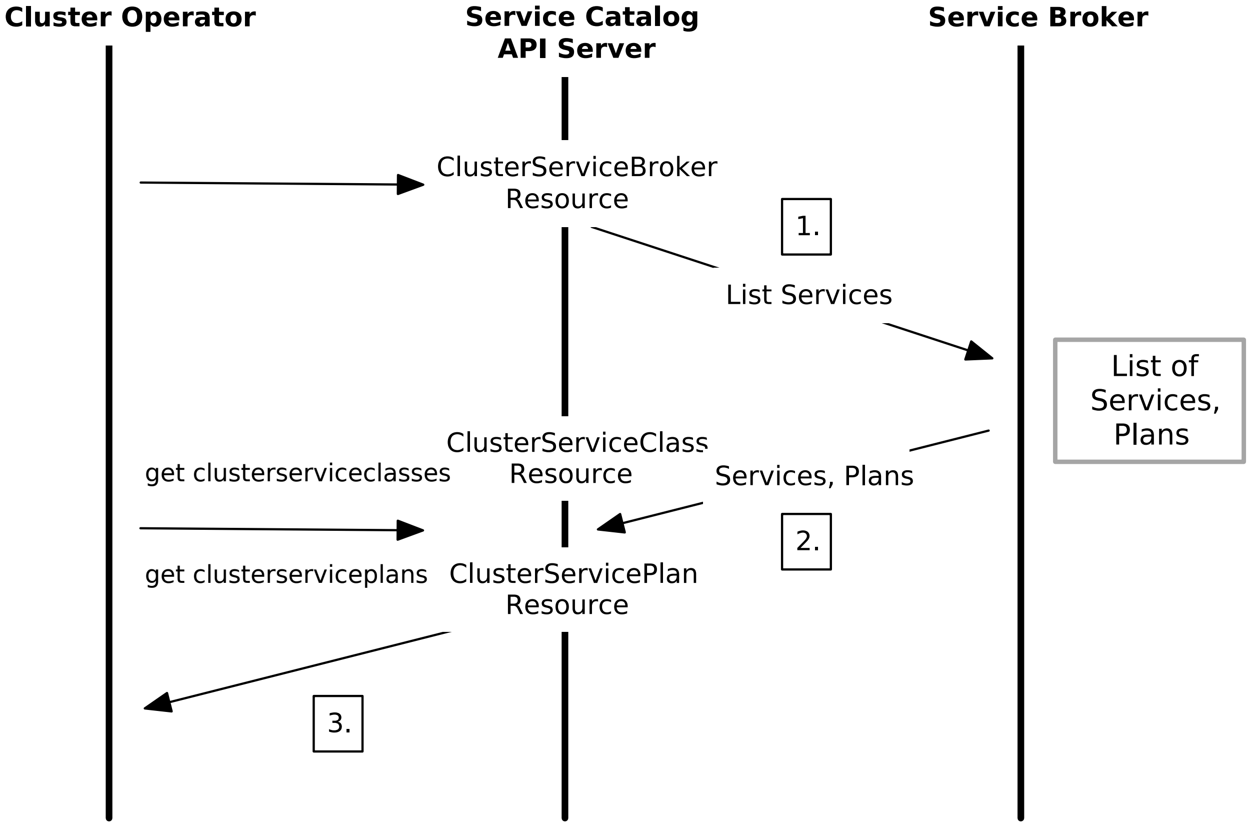
Listing managed services and Service Plans

First, a cluster operator must create a `ClusterServiceBroker` resource within the `servicecatalog.k8s.io` group. This resource contains the URL and connection details necessary to access a service broker endpoint.

This is an example of a `ClusterServiceBroker` resource:

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ClusterServiceBroker
metadata:
  name: cloud-broker
spec:
  # Points to the endpoint of a service broker. (This example is not a working URL.)
  url:  https://servicebroker.somecloudprovider.com/v1alpha1/projects/service-catalog/brokers/default
  #####
  # Additional values can be added here, which may be used to communicate
  # with the service broker, such as bearer token info or a caBundle for TLS.
  #####
```

The following is a sequence diagram illustrating the steps involved in listing managed services and Plans available from a service broker:


Cluster Operator
Service Catalog API Server
Service Broker
ClusterServiceBroker Resource
1.
List Services
List of Services, Plans
ClusterServiceClass Resource
get clusterserviceclasses
Services, Plans
2.
get clusterserviceplans
ClusterServicePlan Resource
3.

1. Once the `ClusterServiceBroker` resource is added to Service Catalog, it triggers a call to the external service broker for a list of available services.
2. The service broker returns a list of available managed services and a list of Service Plans, which are cached locally as `ClusterServiceClass` and `ClusterServicePlan` resources respectively.

3. A cluster operator can then get the list of available managed services using the following command:

