Containers

Technology for packaging an application along with its runtime dependencies.

- 1: Images
- 2: Container Environment
- 3: Runtime Class
- 4: Container Lifecycle Hooks

Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it.

Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments.

Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node.

Container images

A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.

Containers are intended to be stateless and immutable: you should not change the code of a container that is already running. If you have a containerized application and want to make changes, the correct process is to build a new image that includes the change, then recreate the container to start from the updated image.

Container runtimes

A fundamental component that empowers Kubernetes to run containers effectively. It is responsible for managing the execution and lifecycle of containers within the Kubernetes environment.

Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

Usually, you can allow your cluster to pick the default container runtime for a Pod. If you need to use more than one container runtime in your cluster, you can specify the RuntimeClass for a Pod to make sure that Kubernetes runs those containers using a particular container runtime.

You can also use RuntimeClass to run different Pods with the same container runtime but with different settings.

1 - Images

A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment.

You typically create a container image of your application and push it to a registry before referring to it in a Pod.

This page provides an outline of the container image concept.

> **Note:** If you are looking for the container images for a Kubernetes release (such as v1.29, the latest minor release), visit Download Kubernetes.

Image names

Container images are usually given a name such as `pause`, `example/mycontainer`, or `kube-apiserver`. Images can also include a registry hostname; for example: `fictional.registry.example/imagename`, and possibly a port number as well; for example: `fictional.registry.example:10443/imagename`.

If you don't specify a registry hostname, Kubernetes assumes that you mean the Docker public registry.

After the image name part you can add a *tag* (in the same way you would when using with commands like `docker` or `podman`). Tags let you identify different versions of the same series of images.

Image tags consist of lowercase and uppercase letters, digits, underscores (`_`), periods (`.`), and dashes (`-`).
There are additional rules about where you can place the separator characters (`_`, `-`, and `.`) inside an image tag.
If you don't specify a tag, Kubernetes assumes you mean the tag `latest`.

Updating images

When you first create a Deployment, StatefulSet, Pod, or other object that includes a Pod template, then by default the pull policy of all containers in that pod will be set to `IfNotPresent` if it is not explicitly specified. This policy causes the kubelet to skip pulling an image if it already exists.

Image pull policy

The `imagePullPolicy` for a container and the tag of the image affect when the kubelet attempts to pull (download) the specified image.

Here's a list of the values you can set for `imagePullPolicy` and the effects these values have:

`IfNotPresent`

the image is pulled only if it is not already present locally.

`Always`

every time the kubelet launches a container, the kubelet queries the container image registry to resolve the name to an image digest. If the kubelet has a container image with that exact digest cached locally, the kubelet uses its cached image; otherwise, the kubelet pulls the image with the resolved digest, and uses that image to launch the container.

`Never`

the kubelet does not try fetching the image. If the image is somehow already present locally, the kubelet attempts to start the container; otherwise, startup fails. See pre-pulled images for more details.

The caching semantics of the underlying image provider make even `imagePullPolicy: Always` efficient, as long as the registry is reliably accessible. Your container runtime can notice that the image layers already exist on the node so that they don't need to be downloaded again.

> **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.
>
> Instead, specify a meaningful tag such as `v1.42.0` and/or a digest.

To make sure the Pod always uses the same version of a container image, you can specify the image's digest; replace `<image-name>:<tag>` with `<image-name>@<digest>` (for example, `image@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2`).

When using image tags, if the image registry were to change the code that the tag on that image represents, you might end up with a mix of Pods running the old and new code. An image digest uniquely identifies a specific version of the image, so Kubernetes runs the same code every time it starts a container with that image name and digest specified. Specifying an image by digest fixes the code that you run so that a change at the registry cannot lead to that mix of versions.

There are third-party admission controllers that mutate Pods (and pod templates) when they are created, so that the running workload is defined based on an image digest rather than a tag. That might be useful if you want to make sure that all your workload is running the same code no matter what tag changes happen at the registry.

