Configuration

Resources that Kubernetes provides for configuring Pods.

1 - 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 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. When you need to update a running service without downtime, use a Deployment.

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.

- Use the Kubernetes common labels for common use cases. These standardized labels enrich the metadata in a way that allows tools, including `kubectl` and dashboard, to work in an interoperable way.

- 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`: 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 downloads (pulls) the image with the resolved digest, and uses that image to launch the container.
- `imagePullPolicy` is omitted and either the image tag is `:latest` or it is omitted: `imagePullPolicy` is automatically set to `Always`. Note that this will *not* be updated to `IfNotPresent` if the tag changes value.
- `imagePullPolicy` is omitted and the image tag is present but not `:latest`: `imagePullPolicy` is automatically set to `IfNotPresent`. Note that this will *not* be updated to `Always` if the tag is later removed or changed to `:latest`.
- `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; replace `<image-name>:<tag>` with `<image-name>@<digest>` (for example, `image@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, as long as the registry is reliably accessible. 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 create deployment` 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.

2 - ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

> **Caution:** ConfigMap does not provide secrecy or encryption. If the data you want to store are confidential, use a Secret rather than a ConfigMap, or use additional (third party) tools to keep your data private.

Motivation

Use a ConfigMap for setting configuration data separately from application code.

For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named `DATABASE_HOST`. Locally, you set that variable to `localhost`. In the cloud, you set it to refer to a Kubernetes Service that exposes the database component to your cluster. This lets you fetch a container image running in the cloud and debug the exact same code locally if needed.

A ConfigMap is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB. If you need to store settings that are larger than this limit, you may want to consider mounting a volume or use a separate database or file service.

ConfigMap object

A ConfigMap is an API object that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a `spec`, a ConfigMap has `data` and `binaryData` fields. These fields accept key-value pairs as their values. Both the `data` field and the `binaryData` are optional. The `data` field is designed to contain UTF-8 byte sequences while the `binaryData` field is designed to contain binary data as base64-encoded strings.

The name of a ConfigMap must be a valid DNS subdomain name.

Each key under the `data` or the `binaryData` field must consist of alphanumeric characters, `-`, `_` or `.`. The keys stored in `data` must not overlap with the keys in the `binaryData` field.

Starting from v1.19, you can add an `immutable` field to a ConfigMap definition to create an immutable ConfigMap.

ConfigMaps and Pods

You can write a Pod `spec` that refers to a ConfigMap and configures the container(s) in that Pod based on the data in the ConfigMap. The Pod and the ConfigMap must be in the same namespace.

Here's an example ConfigMap that has some keys with single values, and other keys where the value looks like a fragment of a configuration format.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
```

```
  ui_properties_file_name: "user-interface.properties"

  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Inside a container command and args
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

These different methods lend themselves to different ways of modeling the data being consumed. For the first three methods, the kubelet uses the data from the ConfigMap when it launches container(s) for a Pod.

The fourth method means you have to write code to read the ConfigMap and its data. However, because you're using the Kubernetes API directly, your application can subscribe to get updates whenever the ConfigMap changes, and react when that happens. By accessing the Kubernetes API directly, this technique also lets you access a ConfigMap in a different namespace.

Here's an example Pod that uses values from `game-demo` to configure a Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
                                     # from the key name in the ConfigMap.
          valueFrom:
            configMapKeyRef:
              name: game-demo           # The ConfigMap this value comes from.
              key: player_initial_lives # The key to fetch.
        - name: UI_PROPERTIES_FILE_NAME
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: ui_properties_file_name
      volumeMounts:
      - name: config
        mountPath: "/config"
        readOnly: true
  volumes:
    # You set volumes at the Pod level, then mount them into containers inside that
    - name: config
      configMap:
        # Provide the name of the ConfigMap you want to mount.
        name: game-demo
        # An array of keys from the ConfigMap to create as files
        items:
        - key: "game.properties"
          path: "game.properties"
```

```
        - key: "user-interface.properties"
          path: "user-interface.properties"
```

A ConfigMap doesn't differentiate between single line property values and multi-line file-like values. What matters is how Pods and other objects consume those values.

For this example, defining a volume and mounting it inside the `demo` container as `/config` creates two files, `/config/game.properties` and `/config/user-interface.properties`, even though there are four keys in the ConfigMap. This is because the Pod definition specifies an `items` array in the `volumes` section. If you omit the `items` array entirely, every key in the ConfigMap becomes a file with the same name as the key, and you get 4 files.

Using ConfigMaps

ConfigMaps can be mounted as data volumes. ConfigMaps can also be used by other parts of the system, without being directly exposed to the Pod. For example, ConfigMaps can hold data that other parts of the system should use for configuration.

The most common way to use ConfigMaps is to configure settings for containers running in a Pod in the same namespace. You can also use a ConfigMap separately.

For example, you might encounter addons or operators that adjust their behavior based on a ConfigMap.

Using ConfigMaps as files from a Pod

To consume a ConfigMap in a volume in a Pod:

1. Create a ConfigMap or use an existing one. Multiple Pods can reference the same ConfigMap.
2. Modify your Pod definition to add a volume under `.spec.volumes[]`. Name the volume anything, and have a `.spec.volumes[].configMap.name` field set to reference your ConfigMap object.
3. Add a `.spec.containers[].volumeMounts[]` to each container that needs the ConfigMap. Specify `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the ConfigMap to appear.
4. Modify your image or command line so that the program looks for files in that directory. Each key in the ConfigMap `data` map becomes the filename under `mountPath`.

This is an example of a Pod that mounts a ConfigMap 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
    configMap:
      name: myconfigmap
```

Each ConfigMap 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 ConfigMap.