```
kubectl get clusterserviceclasses -o=custom-columns=SERVICE\
NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName
```

It should output a list of service names with a format similar to:

```
SERVICE NAME                            EXTERNAL NAME
4f6e6cf6-ffdd-425f-a2c7-3c9258ad2468    cloud-provider-service
...                                     ...
```

They can also view the Service Plans available using the following command:

```
kubectl get clusterserviceplans -o=custom-columns=PLAN\
NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName
```

It should output a list of plan names with a format similar to:

```
PLAN NAME                              EXTERNAL NAME
86064792-7ea2-467b-af93-ac9694d96d52   service-plan-name
...                                    ...
```

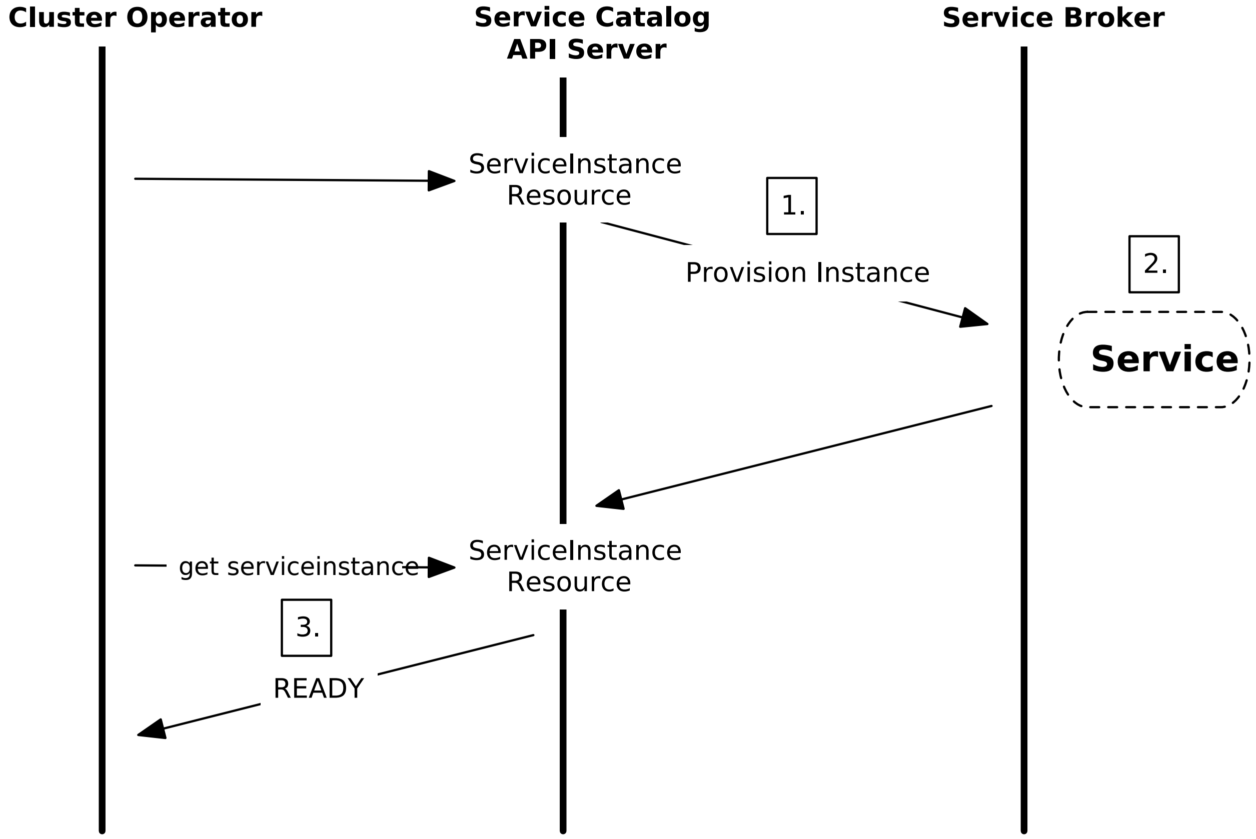
Provisioning a new instance

A cluster operator can initiate the provisioning of a new instance by creating a `ServiceInstance` resource.

This is an example of a `ServiceInstance` resource:

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceInstance
metadata:
  name: cloud-queue-instance
  namespace: cloud-apps
spec:
  # References one of the previously returned services
  clusterServiceClassExternalName: cloud-provider-service
  clusterServicePlanExternalName: service-plan-name
  #####
  # Additional parameters can be added here,
  # which may be used by the service broker.
  #####
```

