Authorization

Details of Kubernetes authorization mechanisms and supported authorization modes.

Kubernetes authorization takes place following authentication. Usually, a client making a request must be authenticated (logged in) before its request can be allowed; however, Kubernetes also allows anonymous requests in some circumstances.

For an overview of how authorization fits into the wider context of API access control, read Controlling Access to the Kubernetes API.

Authorization verdicts

Kubernetes authorization of API requests takes place within the API server. The API server evaluates all of the request attributes against all policies, potentially also consulting external services, and then allows or denies the request.

All parts of an API request must be allowed by some authorization mechanism in order to proceed. In other words: access is denied by default.

Note:

Access controls and policies that depend on specific fields of specific kinds of objects are handled by admission controllers.

Kubernetes admission control happens after authorization has completed (and, therefore, only when the authorization decision was to allow the request).

When multiple authorization modules are configured, each is checked in sequence. If any authorizer *approves* or *denies* a request, that decision is immediately returned and no other authorizer is consulted. If all modules have *no opinion* on the request, then the request is denied. An overall deny verdict means that the API server rejects the request and responds with an HTTP 403 (Forbidden) status.

Request attributes used in authorization

Kubernetes reviews only the following API request attributes:

- **user** - The `user` string provided during authentication.
- **group** - The list of group names to which the authenticated user belongs.
- **extra** - A map of arbitrary string keys to string values, provided by the authentication layer.
- **API** - Indicates whether the request is for an API resource.
- **Request path** - Path to miscellaneous non-resource endpoints like `/api` or `/healthz`.
- **API request verb** - API verbs like `get`, `list`, `create`, `update`, `patch`, `watch`, `delete`, and `deletecollection` are used for resource requests. To determine the request verb for a resource API endpoint, see request verbs and authorization.
- **HTTP request verb** - Lowercased HTTP methods like `get`, `post`, `put`, and `delete` are used for non-resource requests.
- **Resource** - The ID or name of the resource that is being accessed (for resource requests only) -- For resource requests using `get`, `update`, `patch`, and `delete` verbs, you must provide the resource name.
- **Subresource** - The subresource that is being accessed (for resource requests only).
- **Namespace** - The namespace of the object that is being accessed (for namespaced resource requests only).
- **API group** - The API Group being accessed (for resource requests only). An empty string designates the *core* API group.

Request verbs and authorization

Non-resource requests

Requests to endpoints other than `/api/v1/...` or `/apis/<group>/<version>/...` are considered *non-resource requests*, and use the lower-cased HTTP method of the request as the verb. For example, making a `GET` request using HTTP to endpoints such as `/api` or `/healthz` would use **get** as the verb.

Resource requests

To determine the request verb for a resource API endpoint, Kubernetes maps the HTTP verb used and considers whether or not the request acts on an individual resource or on a collection of resources:

HTTP verb	request verb
`POST`	**create**
`GET`, `HEAD`	**get** (for individual resources), **list** (for collections, including full object content), **watch** (for watching an individual resource or collection of resources)
`PUT`	**update**
`PATCH`	**patch**
`DELETE`	**delete** (for individual resources), **deletecollection** (for collections)

Caution:

+The **get**, **list** and **watch** verbs can all return the full details of a resource. In terms of access to the returned data they are equivalent. For example, **list** on `secrets` will reveal the **data** attributes of any returned resources.

Kubernetes sometimes checks authorization for additional permissions using specialized verbs. For example:

- Special cases of authentication

    - **impersonate** verb on `users`, `groups`, and `serviceaccounts` in the core API group, and the `userextras` in the `authentication.k8s.io` API group.
- Authorization of CertificateSigningRequests
    - **approve** verb for CertificateSigningRequests, and **update** for revisions to existing approvals
- RBAC
    - **bind** and **escalate** verbs on `roles` and `clusterroles` resources in the `rbac.authorization.k8s.io` API group.

Authorization context

Kubernetes expects attributes that are common to REST API requests. This means that Kubernetes authorization works with existing organization-wide or cloud-provider-wide access control systems which may handle other APIs besides the Kubernetes API.

Authorization modes

The Kubernetes API server may authorize a request using one of several authorization modes:

`AlwaysAllow`
: This mode allows all requests, which brings security risks. Use this authorization mode only if you do not require authorization for your API requests (for example, for testing).

`AlwaysDeny`
: This mode blocks all requests. Use this authorization mode only for testing.

`ABAC` (attribute-based access control)
: Kubernetes ABAC mode defines an access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together. The policies can use any type of attributes (user attributes, resource attributes, object, environment attributes, etc).

`RBAC` (role-based access control)
: Kubernetes RBAC is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise. In this context, access is the ability of an individual user to perform a specific task, such as view, create, or modify a file.
In this mode, Kubernetes uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing you to dynamically configure permission policies through the Kubernetes API.

`Node`
: A special-purpose authorization mode that grants permissions to kubelets based on the pods they are scheduled to run. To learn more about the Node authorization mode, see Node Authorization.

`Webhook`
: Kubernetes webhook mode for authorization makes a synchronous HTTP callout, blocking the request until the remote HTTP service responds to the query.You can write your own software to handle the callout, or use solutions from the ecosystem.

Warning:

Enabling the `AlwaysAllow` mode bypasses authorization; do not use this on a cluster where you do not trust **all** potential API clients, including the workloads that you run.

Authorization mechanisms typically return either a *deny* or *no opinion* result; see authorization verdicts for more on this. Activating the `AlwaysAllow` means that if all other authorizers return “no opinion”, the request is allowed. For example, `--authorization-mode=AlwaysAllow,RBAC` has the same effect as `--authorization-mode=AlwaysAllow` because Kubernetes RBAC does not provide negative (deny) access rules.

You should not use the `AlwaysAllow` mode on a Kubernetes cluster where the API server is reachable from the public internet.

The system:masters group

The `system:masters` group is a built-in Kubernetes group that grants unrestricted access to the API server. Any user assigned to this group has full cluster administrator privileges, bypassing any authorization restrictions imposed by the RBAC or Webhook mechanisms. Avoid adding users to this group. If you do need to grant a user cluster-admin rights, you can create a ClusterRoleBinding to the built-in `cluster-admin` ClusterRole.

Authorization mode configuration

You can configure the Kubernetes API server's authorizer chain using either a configuration file only or command line arguments.

You have to pick one of the two configuration approaches; setting both `--authorization-config` path and configuring an authorization webhook using the `--authorization-mode` and `--authorization-webhook-*` command line arguments is not allowed. If you try this, the API server reports an error message during startup, then exits immediately.

Configuring the API Server using an authorization config file

FEATURE STATE: `Kubernetes v1.32 [stable]` (enabled by default: true)

Kubernetes lets you configure authorization chains that can include multiple webhooks. The authorization items in that chain can have well-defined parameters that validate requests in a particular order, offering you fine-grained control, such as explicit Deny on failures.

The configuration file approach even allows you to specify CEL rules to pre-filter requests before they are dispatched to webhooks, helping you to prevent unnecessary invocations. The API server also automatically reloads the authorizer chain when the configuration file is modified.

You specify the path to the authorization configuration using the `--authorization-config` command line argument.

If you want to use command line arguments instead of a configuration file, that's also a valid and supported approach. Some authorization capabilities (for example: multiple webhooks, webhook failure policy, and pre-filter rules) are only available if you use an authorization configuration file.

Example configuration

```
---
```

```
## DO NOT USE THE CONFIG AS IS. THIS IS AN EXAMPLE.apiVersion: apiserver.config.k8s.io/v1kind: AuthorizationConfigurationauthoriz
```

When configuring the authorizer chain using a configuration file, make sure all the control plane nodes have the same file contents. Take a note of the API server configuration when upgrading / downgrading your clusters. For example, if upgrading from Kubernetes 1.33 to Kubernetes 1.34, you would need to make sure the config file is in a format that Kubernetes 1.34 can understand, before you upgrade the cluster. If you downgrade to 1.33, you would need to set the configuration appropriately.

Authorization configuration and reloads

Kubernetes reloads the authorization configuration file when the API server observes a change to the file, and also on a 60 second schedule if no change events were observed.

Note:

You must ensure that all non-webhook authorizer types remain unchanged in the file on reload.

A reload **must not** add or remove Node or RBAC authorizers (they can be reordered, but cannot be added or removed).

Command line authorization mode configuration

You can use the following modes:

- `--authorization-mode=ABAC` (Attribute-based access control mode)
- `--authorization-mode=RBAC` (Role-based access control mode)
- `--authorization-mode=Node` (Node authorizer)
- `--authorization-mode=Webhook` (Webhook authorization mode)
- `--authorization-mode=AlwaysAllow` (always allows requests; carries security risks)
- `--authorization-mode=AlwaysDeny` (always denies requests)

You can choose more than one authorization mode; for example: `--authorization-mode=Node,RBAC,Webhook`

Kubernetes checks authorization modules based on the order that you specify them on the API server's command line, so an earlier module has higher priority to allow or deny a request.

You cannot combine the `--authorization-mode` command line argument with the `--authorization-config` command line argument used for configuring authorization using a local file.

For more information on command line arguments to the API server, read the `kube-apiserver` reference.

Privilege escalation via workload creation or edits

Users who can create/edit pods in a namespace, either directly or through an object that enables indirect workload management, may be able to escalate their privileges in that namespace. The potential routes to privilege escalation include Kubernetes API extensions and their associated controllers.

Caution:

As a cluster administrator, use caution when granting access to create or edit workloads. Some details of how these can be misused are documented in escalation paths.

Escalation paths

There are different ways that an attacker or untrustworthy user could gain additional privilege within a namespace, if you allow them to run arbitrary Pods in that namespace:

- Mounting arbitrary Secrets in that namespace
  - Can be used to access confidential information meant for other workloads
  - Can be used to obtain a more privileged ServiceAccount's service account token
- Using arbitrary ServiceAccounts in that namespace
  - Can perform Kubernetes API actions as another workload (impersonation)
  - Can perform any privileged actions that ServiceAccount has
- Mounting or using ConfigMaps meant for other workloads in that namespace
  - Can be used to obtain information meant for other workloads, such as database host names.
- Mounting volumes meant for other workloads in that namespace
  - Can be used to obtain information meant for other workloads, and change it.

Caution:

As a system administrator, you should be cautious when deploying CustomResourceDefinitions that let users make changes to the above areas. These may open privilege escalations paths. Consider the consequences of this kind of change when deciding on your authorization controls.

Checking API access

`kubectl` provides the `auth can-i` subcommand for quickly querying the API authorization layer. The command uses the `SelfSubjectAccessReview` API to determine if the current user can perform a given action, and works regardless of the authorization mode used.

```
kubectl auth can-i create deployments --namespace dev
```

The output is similar to this:

```
yes
```

```
kubectl auth can-i create deployments --namespace prod
```

The output is similar to this:

```
no
```

Administrators can combine this with user impersonation to determine what action other users can perform.

```
kubectl auth can-i list secrets --namespace dev --as dave
```

The output is similar to this:

```
no
```

Similarly, to check whether a ServiceAccount named `dev-sa` in Namespace `dev` can list Pods in the Namespace `target`:

```
kubectl auth can-i list pods \
    --namespace target \    --as system:serviceaccount:dev:dev-sa
```

The output is similar to this:

```
yes
```

SelfSubjectAccessReview is part of the `authorization.k8s.io` API group, which exposes the API server authorization to external services. Other resources in this group include:

SubjectAccessReview
: Access review for any user, not only the current one. Useful for delegating authorization decisions to the API server. For example, the kubelet and extension API servers use this to determine user access to their own APIs.

LocalSubjectAccessReview
: Like SubjectAccessReview but restricted to a specific namespace.

SelfSubjectRulesReview
: A review which returns the set of actions a user can perform within a namespace. Useful for users to quickly summarize their own access, or for UIs to hide/show actions.

These APIs can be queried by creating normal Kubernetes resources, where the response `status` field of the returned object is the result of the query. For example:

```
kubectl create -f - -o yaml << EOF
apiVersion: authorization.k8s.io/v1
kind: SelfSubjectAccessReview
spec:
  resourceAttributes:
    group: apps
    resource: deployments
    verb: create
    namespace: dev
EOF
```

The generated SelfSubjectAccessReview is similar to:

```
apiVersion: authorization.k8s.io/v1
kind: SelfSubjectAccessReviewmetadata:  creationTimestamp: nullspec:  resourceAttributes:    group: apps    resource: deployments
```

What's next

- To learn more about Authentication, see Authentication.
- For an overview, read Controlling Access to the Kubernetes API.
- To learn more about Admission Control, see Using Admission Controllers.
- Read more about Common Expression Language in Kubernetes.

TLS bootstrapping

In a Kubernetes cluster, the components on the worker nodes - kubelet and kube-proxy - need to communicate with Kubernetes control plane components, specifically kube-apiserver. In order to ensure that communication is kept private, not interfered with, and ensure that each component of the cluster is talking to another trusted component, we strongly recommend using client TLS certificates on nodes.

The normal process of bootstrapping these components, especially worker nodes that need certificates so they can communicate safely with kube-apiserver, can be a challenging process as it is often outside of the scope of Kubernetes and requires significant additional work. This in turn, can make it challenging to initialize or scale a cluster.

In order to simplify the process, beginning in version 1.4, Kubernetes introduced a certificate request and signing API. The proposal can be found here.

This document describes the process of node initialization, how to set up TLS client certificate bootstrapping for kubelets, and how it works.

Initialization process

When a worker node starts up, the kubelet does the following:

1. Look for its `kubeconfig` file
2. Retrieve the URL of the API server and credentials, normally a TLS key and signed certificate from the `kubeconfig` file

3. Attempt to communicate with the API server using the credentials.

Assuming that the kube-apiserver successfully validates the kubelet's credentials, it will treat the kubelet as a valid node, and begin to assign pods to it.

Note that the above process depends upon:

- Existence of a key and certificate on the local host in the `kubeconfig`
- The certificate having been signed by a Certificate Authority (CA) trusted by the kube-apiserver

All of the following are responsibilities of whoever sets up and manages the cluster:

1. Creating the CA key and certificate
2. Distributing the CA certificate to the control plane nodes, where kube-apiserver is running
3. Creating a key and certificate for each kubelet; strongly recommended to have a unique one, with a unique CN, for each kubelet
4. Signing the kubelet certificate using the CA key
5. Distributing the kubelet key and signed certificate to the specific node on which the kubelet is running

The TLS Bootstrapping described in this document is intended to simplify, and partially or even completely automate, steps 3 onwards, as these are the most common when initializing or scaling a cluster.

Bootstrap initialization

In the bootstrap initialization process, the following occurs:

1. kubelet begins
2. kubelet sees that it does *not* have a `kubeconfig` file
3. kubelet searches for and finds a `bootstrap-kubeconfig` file
4. kubelet reads its bootstrap file, retrieving the URL of the API server and a limited usage "token"
5. kubelet connects to the API server, authenticates using the token
6. kubelet now has limited credentials to create and retrieve a certificate signing request (CSR)
7. kubelet creates a CSR for itself with the signerName set to `kubernetes.io/kube-apiserver-client-kubelet`
8. CSR is approved in one of two ways:
    - If configured, kube-controller-manager automatically approves the CSR
    - If configured, an outside process, possibly a person, approves the CSR using the Kubernetes API or via `kubectl`
9. Certificate is created for the kubelet
10. Certificate is issued to the kubelet
11. kubelet retrieves the certificate
12. kubelet creates a proper `kubeconfig` with the key and signed certificate
13. kubelet begins normal operation
14. Optional: if configured, kubelet automatically requests renewal of the certificate when it is close to expiry
15. The renewed certificate is approved and issued, either automatically or manually, depending on configuration.

The rest of this document describes the necessary steps to configure TLS Bootstrapping, and its limitations.

Configuration

To configure for TLS bootstrapping and optional automatic approval, you must configure options on the following components:

- kube-apiserver
- kube-controller-manager
- kubelet
- in-cluster resources: `ClusterRoleBinding` and potentially `ClusterRole`

In addition, you need your Kubernetes Certificate Authority (CA).

Certificate Authority

As without bootstrapping, you will need a Certificate Authority (CA) key and certificate. As without bootstrapping, these will be used to sign the kubelet certificate. As before, it is your responsibility to distribute them to control plane nodes.

For the purposes of this document, we will assume these have been distributed to control plane nodes at `/var/lib/kubernetes/ca.pem` (certificate) and `/var/lib/kubernetes/ca-key.pem` (key). We will refer to these as "Kubernetes CA certificate and key".

All Kubernetes components that use these certificates - kubelet, kube-apiserver, kube-controller-manager - assume the key and certificate to be PEM-encoded.

kube-apiserver configuration

The kube-apiserver has several requirements to enable TLS bootstrapping:

- Recognizing CA that signs the client certificate
- Authenticating the bootstrapping kubelet to the `system:bootstrappers` group
- Authorize the bootstrapping kubelet to create a certificate signing request (CSR)

Recognizing client certificates

This is normal for all client certificate authentication. If not already set, add the `--client-ca-file=FILENAME` flag to the kube-apiserver command to enable client certificate authentication, referencing a certificate authority bundle containing the signing certificate, for example `--client-ca-file=/var/lib/kubernetes/ca.pem`.

Initial bootstrap authentication

In order for the bootstrapping kubelet to connect to kube-apiserver and request a certificate, it must first authenticate to the server. You can use any authenticator that can authenticate the kubelet.

While any authentication strategy can be used for the kubelet's initial bootstrap credentials, the following two authenticators are recommended for ease of provisioning.

1. Bootstrap Tokens
2. Token authentication file

Using bootstrap tokens is a simpler and more easily managed method to authenticate kubelets, and does not require any additional flags when starting kube-apiserver.

Whichever method you choose, the requirement is that the kubelet be able to authenticate as a user with the rights to:

1. create and retrieve CSRs
2. be automatically approved to request node client certificates, if automatic approval is enabled.

A kubelet authenticating using bootstrap tokens is authenticated as a user in the group `system:bootstrappers`, which is the standard method to use.

As this feature matures, you should ensure tokens are bound to a Role Based Access Control (RBAC) policy which limits requests (using the bootstrap token) strictly to client requests related to certificate provisioning. With RBAC in place, scoping the tokens to a group allows for great flexibility. For example, you could disable a particular bootstrap group's access when you are done provisioning the nodes.

Bootstrap tokens

Bootstrap tokens are described in detail here. These are tokens that are stored as secrets in the Kubernetes cluster, and then issued to the individual kubelet. You can use a single token for an entire cluster, or issue one per worker node.

The process is two-fold:

1. Create a Kubernetes secret with the token ID, secret and scope(s).
2. Issue the token to the kubelet

From the kubelet's perspective, one token is like another and has no special meaning. From the kube-apiserver's perspective, however, the bootstrap token is special. Due to its `type`, `namespace` and `name`, kube-apiserver recognizes it as a special token, and grants anyone authenticating with that token special bootstrap rights, notably treating them as a member of the `system:bootstrappers` group. This fulfills a basic requirement for TLS bootstrapping.

The details for creating the secret are available here.

If you want to use bootstrap tokens, you must enable it on kube-apiserver with the flag:

```
--enable-bootstrap-token-auth=true
```

Token authentication file

kube-apiserver has the ability to accept tokens as authentication. These tokens are arbitrary but should represent at least 128 bits of entropy derived from a secure random number generator (such as `/dev/urandom` on most modern Linux systems). There are multiple ways you can generate a token. For example:

```
head -c 16 /dev/urandom | od -An -t x | tr -d ' '
```

This will generate tokens that look like `02b50b05283e98dd0fd71db496ef01e8`.

The token file should look like the following example, where the first three values can be anything and the quoted group name should be as depicted:

```
02b50b05283e98dd0fd71db496ef01e8,kubelet-bootstrap,10001,"system:bootstrappers"
```

Add the `--token-auth-file=FILENAME` flag to the kube-apiserver command (in your systemd unit file perhaps) to enable the token file. See docs here for further details.

Authorize kubelet to create CSR

Now that the bootstrapping node is *authenticated* as part of the `system:bootstrappers` group, it needs to be *authorized* to create a certificate signing request (CSR) as well as retrieve it when done. Fortunately, Kubernetes ships with a `ClusterRole` with precisely these (and only these) permissions, `system:node-bootstrapper`.