Mounted ConfigMaps are updated automatically

When a ConfigMap currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap. The type of the cache is configurable using the `ConfigMapAndSecretChangeDetectionStrategy` field in the KubeletConfiguration struct. A ConfigMap can be either propagated by watch (default), ttl-based, or by redirecting all requests directly to the API server. As a result, the total delay from the moment when the ConfigMap 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).

ConfigMaps consumed as environment variables are not updated automatically and require a pod restart.

Immutable ConfigMaps

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

The Kubernetes beta feature *Immutable Secrets and ConfigMaps* provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use ConfigMaps (at least tens of thousands of unique ConfigMap to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for ConfigMaps marked as immutable.

This feature is controlled by the `ImmutableEphemeralVolumes` feature gate. You can create an immutable ConfigMap by setting the `immutable` field to `true`. For example:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  ...
data:
  ...
immutable: true
```

Once a ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the `data` or the `binaryData` field. You can only delete and recreate the ConfigMap. Because existing Pods maintain a mount point to the deleted ConfigMap, it is recommended to recreate these pods.

What's next

- Read about Secrets.
- Read Configure a Pod to Use a ConfigMap.
- Read The Twelve-Factor App to understand the motivation for separating code from configuration.

3 - 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 Pod definition or in a container image. See Secrets design document for more information.

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.

> **Caution:**
> Kubernetes Secrets are, by default, stored as unencrypted base64-encoded strings. By default they can be retrieved - as plain text - by anyone with API access, or anyone with access to Kubernetes' underlying data store, etcd. In order to safely use Secrets, it is recommended you (at a minimum):
>
> 1. Enable Encryption at Rest for Secrets.
> 2. Enable or configure RBAC rules that restrict reading and writing the Secret. Be aware that secrets can be obtained implicitly by anyone with the permission to create a Pod.

Overview of Secrets

To use a Secret, a Pod needs to reference the Secret. A Secret can be used with a Pod in three ways:

- As files in a volume mounted on one or more of its containers.
- As container environment variable.
- By the kubelet when pulling images for the Pod.

The name of a Secret object must be a valid DNS subdomain name. You can specify the `data` and/or the `stringData` field when creating a configuration file for a Secret. The `data` and the `stringData` fields are optional. The values for all keys in the `data` field have to be base64-encoded strings. If the conversion to base64 string is not desirable, you can choose to specify the `stringData` field instead, which accepts arbitrary strings as values.

The keys of `data` and `stringData` must consist of alphanumeric characters, `-`, `_` or `.`. All key-value pairs in the `stringData` field are internally merged into the `data` field. If a key appears in both the `data` and the `stringData` field, the value specified in the `stringData` field takes precedence.

Types of Secret

When creating a Secret, you can specify its type using the `type` field of the `Secret` resource, or certain equivalent `kubectl` command line flags (if available). The Secret type is used to facilitate programmatic handling of the Secret data.

Kubernetes provides several builtin types for some common usage scenarios. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them.

Builtin Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	service account token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file

Builtin Type	Usage
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

You can define and use your own Secret type by assigning a non-empty string as the `type` value for a Secret object. An empty string is treated as an `Opaque` type. Kubernetes doesn't impose any constraints on the type name. However, if you are using one of the builtin types, you must meet all the requirements defined for that type.

Opaque secrets

`Opaque` is the default Secret type if omitted from a Secret configuration file. When you create a Secret using `kubectl`, you will use the `generic` subcommand to indicate an `Opaque` Secret type. For example, the following command creates an empty Secret of type `Opaque`.

```
kubectl create secret generic empty-secret
kubectl get secret empty-secret
```

The output looks like:

```
NAME           TYPE     DATA   AGE
empty-secret   Opaque   0      2m6s
```

The `DATA` column shows the number of data items stored in the Secret. In this case, `0` means we have created an empty Secret.

Service account token Secrets

A `kubernetes.io/service-account-token` type of Secret is used to store a token that identifies a service account. When using this Secret type, you need to ensure that the `kubernetes.io/service-account.name` annotation is set to an existing service account name. A Kubernetes controller fills in some other fields such as the `kubernetes.io/service-account.uid` annotation and the `token` key in the `data` field set to actual token content.

The following example configuration declares a service account token Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name"
type: kubernetes.io/service-account-token
data:
  # You can include additional key value pairs as you do with Opaque Secrets
  extra: YmFyCg==
```

When creating a `Pod`, Kubernetes automatically creates a service account Secret and automatically modifies your Pod to use this Secret. The service account token Secret contains credentials for accessing the API.

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. You can also check the `automountServiceAccountToken` field and the `serviceAccountName` field of the `Pod` for information on referencing service account from Pods.

Docker config Secrets

You can use one of the following `type` values to create a Secret to store the credentials for accessing a Docker registry for images.

- `kubernetes.io/dockercfg`
- `kubernetes.io/dockerconfigjson`

The `kubernetes.io/dockercfg` type is reserved to store a serialized `~/.dockercfg` which is the legacy format for configuring Docker command line. When using this Secret type, you have to ensure the Secret `data` field contains a `.dockercfg` key whose value is content of a `~/.dockercfg` file encoded in the base64 format.

The `kubernetes.io/dockerconfigjson` type is designed for storing a serialized JSON that follows the same format rules as the `~/.docker/config.json` file which is a new format for `~/.dockercfg`. When using this Secret type, the `data` field of the Secret object must contain a `.dockerconfigjson` key, in which the content for the `~/.docker/config.json` file is provided as a base64 encoded string.