The following sequence diagram illustrates the steps involved in provisioning a new instance of a managed service:



1. When the `ServiceInstance` resource is created, Service Catalog initiates a call to the external service broker to provision an instance of the service.
2. The service broker creates a new instance of the managed service and returns an HTTP response.
3. A cluster operator can then check the status of the instance to see if it is ready.

Binding to a managed service

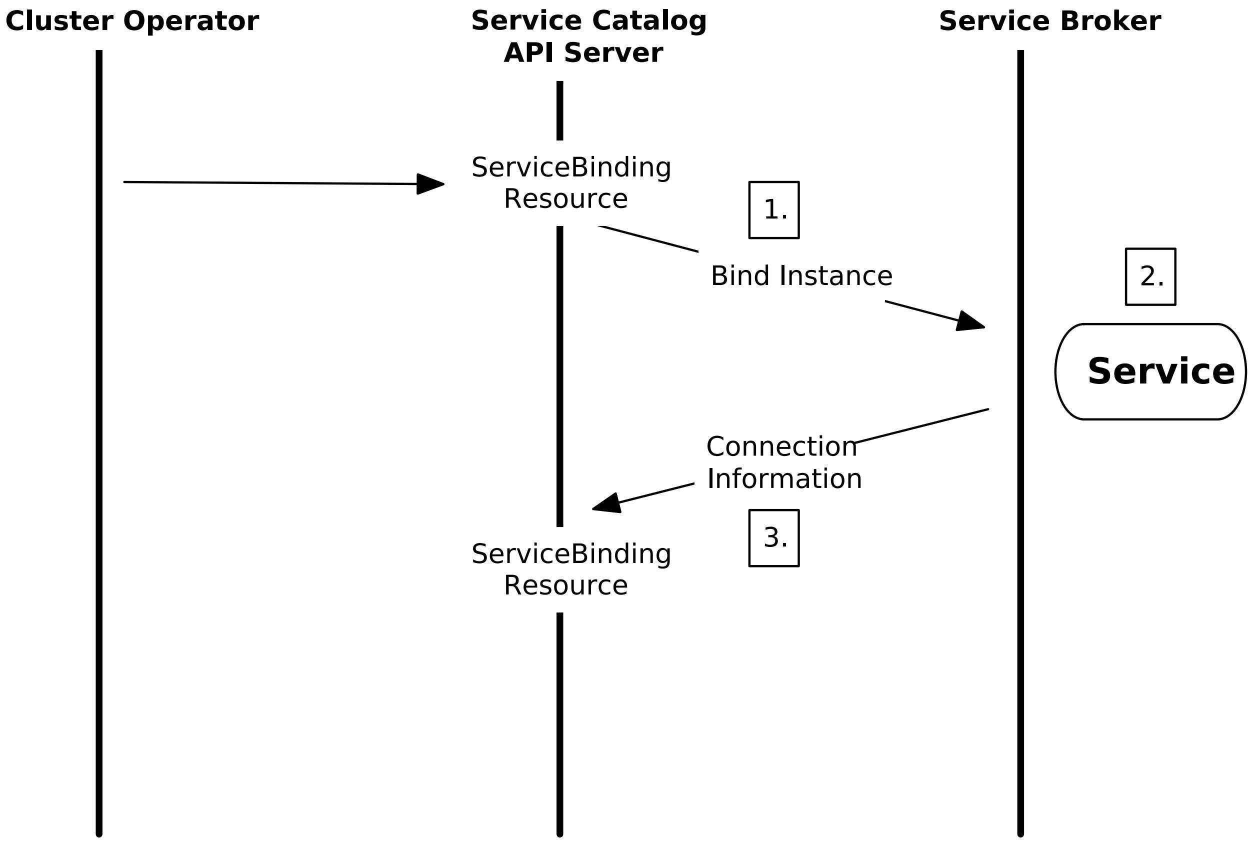
After a new instance has been provisioned, a cluster operator must bind to the managed service to get the connection credentials and service account details necessary for the application to use the service. This is done by creating a `ServiceBinding` resource.