To do this, you only need to create a `ClusterRoleBinding` that binds the `system:bootstrappers` group to the cluster role `system:node-bootstrapper`.

```
# enable bootstrapping nodes to create CSR
apiVersion: rbac.authorization.k8s.io/v1kind: ClusterRoleBindingmetadata:  name: create-csrs-for-bootstrappingsubjects:- kind: Grou
```

kube-controller-manager configuration

While the apiserver receives the requests for certificates from the kubelet and authenticates those requests, the controller-manager is responsible for issuing actual signed certificates.

The controller-manager performs this function via a certificate-issuing control loop. This takes the form of a cfssl local signer using assets on disk. Currently, all certificates issued have one year validity and a default set of key usages.

In order for the controller-manager to sign certificates, it needs the following:

- access to the "Kubernetes CA key and certificate" that you created and distributed
- enabling CSR signing

Access to key and certificate

As described earlier, you need to create a Kubernetes CA key and certificate, and distribute it to the control plane nodes. These will be used by the controller-manager to sign the kubelet certificates.

Since these signed certificates will, in turn, be used by the kubelet to authenticate as a regular kubelet to kube-apiserver, it is important that the CA provided to the controller-manager at this stage also be trusted by kube-apiserver for authentication. This is provided to kube-apiserver with the flag `--client-ca-file=FILENAME` (for example, `--client-ca-file=/var/lib/kubernetes/ca.pem`), as described in the kube-apiserver configuration section.

To provide the Kubernetes CA key and certificate to kube-controller-manager, use the following flags:

```
--cluster-signing-cert-file="/etc/path/to/kubernetes/ca/ca.crt" --cluster-signing-key-file="/etc/path/to/kubernetes/ca/ca.key"
```

For example:

```
--cluster-signing-cert-file="/var/lib/kubernetes/ca.pem" --cluster-signing-key-file="/var/lib/kubernetes/ca-key.pem"
```

The validity duration of signed certificates can be configured with flag:

```
--cluster-signing-duration
```

Approval

In order to approve CSRs, you need to tell the controller-manager that it is acceptable to approve them. This is done by granting RBAC permissions to the correct group.

There are two distinct sets of permissions:

- `nodeclient`: If a node is creating a new certificate for a node, then it does not have a certificate yet. It is authenticating using one of the tokens listed above, and thus is part of the group `system:bootstrappers`.
- `selfnodeclient`: If a node is renewing its certificate, then it already has a certificate (by definition), which it uses continuously to authenticate as part of the group `system:nodes`.

To enable the kubelet to request and receive a new certificate, create a `ClusterRoleBinding` that binds the group in which the bootstrapping node is a member `system:bootstrappers` to the `ClusterRole` that grants it permission, `system:certificates.k8s.io:certificatesigningrequests:nodeclient`:

```
# Approve all CSRs for the group "system:bootstrappers"
apiVersion: rbac.authorization.k8s.io/v1kind: ClusterRoleBindingmetadata:  name: auto-approve-csrs-for-groupsubjects:- kind: Group
```

To enable the kubelet to renew its own client certificate, create a `ClusterRoleBinding` that binds the group in which the fully functioning node is a member `system:nodes` to the `ClusterRole` that grants it permission, `system:certificates.k8s.io:certificatesigningrequests:selfnodeclient`:

```
# Approve renewal CSRs for the group "system:nodes"
apiVersion: rbac.authorization.k8s.io/v1kind: ClusterRoleBindingmetadata:  name: auto-approve-renewals-for-nodessubjects:- kind: G
```

The `csrapproving` controller that ships as part of kube-controller-manager and is enabled by default. The controller uses the `SubjectAccessReview` API to determine if a given user is authorized to request a CSR, then approves based on the authorization outcome. To prevent conflicts with other approvers, the built-in approver doesn't explicitly deny CSRs. It only ignores unauthorized requests. The controller also prunes expired certificates as part of garbage collection.

kubelet configuration

Finally, with the control plane nodes properly set up and all of the necessary authentication and authorization in place, we can configure the kubelet.

The kubelet requires the following configuration to bootstrap:

- A path to store the key and certificate it generates (optional, can use default)
- A path to a `kubeconfig` file that does not yet exist; it will place the bootstrapped config file here
- A path to a bootstrap `kubeconfig` file to provide the URL for the server and bootstrap credentials, e.g. a bootstrap token
- Optional: instructions to rotate certificates

The bootstrap `kubeconfig` should be in a path available to the kubelet, for example `/var/lib/kubelet/bootstrap-kubeconfig`.

Its format is identical to a normal `kubeconfig` file. A sample file might look as follows:

```
apiVersion: v1
kind: Configclusters:- cluster:    certificate-authority: /var/lib/kubernetes/ca.pem    server: https://my.server.example.com:6443
```

The important elements to note are:

- `certificate-authority`: path to a CA file, used to validate the server certificate presented by kube-apiserver
- `server`: URL to kube-apiserver
- `token`: the token to use

The format of the token does not matter, as long as it matches what kube-apiserver expects. In the above example, we used a bootstrap token. As stated earlier, *any* valid authentication method can be used, not only tokens.

Because the bootstrap `kubeconfig` *is* a standard `kubeconfig`, you can use `kubectl` to generate it. To create the above example file:

```
kubectl config --kubeconfig=/var/lib/kubelet/bootstrap-kubeconfig set-cluster bootstrap --server='https://my.server.example.com:64
kubectl config --kubeconfig=/var/lib/kubelet/bootstrap-kubeconfig set-credentials kubelet-bootstrap --token=07401b.f395accd246ae52
kubectl config --kubeconfig=/var/lib/kubelet/bootstrap-kubeconfig set-context bootstrap --user=kubelet-bootstrap --cluster=bootstr
kubectl config --kubeconfig=/var/lib/kubelet/bootstrap-kubeconfig use-context bootstrap
```

To indicate to the kubelet to use the bootstrap `kubeconfig`, use the following kubelet flag:

```
--bootstrap-kubeconfig="/var/lib/kubelet/bootstrap-kubeconfig" --kubeconfig="/var/lib/kubelet/kubeconfig"
```

When starting the kubelet, if the file specified via `--kubeconfig` does not exist, the bootstrap kubeconfig specified via `--bootstrap-kubeconfig` is used to request a client certificate from the API server. On approval of the certificate request and receipt back by the kubelet, a kubeconfig file referencing the generated key and obtained certificate is written to the path specified by `--kubeconfig`. The certificate and key file will be placed in the directory specified by `--cert-dir`.

Client and serving certificates

All of the above relate to kubelet *client* certificates, specifically, the certificates a kubelet uses to authenticate to kube-apiserver.

A kubelet also can use *serving* certificates. The kubelet itself exposes an https endpoint for certain features. To secure these, the kubelet can do one of:

- use provided key and certificate, via the `--tls-private-key-file` and `--tls-cert-file` flags
- create self-signed key and certificate, if a key and certificate are not provided
- request serving certificates from the cluster server, via the CSR API

The client certificate provided by TLS bootstrapping is signed, by default, for `client auth` only, and thus cannot be used as serving certificates, or `server auth`.

However, you *can* enable its server certificate, at least partially, via certificate rotation.

Certificate rotation

Kubernetes v1.8 and higher kubelet implements features for enabling rotation of its client and/or serving certificates. Note, rotation of serving certificate is a **beta** feature and requires the `RotateKubeletServerCertificate` feature flag on the kubelet (enabled by default).

You can configure the kubelet to rotate its client certificates by creating new CSRs as its existing credentials expire. To enable this feature, use the `rotateCertificates` field of kubelet configuration file or pass the following command line argument to the kubelet (deprecated):

```
--rotate-certificates
```

Enabling `RotateKubeletServerCertificate` causes the kubelet **both** to request a serving certificate after bootstrapping its client credentials **and** to rotate that certificate. To enable this behavior, use the field `serverTLSBootstrap` of the kubelet configuration file or pass the following command line argument to the kubelet (deprecated):

```
--rotate-server-certificates
```

Note:

The CSR approving controllers implemented in core Kubernetes do not approve node *serving* certificates for security reasons. To use `RotateKubeletServerCertificate` operators need to run a custom approving controller, or manually approve the serving certificate requests.

A deployment-specific approval process for kubelet serving certificates should typically only approve CSRs which:

1. are requested by nodes (ensure the `spec.username` field is of the form `system:node:<nodeName>` and `spec.groups` contains `system:nodes`)
2. request usages for a serving certificate (ensure `spec.usages` contains `server auth`, optionally contains `digital signature` and `key encipherment`, and contains no other usages)
3. only have IP and DNS subjectAltNames that belong to the requesting node, and have no URI and Email subjectAltNames (parse the x509 Certificate Signing Request in `spec.request` to verify `subjectAltNames`)

Other authenticating components

All of TLS bootstrapping described in this document relates to the kubelet. However, other components may need to communicate directly with kube-apiserver. Notable is kube-proxy, which is part of the Kubernetes node components and runs on every node, but may also include other components such as monitoring or networking.

Like the kubelet, these other components also require a method of authenticating to kube-apiserver. You have several options for generating these credentials:

- The old way: Create and distribute certificates the same way you did for kubelet before TLS bootstrapping
- DaemonSet: Since the kubelet itself is loaded on each node, and is sufficient to start base services, you can run kube-proxy and other node-specific services not as a standalone process, but rather as a daemonset in the `kube-system` namespace. Since it will be in-cluster, you can give it a proper service account with appropriate permissions to perform its activities. This may be the simplest way to configure such services.

kubectl approval

CSRs can be approved outside of the approval flows built into the controller manager.

The signing controller does not immediately sign all certificate requests. Instead, it waits until they have been flagged with an "Approved" status by an appropriately-privileged user. This flow is intended to allow for automated approval handled by an external approval controller or the approval controller implemented in the core controller-manager. However cluster administrators can also manually approve certificate requests using kubectl. An administrator can list CSRs with `kubectl get csr` and describe one in detail with `kubectl describe csr <name>`. An administrator can approve or deny a CSR with `kubectl certificate approve <name>` and `kubectl certificate deny <name>`.

Admission Control in Kubernetes

This page provides an overview of *admission controllers*.

An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the resource, but after the request is authenticated and authorized.

Several important features of Kubernetes require an admission controller to be enabled in order to properly support the feature. As a result, a Kubernetes API server that is not properly configured with the right set of admission controllers is an incomplete server that will not support all the features you expect.

What are they?

Admission controllers are code within the Kubernetes API server that check the data arriving in a request to modify a resource.

Admission controllers apply to requests that create, delete, or modify objects. Admission controllers can also block custom verbs, such as a request to connect to a pod via an API server proxy. Admission controllers do *not* (and cannot) block requests to read (**get**, **watch** or **list**) objects, because reads bypass the admission control layer.

Admission control mechanisms may be *validating*, *mutating*, or both. Mutating controllers may modify the data for the resource being modified; validating controllers may not.

The admission controllers in Kubernetes 1.34 consist of the list below, are compiled into the `kube-apiserver` binary, and may only be configured by the cluster administrator.

Admission control extension points

Within the full list, there are three special controllers: MutatingAdmissionWebhook, ValidatingAdmissionWebhook, and ValidatingAdmissionPolicy. The two webhook controllers execute the mutating and validating (respectively) admission control webhooks which are configured in the API. ValidatingAdmissionPolicy provides a way to embed declarative validation code within the API, without relying on any external HTTP callouts.

You can use these three admission controllers to customize cluster behavior at admission time.

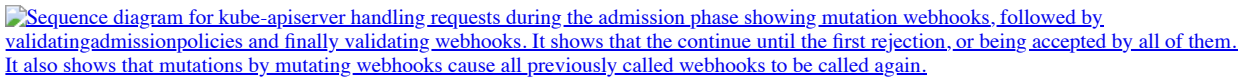
Admission control phases

The admission control process proceeds in two phases. In the first phase, mutating admission controllers are run. In the second phase, validating admission controllers are run. Note again that some of the controllers are both.

If any of the controllers in either phase reject the request, the entire request is rejected immediately and an error is returned to the end-user.

Finally, in addition to sometimes mutating the object in question, admission controllers may sometimes have side effects, that is, mutate related resources as part of request processing. Incrementing quota usage is the canonical example of why this is necessary. Any such side-effect needs a corresponding reclamation or reconciliation process, as a given admission controller does not know for sure that a given request will pass all of the other admission controllers.

The ordering of these calls can be seen below.


Sequence diagram for kube-apiserver handling requests during the admission phase showing mutation webhooks, followed by validatingadmissionpolicies and finally validating webhooks. It shows that the continue until the first rejection, or being accepted by all of them. It also shows that mutations by mutating webhooks cause all previously called webhooks to be called again.

Why do I need them?

Several important features of Kubernetes require an admission controller to be enabled in order to properly support the feature. As a result, a Kubernetes API server that is not properly configured with the right set of admission controllers is an incomplete server and will not support all the features you expect.

How do I turn on an admission controller?

The Kubernetes API server flag `enable-admission-plugins` takes a comma-delimited list of admission control plugins to invoke prior to modifying objects in the cluster. For example, the following command line enables the `NamespaceLifecycle` and the `LimitRanger` admission control plugins:

```
kube-apiserver --enable-admission-plugins=NamespaceLifecycle,LimitRanger ...
```

Note:

Depending on the way your Kubernetes cluster is deployed and how the API server is started, you may need to apply the settings in different ways. For example, you may have to modify the systemd unit file if the API server is deployed as a systemd service, you may modify the manifest file for the API server if Kubernetes is deployed in a self-hosted way.

How do I turn off an admission controller?

The Kubernetes API server flag `disable-admission-plugins` takes a comma-delimited list of admission control plugins to be disabled, even if they are in the list of plugins enabled by default.

```
kube-apiserver --disable-admission-plugins=PodNodeSelector,AlwaysDeny ...
```

Which plugins are enabled by default?

To see which admission plugins are enabled:

```
kube-apiserver -h | grep enable-admission-plugins
```

In Kubernetes 1.34, the default ones are:

```
CertificateApproval, CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass, DefaultStorageClass, DefaultTolerationI
```

What does each admission controller do?

AlwaysAdmit

FEATURE STATE: `Kubernetes v1.13 [deprecated]`

Type: Validating.

This admission controller allows all pods into the cluster. It is **deprecated** because its behavior is the same as if there were no admission controller at all.

AlwaysDeny

FEATURE STATE: `Kubernetes v1.13 [deprecated]`

Type: Validating.

Rejects all requests. AlwaysDeny is **deprecated** as it has no real meaning.

AlwaysPullImages

Type: Mutating and Validating.

This admission controller modifies every new Pod to force the image pull policy to `Always`. This is useful in a multitenant cluster so that users can be assured that their private images can only be used by those who have the credentials to pull them. Without this admission controller, once an image has been pulled to a node, any pod from any user can use it by knowing the image's name (assuming the Pod is scheduled onto the right node), without any authorization check against the image. When this admission controller is enabled, images are always pulled prior to starting containers, which means valid credentials are required.

CertificateApproval

Type: Validating.

This admission controller observes requests to approve CertificateSigningRequest resources and performs additional authorization checks to ensure the approving user has permission to **approve** certificate requests with the `spec.signerName` requested on the CertificateSigningRequest resource.

See Certificate Signing Requests for more information on the permissions required to perform different actions on CertificateSigningRequest resources.

CertificateSigning

Type: Validating.

This admission controller observes updates to the `status.certificate` field of CertificateSigningRequest resources and performs an additional authorization checks to ensure the signing user has permission to **sign** certificate requests with the `spec.signerName` requested on the CertificateSigningRequest resource.

See Certificate Signing Requests for more information on the permissions required to perform different actions on CertificateSigningRequest resources.

CertificateSubjectRestriction

Type: Validating.

This admission controller observes creation of CertificateSigningRequest resources that have a `spec.signerName` of `kubernetes.io/kube-apiserver-client`. It rejects any request that specifies a 'group' (or 'organization attribute') of `system:masters`.

DefaultIngressClass

Type: Mutating.

This admission controller observes creation of `Ingress` objects that do not request any specific ingress class and automatically adds a default ingress class to them. This way, users that do not request any special ingress class do not need to care about them at all and they will get the default one.

This admission controller does not do anything when no default ingress class is configured. When more than one ingress class is marked as default, it rejects any creation of `Ingress` with an error and an administrator must revisit their `IngressClass` objects and mark only one as default (with the annotation "ingressclass.kubernetes.io/is-default-class"). This admission controller ignores any `Ingress` updates; it acts only on creation.

See the Ingress documentation for more about ingress classes and how to mark one as default.

DefaultStorageClass

Type: Mutating.

This admission controller observes creation of `PersistentVolumeClaim` objects that do not request any specific storage class and automatically adds a default storage class to them. This way, users that do not request any special storage class do not need to care about them at all and they will get the default one.

This admission controller does nothing when no default `StorageClass` exists. When more than one storage class is marked as default, and you then create a `PersistentVolumeClaim` with no `storageClassName` set, Kubernetes uses the most recently created default `StorageClass`. When a `PersistentVolumeClaim` is created with a specified `volumeName`, it remains in a pending state if the static volume's `storageClassName` does not match the `storageClassName` on the `PersistentVolumeClaim` after any default StorageClass is applied to it. This admission controller ignores any `PersistentVolumeClaim` updates; it acts only on creation.

See persistent volume documentation about persistent volume claims and storage classes and how to mark a storage class as default.

DefaultTolerationSeconds

Type: Mutating.

This admission controller sets the default forgiveness toleration for pods to tolerate the taints `notready:NoExecute` and `unreachable:NoExecute` based on the k8s-apiserver input parameters `default-not-ready-toleration-seconds` and `default-unreachable-toleration-seconds` if the pods don't already have toleration for taints `node.kubernetes.io/not-ready:NoExecute` or `node.kubernetes.io/unreachable:NoExecute`. The default value for `default-not-ready-toleration-seconds` and `default-unreachable-toleration-seconds` is 5 minutes.

DenyServiceExternalIPs

Type: Validating.

This admission controller rejects all net-new usage of the `Service` field `externalIPs`. This feature is very powerful (allows network traffic interception) and not well controlled by policy. When enabled, users of the cluster may not create new Services which use `externalIPs` and may not add new values to `externalIPs` on existing `Service` objects. Existing uses of `externalIPs` are not affected, and users may remove values from `externalIPs` on existing `Service` objects.

Most users do not need this feature at all, and cluster admins should consider disabling it. Clusters that do need to use this feature should consider using some custom policy to manage usage of it.

This admission controller is disabled by default.

EventRateLimit

FEATURE STATE: `Kubernetes v1.13 [alpha]`

Type: Validating.

This admission controller mitigates the problem where the API server gets flooded by requests to store new Events. The cluster admin can specify event rate limits by:

- Enabling the `EventRateLimit` admission controller;
- Referencing an `EventRateLimit` configuration file from the file provided to the API server's command line flag `--admission-control-config-file`:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:  - name: EventRateLimit    path: eventconfig.yaml...
```

There are four types of limits that can be specified in the configuration:

- `Server`: All Event requests (creation or modifications) received by the API server share a single bucket.
- `Namespace`: Each namespace has a dedicated bucket.
- `User`: Each user is allocated a bucket.
- `SourceAndObject`: A bucket is assigned by each combination of source and involved object of the event.

Below is a sample `eventconfig.yaml` for such a configuration:

```
apiVersion: eventratelimit.admission.k8s.io/v1alpha1
kind: Configurationlimits:  - type: Namespace    qps: 50    burst: 100    cacheSize: 2000  - type: User    qps: 10    burst: 50
```

See the EventRateLimit Config API (v1alpha1) for more details.

This admission controller is disabled by default.

ExtendedResourceToleration

Type: Mutating.

This plug-in facilitates creation of dedicated nodes with extended resources. If operators want to create dedicated nodes with extended resources (like GPUs, FPGAs etc.), they are expected to taint the node with the extended resource name as the key. This admission controller, if enabled, automatically adds tolerations for such taints to pods requesting extended resources, so users don't have to manually add these tolerations.

This admission controller is disabled by default.

ImagePolicyWebhook

Type: Validating.

The ImagePolicyWebhook admission controller allows a backend webhook to make admission decisions.

This admission controller is disabled by default.

Configuration file format

ImagePolicyWebhook uses a configuration file to set options for the behavior of the backend. This file may be json or yaml and has the following format:

```yaml
imagePolicy:
  kubeConfigFile: /path/to/kubeconfig/for/backend
  # time in s to cache approval
  allowTTL: 50
  # time in s to cache denial
  denyTTL: 50
  # time in ms to wait between retries
  retryBackoff: 500
  # determines behavior if the webhook backend fails
  defaultAllow: true
```

Reference the ImagePolicyWebhook configuration file from the file provided to the API server's command line flag `--admission-control-config-file`:

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:  - name: ImagePolicyWebhook    path: imagepolicyconfig.yaml...
```

Alternatively, you can embed the configuration directly in the file:

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:  - name: ImagePolicyWebhook    configuration:      imagePolicy:        kubeConfigFile: <path-t
```

The ImagePolicyWebhook config file must reference a kubeconfig formatted file which sets up the connection to the backend. It is required that the backend communicate over TLS.

The kubeconfig file's `cluster` field must point to the remote service, and the `user` field must contain the returned authorizer.