Below is an example for a `kubernetes.io/dockercfg` type of Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-dockercfg
type: kubernetes.io/dockercfg
data:
  .dockercfg: |
        "<base64 encoded ~/.dockercfg file>"
```

> **Note:** If you do not want to perform the base64 encoding, you can choose to use the `stringData` field instead.

When you create these types of Secrets using a manifest, the API server checks whether the expected key does exists in the `data` field, and it verifies if the value provided can be parsed as a valid JSON. The API server doesn't validate if the JSON actually is a Docker config file.

When you do not have a Docker config file, or you want to use `kubectl` to create a Docker registry Secret, you can do:

```
kubectl create secret docker-registry secret-tiger-docker \
  --docker-username=tiger \
  --docker-password=pass113 \
  --docker-email=tiger@acme.com
```

This command creates a Secret of type `kubernetes.io/dockerconfigjson`. If you dump the `.dockerconfigjson` content from the `data` field, you will get the following JSON content which is a valid Docker configuration created on the fly:

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "username": "tiger",
```

```
      "password": "pass113",
      "email": "tiger@acme.com",
      "auth": "dGlnZXI6cGFzczExMw=="
    }
  }
}
```

Basic authentication Secret

The `kubernetes.io/basic-auth` type is provided for storing credentials needed for basic authentication. When using this Secret type, the `data` field of the Secret must contain the following two keys:

- `username`: the user name for authentication;
- `password`: the password or token for authentication.

Both values for the above two keys are base64 encoded strings. You can, of course, provide the clear text content using the `stringData` for Secret creation.

The following YAML is an example config for a basic authentication Secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: admin
  password: t0p-Secret
```

The basic authentication Secret type is provided only for user's convenience. You can create an `Opaque` for credentials used for basic authentication. However, using the builtin Secret type helps unify the formats of your credentials and the API server does verify if the required keys are provided in a Secret configuration.

SSH authentication secrets

The builtin type `kubernetes.io/ssh-auth` is provided for storing data used in SSH authentication. When using this Secret type, you will have to specify a `ssh-privatekey` key-value pair in the `data` (or `stringData`) field as the SSH credential to use.

The following YAML is an example config for a SSH authentication Secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # the data is abbreviated in this example
  ssh-privatekey: |
          MIIEpQIBAAKCAQEAulqb/Y ...
```

The SSH authentication Secret type is provided only for user's convenience. You can create an `Opaque` for credentials used for SSH authentication. However, using the builtin Secret type helps unify the formats of your credentials and the API server does verify if the required keys are provided in a Secret configuration.

> **Caution:** SSH private keys do not establish trusted communication between an SSH client and host server on their own. A secondary means of establishing trust is needed to mitigate "man in the middle" attacks, such as a `known_hosts` file added to a ConfigMap.

TLS secrets

Kubernetes provides a builtin Secret type `kubernetes.io/tls` for storing a certificate and its associated key that are typically used for TLS . This data is primarily used with TLS termination of the Ingress resource, but may be used with other resources or directly by a workload. When using this type of Secret, the `tls.key` and the `tls.crt` key must be provided in the `data` (or `stringData`) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

The following YAML contains an example config for a TLS Secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-tls
type: kubernetes.io/tls
data:
  # the data is abbreviated in this example
  tls.crt: |
        MIIC2DCCAcCgAwIBAgIBATANBgkqh ...
  tls.key: |
        MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ ...
```

The TLS Secret type is provided for user's convenience. You can create an `Opaque` for credentials used for TLS server and/or client. However, using the builtin Secret type helps ensure the consistency of Secret format in your project; the API server does verify if the required keys are provided in a Secret configuration.

When creating a TLS Secret using `kubectl`, you can use the `tls` subcommand as shown in the following example:

```
kubectl create secret tls my-tls-secret \
  --cert=path/to/cert/file \
  --key=path/to/key/file
```

The public/private key pair must exist before hand. The public key certificate for `--cert` must be .PEM encoded (Base64-encoded DER format), and match the given private key for `--key` . The private key must be in what is commonly called PEM private key format, unencrypted. In both cases, the initial and the last lines from PEM (for example, `--------BEGIN CERTIFICATE-----` and `-------END CERTIFICATE----` for a cetificate) are *not* included.

Bootstrap token Secrets

A bootstrap token Secret can be created by explicitly specifying the Secret `type` to `bootstrap.kubernetes.io/token` . This type of Secret is designed for tokens used during the node bootstrap process. It stores tokens used to sign well known ConfigMaps.

A bootstrap token Secret is usually created in the `kube-system` namespace and named in the form `bootstrap-token-<token-id>` where `<token-id>` is a 6 character string of the token ID.

As a Kubernetes manifest, a bootstrap token Secret might look like the following:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: bootstrap-token-5emitj
  namespace: kube-system
type: bootstrap.kubernetes.io/token
data:
  auth-extra-groups: c3lzdGVtOmJvb3RzdHJhcHBlcnM6a3ViZWFkbTpkZWZhdWx0LW5vZGUtdG9rZW4
```

```
  expiration: MjAyMC0wOS0xM1QwNDozOToxMFo=
  token-id: NWVtaXRq
  token-secret: a3E0Z2lodnN6emduMXAwcg==
  usage-bootstrap-authentication: dHJ1ZQ==
  usage-bootstrap-signing: dHJ1ZQ==
```

A bootstrap type Secret has the following keys specified under `data`:

- `token-id`: A random 6 character string as the token identifier. Required.
- `token-secret`: A random 16 character string as the actual token secret. Required.
- `description`: A human-readable string that describes what the token is used for. Optional.
- `expiration`: An absolute UTC time using RFC3339 specifying when the token should be expired. Optional.
- `usage-bootstrap-<usage>`: A boolean flag indicating additional usage for the bootstrap token.
- `auth-extra-groups`: A comma-separated list of group names that will be authenticated as in addition to the `system:bootstrappers` group.