The following is an example of a `ServiceBinding` resource:

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceBinding
metadata:
  name: cloud-queue-binding
  namespace: cloud-apps
spec:
  instanceRef:
    name: cloud-queue-instance
  #####
  # Additional information can be added here, such as a
secretName or
```

```
  # service account parameters, which may be used by the service broker.
  #####
```

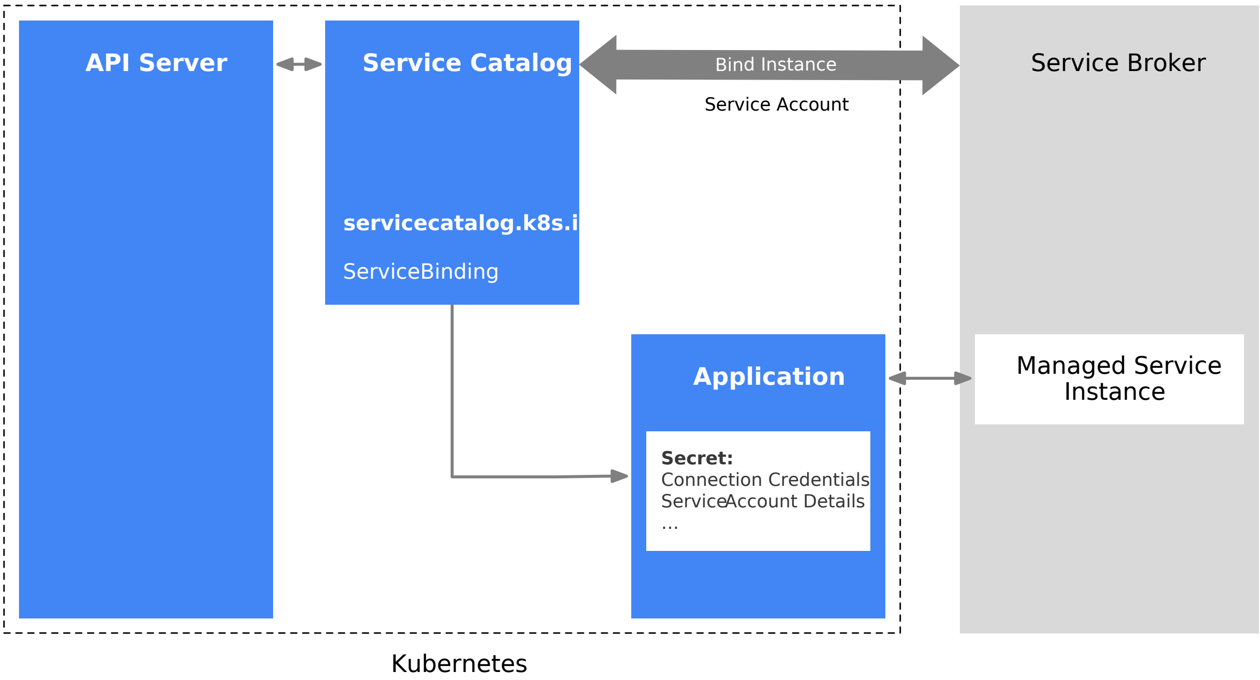
The following sequence diagram illustrates the steps involved in binding to a managed service instance:



1. After the `ServiceBinding` is created, Service Catalog makes a call to the external service broker requesting the information necessary to bind with the service instance.
2. The service broker enables the application permissions/roles for the appropriate service account.
3. The service broker returns the information necessary to connect and access the managed service instance. This is provider and service-specific so the information returned may differ between Service Providers and their managed services.

Mapping the connection credentials

After binding, the final step involves mapping the connection credentials and service-specific information into the application. These pieces of information are stored in secrets that the application in the cluster can access and use to connect directly with the managed service.



Pod configuration File

One method to perform this mapping is to use a declarative Pod configuration.

The following example describes how to map service account credentials into the application. A key called `sa-key` is stored in a volume named `provider-cloud-key`, and the application mounts this volume at `/var/secrets/provider/key.json`. The environment variable `PROVIDER_APPLICATION_CREDENTIALS` is mapped from the value of the mounted file.

```
...
    spec:
      volumes:
        - name: provider-cloud-key
          secret:
            secretName: sa-key
      containers:
...
          volumeMounts:
          - name: provider-cloud-key
            mountPath: /var/secrets/provider
          env:
          - name: PROVIDER_APPLICATION_CREDENTIALS
            value: "/var/secrets/provider/key.json"
```

The following example describes how to map secret values into application environment variables. In this example, the messaging queue topic name is mapped from a secret named `provider-queue-credentials` with a key named `topic` to the environment variable `TOPIC`.