```yaml
# clusters refers to the remote service.
clusters:  - name: name-of-remote-imagepolicy-service    cluster:      certificate-authority: /path/to/ca.pem    # CA for verifyin
```

For additional HTTP configuration, refer to the kubeconfig documentation.

Request payloads

When faced with an admission decision, the API Server POSTs a JSON serialized `imagepolicy.k8s.io/v1alpha1 ImageReview` object describing the action. This object contains fields describing the containers being admitted, as well as any pod annotations that match `*.image-policy.k8s.io/*`.

Note:

The webhook API objects are subject to the same versioning compatibility rules as other Kubernetes API objects. Implementers should be aware of looser compatibility promises for alpha objects and check the `apiVersion` field of the request to ensure correct deserialization. Additionally, the API Server must enable the `imagepolicy.k8s.io/v1alpha1` API extensions group (`--runtime-config=imagepolicy.k8s.io/v1alpha1=true`).

An example request body:

```json
{
  "apiVersion": "imagepolicy.k8s.io/v1alpha1",
  "kind": "ImageReview",
  "spec": {
    "containers": [
      {
        "image": "myrepo/myimage:v1"
      },
      {
        "image": "myrepo/myimage@sha256:beb6bd6a68f114c1dc2ea4b28db81bdf91de202a9014972bec5e4d9171d90ed"
      }
    ],
    "annotations": {
      "mycluster.image-policy.k8s.io/ticket-1234": "break-glass"
    },
    "namespace": "mynamespace"
  }
}
```

The remote service is expected to fill the `status` field of the request and respond to either allow or disallow access. The response body's `spec` field is ignored, and may be omitted. A permissive response would return:

```json
{
  "apiVersion": "imagepolicy.k8s.io/v1alpha1",
  "kind": "ImageReview",
  "status": {
    "allowed": true
  }
}
```

To disallow access, the service would return:

```json
{
  "apiVersion": "imagepolicy.k8s.io/v1alpha1",
  "kind": "ImageReview",
  "status": {
    "allowed": false,
    "reason": "image currently blacklisted"
  }
```

}

For further documentation refer to the `imagepolicy.v1alpha1` API.

Extending with Annotations

All annotations on a Pod that match `*.image-policy.k8s.io/*` are sent to the webhook. Sending annotations allows users who are aware of the image policy backend to send extra information to it, and for different backends implementations to accept different information.

Examples of information you might put here are:

- request to "break glass" to override a policy, in case of emergency.
- a ticket number from a ticket system that documents the break-glass request
- provide a hint to the policy server as to the imageID of the image being provided, to save it a lookup

In any case, the annotations are provided by the user and are not validated by Kubernetes in any way.

LimitPodHardAntiAffinityTopology

Type: Validating.

This admission controller denies any pod that defines `AntiAffinity` topology key other than `kubernetes.io/hostname` in `requiredDuringSchedulingRequiredDuringExecution`.

This admission controller is disabled by default.

LimitRanger

Type: Mutating and Validating.

This admission controller will observe the incoming request and ensure that it does not violate any of the constraints enumerated in the `LimitRange` object in a `Namespace`. If you are using `LimitRange` objects in your Kubernetes deployment, you MUST use this admission controller to enforce those constraints. LimitRanger can also be used to apply default resource requests to Pods that don't specify any; currently, the default LimitRanger applies a 0.1 CPU requirement to all Pods in the `default` namespace.

See the LimitRange API reference and the example of LimitRange for more details.

MutatingAdmissionWebhook

Type: Mutating.

This admission controller calls any mutating webhooks which match the request. Matching webhooks are called in serial; each one may modify the object if it desires.

This admission controller (as implied by the name) only runs in the mutating phase.

If a webhook called by this has side effects (for example, decrementing quota) it *must* have a reconciliation system, as it is not guaranteed that subsequent webhooks or validating admission controllers will permit the request to finish.

If you disable the MutatingAdmissionWebhook, you must also disable the `MutatingWebhookConfiguration` object in the `admissionregistration.k8s.io/v1` group/version via the `--runtime-config` flag, both are on by default.

Use caution when authoring and installing mutating webhooks

- Users may be confused when the objects they try to create are different from what they get back.
- Built in control loops may break when the objects they try to create are different when read back.
  - Setting originally unset fields is less likely to cause problems than overwriting fields set in the original request. Avoid doing the latter.
- Future changes to control loops for built-in resources or third-party resources may break webhooks that work well today. Even when the webhook installation API is finalized, not all possible webhook behaviors will be guaranteed to be supported indefinitely.

NamespaceAutoProvision

Type: Mutating.

This admission controller examines all incoming requests on namespaced resources and checks if the referenced namespace does exist. It creates a namespace if it cannot be found. This admission controller is useful in deployments that do not want to restrict creation of a namespace prior to its usage.

NamespaceExists

Type: Validating.

This admission controller checks all requests on namespaced resources other than `Namespace` itself. If the namespace referenced from a request doesn't exist, the request is rejected.

NamespaceLifecycle

Type: Validating.

This admission controller enforces that a `Namespace` that is undergoing termination cannot have new objects created in it, and ensures that requests in a non-existent `Namespace` are rejected. This admission controller also prevents deletion of three system reserved namespaces `default`, `kube-system`, `kube-public`.

A `Namespace` deletion kicks off a sequence of operations that remove all objects (pods, services, etc.) in that namespace. In order to enforce integrity of that process, we strongly recommend running this admission controller.

NodeRestriction

Type: Validating.

This admission controller limits the `Node` and `Pod` objects a kubelet can modify. In order to be limited by this admission controller, kubelets must use credentials in the `system:nodes` group, with a username in the form `system:node:<nodeName>`. Such kubelets will only be allowed to modify their own `Node` API object, and only modify `Pod` API objects that are bound to their node. kubelets are not allowed to update or remove taints from their `Node` API object.

The `NodeRestriction` admission plugin prevents kubelets from deleting their `Node` API object, and enforces kubelet modification of labels under the `kubernetes.io/` or `k8s.io/` prefixes as follows:

- **Prevents** kubelets from adding/removing/updating labels with a `node-restriction.kubernetes.io/` prefix. This label prefix is reserved for administrators to label their `Node` objects for workload isolation purposes, and kubelets will not be allowed to modify labels with that prefix.
- **Allows** kubelets to add/remove/update these labels and label prefixes:
  - `kubernetes.io/hostname`
  - `kubernetes.io/arch`
  - `kubernetes.io/os`
  - `beta.kubernetes.io/instance-type`
  - `node.kubernetes.io/instance-type`
  - `failure-domain.beta.kubernetes.io/region` (deprecated)
  - `failure-domain.beta.kubernetes.io/zone` (deprecated)
  - `topology.kubernetes.io/region`
  - `topology.kubernetes.io/zone`
  - `kubelet.kubernetes.io/`-prefixed labels
  - `node.kubernetes.io/`-prefixed labels

Use of any other labels under the `kubernetes.io` or `k8s.io` prefixes by kubelets is reserved, and may be disallowed or allowed by the `NodeRestriction` admission plugin in the future.

Future versions may add additional restrictions to ensure kubelets have the minimal set of permissions required to operate correctly.

OwnerReferencesPermissionEnforcement

Type: Validating.

This admission controller protects the access to the `metadata.ownerReferences` of an object so that only users with **delete** permission to the object can change it. This admission controller also protects the access to `metadata.ownerReferences[x].blockOwnerDeletion` of an object, so that only users with **update** permission to the `finalizers` subresource of the referenced *owner* can change it.

PersistentVolumeClaimResize

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

Type: Validating.

This admission controller implements additional validations for checking incoming `PersistentVolumeClaim` resize requests.

Enabling the `PersistentVolumeClaimResize` admission controller is recommended. This admission controller prevents resizing of all claims by default unless a claim's `StorageClass` explicitly enables resizing by setting `allowVolumeExpansion` to `true`.

For example: all `PersistentVolumeClaim`s created from the following `StorageClass` support volume expansion:

```
apiVersion: storage.k8s.io/v1
kind: StorageClassmetadata:  name: gluster-vol-defaultprovisioner: kubernetes.io/glusterfsparameters:  resturl: "http://192.168.10
```

For more information about persistent volume claims, see PersistentVolumeClaims.

PodNodeSelector

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

Type: Validating.

This admission controller defaults and limits what node selectors may be used within a namespace by reading a namespace annotation and a global configuration.

This admission controller is disabled by default.

Configuration file format

`PodNodeSelector` uses a configuration file to set options for the behavior of the backend. Note that the configuration file format will move to a versioned file in a future release. This file may be json or yaml and has the following format:

```
podNodeSelectorPluginConfig:
  clusterDefaultNodeSelector: name-of-node-selector
  namespace1: name-of-node-selector
  namespace2: name-of-node-selector
```

Reference the `PodNodeSelector` configuration file from the file provided to the API server's command line flag `--admission-control-config-file`:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:- name: PodNodeSelector  path: podnodeselector.yaml...
```

Configuration Annotation Format

`PodNodeSelector` uses the annotation key `scheduler.alpha.kubernetes.io/node-selector` to assign node selectors to namespaces.

```
apiVersion: v1
kind: Namespacemetadata:  annotations:    scheduler.alpha.kubernetes.io/node-selector: name-of-node-selector  name: namespace3
```

Internal Behavior

This admission controller has the following behavior:

1. If the `Namespace` has an annotation with a key `scheduler.alpha.kubernetes.io/node-selector`, use its value as the node selector.
2. If the namespace lacks such an annotation, use the `clusterDefaultNodeSelector` defined in the `PodNodeSelector` plugin configuration file as the node selector.
3. Evaluate the pod's node selector against the namespace node selector for conflicts. Conflicts result in rejection.
4. Evaluate the pod's node selector against the namespace-specific allowed selector defined the plugin configuration file. Conflicts result in rejection.

Note:

PodNodeSelector allows forcing pods to run on specifically labeled nodes. Also see the PodTolerationRestriction admission plugin, which allows preventing pods from running on specifically tainted nodes.

PodSecurity

FEATURE STATE: `Kubernetes v1.25 [stable]`

Type: Validating.

The PodSecurity admission controller checks new Pods before they are admitted, determines if it should be admitted based on the requested security context and the restrictions on permitted Pod Security Standards for the namespace that the Pod would be in.

See the Pod Security Admission documentation for more information.

PodSecurity replaced an older admission controller named PodSecurityPolicy.

PodTolerationRestriction

FEATURE STATE: `Kubernetes v1.7 [alpha]`

Type: Mutating and Validating.

The PodTolerationRestriction admission controller verifies any conflict between tolerations of a pod and the tolerations of its namespace. It rejects the pod request if there is a conflict. It then merges the tolerations annotated on the namespace into the tolerations of the pod. The resulting tolerations are checked against a list of allowed tolerations annotated to the namespace. If the check succeeds, the pod request is admitted otherwise it is rejected.

If the namespace of the pod does not have any associated default tolerations or allowed tolerations annotated, the cluster-level default tolerations or cluster-level list of allowed tolerations are used instead if they are specified.

Tolerations to a namespace are assigned via the `scheduler.alpha.kubernetes.io/defaultTolerations` annotation key. The list of allowed tolerations can be added via the `scheduler.alpha.kubernetes.io/tolerationsWhitelist` annotation key.

Example for namespace annotations:

```
apiVersion: v1
kind: Namespacemetadata:  name: apps-that-need-nodes-exclusively  annotations:    scheduler.alpha.kubernetes.io/defaultTolerations
```

This admission controller is disabled by default.

PodTopologyLabels

FEATURE STATE: `Kubernetes v1.33 [alpha]` (enabled by default: false)

Type: Mutating

The PodTopologyLabels admission controller mutates the `pods/binding` subresources for all pods bound to a Node, adding topology labels matching those of the bound Node. This allows Node topology labels to be available as pod labels, which can be surfaced to running containers using the Downward API. The labels available as a result of this controller are the topology.kubernetes.io/region and topology.kuberentes.io/zone labels.

Note:

If any mutating admission webhook adds or modifies labels of the `pods/binding` subresource, these changes will propagate to pod labels as a result of this controller, overwriting labels with conflicting keys.

This admission controller is enabled when the `PodTopologyLabelsAdmission` feature gate is enabled.

Priority

Type: Mutating and Validating.

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.

ResourceQuota

Type: Validating.

This admission controller will observe the incoming request and ensure that it does not violate any of the constraints enumerated in the `ResourceQuota` object in a `Namespace`. If you are using `ResourceQuota` objects in your Kubernetes deployment, you MUST use this admission controller to enforce quota constraints.

See the ResourceQuota API reference and the example of Resource Quota for more details.

RuntimeClass

Type: Mutating and Validating.

If you define a RuntimeClass with Pod overhead configured, this admission controller checks incoming Pods. When enabled, this admission controller rejects any Pod create requests that have the overhead already set. For Pods that have a RuntimeClass configured and selected in their `.spec`, this admission controller sets `.spec.overhead` in the Pod based on the value defined in the corresponding RuntimeClass.

See also Pod Overhead for more information.

ServiceAccount

Type: Mutating and Validating.

This admission controller implements automation for serviceAccounts. The Kubernetes project strongly recommends enabling this admission controller. You should enable this admission controller if you intend to make any use of Kubernetes `ServiceAccount` objects.

To enhance the security measures around Secrets, use separate namespaces to isolate access to mounted secrets.

StorageObjectInUseProtection

Type: Mutating.

The `StorageObjectInUseProtection` plugin adds the `kubernetes.io/pvc-protection` or `kubernetes.io/pv-protection` finalizers to newly created Persistent Volume Claims (PVCs) or Persistent Volumes (PV). In case a user deletes a PVC or PV the PVC or PV is not removed until the finalizer is removed from the PVC or PV by PVC or PV Protection Controller. Refer to the Storage Object in Use Protection for more detailed information.

TaintNodesByCondition

Type: Mutating.

This admission controller taints newly created Nodes as `NotReady` and `NoSchedule`. That tainting avoids a race condition that could cause Pods to be scheduled on new Nodes before their taints were updated to accurately reflect their reported conditions.

ValidatingAdmissionPolicy

Type: Validating.

This admission controller implements the CEL validation for incoming matched requests. It is enabled when both feature gate `validatingadmissionpolicy` and `admissionregistration.k8s.io/v1alpha1` group/version are enabled. If any of the ValidatingAdmissionPolicy fails, the request fails.

ValidatingAdmissionWebhook

Type: Validating.

This admission controller calls any validating webhooks which match the request. Matching webhooks are called in parallel; if any of them rejects the request, the request fails. This admission controller only runs in the validation phase; the webhooks it calls may not mutate the object, as opposed to the webhooks called by the `MutatingAdmissionWebhook` admission controller.

If a webhook called by this has side effects (for example, decrementing quota) it *must* have a reconciliation system, as it is not guaranteed that subsequent webhooks or other validating admission controllers will permit the request to finish.

If you disable the ValidatingAdmissionWebhook, you must also disable the `ValidatingWebhookConfiguration` object in the `admissionregistration.k8s.io/v1` group/version via the `--runtime-config` flag.

Is there a recommended set of admission controllers to use?

Yes. The recommended admission controllers are enabled by default (shown here), so you do not need to explicitly specify them. You can enable additional admission controllers beyond the default set using the `--enable-admission-plugins` flag (**order doesn't matter**).

Mapping PodSecurityPolicies to Pod Security Standards

The tables below enumerate the configuration parameters on `PodSecurityPolicy` objects, whether the field mutates and/or validates pods, and how the configuration values map to the Pod Security Standards.

For each applicable parameter, the allowed values for the Baseline and Restricted profiles are listed. Anything outside the allowed values for those profiles would fall under the Privileged profile. "No opinion" means all values are allowed under all Pod Security Standards.

For a step-by-step migration guide, see Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller.

PodSecurityPolicy Spec

The fields enumerated in this table are part of the `PodSecurityPolicySpec`, which is specified under the `.spec` field path.

PodSecurityPolicySpec	Type	Pod Security Standards Equivalent
`privileged`	Validating	**Baseline & Restricted**: `false` / undefined / nil
`defaultAddCapabilities`	Mutating & Validating	Requirements match `allowedCapabilities` below.
`allowedCapabilities`	Validating	**Baseline**: subset of • `AUDIT_WRITE` • `CHOWN` • `DAC_OVERRIDE` • `FOWNER` • `FSETID` • `KILL` • `MKNOD` • `NET_BIND_SERVICE` • `SETFCAP` • `SETGID` • `SETPCAP` • `SETUID` • `SYS_CHROOT` **Restricted**: empty / undefined / nil OR a list containing *only* `NET_BIND_SERVICE`
`requiredDropCapabilities`	Mutating & Validating	**Baseline**: no opinion **Restricted**: must include `ALL`
`volumes`	Validating	**Baseline**: anything except • `hostPath` • `*` **Restricted**: subset of • `configMap` • `csi` • `downwardAPI` • `emptyDir` • `ephemeral` • `persistentVolumeClaim` • `projected` • `secret`
`hostNetwork`	Validating	**Baseline & Restricted**: `false` / undefined / nil
`hostPorts`	Validating	**Baseline & Restricted**: undefined / nil / empty
`hostPID`	Validating	**Baseline & Restricted**: `false` / undefined / nil
`hostIPC`	Validating	**Baseline & Restricted**: `false` / undefined / nil
`seLinux`	Mutating & Validating	**Baseline & Restricted**: `seLinux.rule` is `MustRunAs`, with the following `options` • `user` is unset (`""` / undefined / nil) • `role` is unset (`""` / undefined / nil) • `type` is unset or one of: `container_t, container_init_t, container_kvm_t, container_engine_t` • `level` is anything
`runAsUser`	Mutating & Validating	**Baseline**: Anything **Restricted**: `rule` is `MustRunAsNonRoot`
`runAsGroup`	Mutating (MustRunAs) & Validating	*No opinion*
`supplementalGroups`	Mutating & Validating	*No opinion*
`fsGroup`	Mutating & Validating	*No opinion*
`readOnlyRootFilesystem`	Mutating & Validating	*No opinion*
`defaultAllowPrivilegeEscalation`	Mutating	*No opinion (non-validating)* *Only mutating if set to `false`*
`allowPrivilegeEscalation`	Mutating & Validating	**Baseline**: No opinion **Restricted**: `false`
`allowedHostPaths`	Validating	*No opinion (volumes takes precedence)*

`allowedFlexVolumes`	Validating	*No opinion (volumes takes precedence)*
`allowedCSIDrivers`	Validating	*No opinion (volumes takes precedence)*
`allowedUnsafeSysctls`	Validating	**Baseline & Restricted**: undefined / nil / empty
`forbiddenSysctls`	Validating	*No opinion*
`allowedProcMountTypes` *(alpha feature)*	Validating	**Baseline & Restricted**: `["Default"]` OR undefined / nil / empty
`runtimeClass .defaultRuntimeClassName`	Mutating	*No opinion*
`runtimeClass .allowedRuntimeClassNames`	Validating	*No opinion*

PodSecurityPolicy annotations

The annotations enumerated in this table can be specified under `.metadata.annotations` on the PodSecurityPolicy object.

PSP Annotation	Type	Pod Security Standards Equivalent
`seccomp.security.alpha.kubernetes.io /defaultProfileName`	Mutating	*No opinion*
`seccomp.security.alpha.kubernetes.io /allowedProfileNames`	Validating	**Baseline**: `"runtime/default,"` *(Trailing comma to allow unset)* **Restricted**: `"runtime/default"` *(No trailing comma)* *`localhost/*` values are also permitted for both Baseline & Restricted.*
`apparmor.security.beta.kubernetes.io /defaultProfileName`	Mutating	*No opinion*
`apparmor.security.beta.kubernetes.io /allowedProfileNames`	Validating	**Baseline**: `"runtime/default,"` *(Trailing comma to allow unset)* **Restricted**: `"runtime/default"` *(No trailing comma)* *`localhost/*` values are also permitted for both Baseline & Restricted.*

Kubelet authentication/authorization

Overview

A kubelet's HTTPS endpoint exposes APIs which give access to data of varying sensitivity, and allow you to perform operations with varying levels of power on the node and within containers.