The above YAML may look confusing because the values are all in base64 encoded strings. In fact, you can create an identical Secret using the following YAML:

```
apiVersion: v1
kind: Secret
metadata:
  # Note how the Secret is named
  name: bootstrap-token-5emitj
  # A bootstrap token Secret usually resides in the kube-system namespace
  namespace: kube-system
type: bootstrap.kubernetes.io/token
stringData:
  auth-extra-groups: "system:bootstrappers:kubeadm:default-node-token"
  expiration: "2020-09-13T04:39:10Z"
  # This token ID is used in the name
  token-id: "5emitj"
  token-secret: "kq4gihvszzgn1p0r"
  # This token can be used for authentication
  usage-bootstrap-authentication: "true"
  # and it can be used for signing
  usage-bootstrap-signing: "true"
```

Creating a Secret

There are several options to create a Secret:

- create Secret using `kubectl` command
- create Secret from config file
- create Secret using kustomize

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 w
# 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 variables 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: 0400
```

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.

Note if you `kubectl exec` into the Pod, you need to follow the symlink to find the expected file mode. For example,

Check the secrets file mode on the pod.

```
kubectl exec mypod -it sh

cd /etc/foo
ls -l
```

The output is similar to this:

```
total 0
lrwxrwxrwx 1 root root 15 May 18 00:18 password -> ..data/password
lrwxrwxrwx 1 root root 15 May 18 00:18 username -> ..data/username
```

Follow the symlink to find the correct file mode.

```
cd /etc/foo/..data
ls -l
```

The output is similar to this:

```
total 8
-r-------- 1 root root 12 May 18 00:18 password
-r-------- 1 root root  5 May 18 00:18 username
```

You can also use mapping, as in the previous example, and specify different permissions for different files like this:

```yaml
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: 0777
```

In this case, the file resulting in `/etc/foo/my-group/my-username` will have permission value of `0777`. If you use JSON, owing to JSON limitations, you must specify the mode in decimal notation, `511`.

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 by 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 variable 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:

```yaml
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 the 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
```

Environment variables are not updated after a secret update

If a container already consumes a Secret in an environment variable, a Secret update will not be seen by the container unless it is restarted. There are third party solutions for triggering restarts when secrets change.

Immutable Secrets

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

The Kubernetes beta feature *Immutable Secrets and ConfigMaps* provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use Secrets (at least tens of thousands of unique Secret to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages

- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for secrets marked as immutable.

This feature is controlled by the `ImmutableEphemeralVolumes` feature gate, which is enabled by default since v1.19. You can create an immutable Secret by setting the `immutable` field to `true`. For example,

```yaml
apiVersion: v1
kind: Secret
metadata:
  ...
data:
  ...
immutable: true
```

> **Note:** Once a Secret or ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the `data` field. You can only delete and recreate the Secret. Existing Pods maintain a mount point to the deleted Secret - it is recommended to recreate these pods.

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.

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 namespace. 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
0s         0s          1         dapi-test-pod   Pod
```

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: As container environment variables

Create a secret

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  USER_NAME: YWRtaW4=
  PASSWORD: MWYyZDFlMmU2N2Rm
```

Create the Secret:

```
kubectl apply -f mysecret.yaml
```

Use `envFrom` to define all of the Secret's data as container environment variables. The key from the Secret becomes the environment variable name in the Pod.

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
      - secretRef:
          name: mysecret
  restartPolicy: Never
```

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/.ss
```

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
```

The output is similar to:

```
secret "prod-db-secret" created
```

You can also create a secret for test environment credentials.

```
kubectl create secret generic test-db-secret --from-literal=username=testuser --from
```

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 --fr
```

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`:

```yaml
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]`

You can enable encryption at rest for secret data, so that the secrets are not stored in the clear into etcd.

Risks

- In the API server, secret data is stored in etcd; therefore:
  - Administrators should enable encryption at rest for cluster data (requires v1.13 or later).
  - Administrators should limit access to etcd to admin users.
  - Administrators may want to wipe/shred disks used by etcd when no longer in use.
  - If running etcd in a cluster, administrators should make sure to use SSL/TLS for etcd peer-to-peer communication.
- If you configure the secret through a manifest (JSON or YAML) file which has the secret data encoded as base64, sharing this file or checking it in to a source repository means the secret is compromised. Base64 encoding is *not* an encryption method and is considered the same as plain text.
- Applications still need to protect the value of secret after reading it from the volume, such as not accidentally logging it or transmitting it to an untrusted party.
- A user who can create a Pod that uses a secret can also see the value of that secret. Even if the API server policy does not allow that user to read the Secret, the user could run a Pod

which exposes the secret.

- Currently, anyone with root permission on any node can read *any* secret from the API server, by impersonating the kubelet. It is a planned feature to only send secrets to nodes that actually require them, to restrict the impact of a root exploit on a single node.

What's next

- Learn how to manage Secret using `kubectl`
- Learn how to manage Secret using config file
- Learn how to manage Secret using kustomize

4 - Managing Resources for Containers

When you specify a Pod, you can optionally specify how much of each resource a Container needs. The most common resources to specify are CPU and memory (RAM); there are others.

When you specify the resource *request* for Containers in a Pod, the scheduler uses this information to decide which node to place the Pod on. When you specify a resource *limit* for a Container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the limit you set. The kubelet also reserves at least the *request* amount of that system resource specifically for that container to use.

Requests and limits

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its `request` for that resource specifies. However, a container is not allowed to use more than its resource `limit`.

For example, if you set a `memory` request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM.