```
...
          env:
          - name: "TOPIC"
            valueFrom:
                secretKeyRef:
                   name: provider-queue-credentials
                   key: topic
```

What's next

- If you are familiar with Helm ChartsA package of pre-configured Kubernetes resources that can be managed with the Helm tool. , install Service Catalog using Helm into your Kubernetes cluster. Alternatively, you can install Service Catalog using the SC tool.
- View sample service brokers.
- Explore the kubernetes-incubator/service-catalog project.
- View svc-cat.io.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Create an Issue Edit This Page
Page last modified on November 27, 2018 at 5:29 PM PST by fix some 404 errors (#11352) (Page History)

Edit This Page

Poseidon-Firmament - An alternate scheduler

Current release of Poseidon-Firmament scheduler is an `alpha` release.

Poseidon-Firmament scheduler is an alternate scheduler that can be deployed alongside the default Kubernetes scheduler.

Introduction

Poseidon is a service that acts as the integration glue for the Firmament scheduler with Kubernetes. Poseidon-Firmament scheduler augments the current Kubernetes scheduling capabilities. It incorporates novel flow network graph based scheduling capabilities alongside the default Kubernetes Scheduler. Firmament scheduler models workloads and clusters as flow networks and runs min-cost flow optimizations over these networks to make scheduling decisions.

It models the scheduling problem as a constraint-based optimization over a flow network graph. This is achieved by reducing scheduling to a min-cost max-flow optimization problem. The Poseidon-Firmament scheduler dynamically refines the workload placements.

Poseidon-Firmament scheduler runs alongside the default Kubernetes Scheduler as an alternate scheduler, so multiple schedulers run simultaneously.

Key Advantages

Flow graph scheduling based Poseidon-Firmament scheduler provides the following key advantages:

- Workloads (pods) are bulk scheduled to enable scheduling at massive scale..
- Based on the extensive performance test results, Poseidon-Firmament scales much better than the Kubernetes default scheduler as the number of nodes increase in a cluster. This is due to the fact that Poseidon-Firmament is able to amortize more and more work across workloads.

- Poseidon-Firmament Scheduler outperforms the Kubernetes default scheduler by a wide margin when it comes to throughput performance numbers for scenarios where compute resource requirements are somewhat uniform across jobs (Replicasets/Deployments/Jobs). Poseidon-Firmament scheduler end-to-end throughput performance numbers, including bind time, consistently get better as the number of nodes in a cluster increase. For example, for a 2,700 node cluster (shown in the graphs here), Poseidon-Firmament scheduler achieves a

7X or greater end-to-end throughput than the Kubernetes default scheduler, which includes bind time.

- Availability of complex rule constraints.

- Scheduling in Poseidon-Firmament is dynamic; it keeps cluster resources in a global optimal state during every scheduling run.

- Highly efficient resource utilizations.

Poseidon-Firmament Scheduler - How it works

As part of the Kubernetes multiple schedulers support, each new pod is typically scheduled by the default scheduler. Kubernetes can be instructed to use another scheduler by specifying the name of another custom scheduler ("poseidon" in our case) in the **schedulerName** field of the PodSpec at the time of pod creation. In this case, the default scheduler will ignore that Pod and allow Poseidon scheduler to schedule the Pod on a relevant node.