This document describes how to authenticate and authorize access to the kubelet's HTTPS endpoint.

Kubelet authentication

By default, requests to the kubelet's HTTPS endpoint that are not rejected by other configured authentication methods are treated as anonymous requests, and given a username of `system:anonymous` and a group of `system:unauthenticated`.

To disable anonymous access and send `401 Unauthorized` responses to unauthenticated requests:

- start the kubelet with the `--anonymous-auth=false` flag

To enable X509 client certificate authentication to the kubelet's HTTPS endpoint:

- start the kubelet with the `--client-ca-file` flag, providing a CA bundle to verify client certificates with
- start the apiserver with `--kubelet-client-certificate` and `--kubelet-client-key` flags
- see the apiserver authentication documentation for more details

To enable API bearer tokens (including service account tokens) to be used to authenticate to the kubelet's HTTPS endpoint:

- ensure the `authentication.k8s.io/v1beta1` API group is enabled in the API server
- start the kubelet with the `--authentication-token-webhook` and `--kubeconfig` flags
- the kubelet calls the `TokenReview` API on the configured API server to determine user information from bearer tokens

Kubelet authorization

Any request that is successfully authenticated (including an anonymous request) is then authorized. The default authorization mode is `AlwaysAllow`, which allows all requests.

There are many possible reasons to subdivide access to the kubelet API:

- anonymous auth is enabled, but anonymous users' ability to call the kubelet API should be limited
- bearer token auth is enabled, but arbitrary API users' (like service accounts) ability to call the kubelet API should be limited
- client certificate auth is enabled, but only some of the client certificates signed by the configured CA should be allowed to use the kubelet API

To subdivide access to the kubelet API, delegate authorization to the API server:

- ensure the `authorization.k8s.io/v1beta1` API group is enabled in the API server
- start the kubelet with the `--authorization-mode=Webhook` and the `--kubeconfig` flags

- the kubelet calls the `SubjectAccessReview` API on the configured API server to determine whether each request is authorized

The kubelet authorizes API requests using the same request attributes approach as the apiserver.

The verb is determined from the incoming request's HTTP verb:

HTTP verb	request verb
POST	create
GET, HEAD	get
PUT	update
PATCH	patch
DELETE	delete

The resource and subresource is determined from the incoming request's path:

Kubelet API	resource	subresource
/stats/*	nodes	stats
/metrics/*	nodes	metrics
/logs/*	nodes	log
/spec/*	nodes	spec
/checkpoint/*	nodes	checkpoint
all others	nodes	proxy

The namespace and API group attributes are always an empty string, and the resource name is always the name of the kubelet's `Node` API object.

When running in this mode, ensure the user identified by the `--kubelet-client-certificate` and `--kubelet-client-key` flags passed to the apiserver is authorized for the following attributes:

- verb=*, resource=nodes, subresource=proxy
- verb=*, resource=nodes, subresource=stats
- verb=*, resource=nodes, subresource=log
- verb=*, resource=nodes, subresource=spec
- verb=*, resource=nodes, subresource=metrics

Fine-grained authorization

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: true)

When the feature gate `KubeletFineGrainedAuthz` is enabled kubelet performs a fine-grained check before falling back to the `proxy` subresource for the `/pods`, `/runningPods`, `/configz` and `/healthz` endpoints. The resource and subresource are determined from the incoming request's path:

Kubelet API	resource	subresource
/stats/*	nodes	stats
/metrics/*	nodes	metrics
/logs/*	nodes	log
/pods	nodes	pods, proxy
/runningPods/	nodes	pods, proxy
/healthz	nodes	healthz, proxy
/configz	nodes	configz, proxy
all others	nodes	proxy

When the feature-gate `KubeletFineGrainedAuthz` is enabled, ensure the user identified by the `--kubelet-client-certificate` and `--kubelet-client-key` flags passed to the API server is authorized for the following attributes:

- verb=*, resource=nodes, subresource=proxy
- verb=*, resource=nodes, subresource=stats
- verb=*, resource=nodes, subresource=log
- verb=*, resource=nodes, subresource=metrics
- verb=*, resource=nodes, subresource=configz
- verb=*, resource=nodes, subresource=healthz
- verb=*, resource=nodes, subresource=pods

If RBAC authorization is used, enabling this gate also ensure that the builtin `system:kubelet-api-admin` ClusterRole is updated with permissions to access all the above mentioned subresources.

Managing Service Accounts

A *ServiceAccount* provides an identity for processes that run in a Pod.

A process inside a Pod can use the identity of its associated service account to authenticate to the cluster's API server.

For an introduction to service accounts, read configure service accounts.

This task guide explains some of the concepts behind ServiceAccounts. The guide also explains how to obtain or revoke tokens that represent ServiceAccounts, and how to (optionally) bind a ServiceAccount's validity to the lifetime of an API object.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To be able to follow these steps exactly, ensure you have a namespace named `examplens`. If you don't, create one by running:

```
kubectl create namespace examplens
```

User accounts versus service accounts

Kubernetes distinguishes between the concept of a user account and a service account for a number of reasons:

- User accounts are for humans. Service accounts are for application processes, which (for Kubernetes) run in containers that are part of pods.
- User accounts are intended to be global: names must be unique across all namespaces of a cluster. No matter what namespace you look at, a particular username that represents a user represents the same user. In Kubernetes, service accounts are namespaced: two different namespaces can contain ServiceAccounts that have identical names.
- Typically, a cluster's user accounts might be synchronised from a corporate database, where new user account creation requires special privileges and is tied to complex business processes. By contrast, service account creation is intended to be more lightweight, allowing cluster users to create service accounts for specific tasks on demand. Separating ServiceAccount creation from the steps to onboard human users makes it easier for workloads to follow the principle of least privilege.
- Auditing considerations for humans and service accounts may differ; the separation makes that easier to achieve.
- A configuration bundle for a complex system may include definition of various service accounts for components of that system. Because service accounts can be created without many constraints and have namespaced names, such configuration is usually portable.

Bound service account tokens

ServiceAccount tokens can be bound to API objects that exist in the kube-apiserver. This can be used to tie the validity of a token to the existence of another API object. Supported object types are as follows:

- Pod (used for projected volume mounts, see below)
- Secret (can be used to allow revoking a token by deleting the Secret)
- Node (can be used to auto-revoke a token when its Node is deleted; creating new node-bound tokens is GA in v1.33+)

When a token is bound to an object, the object's `metadata.name` and `metadata.uid` are stored as extra 'private claims' in the issued JWT.

When a bound token is presented to the kube-apiserver, the service account authenticator will extract and verify these claims. If the referenced object or the ServiceAccount is pending deletion (for example, due to finalizers), then for any instant that is 60 seconds (or more) after the `.metadata.deletionTimestamp` date, authentication with that token would fail. If the referenced object no longer exists (or its `metadata.uid` does not match), the request will not be authenticated.

Additional metadata in Pod bound tokens

FEATURE STATE: `Kubernetes v1.32 [stable]` (enabled by default: true)

When a service account token is bound to a Pod object, additional metadata is also embedded into the token that indicates the value of the bound pod's `spec.nodeName` field, and the uid of that Node, if available.

This node information is **not** verified by the kube-apiserver when the token is used for authentication. It is included so integrators do not have to fetch Pod or Node API objects to check the associated Node name and uid when inspecting a JWT.

Verifying and inspecting private claims

The TokenReview API can be used to verify and extract private claims from a token:

1. First, assume you have a pod named `test-pod` and a service account named `my-sa`.

2. Create a token that is bound to this Pod:

   ```
   kubectl create token my-sa --bound-object-kind="Pod" --bound-object-name="test-pod"
   ```

3. Copy this token into a new file named `tokenreview.yaml`:

   ```
   apiVersion: authentication.k8s.io/v1
   kind: TokenReviewspec:  token: <token from step 2>
   ```

4. Submit this resource to the apiserver for review:

   ```
   # use '-o yaml' to inspect the output
   kubectl create -o yaml -f tokenreview.yaml
   ```

   You should see an output like below:

   ```
   apiVersion: authentication.k8s.io/v1
   kind: TokenReviewmetadata:  creationTimestamp: nullspec:  token: <token>status:  audiences:  - https://kubernetes.default.svc
   ```

Note:

Despite using `kubectl create -f` to create this resource, and defining it similar to other resource types in Kubernetes, TokenReview is a special type and the kube-apiserver does not actually persist the TokenReview object into etcd. Hence `kubectl get tokenreview` is not a valid command.

Schema for service account private claims

The schema for the Kubernetes-specific claims within JWT tokens is not currently documented, however the relevant code area can be found in the serviceaccount package in the Kubernetes codebase.

You can inspect a JWT using standard JWT decoding tool. Below is an example of a JWT for the `my-serviceaccount` ServiceAccount, bound to a Pod object named `my-pod` which is scheduled to the Node `my-node`, in the `my-namespace` namespace:

```
{
  "aud": [
    "https://my-audience.example.com"
  ],
  "exp": 1729605240,
  "iat": 1729601640,
  "iss": "https://my-cluster.example.com",
  "jti": "aed34954-b33a-4142-b1ec-389d6bbb4936",
  "kubernetes.io": {
    "namespace": "my-namespace",
    "node": {
      "name": "my-node",
      "uid": "646e7c5e-32d6-4d42-9dbd-e504e6cbe6b1"
    },
    "pod": {
      "name": "my-pod",
      "uid": "5e0bd49b-f040-43b0-99b7-22765a53f7f3"
    },
    "serviceaccount": {
      "name": "my-serviceaccount",
      "uid": "14ee3fa4-a7e2-420f-9f9a-dbc4507c3798"
    }
  },
  "nbf": 1729601640,
  "sub": "system:serviceaccount:my-namespace:my-serviceaccount"
}
```

Note:

The `aud` and `iss` fields in this JWT may differ between different Kubernetes clusters depending on your configuration.

The presence of both the `pod` and `node` claim implies that this token is bound to a *Pod* object. When verifying Pod bound ServiceAccount tokens, the API server **does not** verify the existence of the referenced Node object.

Services that run outside of Kubernetes and want to perform offline validation of JWTs may use this schema, along with a compliant JWT validator configured with OpenID Discovery information from the API server, to verify presented JWTs without requiring use of the TokenReview API.

Services that verify JWTs in this way **do not verify** the claims embedded in the JWT token to be current and still valid. This means if the token is bound to an object, and that object no longer exists, the token will still be considered valid (until the configured token expires).

Clients that require assurance that a token's bound claims are still valid **MUST** use the TokenReview API to present the token to the `kube-apiserver` for it to verify and expand the embedded claims, using similar steps to the Verifying and inspecting private claims section above, but with a supported client library. For more information on JWTs and their structure, see the JSON Web Token RFC.

Bound service account token volume mechanism

FEATURE STATE: `Kubernetes v1.22 [stable]` (enabled by default: true)

By default, the Kubernetes control plane (specifically, the ServiceAccount admission controller) adds a projected volume to Pods, and this volume includes a token for Kubernetes API access.

Here's an example of how that looks for a launched Pod:

```
...
  - name: kube-api-access-<random-suffix>
    projected:
      sources:
        - serviceAccountToken:
            path: token # must match the path the app expects
        - configMap:
            items:
              - key: ca.crt
                path: ca.crt
            name: kube-root-ca.crt
        - downwardAPI:
            items:
              - fieldRef:
                  apiVersion: v1
                  fieldPath: metadata.namespace
                path: namespace
```

That manifest snippet defines a projected volume that consists of three sources. In this case, each source also represents a single path within that volume. The three sources are:

1. A `serviceAccountToken` source, that contains a token that the kubelet acquires from kube-apiserver. The kubelet fetches time-bound tokens using the TokenRequest API. A token served for a TokenRequest expires either when the pod is deleted or after a defined lifespan (by default, that is 1 hour). The kubelet also refreshes that token before the token expires. The token is bound to the specific Pod and has the kube-apiserver as its audience. This mechanism superseded an earlier mechanism that added a volume based on a Secret, where the Secret represented the ServiceAccount for the Pod, but did not expire.
2. A `configMap` source. The ConfigMap contains a bundle of certificate authority data. Pods can use these certificates to make sure that they are connecting to your cluster's kube-apiserver (and not to middlebox or an accidentally misconfigured peer).
3. A `downwardAPI` source that looks up the name of the namespace containing the Pod, and makes that name information available to application code running inside the Pod.

Any container within the Pod that mounts this particular volume can access the above information.

Note:

There is no specific mechanism to invalidate a token issued via TokenRequest. If you no longer trust a bound service account token for a Pod, you can delete that Pod. Deleting a Pod expires its bound service account tokens.

Manual Secret management for ServiceAccounts

Versions of Kubernetes before v1.22 automatically created credentials for accessing the Kubernetes API. This older mechanism was based on creating token Secrets that could then be mounted into running Pods.

In more recent versions, including Kubernetes v1.34, API credentials are obtained directly using the TokenRequest API, and are mounted into Pods using a projected volume. The tokens obtained using this method have bounded lifetimes, and are automatically invalidated when the Pod they are mounted into is deleted.

You can still manually create a Secret to hold a service account token; for example, if you need a token that never expires.

Once you manually create a Secret and link it to a ServiceAccount, the Kubernetes control plane automatically populates the token into that Secret.

Note:

Although the manual mechanism for creating a long-lived ServiceAccount token exists, using TokenRequest to obtain short-lived API access tokens is recommended instead.

Auto-generated legacy ServiceAccount token clean up

Before version 1.24, Kubernetes automatically generated Secret-based tokens for ServiceAccounts. To distinguish between automatically generated tokens and manually created ones, Kubernetes checks for a reference from the ServiceAccount's secrets field. If the Secret is referenced in the `secrets` field, it is considered an auto-generated legacy token. Otherwise, it is considered a manually created legacy token. For example:

```
apiVersion: v1
kind: ServiceAccountmetadata:  name: build-robot  namespace: defaultsecrets:  - name: build-robot-secret # usually NOT present for
```

Beginning from version 1.29, legacy ServiceAccount tokens that were generated automatically will be marked as invalid if they remain unused for a certain period of time (set to default at one year). Tokens that continue to be unused for this defined period (again, by default, one year) will subsequently be purged by the control plane.

If users use an invalidated auto-generated token, the token validator will

1. add an audit annotation for the key-value pair `authentication.k8s.io/legacy-token-invalidated: <secret name>/<namespace>`,
2. increment the `invalid_legacy_auto_token_uses_total` metric count,
3. update the Secret label `kubernetes.io/legacy-token-last-used` with the new date,
4. return an error indicating that the token has been invalidated.

When receiving this validation error, users can update the Secret to remove the `kubernetes.io/legacy-token-invalid-since` label to temporarily allow use of this token.

Here's an example of an auto-generated legacy token that has been marked with the `kubernetes.io/legacy-token-last-used` and `kubernetes.io/legacy-token-invalid-since` labels:

```
apiVersion: v1
kind: Secretmetadata:  name: build-robot-secret  namespace: default  labels:    kubernetes.io/legacy-token-last-used: 2022-10-24
```

Control plane details

ServiceAccount controller

A ServiceAccount controller manages the ServiceAccounts inside namespaces, and ensures a ServiceAccount named "default" exists in every active namespace.

Token controller

The service account token controller runs as part of `kube-controller-manager`. This controller acts asynchronously. It:

- watches for ServiceAccount deletion and deletes all corresponding ServiceAccount token Secrets.
- watches for ServiceAccount token Secret addition, and ensures the referenced ServiceAccount exists, and adds a token to the Secret if needed.
- watches for Secret deletion and removes a reference from the corresponding ServiceAccount if needed.

You must pass a service account private key file to the token controller in the `kube-controller-manager` using the `--service-account-private-key-file` flag. The private key is used to sign generated service account tokens. Similarly, you must pass the corresponding public key to the `kube-apiserver` using the `--service-account-key-file` flag. The public key will be used to verify the tokens during authentication.

FEATURE STATE: `Kubernetes v1.34 [beta]` (enabled by default: true)

An alternate setup to setting `--service-account-private-key-file` and `--service-account-key-file` flags is to configure an external JWT signer for external ServiceAccount token signing and key management. Note that these setups are mutually exclusive and cannot be configured together.

ServiceAccount admission controller

The modification of pods is implemented via a plugin called an Admission Controller. It is part of the API server. This admission controller acts synchronously to modify pods as they are created. When this plugin is active (and it is by default on most distributions), then it does the following when a Pod is created:

1. If the pod does not have a `.spec.serviceAccountName` set, the admission controller sets the name of the ServiceAccount for this incoming Pod to `default`.
2. The admission controller ensures that the ServiceAccount referenced by the incoming Pod exists. If there is no ServiceAccount with a matching name, the admission controller rejects the incoming Pod. That check applies even for the `default` ServiceAccount.
3. Provided that neither the ServiceAccount's `automountServiceAccountToken` field nor the Pod's `automountServiceAccountToken` field is set to `false`:
   - the admission controller mutates the incoming Pod, adding an extra volume that contains a token for API access.
   - the admission controller adds a `volumeMount` to each container in the Pod, skipping any containers that already have a volume mount defined for the path `/var/run/secrets/kubernetes.io/serviceaccount`. For Linux containers, that volume is mounted at `/var/run/secrets/kubernetes.io/serviceaccount`; on Windows nodes, the mount is at the equivalent path.
4. If the spec of the incoming Pod doesn't already contain any `imagePullSecrets`, then the admission controller adds `imagePullSecrets`, copying them from the `ServiceAccount`.

Legacy ServiceAccount token tracking controller

FEATURE STATE: `Kubernetes v1.28 [stable]` (enabled by default: true)

This controller generates a ConfigMap called `kube-system/kube-apiserver-legacy-service-account-token-tracking` in the `kube-system` namespace. The ConfigMap records the timestamp when legacy service account tokens began to be monitored by the system.

Legacy ServiceAccount token cleaner

FEATURE STATE: `Kubernetes v1.30 [stable]` (enabled by default: true)

The legacy ServiceAccount token cleaner runs as part of the `kube-controller-manager` and checks every 24 hours to see if any auto-generated legacy ServiceAccount token has not been used in a *specified amount of time*. If so, the cleaner marks those tokens as invalid.

The cleaner works by first checking the ConfigMap created by the control plane (provided that `LegacyServiceAccountTokenTracking` is enabled). If the current time is a *specified amount of time* after the date in the ConfigMap, the cleaner then loops through the list of Secrets in the cluster and evaluates each Secret that has the type `kubernetes.io/service-account-token`.

If a Secret meets all of the following conditions, the cleaner marks it as invalid:

- The Secret is auto-generated, meaning that it is bi-directionally referenced by a ServiceAccount.
- The Secret is not currently mounted by any pods.
- The Secret has not been used in a *specified amount of time* since it was created or since it was last used.

The cleaner marks a Secret invalid by adding a label called `kubernetes.io/legacy-token-invalid-since` to the Secret, with the current date as the value. If an invalid Secret is not used in a *specified amount of time*, the cleaner will delete it.

Note:

All the *specified amount of time* above defaults to one year. The cluster administrator can configure this value through the `--legacy-service-account-token-clean-up-period` command line argument for the `kube-controller-manager` component.

TokenRequest API

FEATURE STATE: `Kubernetes v1.22 [stable]`

You use the TokenRequest subresource of a ServiceAccount to obtain a time-bound token for that ServiceAccount. You don't need to call this to obtain an API token for use within a container, since the kubelet sets this up for you using a *projected volume*.

If you want to use the TokenRequest API from `kubectl`, see Manually create an API token for a ServiceAccount.

The Kubernetes control plane (specifically, the ServiceAccount admission controller) adds a projected volume to Pods, and the kubelet ensures that this volume contains a token that lets containers authenticate as the right ServiceAccount.

(This mechanism superseded an earlier mechanism that added a volume based on a Secret, where the Secret represented the ServiceAccount for the Pod but did not expire.)