If you set a `memory` limit of 4GiB for that Container, the kubelet (and container runtime) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an out of memory (OOM) error.

Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions.

> **Note:** If a Container specifies its own memory limit, but does not specify a memory request, Kubernetes automatically assigns a memory request that matches the limit. Similarly, if a Container specifies its own CPU limit, but does not specify a CPU request, Kubernetes automatically assigns a CPU request that matches the limit.

Resource types

CPU and *memory* are each a *resource type*. A resource type has a base unit. CPU represents compute processing and is specified in units of Kubernetes CPUs. 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 *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.

Resource units in Kubernetes

Meaning of CPU

Limits and requests for CPU resources are measured in *cpu* units. One cpu, in Kubernetes, is equivalent to **1 vCPU/Core** for cloud providers and **1 hyperthread** on bare-metal Intel processors.

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 number 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.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: log-aggregator
```

```
    image: images.my-company.example/log-aggregator:v6
    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 & memory resource usage

The resource usage of a Pod is reported as part of the Pod status.

If optional tools for monitoring are available in your cluster, then Pod resource usage can be retrieved either from the Metrics API directly or from your monitoring tools.

Local ephemeral storage

FEATURE STATE: `Kubernetes v1.10 [beta]`

Nodes have local ephemeral storage, backed by locally-attached writeable devices or, sometimes, by RAM. "Ephemeral" means that there is no long-term guarantee about durability.

Pods use ephemeral local storage for scratch space, caching, and for logs. The kubelet can provide scratch space to Pods using local ephemeral storage to mount `emptyDir` volumes into containers.

The kubelet also uses this kind of storage to hold node-level container logs, container images, and the writable layers of running containers.

> **Caution:** If a node fails, the data in its ephemeral storage can be lost.
> Your applications cannot expect any performance SLAs (disk IOPS for example) from local ephemeral storage.

As a beta feature, Kubernetes lets you track, reserve and limit the amount of ephemeral local storage a Pod can consume.

Configurations for local ephemeral storage

Kubernetes supports two ways to configure local ephemeral storage on a node:

Single filesystem | Two filesystems

In this configuration, you place all different kinds of ephemeral local data (`emptyDir` volumes, writeable layers, container images, logs) into one filesystem. The most effective way to configure the kubelet means dedicating this filesystem to Kubernetes (kubelet) data.

The kubelet also writes node-level container logs and treats these similarly to ephemeral local storage.

The kubelet writes logs to files inside its configured log directory (`/var/log` by default); and has a base directory for other locally stored data (`/var/lib/kubelet` by default).

Typically, both `/var/lib/kubelet` and `/var/log` are on the system root filesystem, and the kubelet is designed with that layout in mind.

Your node can have as many other filesystems, not used for Kubernetes, as you like.

The kubelet can measure how much local storage it is using. It does this provided that:

- the `LocalStorageCapacityIsolation` feature gate is enabled (the feature is on by default), and
- you have set up the node using one of the supported configurations for local ephemeral storage.

If you have a different configuration, then the kubelet does not apply resource limits for ephemeral local storage.

> **Note:** The kubelet tracks `tmpfs` emptyDir volumes as container memory use, rather than as local ephemeral storage.

Setting requests and limits for local ephemeral storage

You can use *ephemeral-storage* for managing local ephemeral storage. Each Container of a Pod can specify one or more of the following:

- `spec.containers[].resources.limits.ephemeral-storage`
- `spec.containers[].resources.requests.ephemeral-storage`

Limits and requests for `ephemeral-storage` are measured in bytes. You can express storage as a plain integer or as a fixed-point number 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
```

In the following example, the Pod has two Containers. Each Container has a request of 2GiB of local ephemeral storage. Each Container has a limit of 4GiB of local ephemeral storage. Therefore, the Pod has a request of 4GiB of local ephemeral storage, and a limit of 8GiB of local ephemeral storage.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
```

How Pods with ephemeral-storage 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 amount of local ephemeral storage it can provide for Pods. For more information, see Node Allocatable.

The scheduler ensures that the sum of the resource requests of the scheduled Containers is less than the capacity of the node.

Ephemeral storage consumption management

If the kubelet is managing local ephemeral storage as a resource, then the kubelet measures storage use in:

- `emptyDir` volumes, except *tmpfs* `emptyDir` volumes
- directories holding node-level logs
- writeable container layers

If a Pod is using more ephemeral storage than you allow it to, the kubelet sets an eviction signal that triggers Pod eviction.

For container-level isolation, if a Container's writable layer and log usage exceeds its storage limit, the kubelet marks the Pod for eviction.

For pod-level isolation the kubelet works out an overall Pod storage limit by summing the limits for the containers in that Pod. In this case, if the sum of the local ephemeral storage usage from all containers and also the Pod's `emptyDir` volumes exceeds the overall Pod storage limit, then the kubelet also marks the Pod for eviction.

Caution:
If the kubelet is not measuring local ephemeral storage, then a Pod that exceeds its local storage limit will not be evicted for breaching local storage resource limits.

However, if the filesystem space for writeable container layers, node-level logs, or `emptyDir` volumes falls low, the node taints itself as short on local storage and this taint triggers eviction for any Pods that don't specifically tolerate the taint.

See the supported configurations for ephemeral local storage.

The kubelet supports different ways to measure Pod storage use:

Periodic scanning | Filesystem project quota

The kubelet performs regular, scheduled checks that scan each `emptyDir` volume, container log directory, and writeable container layer.

The scan measures how much space is used.

Note:
In this mode, the kubelet does not track open file descriptors for deleted files.

If you (or a container) create a file inside an `emptyDir` volume, something then opens that file, and you delete the file while it is still open, then the inode for the deleted file stays until you close that file but the kubelet does not categorize the space as in use.