Default image pull policy

When you (or a controller) submit a new Pod to the API server, your cluster sets the `imagePullPolicy` field when specific conditions are met:

- if you omit the `imagePullPolicy` field, and you specify the digest for the container image, the `imagePullPolicy` is automatically set to `IfNotPresent`.
- if you omit the `imagePullPolicy` field, and the tag for the container image is `:latest`, `imagePullPolicy` is automatically set to `Always`;
- if you omit the `imagePullPolicy` field, and you don't specify the tag for the container image, `imagePullPolicy` is automatically set to `Always`;
- if you omit the `imagePullPolicy` field, and you specify the tag for the container image that isn't `:latest`, the `imagePullPolicy` is automatically set to `IfNotPresent`.

> **Note:**
> The value of `imagePullPolicy` of the container is always set when the object is first *created*, and is not updated if the image's tag or digest later changes.
>
> For example, if you create a Deployment with an image whose tag is *not* `:latest`, and later update that Deployment's image to a `:latest` tag, the `imagePullPolicy` field will *not* change to `Always`. You must manually change the pull policy of any object after its initial creation.

Required image pull

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; Kubernetes will set the policy to `Always` when you submit the Pod.
- Omit the `imagePullPolicy` and the tag for the image to use; Kubernetes will set the policy to `Always` when you submit the Pod.
- Enable the AlwaysPullImages admission controller.

ImagePullBackOff

When a kubelet starts creating containers for a Pod using a container runtime, it might be possible the container is in Waiting state because of `ImagePullBackOff`.

The status `ImagePullBackOff` means that a container could not start because Kubernetes could not pull a container image (for reasons such as invalid image name, or pulling from a private registry without `imagePullSecret`). The `BackOff` part indicates that Kubernetes will keep trying to pull the image, with an increasing back-off delay.

Kubernetes raises the delay between each attempt until it reaches a compiled-in limit, which is 300 seconds (5 minutes).

Image pull per runtime class

FEATURE STATE: `Kubernetes v1.29 [alpha]`

Kubernetes includes alpha support for performing image pulls based on the RuntimeClass of a Pod.

If you enable the `RuntimeClassInImageCriApi` feature gate, the kubelet references container images by a tuple of (image name, runtime handler) rather than just the image name or digest. Your container runtime may adapt its behavior based on the selected runtime handler. Pulling images based on runtime class will be helpful for VM based containers like windows hyperV containers.

Serial and parallel image pulls

By default, kubelet pulls images serially. In other words, kubelet sends only one image pull request to the image service at a time. Other image pull requests have to wait until the one being processed is complete.

Nodes make image pull decisions in isolation. Even when you use serialized image pulls, two different nodes can pull the same image in parallel.

If you would like to enable parallel image pulls, you can set the field `serializeImagePulls` to false in the kubelet configuration. With `serializeImagePulls` set to false, image pull requests will be sent to the image service immediately, and multiple images will be pulled at the same time.

When enabling parallel image pulls, please make sure the image service of your container runtime can handle parallel image pulls.

The kubelet never pulls multiple images in parallel on behalf of one Pod. For example, if you have a Pod that has an init container and an application container, the image pulls for the two containers will not be parallelized. However, if you have two Pods that use different images, the kubelet pulls the images in parallel on behalf of the two different Pods, when parallel image pulls is enabled.

Maximum parallel image pulls

FEATURE STATE: `Kubernetes v1.27 [alpha]`

When `serializeImagePulls` is set to false, the kubelet defaults to no limit on the maximum number of images being pulled at the same time. If you would like to limit the number of parallel image pulls, you can set the field `maxParallelImagePulls` in kubelet configuration. With `maxParallelImagePulls` set to *n*, only *n* images can be pulled at the same time, and any image pull beyond *n* will have to wait until at least one ongoing image pull is complete.

Limiting the number parallel image pulls would prevent image pulling from consuming too much network bandwidth or disk I/O, when parallel image pulling is enabled.

You can set `maxParallelImagePulls` to a positive number that is greater than or equal to 1. If you set `maxParallelImagePulls` to be greater than or equal to 2, you must set the `serializeImagePulls` to false. The kubelet will fail to start with invalid `maxParallelImagePulls` settings.

Multi-architecture images with image indexes

As well as providing binary images, a container registry can also serve a container image index. An image index can point to multiple image manifests for architecture-specific versions of a container. The idea is that you can have a name for an image (for example: `pause`, `example/mycontainer`, `kube-apiserver`) and allow different systems to fetch the right binary image for the machine architecture they are using.