Here's an example of how that looks for a launched Pod:

```
...
  - name: kube-api-access-<random-suffix>
    projected:
      defaultMode: 420 # decimal equivalent of octal 0644
      sources:
```

```
    - serviceAccountToken:
        expirationSeconds: 3607
        path: token
    - configMap:
        items:
          - key: ca.crt
            path: ca.crt
        name: kube-root-ca.crt
    - downwardAPI:
        items:
          - fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
            path: namespace
```

That manifest snippet defines a projected volume that combines information from three sources:

1. A `serviceAccountToken` source, that contains a token that the kubelet acquires from kube-apiserver. The kubelet fetches time-bound tokens using the TokenRequest API. A token served for a TokenRequest expires either when the pod is deleted or after a defined lifespan (by default, that is 1 hour). The token is bound to the specific Pod and has the kube-apiserver as its audience.
2. A `configMap` source. The ConfigMap contains a bundle of certificate authority data. Pods can use these certificates to make sure that they are connecting to your cluster's kube-apiserver (and not to a middlebox or an accidentally misconfigured peer).
3. A `downwardAPI` source. This `downwardAPI` volume makes the name of the namespace containing the Pod available to application code running inside the Pod.

Any container within the Pod that mounts this volume can access the above information.

Create additional API tokens

Caution:

Only create long-lived API tokens if the token request mechanism is not suitable. The token request mechanism provides time-limited tokens; because these expire, they represent a lower risk to information security.

To create a non-expiring, persisted API token for a ServiceAccount, create a Secret of type `kubernetes.io/service-account-token` with an annotation referencing the ServiceAccount. The control plane then generates a long-lived token and updates that Secret with that generated token data.

Here is a sample manifest for such a Secret:

secret/serviceaccount/mysecretname.yaml Copy secret/serviceaccount/mysecretname.yaml to clipboard

```
apiVersion: v1
kind: Secrettype: kubernetes.io/service-account-tokenmetadata:  name: mysecretname  annotations:    kubernetes.io/service-account.
```

To create a Secret based on this example, run:

```
kubectl -n examplens create -f https://k8s.io/examples/secret/serviceaccount/mysecretname.yaml
```

To see the details for that Secret, run:

```
kubectl -n examplens describe secret mysecretname
```

The output is similar to:

```
Name:           mysecretname
Namespace:      examplens
Labels:         <none>
Annotations:    kubernetes.io/service-account.name=myserviceaccount
                kubernetes.io/service-account.uid=8a85c4c4-8483-11e9-bc42-526af7764f64

Type:   kubernetes.io/service-account-token

Data
====
ca.crt:         1362 bytes
namespace:      9 bytes
token:          ...
```

If you launch a new Pod into the `examplens` namespace, it can use the `myserviceaccount` service-account-token Secret that you just created.

Caution:

Do not reference manually created Secrets in the `secrets` field of a ServiceAccount. Or the manually created Secrets will be cleaned if it is not used for a long time. Please refer to auto-generated legacy ServiceAccount token clean up.

Delete/invalidate a ServiceAccount token

Delete/invalidate a long-lived/legacy ServiceAccount token

If you know the name of the Secret that contains the token you want to remove:

```
kubectl delete secret name-of-secret
```

Otherwise, first find the Secret for the ServiceAccount.

```
# This assumes that you already have a namespace named 'examplens'
```

```
kubectl -n examplens get serviceaccount/example-automated-thing -o yaml
```

The output is similar to:

```
apiVersion: v1
kind: ServiceAccountmetadata:  annotations:    kubectl.kubernetes.io/last-applied-configuration: |      {"apiVersion":"v1","kind":
```

Then, delete the Secret you now know the name of:

```
kubectl -n examplens delete secret/example-automated-thing-token-zyxwv
```

Delete/invalidate a short-lived ServiceAccount token

Short lived ServiceAccount tokens automatically expire after the time-limit specified during their creation. There is no central record of tokens issued, so there is no way to revoke individual tokens.

If you have to revoke a short-lived token before its expiration, you can delete and re-create the ServiceAccount it is associated to. This will change its UID and hence invalidate **all** ServiceAccount tokens that were created for it.

External ServiceAccount token signing and key management

FEATURE STATE: `Kubernetes v1.34 [beta]` (enabled by default: true)

The kube-apiserver can be configured to use external signer for token signing and token verifying key management. This feature enables kubernetes distributions to integrate with key management solutions of their choice (for example, HSMs, cloud KMSes) for service account credential signing and verification. To configure kube-apiserver to use external-jwt-signer set the `--service-account-signing-endpoint` flag to the location of a Unix domain socket (UDS) on a filesystem, or be prefixed with an @ symbol and name a UDS in the abstract socket namespace. At the configured UDS shall be an RPC server which implements an `ExternalJWTSigner` gRPC service.

The external-jwt-signer must be healthy and be ready to serve supported service account keys for the kube-apiserver to start.

Note:

The kube-apiserver flags `--service-account-key-file` and `--service-account-signing-key-file` will continue to be used for reading from files unless `--service-account-signing-endpoint` is set; they are mutually exclusive ways of supporting JWT signing and authentication.

An external signer provides a `v1.ExternalJWTSigner` gRPC service that implements 3 methods:

Metadata

Metadata is meant to be called once by `kube-apiserver` on startup. This enables the external signer to share metadata with kube-apiserver, like the max token lifetime that signer supports.

```
rpc Metadata(MetadataRequest) returns (MetadataResponse) {}
```

```
message MetadataRequest {}message MetadataResponse {  // used by kube-apiserver for defaulting/validation of JWT lifetime while ac
```

FetchKeys

FetchKeys returns the set of public keys that are trusted to sign Kubernetes service account tokens. Kube-apiserver will call this RPC:

- Every time it tries to validate a JWT from the service account issuer with an unknown key ID, and
- Periodically, so it can serve reasonably-up-to-date keys from the OIDC JWKs endpoint.

```
rpc FetchKeys(FetchKeysRequest) returns (FetchKeysResponse) {}
```

```
message FetchKeysRequest {}message FetchKeysResponse {  repeated Key keys = 1;  // The timestamp when this data was pulled from th
```

Sign

Sign takes a serialized JWT payload, and returns the serialized header and signature. `kube-apiserver` then assembles the JWT from the header, payload, and signature.

```
rpc Sign(SignJWTRequest) returns (SignJWTResponse) {}
```

```
message SignJWTRequest {  // URL-safe base64 wrapped payload to be signed.  // Exactly as it appears in the second segment of the 
```

Clean up

If you created a namespace `examplens` to experiment with, you can remove it:

```
kubectl delete namespace examplens
```

What's next

- Read more details about projected volumes.

Validating Admission Policy

FEATURE STATE: `Kubernetes v1.30 [stable]`

This page provides an overview of Validating Admission Policy.

What is Validating Admission Policy?

Validating admission policies offer a declarative, in-process alternative to validating admission webhooks.

Validating admission policies use the Common Expression Language (CEL) to declare the validation rules of a policy. Validation admission policies are highly configurable, enabling policy authors to define policies that can be parameterized and scoped to resources as needed by cluster administrators.

What Resources Make a Policy

A policy is generally made up of three resources:

- The `ValidatingAdmissionPolicy` describes the abstract logic of a policy (think: "this policy makes sure a particular label is set to a particular value").
- A parameter resource provides information to a ValidatingAdmissionPolicy to make it a concrete statement (think "the `owner` label must be set to something that ends in `.company.com`"). A native type such as ConfigMap or a CRD defines the schema of a parameter resource. `ValidatingAdmissionPolicy` objects specify what Kind they are expecting for their parameter resource.
- A `ValidatingAdmissionPolicyBinding` links the above resources together and provides scoping. If you only want to require an `owner` label to be set for `Pods`, the binding is where you would specify this restriction.

At least a `ValidatingAdmissionPolicy` and a corresponding `ValidatingAdmissionPolicyBinding` must be defined for a policy to have an effect.

If a `ValidatingAdmissionPolicy` does not need to be configured via parameters, simply leave `spec.paramKind` in `ValidatingAdmissionPolicy` not specified.

Getting Started with Validating Admission Policy

Validating Admission Policy is part of the cluster control-plane. You should write and deploy them with great caution. The following describes how to quickly experiment with Validating Admission Policy.

Creating a ValidatingAdmissionPolicy

The following is an example of a ValidatingAdmissionPolicy.

validatingadmissionpolicy/basic-example-policy.yaml Copy validatingadmissionpolicy/basic-example-policy.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "demo-policy.example.com"spec:  failurePolicy: Fail  matchConstraints:    resourceI
```

`spec.validations` contains CEL expressions which use the Common Expression Language (CEL) to validate the request. If an expression evaluates to false, the validation check is enforced according to the `spec.failurePolicy` field.

Note:

You can quickly test CEL expressions in CEL Playground.

To configure a validating admission policy for use in a cluster, a binding is required. The following is an example of a ValidatingAdmissionPolicyBinding.:

validatingadmissionpolicy/basic-example-binding.yaml Copy validatingadmissionpolicy/basic-example-binding.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicyBindingmetadata:  name: "demo-binding-test.example.com"spec:  policyName: "demo-policy.example.com"
```

When trying to create a deployment with replicas set not satisfying the validation expression, an error will return containing message:

```
ValidatingAdmissionPolicy 'demo-policy.example.com' with binding 'demo-binding-test.example.com' denied request: failed expression
```

The above provides a simple example of using ValidatingAdmissionPolicy without a parameter configured.

Validation actions

Each `ValidatingAdmissionPolicyBinding` must specify one or more `validationActions` to declare how `validations` of a policy are enforced.

The supported `validationActions` are:

- `Deny`: Validation failure results in a denied request.
- `Warn`: Validation failure is reported to the request client as a warning.
- `Audit`: Validation failure is included in the audit event for the API request.

For example, to both warn clients about a validation failure and to audit the validation failures, use:

```
validationActions: [Warn, Audit]
```

`Deny` and `Warn` may not be used together since this combination needlessly duplicates the validation failure both in the API response body and the HTTP warning headers.

A `validation` that evaluates to false is always enforced according to these actions. Failures defined by the `failurePolicy` are enforced according to these actions only if the `failurePolicy` is set to `Fail` (or not specified), otherwise the failures are ignored.

See Audit Annotations: validation failures for more details about the validation failure audit annotation.

Parameter resources

Parameter resources allow a policy configuration to be separate from its definition. A policy can define paramKind, which outlines GVK of the parameter resource, and then a policy binding ties a policy by name (via policyName) to a particular parameter resource via paramRef.

If parameter configuration is needed, the following is an example of a ValidatingAdmissionPolicy with parameter configuration.

validatingadmissionpolicy/policy-with-param.yaml Copy validatingadmissionpolicy/policy-with-param.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "replicalimit-policy.example.com"spec:  failurePolicy: Fail  paramKind:    apiVers
```

The `spec.paramKind` field of the ValidatingAdmissionPolicy specifies the kind of resources used to parameterize this policy. For this example, it is configured by ReplicaLimit custom resources. Note in this example how the CEL expression references the parameters via the CEL params variable, e.g. `params.maxReplicas`. `spec.matchConstraints` specifies what resources this policy is designed to validate. Note that the native types such like `ConfigMap` could also be used as parameter reference.

The `spec.validations` fields contain CEL expressions. If an expression evaluates to false, the validation check is enforced according to the `spec.failurePolicy` field.

The validating admission policy author is responsible for providing the ReplicaLimit parameter CRD.

To configure an validating admission policy for use in a cluster, a binding and parameter resource are created. The following is an example of a ValidatingAdmissionPolicyBinding that uses a **cluster-wide** param - the same param will be used to validate every resource request that matches the binding:

validatingadmissionpolicy/binding-with-param.yaml Copy validatingadmissionpolicy/binding-with-param.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicyBindingmetadata:  name: "replicalimit-binding-test.example.com"spec:  policyName: "replicalimit-pol
```

Notice this binding applies a parameter to the policy for all resources which are in the `test` environment.

The parameter resource could be as following:

validatingadmissionpolicy/replicalimit-param.yaml Copy validatingadmissionpolicy/replicalimit-param.yaml to clipboard

```
apiVersion: rules.example.com/v1
kind: ReplicaLimitmetadata:  name: "replica-limit-test.example.com"  namespace: "default"maxReplicas: 3
```

This policy parameter resource limits deployments to a max of 3 replicas.

An admission policy may have multiple bindings. To bind all other environments to have a maxReplicas limit of 100, create another ValidatingAdmissionPolicyBinding:

validatingadmissionpolicy/binding-with-param-prod.yaml Copy validatingadmissionpolicy/binding-with-param-prod.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicyBindingmetadata:  name: "replicalimit-binding-nontest"spec:  policyName: "replicalimit-policy.examp
```

Notice this binding applies a different parameter to resources which are not in the `test` environment.

And have a parameter resource:

validatingadmissionpolicy/replicalimit-param-prod.yaml Copy validatingadmissionpolicy/replicalimit-param-prod.yaml to clipboard

```
apiVersion: rules.example.com/v1
kind: ReplicaLimitmetadata:  name: "replica-limit-prod.example.com"maxReplicas: 100
```

For each admission request, the API server evaluates CEL expressions of each (policy, binding, param) combination that match the request. For a request to be admitted it must pass **all** evaluations.

If multiple bindings match the request, the policy will be evaluated for each, and they must all pass evaluation for the policy to be considered passed.

If multiple parameters match a single binding, the policy rules will be evaluated for each param, and they too must all pass for the binding to be considered passed. Bindings can have overlapping match criteria. The policy is evaluated for each matching binding-parameter combination. A policy may even be evaluated multiple times if multiple bindings match it, or a single binding that matches multiple parameters.

The params object representing a parameter resource will not be set if a parameter resource has not been bound, so for policies requiring a parameter resource, it can be useful to add a check to ensure one has been bound. A parameter resource will not be bound and `params` will be null if `paramKind` of the policy, or `paramRef` of the binding are not specified.

For the use cases requiring parameter configuration, we recommend to add a param check in `spec.validations[0].expression`:

```
- expression: "params != null"
  message: "params missing but required to bind to this policy"
```

Optional parameters

It can be convenient to be able to have optional parameters as part of a parameter resource, and only validate them if present. CEL provides `has()`, which checks if the key passed to it exists. CEL also implements Boolean short-circuiting. If the first half of a logical OR evaluates to true, it won't evaluate the

other half (since the result of the entire OR will be true regardless).

Combining the two, we can provide a way to validate optional parameters:

```
!has(params.optionalNumber) || (params.optionalNumber >= 5 && params.optionalNumber <= 10)
```

Here, we first check that the optional parameter is present with `!has(params.optionalNumber)`.

- If `optionalNumber` hasn't been defined, then the expression short-circuits since `!has(params.optionalNumber)` will evaluate to true.
- If `optionalNumber` has been defined, then the latter half of the CEL expression will be evaluated, and optionalNumber will be checked to ensure that it contains a value between 5 and 10 inclusive.

Per-namespace Parameters

As the author of a ValidatingAdmissionPolicy and its ValidatingAdmissionPolicyBinding, you can choose to specify cluster-wide, or per-namespace parameters. If you specify a `namespace` for the binding's `paramRef`, the control plane only searches for parameters in that namespace.

However, if `namespace` is not specified in the ValidatingAdmissionPolicyBinding, the API server can search for relevant parameters in the namespace that a request is against. For example, if you make a request to modify a ConfigMap in the `default` namespace and there is a relevant ValidatingAdmissionPolicyBinding with no `namespace` set, then the API server looks for a parameter object in `default`. This design enables policy configuration that depends on the namespace of the resource being manipulated, for more fine-tuned control.

Parameter selector

In addition to specify a parameter in a binding by `name`, you may choose instead to specify label selector, such that all resources of the policy's `paramKind`, and the param's `namespace` (if applicable) that match the label selector are selected for evaluation. See selector for more information on how label selectors match resources.

If multiple parameters are found to meet the condition, the policy's rules are evaluated for each parameter found and the results will be ANDed together.

If `namespace` is provided, only objects of the `paramKind` in the provided namespace are eligible for selection. Otherwise, when `namespace` is empty and `paramKind` is namespace-scoped, the `namespace` used in the request being admitted will be used.

Authorization checks

We introduced the authorization check for parameter resources. User is expected to have `read` access to the resources referenced by `paramKind` in `ValidatingAdmissionPolicy` and `paramRef` in `ValidatingAdmissionPolicyBinding`.

Note that if a resource in `paramKind` fails resolving via the restmapper, `read` access to all resources of groups is required.

`paramRef`

The `paramRef` field specifies the parameter resource used by the policy. It has the following fields:

- **name**: The name of the parameter resource.
- **namespace**: The namespace of the parameter resource.
- **selector**: A label selector to match multiple parameter resources.
- **parameterNotFoundAction**: (Required) Controls the behavior when the specified parameters are not found.
    - **Allowed Values**:
        - **`Allow`**: The absence of matched parameters is treated as a successful validation by the binding.
        - **`Deny`**: The absence of matched parameters is subject to the `failurePolicy` of the policy.

One of `name` or `selector` must be set, but not both.

Note:

The `parameterNotFoundAction` field in `paramRef` is **required**. It specifies the action to take when no parameters are found matching the `paramRef`. If not specified, the policy binding may be considered invalid and will be ignored or could lead to unexpected behavior.

- **`Allow`**: If set to `Allow`, and no parameters are found, the binding treats the absence of parameters as a successful validation, and the policy is considered to have passed.
- **`Deny`**: If set to `Deny`, and no parameters are found, the binding enforces the `failurePolicy` of the policy. If the `failurePolicy` is `Fail`, the request is rejected.

Make sure to set `parameterNotFoundAction` according to the desired behavior when parameters are missing.

Handling Missing Parameters with `parameterNotFoundAction`

When using `paramRef` with a selector, it's possible that no parameters match the selector. The `parameterNotFoundAction` field determines how the binding behaves in this scenario.

Example:

```
apiVersion: admissionregistration.k8s.io/v1alpha1
kind: ValidatingAdmissionPolicyBindingmetadata:  name: example-bindingspec:  policyName: example-policy  paramRef:    selector:
```

Failure Policy

`failurePolicy` defines how mis-configurations and CEL expressions evaluating to error from the admission policy are handled. Allowed values are `Ignore` or `Fail`.

- `Ignore` means that an error calling the ValidatingAdmissionPolicy is ignored and the API request is allowed to continue.
- `Fail` means that an error calling the ValidatingAdmissionPolicy causes the admission to fail and the API request to be rejected.

Note that the `failurePolicy` is defined inside `ValidatingAdmissionPolicy`:

validatingadmissionpolicy/failure-policy-ignore.yaml  Copy validatingadmissionpolicy/failure-policy-ignore.yaml to clipboard

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicyspec:...failurePolicy: Ignore # The default is "Fail"validations:- expression: "object.spec.xyz == 
```

Validation Expression

`spec.validations[i].expression` represents the expression which will be evaluated by CEL. To learn more, see the CEL language specification CEL expressions have access to the contents of the Admission request/response, organized into CEL variables as well as some other useful variables:

- 'object' - The object from the incoming request. The value is null for DELETE requests.
- 'oldObject' - The existing object. The value is null for CREATE requests.
- 'request' - Attributes of the admission request.
- 'params' - Parameter resource referred to by the policy binding being evaluated. The value is null if `ParamKind` is not specified.
- `namespaceObject` - The namespace, as a Kubernetes resource, that the incoming object belongs to. The value is null if the incoming object is cluster-scoped.
- `authorizer` - A CEL Authorizer. May be used to perform authorization checks for the principal (authenticated user) of the request. See AuthzSelectors and Authz in the Kubernetes CEL library documentation for more details.
- `authorizer.requestResource` - A shortcut for an authorization check configured with the request resource (group, resource, (subresource), namespace, name).

In CEL expressions, variables like `object` and `oldObject` are strongly-typed. You can access any field in the object's schema, such as `object.metadata.labels` and fields in `spec`.

For any Kubernetes object, including schemaless Custom Resources, CEL guarantees access to a minimal set of properties: `apiVersion`, `kind`, `metadata.name`, and `metadata.generateName`.

Equality on arrays with list type of 'set' or 'map' ignores element order, i.e. [1, 2] == [2, 1]. Concatenation on arrays with x-kubernetes-list-type use the semantics of the list type:

- 'set': `X + Y` performs a union where the array positions of all elements in `X` are preserved and non-intersecting elements in `Y` are appended, retaining their partial order.
- 'map': `X + Y` performs a merge where the array positions of all keys in `X` are preserved but the values are overwritten by values in `Y` when the key sets of `X` and `Y` intersect. Elements in `Y` with non-intersecting keys are appended, retaining their partial order.