Extended resources

Extended resources are fully-qualified resource names outside the `kubernetes.io` domain. They allow cluster operators to advertise and users to consume the non-Kubernetes-built-in resources.

There are two steps required to use Extended Resources. First, the cluster operator must advertise an Extended Resource. Second, users must request the Extended Resource in Pods.

Managing extended resources

Node-level extended resources

Node-level extended resources are tied to nodes.

Device plugin managed resources

See Device Plugin for how to advertise device plugin managed resources on each node.

Other resources

To advertise a new node-level extended resource, the cluster operator can submit a `PATCH` HTTP request to the API server to specify the available quantity in the `status.capacity` for a node in the cluster. After this operation, the node's `status.capacity` will include a new resource. The `status.allocatable` field is updated automatically with the new resource asynchronously by the kubelet. Note that because the scheduler uses the node `status.allocatable` value when evaluating Pod fitness, there may be a short delay between patching the node capacity with a new resource and the first Pod that requests the resource to be scheduled on that node.

Example:

Here is an example showing how to use `curl` to form an HTTP request that advertises five "example.com/foo" resources on node `k8s-node-1` whose master is `k8s-master`.

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/example.com~1foo", "value": "5"}]'
http://k8s-master:8080/api/v1/nodes/k8s-node-1/status
```

> **Note:** In the preceding request, `~1` is the encoding for the character `/` in the patch path. The operation path value in JSON-Patch is interpreted as a JSON-Pointer. For more details, see IETF RFC 6901, section 3.

Cluster-level extended resources

Cluster-level extended resources are not tied to nodes. They are usually managed by scheduler extenders, which handle the resource consumption and resource quota.

You can specify the extended resources that are handled by scheduler extenders in scheduler policy configuration.

Example:

The following configuration for a scheduler policy indicates that the cluster-level extended resource "example.com/foo" is handled by the scheduler extender.

- The scheduler sends a Pod to the scheduler extender only if the Pod requests "example.com/foo".
- The `ignoredByScheduler` field specifies that the scheduler does not check the "example.com/foo" resource in its `PodFitsResources` predicate.

```
{
  "kind": "Policy",
  "apiVersion": "v1",
  "extenders": [
    {
      "urlPrefix":"<extender-endpoint>",
      "bindVerb": "bind",
      "managedResources": [
        {
          "name": "example.com/foo",
          "ignoredByScheduler": true
        }
      ]
    }
  ]
}
```

Consuming extended resources

Users can consume extended resources in Pod specs like CPU and memory. The scheduler takes care of the resource accounting so that no more than the available amount is simultaneously allocated to Pods.

The API server restricts quantities of extended resources to whole numbers. Examples of *valid* quantities are `3`, `3000m` and `3Ki`. Examples of *invalid* quantities are `0.5` and `1500m`.

> **Note:** Extended resources replace Opaque Integer Resources. Users can use any domain name prefix other than `kubernetes.io` which is reserved.

To consume an extended resource in a Pod, include the resource name as a key in the `spec.containers[].resources.limits` map in the container spec.

> **Note:** Extended resources cannot be overcommitted, so request and limit must be equal if both are present in a container spec.

A Pod is scheduled only if all of the resource requests are satisfied, including CPU, memory and any extended resources. The Pod remains in the `PENDING` state as long as the resource request cannot be satisfied.

Example:

The Pod below requests 2 CPUs and 1 "example.com/foo" (an extended resource).

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: myimage
    resources:
      requests:
        cpu: 2
        example.com/foo: 1
      limits:
        example.com/foo: 1
```

PID limiting

Process ID (PID) limits allow for the configuration of a kubelet to limit the number of PIDs that a given Pod can consume. See Pid Limiting for information.

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
```

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  Memor
  ---------    ----                                  ------------  ----------  -----
  kube-system  fluentd-gcp-v1.38-28bv1               100m (5%)     0 (0%)      200Mi
  kube-system  kube-dns-3297075139-61lj3             260m (13%)    0 (0%)      100Mi
  kube-system  kube-proxy-e2e-test-...               100m (5%)     0 (0%)      0 (0%
  kube-system  monitoring-influxdb-grafana-v4-z1m12  200m (10%)    200m (10%)  600Mi
  kube-system  node-problem-detector-v0.1-fj7m3      20m (1%)      200m (10%)  20Mi
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 (11%)        1070Mi (13%)
```

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
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {schedul
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet
```

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:
```

```
Container Name: simmemleak
LastState: map[terminated:map[exitCode:137 reason:OOM Killed startedAt:2015-07-07T20
```

You can see that the Container was terminated because of `reason:OOM Killed`, where `OOM` stands for Out Of Memory.

What's next

- Get hands-on experience assigning Memory resources to Containers and Pods.
- Get hands-on experience assigning CPU resources to Containers and Pods.
- For more details about the difference between requests and limits, see Resource QoS.
- Read the Container API reference
- Read the ResourceRequirements API reference
- Read about project quotas in XFS

5 - 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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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`

6 - Pod Priority and Preemption

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

Pods can have *priority*. Priority indicates the importance of a Pod relative to other Pods. If a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible.

> **Warning:**
> In a cluster where not all users are trusted, a malicious user could create Pods at the highest possible priorities, causing other Pods to be evicted/not get scheduled. An administrator can use ResourceQuota to prevent users from creating pods at high priorities.
>
> See limit Priority Class consumption by default for details.

How to use priority and preemption

To use priority and preemption:

1. Add one or more PriorityClasses.
2. Create Pods with `priorityClassName` set to one of the added PriorityClasses. Of course you do not need to create the Pods directly; normally you would add `priorityClassName` to the Pod template of a collection object like a Deployment.

