

Pods

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

A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

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

What is a Pod?

Note: You need to install a [container runtime](#) into each node in the cluster so that Pods can run there.

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 container. Within a Pod's context, the individual applications may have further sub-isolations applied.

A Pod is similar to a set of containers with shared namespaces and shared filesystem volumes.

Using Pods

The following is an example of a Pod which consists of a container running the image `nginx:1.14.2`.

[pods/simple-pod.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

To create the Pod shown above, run the following command:

```
kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml
```

Pods are generally not created directly and are created using workload resources. See [Working with Pods](#) for more information on how Pods are used with workload resources.

Workload resources for managing pods

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

Pods in a Kubernetes cluster are used in two main ways:

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

Note: Grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled.

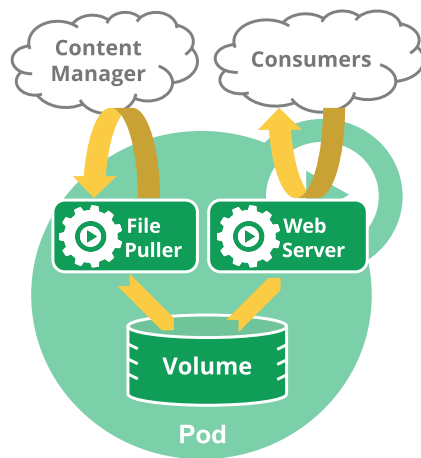
Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as *replication*. Replicated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

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

How Pods manage multiple containers

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

For example, you might have a container that acts as a web server for files in a shared volume, and a separate "sidecar" container that updates those files from a remote source, as in the following diagram:



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

Pods natively provide two kinds of shared resources for their constituent containers: [networking](#) and [storage](#).

Working with Pods

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

Note: Restarting a container in a Pod should not be confused with restarting a Pod. A Pod is not a process, but an environment for running container(s). A Pod persists until it is deleted.

The name of a Pod must be a valid [DNS subdomain](#) value, but this can produce unexpected results for the Pod hostname. For best compatibility, the name should follow the more restrictive rules for a [DNS label](#).

Pod OS

FEATURE STATE: [Kubernetes v1.25](#) [\[stable\]](#)

You should set the `.spec.os.name` field to either `windows` or `linux` to indicate the OS on which you want the pod to run. These two are the only operating systems supported for now by Kubernetes. In future, this list may be expanded.

In Kubernetes v1.27, the value you set for this field has no effect on scheduling of the pods. Setting the `.spec.os.name` helps to identify the pod OS authoritatively and is used for validation. The kubelet refuses to run a Pod where you have specified a Pod OS, if this isn't the same as the operating system for the node where that kubelet is running. The [Pod security standards](#) also use this field to avoid enforcing policies that aren't relevant to that operating system.

Pods and controllers

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

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

- [Deployment](#)
- [StatefulSet](#)
- [DaemonSet](#)

Pod templates

Controllers for [workload](#) resources create Pods from a *pod template* and manage those Pods on your behalf.

PodTemplates are specifications for creating Pods, and are included in workload resources such as [Deployments](#), [Jobs](#), and [DaemonSets](#).

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

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

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
        - name: hello
          image: busybox:1.28
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
          restartPolicy: OnFailure
    # The pod template ends here
```

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

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

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

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

Pod update and replacement

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

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

- Most of the metadata about a Pod is immutable. For example, you cannot change the `namespace`, `name`, `uid`, or `creationTimestamp` fields; the `generation` field is unique. It only accepts updates that increment the field's current value.
- If the `metadata.deletionTimestamp` is set, no new entry can be added to the `metadata.finalizers` list.
- Pod updates may not change fields other than `spec.containers[*].image`, `spec.initContainers[*].image`, `spec.activeDeadlineSeconds` or `spec.tolerations`. For `spec.tolerations`, you can only add new entries.
- When updating the `spec.activeDeadlineSeconds` field, two types of updates are allowed:
 1. setting the unassigned field to a positive number;
 2. updating the field from a positive number to a smaller, non-negative number.

Resource sharing and communication

Pods enable data sharing and communication among their constituent containers.

Storage in Pods

A Pod can specify a set of shared storage volumes. All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See [Storage](#) for more information on how Kubernetes implements shared storage and makes it available to Pods.

Pod networking

Each Pod is assigned a unique IP address for each address family. Every container in a Pod shares the network namespace, including the IP address and network ports. Inside a Pod (and **only** then), the containers that belong to the Pod can communicate with one another using `localhost`. When containers in a Pod communicate with entities *outside the Pod*, they must coordinate how they use the shared network resources (such as ports). Within a Pod, containers share an IP address and port space, and can find each other via `localhost`. The containers in a Pod can also communicate with each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. Containers in different Pods have distinct IP addresses and can not communicate by OS-level IPC without special configuration. Containers that want to interact with a container running in a different Pod can use IP networking to communicate.

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

Privileged mode for containers

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

Any container in a pod can run in privileged mode to use operating system administrative capabilities that would otherwise be inaccessible. This is available for both Windows and Linux.

Linux privileged containers

In Linux, any container in a Pod can enable privileged mode using the `privileged` (Linux) flag on the [security context](#) of the container spec. This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices.

Windows privileged containers

FEATURE STATE: [Kubernetes v1.26](#) [\[stable\]](#)

In Windows, you can create a [Windows HostProcess pod](#) by setting the `windowsOptions.hostProcess` flag on the security context of the pod spec. All containers in these pods must run as Windows HostProcess containers. HostProcess pods run directly on the host and can also be used to perform administrative tasks as is done with Linux privileged containers.

Static Pods

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

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

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

Note: The `spec` of a static Pod cannot refer to other API objects (e.g., [ServiceAccount](#), [ConfigMap](#), [Secret](#), etc).

Container probes

A *probe* is a diagnostic performed periodically by the kubelet on a container. To perform a diagnostic, the kubelet can invoke different actions:

- `ExecAction` (performed with the help of the container runtime)
- `TCPSocketAction` (checked directly by the kubelet)
- `HTTPGetAction` (checked directly by the kubelet)

You can read more about [probes](#) in the Pod Lifecycle documentation.

What's next

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

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

- [Aurora](#)
- [Borg](#)
- [Marathon](#)
- [Omega](#)
- [Tupperware](#).

Feedback

Was this page helpful?

☐ Yes ☐ No

Last modified July 07, 2023 at 6:28 PM PST: [Rewrite note on container runtimes \(dcd3337cbc\)](#)