Validation expression examples

Expression	Purpose
`object.minReplicas <= object.replicas && object.replicas <= object.maxReplicas`	Validate that the three fields defining replicas are ordered appropriately
`'Available' in object.stateCounts`	Validate that an entry with the 'Available' key exists in a map
`(size(object.list1) == 0) != (size(object.list2) == 0)`	Validate that one of two lists is non-empty, but not both
`!('MY_KEY' in object.map1) || object['MY_KEY'].matches('^[a-zA-Z]*$')`	Validate the value of a map for a specific key, if it is in the map
`object.envars.filter(e, e.name == 'MY_ENV').all(e, e.value.matches('^[a-zA-Z]*$')`	Validate the 'value' field of a listMap entry where key field 'name' is 'MY_ENV'
`has(object.expired) && object.created + object.ttl < object.expired`	Validate that 'expired' date is after a 'create' date plus a 'ttl' duration
`object.health.startsWith('ok')`	Validate a 'health' string field has the prefix 'ok'
`object.widgets.exists(w, w.key == 'x' && w.foo < 10)`	Validate that the 'foo' property of a listMap item with a key 'x' is less than 10
`type(object) == string ? object == '100%' : object == 1000`	Validate an int-or-string field for both the int and string cases
`object.metadata.name.startsWith(object.prefix)`	Validate that an object's name has the prefix of another field value
`object.set1.all(e, !(e in object.set2))`	Validate that two listSets are disjoint
`size(object.names) == size(object.details) && object.names.all(n, n in object.details)`	Validate the 'details' map is keyed by the items in the 'names' listSet
`size(object.clusters.filter(c, c.name == object.primary)) == 1`	Validate that the 'primary' property has one and only one occurrence in the 'clusters' listMap

Read Supported evaluation on CEL for more information about CEL rules.

`spec.validation[i].reason` represents a machine-readable description of why this validation failed. If this is the first validation in the list to fail, this reason, as well as the corresponding HTTP response code, are used in the HTTP response to the client. The currently supported reasons are: `Unauthorized`, `Forbidden`, `Invalid`, `RequestEntityTooLarge`. If not set, `StatusReasonInvalid` is used in the response to the client.

Matching requests: `matchConditions`

You can define *match conditions* for a `ValidatingAdmissionPolicy` if you need fine-grained request filtering. These conditions are useful if you find that match rules, `objectSelectors` and `namespaceSelectors` still doesn't provide the filtering you want. Match conditions are CEL expressions. All match conditions must evaluate to true for the resource to be evaluated.

Here is an example illustrating a few different uses for match conditions:

access/validating-admission-policy-match-conditions.yaml Copy access/validating-admission-policy-match-conditions.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "demo-policy.example.com"spec:  failurePolicy: Fail  matchConstraints:    resourcel
```

Match conditions have access to the same CEL variables as validation expressions.

In the event of an error evaluating a match condition the policy is not evaluated. Whether to reject the request is determined as follows:

1. If **any** match condition evaluated to `false` (regardless of other errors), the API server skips the policy.
2. Otherwise:
    - for `failurePolicy: Fail`, reject the request (without evaluating the policy).
    - for `failurePolicy: Ignore`, proceed with the request but skip the policy.

Audit annotations

`auditAnnotations` may be used to include audit annotations in the audit event of the API request.

For example, here is an admission policy with an audit annotation:

access/validating-admission-policy-audit-annotation.yaml  Copy access/validating-admission-policy-audit-annotation.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "demo-policy.example.com"spec:  failurePolicy: Fail  matchConstraints:    resourcel
```

When an API request is validated with this admission policy, the resulting audit event will look like:

```
# the audit event recorded
{
    "kind": "Event",
    "apiVersion": "audit.k8s.io/v1",
    "annotations": {
        "demo-policy.example.com/high-replica-count": "Deployment spec.replicas set to 128"
        # other annotations
        ...
    }
    # other fields
    ...
}
```

In this example the annotation will only be included if the `spec.replicas` of the Deployment is more than 50, otherwise the CEL expression evaluates to null and the annotation will not be included.

Note that audit annotation keys are prefixed by the name of the `ValidatingAdmissionPolicy` and a `/`. If another admission controller, such as an admission webhook, uses the exact same audit annotation key, the value of the first admission controller to include the audit annotation will be included in the audit event and all other values will be ignored.

Message expression

To return a more friendly message when the policy rejects a request, we can use a CEL expression to composite a message with `spec.validations[i].messageExpression`. Similar to the validation expression, a message expression has access to `object`, `oldObject`, `request`, `params`, and `namespaceObject`. Unlike validations, message expression must evaluate to a string.

For example, to better inform the user of the reason of denial when the policy refers to a parameter, we can have the following validation:

access/deployment-replicas-policy.yaml  Copy access/deployment-replicas-policy.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "deploy-replica-policy.example.com"spec:  paramKind:    apiVersion: rules.example.(
```

After creating a params object that limits the replicas to 3 and setting up the binding, when we try to create a deployment with 5 replicas, we will receive the following message.

```
$ kubectl create deploy --image=nginx nginx --replicas=5
error: failed to create deployment: deployments.apps "nginx" is forbidden: ValidatingAdmissionPolicy 'deploy-replica-policy.exampl
```

This is more informative than a static message of "too many replicas".

The message expression takes precedence over the static message defined in `spec.validations[i].message` if both are defined. However, if the message expression fails to evaluate, the static message will be used instead. Additionally, if the message expression evaluates to a multi-line string, the evaluation result will be discarded and the static message will be used if present. Note that static message is validated against multi-line strings.

Type checking

When a policy definition is created or updated, the validation process parses the expressions it contains and reports any syntax errors, rejecting the definition if any errors are found. Afterward, the referred variables are checked for type errors, including missing fields and type confusion, against the matched types of `spec.matchConstraints`. The result of type checking can be retrieved from `status.typeChecking`. The presence of `status.typeChecking` indicates the completion of type checking, and an empty `status.typeChecking` means that no errors were detected.

For example, given the following policy definition:

validatingadmissionpolicy/typechecking.yaml  Copy validatingadmissionpolicy/typechecking.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "deploy-replica-policy.example.com"spec:  matchConstraints:    resourceRules:    -
```

The status will yield the following information:

```yaml
status:
  typeChecking:
    expressionWarnings:
    - fieldRef: spec.validations[0].expression
      warning: |-
        apps/v1, Kind=Deployment: ERROR: <input>:1:7: undefined field 'replicas'
         | object.replicas > 1
         | ......^
```

If multiple resources are matched in `spec.matchConstraints`, all of matched resources will be checked against. For example, the following policy definition

validatingadmissionpolicy/typechecking-multiple-match.yaml Copy validatingadmissionpolicy/typechecking-multiple-match.yaml to clipboard

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicymetadata:  name: "replica-policy.example.com"spec:  matchConstraints:    resourceRules:    - apiGro
```

will have multiple types and type checking result of each type in the warning message.

```yaml
status:
  typeChecking:
    expressionWarnings:
    - fieldRef: spec.validations[0].expression
      warning: |-
        apps/v1, Kind=Deployment: ERROR: <input>:1:7: undefined field 'replicas'
         | object.replicas > 1
         | ......^
        apps/v1, Kind=ReplicaSet: ERROR: <input>:1:7: undefined field 'replicas'
         | object.replicas > 1
         | ......^
```

Type Checking has the following limitation:

- No wildcard matching. If `spec.matchConstraints.resourceRules` contains `"*"` in any of `apiGroups`, `apiVersions` or `resources`, the types that `"*"` matches will not be checked.
- The number of matched types is limited to 10. This is to prevent a policy that manually specifying too many types. to consume excessive computing resources. In the order of ascending group, version, and then resource, 11th combination and beyond are ignored.
- Type Checking does not affect the policy behavior in any way. Even if the type checking detects errors, the policy will continue to evaluate. If errors do occur during evaluate, the failure policy will decide its outcome.
- Type Checking does not apply to CRDs, including matched CRD types and reference of paramKind. The support for CRDs will come in future release.

Variable composition

If an expression grows too complicated, or part of the expression is reusable and computationally expensive to evaluate, you can extract some part of the expressions into variables. A variable is a named expression that can be referred later in `variables` in other expressions.

```yaml
spec:
  variables:
    - name: foo
      expression: "'foo' in object.spec.metadata.labels ? object.spec.metadata.labels['foo'] : 'default'"
  validations:
    - expression: variables.foo == 'bar'
```

A variable is lazily evaluated when it is first referred. Any error that occurs during the evaluation will be reported during the evaluation of the referring expression. Both the result and potential error are memorized and count only once towards the runtime cost.

The order of variables are important because a variable can refer to other variables that are defined before it. This ordering prevents circular references.

The following is a more complex example of enforcing that image repo names match the environment defined in its namespace.

access/image-matches-namespace-environment.policy.yaml Copy access/image-matches-namespace-environment.policy.yaml to clipboard

```yaml
# This policy enforces that all containers of a deployment has the image repo match the environment label of its namespace.
# Except for "exempt" deployments, or any containers that do not belong to the "example.com" organization (e.g. common sidecars).#
```

With the policy bound to the namespace `default`, which is labeled `environment: prod`, the following attempt to create a deployment would be rejected.

```
kubectl create deploy --image=dev.example.com/nginx invalid
```

The error message is similar to this.

```
error: failed to create deployment: deployments.apps "invalid" is forbidden: ValidatingAdmissionPolicy 'image-matches-namespace-en
```

API kinds exempt from admission validation

There are certain API kinds that are exempt from admission-time validation checks. For example, you can't create a ValidatingAdmissionPolicy that prevents changes to ValidatingAdmissionPolicyBindings.

The list of exempt API kinds is:

- ValidatingAdmissionPolicies
- ValidatingAdmissionPolicyBindings
- MutatingAdmissionPolicies
- MutatingAdmissionPolicyBindings
- TokenReviews

- LocalSubjectAccessReviews
- SelfSubjectAccessReviews
- SelfSubjectReviews

Using RBAC Authorization

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization.

RBAC authorization uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing you to dynamically configure policies through the Kubernetes API.

To enable RBAC, start the API server with the `--authorization-config` flag set to a file that includes the `RBAC` authorizer; for example:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AuthorizationConfigurationauthorizers:  ...  - type: RBAC  ...
```

Or, start the API server with the `--authorization-mode` flag set to a comma-separated list that includes `RBAC`; for example:

```
kube-apiserver --authorization-mode=...,RBAC --other-options --more-options
```

API objects

The RBAC API declares four kinds of Kubernetes object: *Role*, *ClusterRole*, *RoleBinding* and *ClusterRoleBinding*. You can describe or amend the RBAC objects using tools such as `kubectl`, just like any other Kubernetes object.

Caution:

These objects, by design, impose access restrictions. If you are making changes to a cluster as you learn, see privilege escalation prevention and bootstrapping to understand how those restrictions can prevent you making some changes.

Role and ClusterRole

An RBAC *Role* or *ClusterRole* contains rules that represent a set of permissions. Permissions are purely additive (there are no "deny" rules).

A Role always sets permissions within a particular namespace; when you create a Role, you have to specify the namespace it belongs in.

ClusterRole, by contrast, is a non-namespaced resource. The resources have different names (Role and ClusterRole) because a Kubernetes object always has to be either namespaced or not namespaced; it can't be both.

ClusterRoles have several uses. You can use a ClusterRole to:

1. define permissions on namespaced resources and be granted access within individual namespace(s)
2. define permissions on namespaced resources and be granted access across all namespaces
3. define permissions on cluster-scoped resources

If you want to define a role within a namespace, use a Role; if you want to define a role cluster-wide, use a ClusterRole.

Role example

Here's an example Role in the "default" namespace that can be used to grant read access to pods:

access/simple-role.yaml Copy access/simple-role.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Rolemetadata:  namespace: default  name: pod-readerrules:- apiGroups: [""] # "" indicates the core API group  resources: ["p
```

ClusterRole example

A ClusterRole can be used to grant the same permissions as a Role. Because ClusterRoles are cluster-scoped, you can also use them to grant access to:

- cluster-scoped resources (like nodes)
- non-resource endpoints (like `/healthz`)
- namespaced resources (like Pods), across all namespaces

  For example: you can use a ClusterRole to allow a particular user to run `kubectl get pods --all-namespaces`

Here is an example of a ClusterRole that can be used to grant read access to secrets in any particular namespace, or across all namespaces (depending on how it is bound):

access/simple-clusterrole.yaml Copy access/simple-clusterrole.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  # "namespace" omitted since ClusterRoles are not namespaced  name: secret-readerrules:- apiGroups: [""
```

The name of a Role or a ClusterRole object must be a valid path segment name.

RoleBinding and ClusterRoleBinding

A role binding grants the permissions defined in a role to a user or set of users. It holds a list of *subjects* (users, groups, or service accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

A RoleBinding may reference any Role in the same namespace. Alternatively, a RoleBinding can reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding. If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding.

The name of a RoleBinding or ClusterRoleBinding object must be a valid path segment name.

RoleBinding examples

Here is an example of a RoleBinding that grants the "pod-reader" Role to the user "jane" within the "default" namespace. This allows "jane" to read pods in the "default" namespace.

access/simple-rolebinding-with-role.yaml Copy access/simple-rolebinding-with-role.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.# You need to already have a Role named "pod-reader" in
```

A RoleBinding can also reference a ClusterRole to grant the permissions defined in that ClusterRole to resources inside the RoleBinding's namespace. This kind of reference lets you define a set of common roles across your cluster, then reuse them within multiple namespaces.

For instance, even though the following RoleBinding refers to a ClusterRole, "dave" (the subject, case sensitive) will only be able to read Secrets in the "development" namespace, because the RoleBinding's namespace (in its metadata) is "development".

access/simple-rolebinding-with-clusterrole.yaml Copy access/simple-rolebinding-with-clusterrole.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "dave" to read secrets in the "development" namespace.# You need to already have a ClusterRole named "s
```

ClusterRoleBinding example

To grant permissions across a whole cluster, you can use a ClusterRoleBinding. The following ClusterRoleBinding allows any user in the group "manager" to read secrets in any namespace.

access/simple-clusterrolebinding.yaml Copy access/simple-clusterrolebinding.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.kind: ClusterRoleBindingmetadata
```

After you create a binding, you cannot change the Role or ClusterRole that it refers to. If you try to change a binding's `roleRef`, you get a validation error. If you do want to change the `roleRef` for a binding, you need to remove the binding object and create a replacement.

There are two reasons for this restriction:

1. Making `roleRef` immutable allows granting someone `update` permission on an existing binding object, so that they can manage the list of subjects, without being able to change the role that is granted to those subjects.
2. A binding to a different role is a fundamentally different binding. Requiring a binding to be deleted/recreated in order to change the `roleRef` ensures the full list of subjects in the binding is intended to be granted the new role (as opposed to enabling or accidentally modifying only the roleRef without verifying all of the existing subjects should be given the new role's permissions).

The `kubectl auth reconcile` command-line utility creates or updates a manifest file containing RBAC objects, and handles deleting and recreating binding objects if required to change the role they refer to. See command usage and examples for more information.

Referring to resources

In the Kubernetes API, most resources are represented and accessed using a string representation of their object name, such as `pods` for a Pod. RBAC refers to resources using exactly the same name that appears in the URL for the relevant API endpoint. Some Kubernetes APIs involve a *subresource*, such as the logs for a Pod. A request for a Pod's logs looks like:

```
GET /api/v1/namespaces/{namespace}/pods/{name}/log
```

In this case, `pods` is the namespaced resource for Pod resources, and `log` is a subresource of `pods`. To represent this in an RBAC role, use a slash (`/`) to delimit the resource and subresource. To allow a subject to read `pods` and also access the `log` subresource for each of those Pods, you write:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Rolemetadata:  namespace: default  name: pod-and-pod-logs-readerrules:- apiGroups: [""]  resources: ["pods", "pods/log"]  ve
```

You can also refer to resources by name for certain requests through the `resourceNames` list. When specified, requests can be restricted to individual instances of a resource. Here is an example that restricts its subject to only `get` or `update` a ConfigMap named `my-configmap`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Rolemetadata:  namespace: default  name: configmap-updaterrules:- apiGroups: [""]  #  # at the HTTP level, the name of the r
```

Note:

You cannot restrict `create` or `deletecollection` requests by their resource name. For `create`, this limitation is because the name of the new object may not be known at authorization time. If you restrict `list` or `watch` by resourceName, clients must include a `metadata.name` field selector in their `list` or `watch` request that matches the specified resourceName in order to be authorized. For example, `kubectl get configmaps --field-selector=metadata.name=my-configmap`

Rather than referring to individual `resources`, `apiGroups`, and `verbs`, you can use the wildcard `*` symbol to refer to all such objects. For `nonResourceURLs`, you can use the wildcard `*` as a suffix glob match. For `resourceNames`, an empty set means that everything is allowed. Here is an example that allows access to perform any current and future action on all current and future resources in the `example.com` API group. This is similar to the built-in `cluster-admin` role.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Rolemetadata:  namespace: default  name: example.com-superuser # DO NOT USE THIS ROLE, IT IS JUST AN EXAMPLErules:- apiGroup
```

Caution:

Using wildcards in resource and verb entries could result in overly permissive access being granted to sensitive resources. For instance, if a new resource type is added, or a new subresource is added, or a new custom verb is checked, the wildcard entry automatically grants access, which may be undesirable. The principle of least privilege should be employed, using specific resources and verbs to ensure only the permissions required for the workload to function correctly are applied.

Aggregated ClusterRoles

You can *aggregate* several ClusterRoles into one combined ClusterRole. A controller, running as part of the cluster control plane, watches for ClusterRole objects with an `aggregationRule` set. The `aggregationRule` defines a label selector that the controller uses to match other ClusterRole objects that should be combined into the `rules` field of this one.

Caution:

The control plane overwrites any values that you manually specify in the `rules` field of an aggregate ClusterRole. If you want to change or add rules, do so in the `ClusterRole` objects that are selected by the `aggregationRule`.

Here is an example aggregated ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: monitoringaggregationRule:  clusterRoleSelectors:  - matchLabels:  rbac.example.com/aggregat
```

If you create a new ClusterRole that matches the label selector of an existing aggregated ClusterRole, that change triggers adding the new rules into the aggregated ClusterRole. Here is an example that adds rules to the "monitoring" ClusterRole, by creating another ClusterRole labeled `rbac.example.com/aggregate-to-monitoring: true`.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: monitoring-endpoints  labels:  rbac.example.com/aggregate-to-monitoring: "true"# When you crea
```

The default user-facing roles use ClusterRole aggregation. This lets you, as a cluster administrator, include rules for custom resources, such as those served by CustomResourceDefinitions or aggregated API servers, to extend the default roles.