Keep reading for more information about these steps.

> **Note:** Kubernetes already ships with two PriorityClasses: `system-cluster-critical` and `system-node-critical`. These are common classes and are used to ensure that critical components are always scheduled first.

PriorityClass

A PriorityClass is a non-namespaced object that defines a mapping from a priority class name to the integer value of the priority. The name is specified in the `name` field of the PriorityClass object's metadata. The value is specified in the required `value` field. The higher the value, the higher the priority. The name of a PriorityClass object must be a valid DNS subdomain name, and it cannot be prefixed with `system-`.

A PriorityClass object can have any 32-bit integer value smaller than or equal to 1 billion. Larger numbers are reserved for critical system Pods that should not normally be preempted or evicted. A cluster admin should create one PriorityClass object for each such mapping that they want.

PriorityClass also has two optional fields: `globalDefault` and `description`. The `globalDefault` field indicates that the value of this PriorityClass should be used for Pods without a `priorityClassName`. Only one PriorityClass with `globalDefault` set to true can exist in the system. If there is no PriorityClass with `globalDefault` set, the priority of Pods with no `priorityClassName` is zero.

The `description` field is an arbitrary string. It is meant to tell users of the cluster when they should use this PriorityClass.

Notes about PodPriority and existing clusters

- If you upgrade an existing cluster without this feature, the priority of your existing Pods is effectively zero.
- Addition of a PriorityClass with `globalDefault` set to `true` does not change the priorities of existing Pods. The value of such a PriorityClass is used only for Pods created after the PriorityClass is added.

- If you delete a PriorityClass, existing Pods that use the name of the deleted PriorityClass remain unchanged, but you cannot create more Pods that use the name of the deleted PriorityClass.

Example PriorityClass

```yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service pods only."
```

Non-preempting PriorityClass

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

Pods with `PreemptionPolicy: Never` will be placed in the scheduling queue ahead of lower-priority pods, but they cannot preempt other pods. A non-preempting pod waiting to be scheduled will stay in the scheduling queue, until sufficient resources are free, and it can be scheduled. Non-preempting pods, like other pods, are subject to scheduler back-off. This means that if the scheduler tries these pods and they cannot be scheduled, they will be retried with lower frequency, allowing other pods with lower priority to be scheduled before them.

Non-preempting pods may still be preempted by other, high-priority pods.

`PreemptionPolicy` defaults to `PreemptLowerPriority`, which will allow pods of that PriorityClass to preempt lower-priority pods (as is existing default behavior). If `PreemptionPolicy` is set to `Never`, pods in that PriorityClass will be non-preempting.

An example use case is for data science workloads. A user may submit a job that they want to be prioritized above other workloads, but do not wish to discard existing work by preempting running pods. The high priority job with `PreemptionPolicy: Never` will be scheduled ahead of other queued pods, as soon as sufficient cluster resources "naturally" become free.

Example Non-preempting PriorityClass

```yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority-nonpreempting
value: 1000000
preemptionPolicy: Never
globalDefault: false
description: "This priority class will not cause other pods to be preempted."
```

Pod priority

After you have one or more PriorityClasses, you can create Pods that specify one of those PriorityClass names in their specifications. The priority admission controller uses the `priorityClassName` field and populates the integer value of the priority. If the priority class is not found, the Pod is rejected.

The following YAML is an example of a Pod configuration that uses the PriorityClass created in the preceding example. The priority admission controller checks the specification and resolves the priority of the Pod to 1000000.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

Effect of Pod priority on scheduling order

When Pod priority is enabled, the scheduler orders pending Pods by their priority and a pending Pod is placed ahead of other pending Pods with lower priority in the scheduling queue. As a result, the higher priority Pod may be scheduled sooner than Pods with lower priority if its scheduling requirements are met. If such Pod cannot be scheduled, scheduler will continue and tries to schedule other lower priority Pods.

Preemption

When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, preemption logic is triggered for the pending Pod. Let's call the pending Pod P. Preemption logic tries to find a Node where removal of one or more Pods with lower priority than P would enable P to be scheduled on that Node. If such a Node is found, one or more lower priority Pods get evicted from the Node. After the Pods are gone, P can be scheduled on the Node.

User exposed information

When Pod P preempts one or more Pods on Node N, `nominatedNodeName` field of Pod P's status is set to the name of Node N. This field helps scheduler track resources reserved for Pod P and also gives users information about preemptions in their clusters.

Please note that Pod P is not necessarily scheduled to the "nominated Node". After victim Pods are preempted, they get their graceful termination period. If another node becomes available while scheduler is waiting for the victim Pods to terminate, scheduler will use the other node to schedule Pod P. As a result `nominatedNodeName` and `nodeName` of Pod spec are not always the same. Also, if scheduler preempts Pods on Node N, but then a higher priority Pod than Pod P arrives, scheduler may give Node N to the new higher priority Pod. In such a case, scheduler clears `nominatedNodeName` of Pod P. By doing this, scheduler makes Pod P eligible to preempt Pods on another Node.

Limitations of preemption

Graceful termination of preemption victims

When Pods are preempted, the victims get their graceful termination period. They have that much time to finish their work and exit. If they don't, they are killed. This graceful termination period creates a time gap between the point that the scheduler preempts Pods and the time when the pending Pod (P) can be scheduled on the Node (N). In the meantime, the scheduler keeps scheduling other pending Pods. As victims exit or get terminated, the scheduler tries to schedule Pods in the pending queue. Therefore, there is usually a time gap between the point that scheduler preempts victims and the time that Pod P is scheduled. In order to minimize this gap, one can set graceful termination period of lower priority Pods to zero or a small number.