Kubernetes itself typically names container images with a 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:

- Configuring Nodes to Authenticate to a Private Registry
  - all pods can read any configured private registries
  - requires node configuration by cluster administrator

- Kubelet Credential Provider to dynamically fetch credentials for private registries
  - kubelet can be configured to use credential provider exec plugin for the respective private registry.
- Pre-pulled Images
  - all pods can use any images cached on a node
  - requires root access to all nodes to set up
- Specifying ImagePullSecrets on a Pod
  - only pods which provide own keys can access the private registry
- Vendor-specific or local extensions
  - if you're using a custom node configuration, you (or your cloud provider) can implement your mechanism for authenticating the node to the container registry.

These options are explained in more detail below.

Configuring nodes to authenticate to a private registry

Specific instructions for setting credentials depends on the container runtime and registry you chose to use. You should refer to your solution's documentation for the most accurate information.

For an example of configuring a private container image registry, see the Pull an Image from a Private Registry task. That example uses a private registry in Docker Hub.

Kubelet credential provider for authenticated image pulls

> **Note:** This approach is especially suitable when kubelet needs to fetch registry credentials dynamically. Most commonly used for registries provided by cloud providers where auth tokens are short-lived.

You can configure the kubelet to invoke a plugin binary to dynamically fetch registry credentials for a container image. This is the most robust and versatile way to fetch credentials for private registries, but also requires kubelet-level configuration to enable.

See Configure a kubelet image credential provider for more details.

Interpretation of config.json

The interpretation of `config.json` varies between the original Docker implementation and the Kubernetes interpretation. In Docker, the `auths` keys can only specify root URLs, whereas Kubernetes allows glob URLs as well as prefix-matched paths. The only limitation is that glob patterns (`*`) have to include the dot (`.`) for each subdomain. The amount of matched subdomains has to be equal to the amount of glob patterns (`*.`), for example:

- `*.kubernetes.io` will *not* match `kubernetes.io` , but `abc.kubernetes.io`
- `*.*.kubernetes.io` will *not* match `abc.kubernetes.io` , but `abc.def.kubernetes.io`
- `prefix.*.io` will match `prefix.kubernetes.io`
- `*-good.kubernetes.io` will match `prefix-good.kubernetes.io`

This means that a `config.json` like this is valid:

```
{
    "auths": {
        "my-registry.io/images": { "auth": "…" },
        "*.my-registry.io/images": { "auth": "…" }
    }
}
```

Image pull operations would now pass the credentials to the CRI container runtime for every valid pattern. For example the following container image names would match successfully:

- `my-registry.io/images`
- `my-registry.io/images/my-image`
- `my-registry.io/images/another-image`
- `sub.my-registry.io/images/my-image`

But not:

- `a.sub.my-registry.io/images/my-image`
- `a.b.sub.my-registry.io/images/my-image`

The kubelet performs image pulls sequentially for every found credential. This means, that multiple entries in `config.json` for different paths are possible, too:

```
{
    "auths": {
        "my-registry.io/images": {
            "auth": "…"
        },
        "my-registry.io/images/subpath": {
            "auth": "…"
        }
    }
}
```

If now a container specifies an image `my-registry.io/images/subpath/my-image` to be pulled, then the kubelet will try to download them from both authentication sources if one of them fails.

Pre-pulled images

> **Note:** This approach is suitable if you can control node configuration. It will not work reliably if your cloud provider manages nodes and replaces them automatically.

By default, the kubelet tries 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 is the recommended approach to run containers based on images in private registries.

Kubernetes supports specifying container image registry keys on a Pod. `imagePullSecrets` must all be in the same namespace as the Pod. The referenced Secrets must be of type `kubernetes.io/dockercfg` or `kubernetes.io/dockerconfigjson`.

Creating a Secret with a Docker config

You need to know the username, registry password and client email address for authenticating to the registry, as well as its hostname. 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 Secrets.
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 only works 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. Each item in the `imagePullSecrets` array can only reference a Secret in the same namespace.