For example: the following ClusterRoles let the "admin" and "edit" default roles manage the custom resource named CronTab, whereas the "view" role can perform only read actions on CronTab resources. You can assume that CronTab objects are named `"crontabs"` in URLs as seen by the API server.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: aggregate-cron-tabs-edit  labels:  # Add these permissions to the "admin" and "edit" default r
```

Role examples

The following examples are excerpts from Role or ClusterRole objects, showing only the `rules` section.

Allow reading `"pods"` resources in the core API Group:

```
rules:
- apiGroups: [""]  #  # at the HTTP level, the name of the resource for accessing Pod  # objects is "pods"  resources: ["pods"]  v
```

Allow reading/writing Deployments (at the HTTP level: objects with `"deployments"` in the resource part of their URL) in the `"apps"` API groups:

```
rules:
- apiGroups: ["apps"]  #  # at the HTTP level, the name of the resource for accessing Deployment  # objects is "deployments"  reso
```

Allow reading Pods in the core API group, as well as reading or writing Job resources in the `"batch"` API group:

```
rules:
- apiGroups: [""]  #  # at the HTTP level, the name of the resource for accessing Pod  # objects is "pods"  resources: ["pods"]  v
```

Allow reading a ConfigMap named "my-config" (must be bound with a RoleBinding to limit to a single ConfigMap in a single namespace):

```
rules:
- apiGroups: [""]  #  # at the HTTP level, the name of the resource for accessing ConfigMap  # objects is "configmaps"  resources:
```

Allow reading the resource `"nodes"` in the core group (because a Node is cluster-scoped, this must be in a ClusterRole bound with a ClusterRoleBinding to be effective):

```
rules:
- apiGroups: [""]  #  # at the HTTP level, the name of the resource for accessing Node  # objects is "nodes"  resources: ["nodes"]
```

Allow GET and POST requests to the non-resource endpoint `/healthz` and all subpaths (must be in a ClusterRole bound with a ClusterRoleBinding to be effective):

```
rules:
- nonResourceURLs: ["/healthz", "/healthz/*"] # '*' in a nonResourceURL is a suffix glob match  verbs: ["get", "post"]
```

Referring to subjects

A RoleBinding or ClusterRoleBinding binds a role to subjects. Subjects can be groups, users or ServiceAccounts.

Kubernetes represents usernames as strings. These can be: plain names, such as "alice"; email-style names, like "bob@example.com"; or numeric user IDs represented as a string. It is up to you as a cluster administrator to configure the authentication modules so that authentication produces usernames in the format you want.

Caution:

The prefix `system:` is reserved for Kubernetes system use, so you should ensure that you don't have users or groups with names that start with `system:` by accident. Other than this special prefix, the RBAC authorization system does not require any format for usernames.

In Kubernetes, Authenticator modules provide group information. Groups, like users, are represented as strings, and that string has no format requirements, other than that the prefix `system:` is reserved.

ServiceAccounts have names prefixed with `system:serviceaccount:`, and belong to groups that have names prefixed with `system:serviceaccounts:`.

Note:

- `system:serviceaccount:` (singular) is the prefix for service account usernames.
- `system:serviceaccounts:` (plural) is the prefix for service account groups.

RoleBinding examples

The following examples are `RoleBinding` excerpts that only show the `subjects` section.

For a user named `alice@example.com`:

```
subjects:
- kind: User  name: "alice@example.com"  apiGroup: rbac.authorization.k8s.io
```

For a group named `frontend-admins`:

```
subjects:
- kind: Group  name: "frontend-admins"  apiGroup: rbac.authorization.k8s.io
```

For the default service account in the "kube-system" namespace:

```
subjects:
- kind: ServiceAccount  name: default  namespace: kube-system
```

For all service accounts in the "qa" namespace:

```
subjects:
- kind: Group  name: system:serviceaccounts:qa  apiGroup: rbac.authorization.k8s.io
```

For all service accounts in any namespace:

```
subjects:
- kind: Group  name: system:serviceaccounts  apiGroup: rbac.authorization.k8s.io
```

For all authenticated users:

```
subjects:
- kind: Group  name: system:authenticated  apiGroup: rbac.authorization.k8s.io
```

For all unauthenticated users:

```
subjects:
- kind: Group  name: system:unauthenticated  apiGroup: rbac.authorization.k8s.io
```

For all users:

```
subjects:
- kind: Group  name: system:authenticated  apiGroup: rbac.authorization.k8s.io- kind: Group  name: system:unauthenticated  apiGrou
```

Default roles and role bindings

API servers create a set of default ClusterRole and ClusterRoleBinding objects. Many of these are `system:` prefixed, which indicates that the resource is directly managed by the cluster control plane. All of the default ClusterRoles and ClusterRoleBindings are labeled with `kubernetes.io/bootstrapping=rbac-defaults`.

Caution:

Take care when modifying ClusterRoles and ClusterRoleBindings with names that have a `system:` prefix. Modifications to these resources can result in non-functional clusters.

Auto-reconciliation

At each start-up, the API server updates default cluster roles with any missing permissions, and updates default cluster role bindings with any missing subjects. This allows the cluster to repair accidental modifications, and helps to keep roles and role bindings up-to-date as permissions and subjects change in new Kubernetes releases.

To opt out of this reconciliation, set the `rbac.authorization.kubernetes.io/autoupdate` annotation on a default cluster role or default cluster RoleBinding to `false`. Be aware that missing default permissions and subjects can result in non-functional clusters.

Auto-reconciliation is enabled by default if the RBAC authorizer is active.

API discovery roles

Default cluster role bindings authorize unauthenticated and authenticated users to read API information that is deemed safe to be publicly accessible (including CustomResourceDefinitions). To disable anonymous unauthenticated access, add `--anonymous-auth=false` flag to the API server configuration.

To view the configuration of these roles via `kubectl` run:

```
kubectl get clusterroles system:discovery -o yaml
```

Note:

If you edit that ClusterRole, your changes will be overwritten on API server restart via auto-reconciliation. To avoid that overwriting, either do not manually edit the role, or disable auto-reconciliation.

Kubernetes RBAC API discovery roles

Default ClusterRole	Default ClusterRoleBinding	Description
system:basic-user	**system:authenticated** group	Allows a user read-only access to basic information about themselves. Prior to v1.14, this role was also bound to `system:unauthenticated` by default.
system:discovery	**system:authenticated** group	Allows read-only access to API discovery endpoints needed to discover and negotiate an API level. Prior to v1.14, this role was also bound to `system:unauthenticated` by default.
system:public-info-viewer	**system:authenticated** and **system:unauthenticated** groups	Allows read-only access to non-sensitive information about the cluster. Introduced in Kubernetes v1.14.

User-facing roles

Some of the default ClusterRoles are not `system:` prefixed. These are intended to be user-facing roles. They include super-user roles (`cluster-admin`), roles intended to be granted cluster-wide using ClusterRoleBindings, and roles intended to be granted within particular namespaces using RoleBindings (`admin`, `edit`, `view`).

User-facing ClusterRoles use ClusterRole aggregation to allow admins to include rules for custom resources on these ClusterRoles. To add rules to the `admin`, `edit`, or `view` roles, create a ClusterRole with one or more of the following labels:

```yaml
metadata:
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
    rbac.authorization.k8s.io/aggregate-to-view: "true"
```

Default ClusterRole	Default ClusterRoleBinding	Description
cluster-admin	**system:masters** group	Allows super-user access to perform any action on any resource. When used in a **ClusterRoleBinding**, it gives full control over every resource in the cluster and in all namespaces. When used in a **RoleBinding**, it gives full control over every resource in the role binding's namespace, including the namespace itself.
admin	None	Allows admin access, intended to be granted within a namespace using a **RoleBinding**. If used in a **RoleBinding**, allows read/write access to most resources in a namespace, including the ability to create roles and role bindings within the namespace. This role does not allow write access to resource quota or to the namespace itself. This role also does not allow write access to EndpointSlices in clusters created using Kubernetes v1.22+. More information is available in the "Write Access for EndpointSlices" section.
edit	None	Allows read/write access to most objects in a namespace. This role does not allow viewing or modifying roles or role bindings. However, this role allows accessing Secrets and running Pods as any ServiceAccount in the namespace, so it can be used to gain the API access levels of any ServiceAccount in the namespace. This role also does not allow write access to EndpointSlices in clusters created using Kubernetes v1.22+. More information is available in the "Write Access for EndpointSlices" section.
view	None	Allows read-only access to see most objects in a namespace. It does not allow viewing roles or role bindings. This role does not allow viewing Secrets, since reading the contents of Secrets enables access to ServiceAccount credentials in the namespace, which would allow API access as any ServiceAccount in the namespace (a form of privilege escalation).

Core component roles

Default ClusterRole	Default ClusterRoleBinding	Description
system:kube-scheduler	**system:kube-scheduler** user	Allows access to the resources required by the scheduler component.
system:volume-scheduler	**system:kube-scheduler** user	Allows access to the volume resources required by the kube-scheduler component.
system:kube-controller-manager	**system:kube-controller-manager** user	Allows access to the resources required by the controller manager component. The permissions required by individual controllers are detailed in the controller roles.
system:node	None	Allows access to resources required by the kubelet, **including read access to all secrets, and write access to all pod status objects**.

Default ClusterRole	Default ClusterRoleBinding	Description
		You should use the Node authorizer and NodeRestriction admission plugin instead of the `system:node` role, and allow granting API access to kubelets based on the Pods scheduled to run on them. The `system:node` role only exists for compatibility with Kubernetes clusters upgraded from versions prior to v1.8.
system:node-proxier	**system:kube-proxy** user	Allows access to the resources required by the kube-proxy component.

Other component roles

Default ClusterRole	Default ClusterRoleBinding	Description
system:auth-delegator	None	Allows delegated authentication and authorization checks. This is commonly used by add-on API servers for unified authentication and authorization.
system:heapster	None	Role for the Heapster component (deprecated).
system:kube-aggregator	None	Role for the kube-aggregator component.
system:kube-dns	**kube-dns** service account in the **kube-system** namespace	Role for the kube-dns component.
system:kubelet-api-admin	None	Allows full access to the kubelet API.
system:node-bootstrapper	None	Allows access to the resources required to perform kubelet TLS bootstrapping.
system:node-problem-detector	None	Role for the node-problem-detector component.
system:persistent-volume-provisioner	None	Allows access to the resources required by most dynamic volume provisioners.
system:monitoring	**system:monitoring** group	Allows read access to control-plane monitoring endpoints (i.e. kube-apiserver liveness and readiness endpoints (`/healthz`, `/livez`, `/readyz`), the individual health-check endpoints (`/healthz/*`, `/livez/*`, `/readyz/*`), `/metrics`), and causes the kube-apiserver to respect the traceparent header provided with requests for tracing. Note that individual health check endpoints and the metric endpoint may expose sensitive information.

Roles for built-in controllers

The Kubernetes controller manager runs controllers that are built in to the Kubernetes control plane. When invoked with `--use-service-account-credentials`, kube-controller-manager starts each controller using a separate service account. Corresponding roles exist for each built-in controller, prefixed with `system:controller:`. If the controller manager is not started with `--use-service-account-credentials`, it runs all control loops using its own credential, which must be granted all the relevant roles. These roles include:

- `system:controller:attachdetach-controller`
- `system:controller:certificate-controller`
- `system:controller:clusterrole-aggregation-controller`
- `system:controller:cronjob-controller`
- `system:controller:daemon-set-controller`
- `system:controller:deployment-controller`
- `system:controller:disruption-controller`
- `system:controller:endpoint-controller`
- `system:controller:expand-controller`
- `system:controller:generic-garbage-collector`
- `system:controller:horizontal-pod-autoscaler`
- `system:controller:job-controller`
- `system:controller:namespace-controller`
- `system:controller:node-controller`
- `system:controller:persistent-volume-binder`
- `system:controller:pod-garbage-collector`
- `system:controller:pv-protection-controller`
- `system:controller:pvc-protection-controller`
- `system:controller:replicaset-controller`
- `system:controller:replication-controller`
- `system:controller:resourcequota-controller`
- `system:controller:root-ca-cert-publisher`
- `system:controller:route-controller`
- `system:controller:service-account-controller`
- `system:controller:service-controller`
- `system:controller:statefulset-controller`
- `system:controller:ttl-controller`

Privilege escalation prevention and bootstrapping

The RBAC API prevents users from escalating privileges by editing roles or role bindings. Because this is enforced at the API level, it applies even when the RBAC authorizer is not in use.

Restrictions on role creation or update

You can only create/update a role if at least one of the following things is true:

1. You already have all the permissions contained in the role, at the same scope as the object being modified (cluster-wide for a ClusterRole, within the same namespace or cluster-wide for a Role).
2. You are granted explicit permission to perform the `escalate` verb on the `roles` or `clusterroles` resource in the `rbac.authorization.k8s.io` API group.

For example, if `user-1` does not have the ability to list Secrets cluster-wide, they cannot create a ClusterRole containing that permission. To allow a user to create/update roles:

1. Grant them a role that allows them to create/update Role or ClusterRole objects, as desired.
2. Grant them permission to include specific permissions in the roles they create/update:
   - implicitly, by giving them those permissions (if they attempt to create or modify a Role or ClusterRole with permissions they themselves have not been granted, the API request will be forbidden)
   - or explicitly allow specifying any permission in a `Role` or `ClusterRole` by giving them permission to perform the `escalate` verb on `roles` or `clusterroles` resources in the `rbac.authorization.k8s.io` API group

Restrictions on role binding creation or update

You can only create/update a role binding if you already have all the permissions contained in the referenced role (at the same scope as the role binding) *or* if you have been authorized to perform the `bind` verb on the referenced role. For example, if `user-1` does not have the ability to list Secrets cluster-wide, they cannot create a ClusterRoleBinding to a role that grants that permission. To allow a user to create/update role bindings:

1. Grant them a role that allows them to create/update RoleBinding or ClusterRoleBinding objects, as desired.
2. Grant them permissions needed to bind a particular role:
   - implicitly, by giving them the permissions contained in the role.
   - explicitly, by giving them permission to perform the `bind` verb on the particular Role (or ClusterRole).

For example, this ClusterRole and RoleBinding would allow `user-1` to grant other users the `admin`, `edit`, and `view` roles in the namespace `user-1-namespace`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: role-grantorrules:- apiGroups: ["rbac.authorization.k8s.io"]  resources: ["rolebindings"]  verbs
```

When bootstrapping the first roles and role bindings, it is necessary for the initial user to grant permissions they do not yet have. To bootstrap initial roles and role bindings:

- Use a credential with the "system:masters" group, which is bound to the "cluster-admin" super-user role by the default bindings.

Command-line utilities

`kubectl create role`

Creates a Role object defining permissions within a single namespace. Examples:

- Create a Role named "pod-reader" that allows users to perform `get`, `watch` and `list` on pods:

  ```
  kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods
  ```

- Create a Role named "pod-reader" with resourceNames specified:

  ```
  kubectl create role pod-reader --verb=get --resource=pods --resource-name=readablepod --resource-name=anotherpod
  ```

- Create a Role named "foo" with apiGroups specified:

  ```
  kubectl create role foo --verb=get,list,watch --resource=replicasets.apps
  ```

- Create a Role named "foo" with subresource permissions:

  ```
  kubectl create role foo --verb=get,list,watch --resource=pods,pods/status
  ```

- Create a Role named "my-component-lease-holder" with permissions to get/update a resource with a specific name:

  ```
  kubectl create role my-component-lease-holder --verb=get,list,watch,update --resource=lease --resource-name=my-component
  ```

`kubectl create clusterrole`

Creates a ClusterRole. Examples:

- Create a ClusterRole named "pod-reader" that allows user to perform `get`, `watch` and `list` on pods:

  ```
  kubectl create clusterrole pod-reader --verb=get,list,watch --resource=pods
  ```

- Create a ClusterRole named "pod-reader" with resourceNames specified:

  ```
  kubectl create clusterrole pod-reader --verb=get --resource=pods --resource-name=readablepod --resource-name=anotherpod
  ```

- Create a ClusterRole named "foo" with apiGroups specified:

  ```
  kubectl create clusterrole foo --verb=get,list,watch --resource=replicasets.apps
  ```

- Create a ClusterRole named "foo" with subresource permissions:

  ```
  kubectl create clusterrole foo --verb=get,list,watch --resource=pods,pods/status
  ```

- Create a ClusterRole named "foo" with nonResourceURL specified:

```
kubectl create clusterrole "foo" --verb=get --non-resource-url=/logs/*
```

- Create a ClusterRole named "monitoring" with an aggregationRule specified:

  ```
  kubectl create clusterrole monitoring --aggregation-rule="rbac.example.com/aggregate-to-monitoring=true"
  ```

`kubectl create rolebinding`

Grants a Role or ClusterRole within a specific namespace. Examples:

- Within the namespace "acme", grant the permissions in the "admin" ClusterRole to a user named "bob":

  ```
  kubectl create rolebinding bob-admin-binding --clusterrole=admin --user=bob --namespace=acme
  ```

- Within the namespace "acme", grant the permissions in the "view" ClusterRole to the service account in the namespace "acme" named "myapp":

  ```
  kubectl create rolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp --namespace=acme
  ```

- Within the namespace "acme", grant the permissions in the "view" ClusterRole to a service account in the namespace "myappnamespace" named "myapp":

  ```
  kubectl create rolebinding myappnamespace-myapp-view-binding --clusterrole=view --serviceaccount=myappnamespace:myapp --names
  ```

`kubectl create clusterrolebinding`

Grants a ClusterRole across the entire cluster (all namespaces). Examples:

- Across the entire cluster, grant the permissions in the "cluster-admin" ClusterRole to a user named "root":

  ```
  kubectl create clusterrolebinding root-cluster-admin-binding --clusterrole=cluster-admin --user=root
  ```

- Across the entire cluster, grant the permissions in the "system:node-proxier" ClusterRole to a user named "system:kube-proxy":

  ```
  kubectl create clusterrolebinding kube-proxy-binding --clusterrole=system:node-proxier --user=system:kube-proxy
  ```

- Across the entire cluster, grant the permissions in the "view" ClusterRole to a service account named "myapp" in the namespace "acme":

  ```
  kubectl create clusterrolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp
  ```

`kubectl auth reconcile`

Creates or updates `rbac.authorization.k8s.io/v1` API objects from a manifest file.

Missing objects are created, and the containing namespace is created for namespaced objects, if required.

Existing roles are updated to include the permissions in the input objects, and remove extra permissions if `--remove-extra-permissions` is specified.

Existing bindings are updated to include the subjects in the input objects, and remove extra subjects if `--remove-extra-subjects` is specified.

Examples:

- Test applying a manifest file of RBAC objects, displaying changes that would be made:

  ```
  kubectl auth reconcile -f my-rbac-rules.yaml --dry-run=client
  ```

- Apply a manifest file of RBAC objects, preserving any extra permissions (in roles) and any extra subjects (in bindings):

  ```
  kubectl auth reconcile -f my-rbac-rules.yaml
  ```

- Apply a manifest file of RBAC objects, removing any extra permissions (in roles) and any extra subjects (in bindings):

  ```
  kubectl auth reconcile -f my-rbac-rules.yaml --remove-extra-subjects --remove-extra-permissions
  ```

ServiceAccount permissions

Default RBAC policies grant scoped permissions to control-plane components, nodes, and controllers, but grant *no permissions* to service accounts outside the `kube-system` namespace (beyond the permissions given by API discovery roles).

This allows you to grant particular roles to particular ServiceAccounts as needed. Fine-grained role bindings provide greater security, but require more effort to administrate. Broader grants can give unnecessary (and potentially escalating) API access to ServiceAccounts, but are easier to administrate.

In order from most secure to least secure, the approaches are:

1. Grant a role to an application-specific service account (best practice)

   This requires the application to specify a `serviceAccountName` in its pod spec, and for the service account to be created (via the API, application manifest, `kubectl create serviceaccount`, etc.).

   For example, grant read-only permission within "my-namespace" to the "my-sa" service account:

   ```
   kubectl create rolebinding my-sa-view \
     --clusterrole=view \  --serviceaccount=my-namespace:my-sa \  --namespace=my-namespace
   ```

2. Grant a role to the "default" service account in a namespace

   If an application does not specify a `serviceAccountName`, it uses the "default" service account.

Note:

Permissions given to the "default" service account are available to any pod in the namespace that does not specify a `serviceAccountName`.

For example, grant read-only permission within "my-namespace" to the "default" service account:

```
kubectl create rolebinding default-view \
  --clusterrole=view \  --serviceaccount=my-namespace:default \  --namespace=my-namespace
```

Many add-ons run as the "default" service account in the `kube-system` namespace. To allow those add-ons to run with super-user access, grant cluster-admin permissions to the "default" service account in the `kube-system` namespace.

Caution:

Enabling this means the `kube-system` namespace contains Secrets that grant super-user access to your cluster's API.

```
kubectl create clusterrolebinding add-on-cluster-admin \
  --clusterrole=cluster-admin \  --serviceaccount=kube-system:default
```

3. Grant a role to all service accounts in a namespace

   If you want all applications in a namespace to have a role, no matter what service account they use, you can grant a role to the service account group for that namespace.

   For example, grant read-only permission within "my-namespace" to all service accounts in that namespace:

   ```
   kubectl create rolebinding serviceaccounts-view \
     --clusterrole=view \  --group=system:serviceaccounts:my-namespace \  --namespace=my-namespace
   ```

4. Grant a limited role to all service accounts cluster-wide (discouraged)

   If you don't want to manage permissions per-namespace, you can grant a cluster-wide role to all service accounts.

   For example, grant read-only permission across all namespaces to all service accounts in the cluster:

   ```
   kubectl create clusterrolebinding serviceaccounts-view \
     --clusterrole=view \ --group=system:serviceaccounts
   ```

5. Grant super-user access to all service accounts cluster-wide (strongly discouraged)

   If you don't care about partitioning permissions at all, you can grant super-user access to all service accounts.