PodDisruptionBudget is supported, but not guaranteed

A PodDisruptionBudget (PDB) allows application owners to limit the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. Kubernetes supports PDB when preempting Pods, but respecting PDB is best effort. The scheduler tries to find victims whose PDB are not violated by preemption, but if no such victims are found, preemption will still happen, and lower priority Pods will be removed despite their PDBs being violated.

Inter-Pod affinity on lower-priority Pods

A Node is considered for preemption only when the answer to this question is yes: "If all the Pods with lower priority than the pending Pod are removed from the Node, can the pending Pod be scheduled on the Node?"

> **Note:** Preemption does not necessarily remove all lower-priority Pods. If the pending Pod can be scheduled by removing fewer than all lower-priority Pods, then only a portion of the lower-priority Pods are removed. Even so, the answer to the preceding question must be yes. If the answer is no, the Node is not considered for preemption.

If a pending Pod has inter-pod affinity to one or more of the lower-priority Pods on the Node, the inter-Pod affinity rule cannot be satisfied in the absence of those lower-priority Pods. In this case, the scheduler does not preempt any Pods on the Node. Instead, it looks for another Node. The scheduler might find a suitable Node or it might not. There is no guarantee that the pending Pod can be scheduled.

Our recommended solution for this problem is to create inter-Pod affinity only towards equal or higher priority Pods.

Cross node preemption

Suppose a Node N is being considered for preemption so that a pending Pod P can be scheduled on N. P might become feasible on N only if a Pod on another Node is preempted. Here's an example:

- Pod P is being considered for Node N.
- Pod Q is running on another Node in the same Zone as Node N.
- Pod P has Zone-wide anti-affinity with Pod Q (`topologyKey: topology.kubernetes.io/zone`).
- There are no other cases of anti-affinity between Pod P and other Pods in the Zone.
- In order to schedule Pod P on Node N, Pod Q can be preempted, but scheduler does not perform cross-node preemption. So, Pod P will be deemed unschedulable on Node N.

If Pod Q were removed from its Node, the Pod anti-affinity violation would be gone, and Pod P could possibly be scheduled on Node N.

We may consider adding cross Node preemption in future versions if there is enough demand and if we find an algorithm with reasonable performance.

Troubleshooting

Pod priority and pre-emption can have unwanted side effects. Here are some examples of potential problems and ways to deal with them.

Pods are preempted unnecessarily

Preemption removes existing Pods from a cluster under resource pressure to make room for higher priority pending Pods. If you give high priorities to certain Pods by mistake, these unintentionally high priority Pods may cause preemption in your cluster. Pod priority is specified by setting the `priorityClassName` field in the Pod's specification. The integer value for priority is then resolved and populated to the `priority` field of `podSpec` .

To address the problem, you can change the `priorityClassName` for those Pods to use lower priority classes, or leave that field empty. An empty `priorityClassName` is resolved to zero by default.

When a Pod is preempted, there will be events recorded for the preempted Pod. Preemption should happen only when a cluster does not have enough resources for a Pod. In such cases, preemption happens only when the priority of the pending Pod (preemptor) is higher than the victim Pods. Preemption must not happen when there is no pending Pod, or when the pending Pods have equal or lower priority than the victims. If preemption happens in such scenarios, please file an issue.

Pods are preempted, but the preemptor is not scheduled

When pods are preempted, they receive their requested graceful termination period, which is by default 30 seconds. If the victim Pods do not terminate within this period, they are forcibly terminated. Once all the victims go away, the preemptor Pod can be scheduled.

While the preemptor Pod is waiting for the victims to go away, a higher priority Pod may be created that fits on the same Node. In this case, the scheduler will schedule the higher priority Pod instead of the preemptor.

This is expected behavior: the Pod with the higher priority should take the place of a Pod with a lower priority.

Higher priority Pods are preempted before lower priority pods

The scheduler tries to find nodes that can run a pending Pod. If no node is found, the scheduler tries to remove Pods with lower priority from an arbitrary node in order to make room for the pending pod. If a node with low priority Pods is not feasible to run the pending Pod, the scheduler may choose another node with higher priority Pods (compared to the Pods on the other node) for preemption. The victims must still have lower priority than the preemptor Pod.

When there are multiple nodes available for preemption, the scheduler tries to choose the node with a set of Pods with lowest priority. However, if such Pods have PodDisruptionBudget that would be violated if they are preempted then the scheduler may choose another node with higher priority Pods.

When multiple nodes exist for preemption and none of the above scenarios apply, the scheduler chooses a node with the lowest priority.

Interactions between Pod priority and quality of service

Pod priority and QoS class are two orthogonal features with few interactions and no default restrictions on setting the priority of a Pod based on its QoS classes. The scheduler's preemption logic does not consider QoS when choosing preemption targets. Preemption considers Pod priority and attempts to choose a set of targets with the lowest priority. Higher-priority Pods are considered for preemption only if the removal of the lowest priority Pods is not sufficient to allow the scheduler to schedule the preemptor Pod, or if the lowest priority Pods are protected by `PodDisruptionBudget`.

The only component that considers both QoS and Pod priority is kubelet out-of-resource eviction. The kubelet ranks Pods for eviction first by whether or not their usage of the starved resource exceeds requests, then by Priority, and then by the consumption of the starved compute resource relative to the Pods' scheduling requests. See evicting end-user pods for more details.

kubelet out-of-resource eviction does not evict Pods when their usage does not exceed their requests. If a Pod with lower priority is not exceeding its requests, it won't be evicted. Another Pod with higher priority that exceeds its requests may be evicted.

What's next

- Read about using ResourceQuotas in connection with PriorityClasses: limit Priority Class consumption by default