```
apiVersion: v1
kind: Pod

...
spec:
    schedulerName: poseidon
```

> **Note:** For details about the design of this project see the design document.

Possible Use Case Scenarios - When to use it

As mentioned earlier, Poseidon-Firmament scheduler enables an extremely high throughput scheduling environment at scale due to its bulk scheduling approach versus Kubernetes pod-at-a-time approach. In our extensive tests, we have observed substantial throughput benefits as long as resource requirements (CPU/Memory) for incoming Pods are uniform across jobs (Replicasets/Deployments/Jobs), mainly due to efficient amortization of work across jobs.

Although, Poseidon-Firmament scheduler is capable of scheduling various types of workloads, such as service, batch, etc., the following are a few use cases where it excels the most:

1. For "Big Data/AI" jobs consisting of large number of tasks, throughput benefits are tremendous.
2. Service or batch jobs where workload resource requirements are uniform across jobs (Replicasets/Deployments/Jobs).

Current Project Stage

- **Alpha Release - Incubation repo.** at https://github.com/kubernetes-sigs/poseidon.
- Currently, Poseidon-Firmament scheduler **does not provide support for high availability**, our implementation assumes that the scheduler cannot fail. The design document describes possible ways to enable high availability, but we leave this to future work.
- We are **not aware of any production deployment** of Poseidon-Firmament scheduler at this time.
- Poseidon-Firmament is supported from Kubernetes release 1.6 and works with all subsequent releases.
- Release process for Poseidon and Firmament repos are in lock step. The current Poseidon release can be found here and the corresponding Firmament release can be found here.

Features Comparison Matrix

Feature	**Kubernetes Default Scheduler**	**Poseidon-Firmament Scheduler**	**Notes**
Node Affinity/ Anti-Affinity	Y	Y	
Pod Affinity/ Anti-Affinity - including support for pod anti-affinity symmetry	Y	Y	Currently, the default scheduler outperforms the Poseidon-Firmament scheduler pod affinity/anti-affinity functionality. We are working towards resolving this.
Taints & Tolerations	Y	Y	
Baseline Scheduling capability in accordance to available compute resources (CPU & Memory) on a node	Y	Y**	Not all Predicates & Priorities are supported at this time.

Feature	Kubernetes Default Scheduler	Poseidon-Firmament Scheduler	Notes
Extreme Throughput at scale	Y**	Y	Bulk scheduling approach scales or increases workload placement. Substantial throughput benefits using Firmament scheduler as long as resource requirements (CPU/Memory) for incoming Pods is uniform across Replicasets/Deployments/Jobs. This is mainly due to efficient amortization of work across Replicasets/Deployments/Jobs . 1) For "Big Data/AI" jobs consisting of large no. of tasks, throughput benefits are tremendous. 2) Substantial throughput benefits also for service or batch job scenarios where workload resource requirements are uniform across Replicasets/ Deployments/Jobs.
Optimal Scheduling	Pod-by-Pod scheduler, processes one pod at a time (may result into sub-optimal scheduling)	Bulk Scheduling (Optimal scheduling)	Pod-by-Pod Kubernetes default scheduler may assign tasks to a sub-optimal machine. By contrast, Firmament considers all unscheduled tasks at the same time together with their soft and hard constraints.
Colocation Interference Avoidance	N	N**	Planned in Poseidon-Firmament.
Priority Pre-emption	Y	N**	Partially exists in Poseidon-Firmament versus extensive support in Kubernetes default scheduler.
Inherent Re-Scheduling	N	Y**	Poseidon-Firmament scheduler supports workload re-scheduling. In each scheduling run it considers all the pods, including running pods, and as a result can migrate or evict pods - a globally optimal scheduling environment.
Gang Scheduling	N	Y	

Feature	**Kubernetes Default Scheduler**	**Poseidon-Firmament Scheduler**	**Notes**
Support for Pre-bound Persistence Volume Scheduling	Y	Y	
Support for Local Volume & Dynamic Persistence Volume Binding Scheduling	Y	N**	Planned.
High Availability	Y	N**	Planned.
Real-time metrics based scheduling	N	Y**	Initially supported using Heapster (now deprecated) for placing pods using actual cluster utilization statistics rather than reservations. Plans to switch over to "metric server".
Support for Max-Pod per node	Y	Y	Poseidon-Firmament scheduler seamlessly co-exists with Kubernetes default scheduler.
Support for Ephemeral Storage, in addition to CPU/Memory	Y	Y	

Installation

For in-cluster installation of Poseidon, please start at the Installation instructions.

Development

For developers, please refer to the Developer Setup instructions.

Latest Throughput Performance Testing Results

Pod-by-pod schedulers, such as the Kubernetes default scheduler, typically process one pod at a time. These schedulers have the following crucial drawbacks:

1. The scheduler commits to a pod placement early and restricts the choices for other pods that wait to be placed.
2. There is limited opportunities for amortizing work across pods because they are considered for placement individually.

These downsides of pod-by-pod schedulers are addressed by batching or bulk scheduling in Poseidon-Firmament scheduler. Processing several pods in a batch allows the scheduler to jointly consider their placement, and thus to find the best trade-off for the whole batch instead of one pod. At the same time it amortizes work across pods resulting in much higher throughput.

> **Note:** Please refer to the latest benchmark results for detailed throughput performance comparison test results between Poseidon-Firmament scheduler and the Kubernetes default scheduler.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.