For example:

```
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.

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 from a public registry
     - No configuration required.
     - Some cloud providers automatically cache or mirror public images, which improves availability and reduces the time to pull images.
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 registry
     - Manual configuration may be required on the nodes that need to access to private registry
   - Or, run an internal private registry behind your firewall with open read access.
     - No Kubernetes configuration is required.
   - Use a hosted container image registry service that controls image access
     - 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.

Legacy built-in kubelet credential provider

In older versions of Kubernetes, the kubelet had a direct integration with cloud provider credentials. This gave it the ability to dynamically fetch credentials for image registries.

There were three built-in implementations of the kubelet credential provider integration: ACR (Azure Container Registry), ECR (Elastic Container Registry), and GCR (Google Container Registry).

For more information on the legacy mechanism, read the documentation for the version of Kubernetes that you are using. Kubernetes v1.26 through to v1.29 do not include the legacy mechanism, so you would need to either:

- configure a kubelet image credential provider on each node
- specify image pull credentials using `imagePullSecrets` and at least one Secret

What's next

- Read the OCI Image Manifest Specification.
- Learn about container image garbage collection.
- Learn more about pulling an Image from a Private Registry.

2 - Container Environment

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

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 container image.

Cluster information

A list of all services that were running when a Container was created is available to that Container as environment variables. This list is limited to services within the same namespace as the new Container's Pod and Kubernetes control plane services.

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.

What's next

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

3 - Runtime Class

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

This page describes the RuntimeClass resource and runtime selection mechanism.

RuntimeClass is a feature for selecting the container runtime configuration. The container runtime configuration is used to run a Pod's containers.

Motivation

You can set a different RuntimeClass between different Pods to provide a balance of performance versus security. For example, if part of your workload deserves a high level of information security assurance, you might choose to schedule those Pods so that they run in a container runtime that uses hardware virtualization. You'd then benefit from the extra isolation of the alternative runtime, at the expense of some additional overhead.

You can also use RuntimeClass to run different Pods with the same container runtime but with different settings.

Setup

1. Configure the CRI implementation on nodes (runtime dependent)
2. Create the corresponding RuntimeClass resources

1. Configure the CRI implementation on nodes

The configurations available through RuntimeClass are Container Runtime Interface (CRI) implementation dependent. See the corresponding documentation (below) for your CRI implementation for how to configure.

> **Note:** RuntimeClass assumes a homogeneous node configuration across the cluster by default (which means that all nodes are configured the same way with respect to container runtimes). To support heterogeneous node configurations, see Scheduling below.

The configurations have a corresponding `handler` name, referenced by the RuntimeClass. The handler must be a valid DNS label name.

2. Create the corresponding RuntimeClass resources

The configurations setup in step 1 should each have an associated `handler` name, which identifies the configuration. For each handler, create a corresponding RuntimeClass object.

The RuntimeClass resource currently only has 2 significant fields: the RuntimeClass name (`metadata.name`) and the handler (`handler`). The object definition looks like this:

```yaml
# RuntimeClass is defined in the node.k8s.io API group
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  # The name the RuntimeClass will be referenced by.
  # RuntimeClass is a non-namespaced resource.
  name: myclass
# The name of the corresponding CRI configuration
handler: myconfiguration
```

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

> **Note:** It is recommended that RuntimeClass write operations (create/update/patch/delete) be restricted to the cluster administrator. This is typically the default. See Authorization Overview for more details.

Usage

Once RuntimeClasses are configured for the cluster, you can specify a `runtimeClassName` in the Pod spec to use it. For example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  runtimeClassName: myclass
  # ...
```

This will instruct the kubelet to use the named RuntimeClass to run this pod. If the named RuntimeClass does not exist, or the CRI cannot run the corresponding handler, the pod will enter the `Failed` terminal phase. Look for a corresponding event for an error message.

If no `runtimeClassName` is specified, the default RuntimeHandler will be used, which is equivalent to the behavior when the RuntimeClass feature is disabled.

CRI Configuration

For more details on setting up CRI runtimes, see CRI installation.

containerd

Runtime handlers are configured through containerd's configuration at `/etc/containerd/config.toml`. Valid handlers are configured under the runtimes section:

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.${HANDLER_NAME}]
```

See containerd's config documentation for more details:

CRI-O

Runtime handlers are configured through CRI-O's configuration at `/etc/crio/crio.conf`. Valid handlers are configured under the crio.runtime table:

```
[crio.runtime.runtimes.${HANDLER_NAME}]
  runtime_path = "${PATH_TO_BINARY}"
```

See CRI-O's config documentation for more details.

Scheduling

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

By specifying the `scheduling` field for a RuntimeClass, you can set constraints to ensure that Pods running with this RuntimeClass are scheduled to nodes that support it. If `scheduling` is not set, this RuntimeClass is assumed to be supported by all nodes.

To ensure pods land on nodes supporting a specific RuntimeClass, that set of nodes should have a common label which is then selected by the `runtimeclass.scheduling.nodeSelector` field. The RuntimeClass's nodeSelector is merged with the pod's nodeSelector in admission, effectively taking the intersection of the set of nodes selected by each. If there is a conflict, the pod will be rejected.

If the supported nodes are tainted to prevent other RuntimeClass pods from running on the node, you can add `tolerations` to the RuntimeClass. As with the `nodeSelector`, the tolerations are merged with the pod's tolerations in admission, effectively taking the union of the set of nodes tolerated by each.

To learn more about configuring the node selector and tolerations, see Assigning Pods to Nodes.

Pod Overhead

FEATURE STATE: `Kubernetes v1.24 [stable]`

You can specify *overhead* resources that are associated with running a Pod. Declaring overhead allows the cluster (including the scheduler) to account for it when making decisions about Pods and resources.

Pod overhead is defined in RuntimeClass through the `overhead` field. Through the use of this field, you can specify the overhead of running pods utilizing this RuntimeClass and ensure these overheads are accounted for in Kubernetes.

What's next

- RuntimeClass Design
- RuntimeClass Scheduling Design
- Read about the Pod Overhead concept
- PodOverhead Feature Design

4 - 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

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 is executed 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 a liveness/startup probe failure, preemption, resource contention and others. A call to the `PreStop` hook fails if the container is already in a terminated or completed state and the hook must complete before the TERM signal to stop the container can be sent. The Pod's termination grace period countdown begins before the `PreStop` hook is executed, so regardless of the outcome of the handler, the container will eventually terminate within the Pod's termination grace period. 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 three 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.
- Sleep - Pauses the container for a specified duration. The "Sleep" action is available when the feature gate `PodLifecycleSleepAction` is enabled.

Hook handler execution

When a Container lifecycle management hook is called, the Kubernetes management system executes the handler according to the hook action, `httpGet` , `tcpSocket` and `sleep` are executed by the kubelet process, and `exec` is executed in the container.

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.

`PreStop` hooks are not executed asynchronously from the signal to stop the Container; the hook must complete its execution before the TERM signal can be sent. If a `PreStop` hook hangs during execution, the Pod's phase will be `Terminating` and remain there until the Pod is killed after its `terminationGracePeriodSeconds` expires. This grace period applies to the total time it takes for both the `PreStop` hook to execute and for the Container to stop normally. If, for example, `terminationGracePeriodSeconds` is 60, and the hook takes 55 seconds to complete, and the Container takes 10 seconds to stop normally after receiving the signal, then the Container will be killed before it can stop normally, since `terminationGracePeriodSeconds` is less than the total time (55+10) it takes for these two things to happen.

If either 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. To generate a failed `FailedPostStartHook` event yourself, modify the lifecycle-events.yaml file to change the postStart command to "badcommand" and apply it. Here is some example output of the resulting events you see from running `kubectl describe pod lifecycle-demo`:

```
Events:
  Type     Reason               Age              From               Message
  ----     ------               ----             ----               -------
  Normal   Scheduled            7s               default-scheduler  Successfully assigned default/lifecycle-demo to ip-XXX-XXX
  Normal   Pulled               6s               kubelet            Successfully pulled image "nginx" in 229.604315ms
  Normal   Pulling              4s (x2 over 6s)  kubelet            Pulling image "nginx"
  Normal   Created              4s (x2 over 5s)  kubelet            Created container lifecycle-demo-container
  Normal   Started              4s (x2 over 5s)  kubelet            Started container lifecycle-demo-container
  Warning  FailedPostStartHook  4s (x2 over 5s)  kubelet            Exec lifecycle hook ([badcommand]) for Container "lifecycl
  Normal   Killing              4s (x2 over 5s)  kubelet            FailedPostStartHook
  Normal   Pulled               4s               kubelet            Successfully pulled image "nginx" in 215.66395ms
  Warning  BackOff              2s (x2 over 3s)  kubelet            Back-off restarting failed container
```

What's next

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