   Warning:

   This allows any application full access to your cluster, and also grants any user with read access to Secrets (or the ability to create any pod) full access to your cluster.

   ```
   kubectl create clusterrolebinding serviceaccounts-cluster-admin \
     --clusterrole=cluster-admin \  --group=system:serviceaccounts
   ```

Write access for EndpointSlices

Kubernetes clusters created before Kubernetes v1.22 include write access to EndpointSlices (and the now-deprecated Endpoints API) in the aggregated "edit" and "admin" roles. As a mitigation for CVE-2021-25740, this access is not part of the aggregated roles in clusters that you create using Kubernetes v1.22 or later.

Existing clusters that have been upgraded to Kubernetes v1.22 will not be subject to this change. The CVE announcement includes guidance for restricting this access in existing clusters.

If you want new clusters to retain this level of access in the aggregated roles, you can create the following ClusterRole:

access/endpoints-aggregated.yaml Copy access/endpoints-aggregated.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  annotations:    kubernetes.io/description: |-      Add endpoints write permissions to the edit and adm
```

Upgrading from ABAC

Clusters that originally ran older Kubernetes versions often used permissive ABAC policies, including granting full API access to all service accounts.

Default RBAC policies grant scoped permissions to control-plane components, nodes, and controllers, but grant *no permissions* to service accounts outside the `kube-system` namespace (beyond the permissions given by API discovery roles).

While far more secure, this can be disruptive to existing workloads expecting to automatically receive API permissions. Here are two approaches for managing this transition:

Parallel authorizers

Run both the RBAC and ABAC authorizers, and specify a policy file that contains the legacy ABAC policy:

```
--authorization-mode=...,RBAC,ABAC --authorization-policy-file=mypolicy.json
```

To explain that first command line option in detail: if earlier authorizers, such as Node, deny a request, then the RBAC authorizer attempts to authorize the API request. If RBAC also denies that API request, the ABAC authorizer is then run. This means that any request allowed by *either* the RBAC or ABAC policies is allowed.

When the kube-apiserver is run with a log level of 5 or higher for the RBAC component (`--vmodule=rbac*=5` or `--v=5`), you can see RBAC denials in the API server log (prefixed with `RBAC`). You can use that information to determine which roles need to be granted to which users, groups, or service accounts.

Once you have granted roles to service accounts and workloads are running with no RBAC denial messages in the server logs, you can remove the ABAC authorizer.

Permissive RBAC permissions

You can replicate a permissive ABAC policy using RBAC role bindings.

Warning:

The following policy allows **ALL** service accounts to act as cluster administrators. Any application running in a container receives service account credentials automatically, and could perform any action against the API, including viewing secrets and modifying permissions. This is not a recommended policy.

```
kubectl create clusterrolebinding permissive-binding \
  --clusterrole=cluster-admin \  --user=admin \  --user=kubelet \  --group=system:serviceaccounts
```

After you have transitioned to use RBAC, you should adjust the access controls for your cluster to ensure that these meet your information security needs.

Dynamic Admission Control

In addition to compiled-in admission plugins, admission plugins can be developed as extensions and run as webhooks configured at runtime. This page describes how to build, configure, use, and monitor admission webhooks.

What are admission webhooks?

Admission webhooks are HTTP callbacks that receive admission requests and do something with them. You can define two types of admission webhooks, validating admission webhook and mutating admission webhook. Mutating admission webhooks are invoked first, and can modify objects sent to the API server to enforce custom defaults. After all object modifications are complete, and after the incoming object is validated by the API server, validating admission webhooks are invoked and can reject requests to enforce custom policies.

Note:

Admission webhooks that need to guarantee they see the final state of the object in order to enforce policy should use a validating admission webhook, since objects can be modified after being seen by mutating webhooks.

Experimenting with admission webhooks

Admission webhooks are essentially part of the cluster control-plane. You should write and deploy them with great caution. Please read the user guides for instructions if you intend to write/deploy production-grade admission webhooks. In the following, we describe how to quickly experiment with admission webhooks.

Prerequisites

- Ensure that MutatingAdmissionWebhook and ValidatingAdmissionWebhook admission controllers are enabled. Here is a recommended set of admission controllers to enable in general.
- Ensure that the `admissionregistration.k8s.io/v1` API is enabled.

Write an admission webhook server

Please refer to the implementation of the admission webhook server that is validated in a Kubernetes e2e test. The webhook handles the `AdmissionReview` request sent by the API servers, and sends back its decision as an `AdmissionReview` object in the same version it received.

See the webhook request section for details on the data sent to webhooks.

See the webhook response section for the data expected from webhooks.

The example admission webhook server leaves the `ClientAuth` field empty, which defaults to `NoClientCert`. This means that the webhook server does not authenticate the identity of the clients, supposedly API servers. If you need mutual TLS or other ways to authenticate the clients, see how to authenticate API servers.

Deploy the admission webhook service

The webhook server in the e2e test is deployed in the Kubernetes cluster, via the deployment API. The test also creates a service as the front-end of the webhook server. See code.

You may also deploy your webhooks outside of the cluster. You will need to update your webhook configurations accordingly.

Configure admission webhooks on the fly

You can dynamically configure what resources are subject to what admission webhooks via ValidatingWebhookConfiguration or MutatingWebhookConfiguration.

The following is an example `ValidatingWebhookConfiguration`, a mutating webhook configuration is similar. See the webhook configuration section for details about each config field.

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationmetadata:  name: "pod-policy.example.com"webhooks:- name: "pod-policy.example.com"  rules:  - 
```

Note:

You must replace the `<CA_BUNDLE>` in the above example by a valid CA bundle which is a PEM-encoded (field value is Base64 encoded) CA bundle for validating the webhook's server certificate.

The `scope` field specifies if only cluster-scoped resources ("Cluster") or namespace-scoped resources ("Namespaced") will match this rule. "`*`" means that there are no scope restrictions.

Note:

When using `clientConfig.service`, the server cert must be valid for `<svc_name>.<svc_namespace>.svc`.

Note:

Default timeout for a webhook call is 10 seconds, You can set the `timeout` and it is encouraged to use a short timeout for webhooks. If the webhook call times out, the request is handled according to the webhook's failure policy.

When an API server receives a request that matches one of the `rules`, the API server sends an `admissionReview` request to webhook as specified in the `clientConfig`.

After you create the webhook configuration, the system will take a few seconds to honor the new configuration.

Authenticate API servers

If your admission webhooks require authentication, you can configure the API servers to use basic auth, bearer token, or a cert to authenticate itself to the webhooks. There are three steps to complete the configuration.

- When starting the API server, specify the location of the admission control configuration file via the `--admission-control-config-file` flag.
- In the admission control configuration file, specify where the MutatingAdmissionWebhook controller and ValidatingAdmissionWebhook controller should read the credentials. The credentials are stored in kubeConfig files (yes, the same schema that's used by kubectl), so the field name is `kubeConfigFile`. Here is an example admission control configuration file:

- apiserver.config.k8s.io/v1
- apiserver.k8s.io/v1alpha1

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:- name: ValidatingAdmissionWebhook  configuration:    apiVersion: apiserver.config.k8s.io/v1

# Deprecated in v1.17 in favor of apiserver.config.k8s.io/v1
apiVersion: apiserver.k8s.io/v1alpha1kind: AdmissionConfigurationplugins:- name: ValidatingAdmissionWebhook  configuration:    # D
```

For more information about `AdmissionConfiguration`, see the AdmissionConfiguration (v1) reference. See the webhook configuration section for details about each config field.

In the kubeConfig file, provide the credentials:

```
apiVersion: v1
kind: Configusers:# name should be set to the DNS name of the service or the host (including port) of the URL the webhook is confi
```

Of course you need to set up the webhook server to handle these authentication requests.

Webhook request and response

Request

Webhooks are sent as POST requests, with `Content-Type: application/json`, with an `AdmissionReview` API object in the `admission.k8s.io` API group serialized to JSON as the body.

Webhooks can specify what versions of `AdmissionReview` objects they accept with the `admissionReviewVersions` field in their configuration:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:- name: my-webhook.example.com  admissionReviewVersions: ["v1", "v1beta1"]
```

`admissionReviewVersions` is a required field when creating webhook configurations. Webhooks are required to support at least one `AdmissionReview` version understood by the current and previous API server.

API servers send the first `AdmissionReview` version in the `admissionReviewVersions` list they support. If none of the versions in the list are supported by the API server, the configuration will not be allowed to be created. If an API server encounters a webhook configuration that was previously created and does not support any of the `AdmissionReview` versions the API server knows how to send, attempts to call to the webhook will fail and be subject to the failure policy.

This example shows the data contained in an `AdmissionReview` object for a request to update the `scale` subresource of an `apps/v1 Deployment`:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "request": {
    # Random uid uniquely identifying this admission call
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002",

    # Fully-qualified group/version/kind of the incoming object
    "kind": {
      "group": "autoscaling",
      "version": "v1",
      "kind": "Scale"
    },

    # Fully-qualified group/version/kind of the resource being modified
    "resource": {
      "group": "apps",
      "version": "v1",
      "resource": "deployments"
    },

    # Subresource, if the request is to a subresource
    "subResource": "scale",

    # Fully-qualified group/version/kind of the incoming object in the original request to the API server
    # This only differs from `kind` if the webhook specified `matchPolicy: Equivalent` and the original
    # request to the API server was converted to a version the webhook registered for
    "requestKind": {
      "group": "autoscaling",
      "version": "v1",
      "kind": "Scale"
    },

    # Fully-qualified group/version/kind of the resource being modified in the original request to the API server
    # This only differs from `resource` if the webhook specified `matchPolicy: Equivalent` and the original
    # request to the API server was converted to a version the webhook registered for
    "requestResource": {
      "group": "apps",
      "version": "v1",
      "resource": "deployments"
    },

    # Subresource, if the request is to a subresource
    # This only differs from `subResource` if the webhook specified `matchPolicy: Equivalent` and the original
    # request to the API server was converted to a version the webhook registered for
    "requestSubResource": "scale",

    # Name of the resource being modified
    "name": "my-deployment",

    # Namespace of the resource being modified, if the resource is namespaced (or is a Namespace object)
    "namespace": "my-namespace",

    # operation can be CREATE, UPDATE, DELETE, or CONNECT
    "operation": "UPDATE",

    "userInfo": {
      # Username of the authenticated user making the request to the API server
      "username": "admin",

      # UID of the authenticated user making the request to the API server
      "uid": "014fbff9a07c",

      # Group memberships of the authenticated user making the request to the API server
      "groups": [
        "system:authenticated",
        "my-admin-group"
      ],

      # Arbitrary extra info associated with the user making the request to the API server
      # This is populated by the API server authentication layer
      "extra": {
        "some-key": [
          "some-value1",
          "some-value2"
        ]
      }
    },

    # object is the new object being admitted. It is null for DELETE operations
    "object": {
      "apiVersion": "autoscaling/v1",
      "kind": "Scale"
    },

    # oldObject is the existing object. It is null for CREATE and CONNECT operations
    "oldObject": {
      "apiVersion": "autoscaling/v1",
      "kind": "Scale"
    },

    # options contain the options for the operation being admitted, like meta.k8s.io/v1 CreateOptions,
    # UpdateOptions, or DeleteOptions. It is null for CONNECT operations
    "options": {
      "apiVersion": "meta.k8s.io/v1",
      "kind": "UpdateOptions"
    },
```

```
    # dryRun indicates the API request is running in dry run mode and will not be persisted
    # Webhooks with side effects should avoid actuating those side effects when dryRun is true
    "dryRun": false
  }
}
```

Response

Webhooks respond with a 200 HTTP status code, `Content-Type: application/json`, and a body containing an `AdmissionReview` object (in the same version they were sent), with the `response` stanza populated, serialized to JSON.

At a minimum, the `response` stanza must contain the following fields:

- `uid`, copied from the `request.uid` sent to the webhook
- `allowed`, either set to `true` or `false`

Example of a minimal response from a webhook to allow a request:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true
  }
}
```

Example of a minimal response from a webhook to forbid a request:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": false
  }
}
```

When rejecting a request, the webhook can customize the http code and message returned to the user using the `status` field. The specified status object is returned to the user. See the API documentation for details about the `status` type. Example of a response to forbid a request, customizing the HTTP status code and message presented to the user:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": false,
    "status": {
      "code": 403,
      "message": "You cannot do this because it is Tuesday and your name starts with A"
    }
  }
}
```

When allowing a request, a mutating admission webhook may optionally modify the incoming object as well. This is done using the `patch` and `patchType` fields in the response. The only currently supported `patchType` is `JSONPatch`. See JSON patch documentation for more details. For `patchType: JSONPatch`, the `patch` field contains a base64-encoded array of JSON patch operations.

As an example, a single patch operation that would set `spec.replicas` would be `[{"op": "add", "path": "/spec/replicas", "value": 3}]`

Base64-encoded, this would be `W3sib3AiOiAiYWRkIiwgInBhdGgiOiAiL3NwZWMvcmVwbGljYXMiLCAidmFsdWUiOiAzfV0=`

So a webhook response to add that label would be:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true,
    "patchType": "JSONPatch",
    "patch": "W3sib3AiOiAiYWRkIiwgInBhdGgiOiAiL3NwZWMvcmVwbGljYXMiLCAidmFsdWUiOiAzfV0="
  }
}
```

Admission webhooks can optionally return warning messages that are returned to the requesting client in HTTP `Warning` headers with a warning code of 299. Warnings can be sent with allowed or rejected admission responses.

If you're implementing a webhook that returns a warning:

- Don't include a "Warning:" prefix in the message
- Use warning messages to describe problems the client making the API request should correct or be aware of
- Limit warnings to 120 characters if possible

Caution:

Individual warning messages over 256 characters may be truncated by the API server before being returned to clients. If more than 4096 characters of warning messages are added (from all sources), additional warning messages are ignored.

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true,
    "warnings": [
      "duplicate envvar entries specified with name MY_ENV",
      "memory request less than 4MB specified for container mycontainer, which will not start successfully"
    ]
  }
}
```

Webhook configuration

To register admission webhooks, create `MutatingWebhookConfiguration` or `ValidatingWebhookConfiguration` API objects. The name of a `MutatingWebhookConfiguration` or a `ValidatingWebhookConfiguration` object must be a valid DNS subdomain name.

Each configuration can contain one or more webhooks. If multiple webhooks are specified in a single configuration, each must be given a unique name. This is required in order to make resulting audit logs and metrics easier to match up to active configurations.

Each webhook defines the following things.

Matching requests: rules

Each webhook must specify a list of rules used to determine if a request to the API server should be sent to the webhook. Each rule specifies one or more operations, apiGroups, apiVersions, and resources, and a resource scope:

- `operations` lists one or more operations to match. Can be `"CREATE"`, `"UPDATE"`, `"DELETE"`, `"CONNECT"`, or `"*"` to match all.
- `apiGroups` lists one or more API groups to match. `""` is the core API group. `"*"` matches all API groups.
- `apiVersions` lists one or more API versions to match. `"*"` matches all API versions.
- `resources` lists one or more resources to match.
  - `"*"` matches all resources, but not subresources.
  - `"*/*"` matches all resources and subresources.
  - `"pods/*"` matches all subresources of pods.
  - `"*/status"` matches all status subresources.
- `scope` specifies a scope to match. Valid values are `"Cluster"`, `"Namespaced"`, and `"*"`. Subresources match the scope of their parent resource. Default is `"*"`.
  - `"Cluster"` means that only cluster-scoped resources will match this rule (Namespace API objects are cluster-scoped).
  - `"Namespaced"` means that only namespaced resources will match this rule.
  - `"*"` means that there are no scope restrictions.

If an incoming request matches one of the specified `operations`, `groups`, `versions`, `resources`, and `scope` for any of a webhook's `rules`, the request is sent to the webhook.

Here are other examples of rules that could be used to specify which resources should be intercepted.

Match `CREATE` or `UPDATE` requests to `apps/v1` and `apps/v1beta1` `deployments` and `replicasets`:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration...webhooks:- name: my-webhook.example.com  rules:  - operations: ["CREATE", "UPDATE"]    apiG
```

Match create requests for all resources (but not subresources) in all API groups and versions:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:  - name: my-webhook.example.com    rules:      - operations: ["CREATE"]        apiGr
```

Match update requests for all `status` subresources in all API groups and versions:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:  - name: my-webhook.example.com    rules:      - operations: ["UPDATE"]        apiGr
```

Matching requests: objectSelector

Webhooks may optionally limit which requests are intercepted based on the labels of the objects they would be sent, by specifying an `objectSelector`. If specified, the objectSelector is evaluated against both the object and oldObject that would be sent to the webhook, and is considered to match if either object matches the selector.

A null object (`oldObject` in the case of create, or `newObject` in the case of delete), or an object that cannot have labels (like a `DeploymentRollback` or a `PodProxyOptions` object) is not considered to match.

Use the object selector only if the webhook is opt-in, because end users may skip the admission webhook by setting the labels.

This example shows a mutating webhook that would match a `CREATE` of any resource (but not subresources) with the label `foo: bar`:

```
apiVersion: admissionregistration.k8s.io/v1
```

```
kind: MutatingWebhookConfigurationwebhooks:- name: my-webhook.example.com  objectSelector:    matchLabels:      foo: bar  rules:  -
```

See labels concept for more examples of label selectors.

Matching requests: namespaceSelector

Webhooks may optionally limit which requests for namespaced resources are intercepted, based on the labels of the containing namespace, by specifying a `namespaceSelector`.

The `namespaceSelector` decides whether to run the webhook on a request for a namespaced resource (or a Namespace object), based on whether the namespace's labels match the selector. If the object itself is a namespace, the matching is performed on object.metadata.labels. If the object is a cluster scoped resource other than a Namespace, `namespaceSelector` has no effect.

This example shows a mutating webhook that matches a `CREATE` of any namespaced resource inside a namespace that does not have a "runlevel" label of "0" or "1":

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfigurationwebhooks:  - name: my-webhook.example.com    namespaceSelector:      matchExpressions:        - l
```

This example shows a validating webhook that matches a `CREATE` of any namespaced resource inside a namespace that is associated with the "environment" of "prod" or "staging":

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:  - name: my-webhook.example.com    namespaceSelector:      matchExpressions:  -
```

See labels concept for more examples of label selectors.

Matching requests: matchPolicy

API servers can make objects available via multiple API groups or versions.

For example, if a webhook only specified a rule for some API groups/versions (like `apiGroups:["apps"], apiVersions:["v1","v1beta1"]`), and a request was made to modify the resource via another API group/version (like `extensions/v1beta1`), the request would not be sent to the webhook.

The `matchPolicy` lets a webhook define how its `rules` are used to match incoming requests. Allowed values are `Exact` or `Equivalent`.

- `Exact` means a request should be intercepted only if it exactly matches a specified rule.
- `Equivalent` means a request should be intercepted if it modifies a resource listed in `rules`, even via another API group or version.

In the example given above, the webhook that only registered for `apps/v1` could use `matchPolicy`:

- `matchPolicy: Exact` would mean the `extensions/v1beta1` request would not be sent to the webhook
- `matchPolicy: Equivalent` means the `extensions/v1beta1` request would be sent to the webhook (with the objects converted to a version the webhook had specified: `apps/v1`)

Specifying `Equivalent` is recommended, and ensures that webhooks continue to intercept the resources they expect when upgrades enable new versions of the resource in the API server.

When a resource stops being served by the API server, it is no longer considered equivalent to other versions of that resource that are still served. For example, `extensions/v1beta1` deployments were first deprecated and then removed (in Kubernetes v1.16).

Since that removal, a webhook with a `apiGroups:["extensions"], apiVersions:["v1beta1"], resources:["deployments"]` rule does not intercept deployments created via `apps/v1` APIs. For that reason, webhooks should prefer registering for stable versions of resources.

This example shows a validating webhook that intercepts modifications to deployments (no matter the API group or version), and is always sent an `apps/v1` `Deployment` object:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:- name: my-webhook.example.com  matchPolicy: Equivalent  rules:  - operations: ["CREA
```

The `matchPolicy` for an admission webhooks defaults to `Equivalent`.

Matching requests: `matchConditions`

FEATURE STATE: `Kubernetes v1.30 [stable]` (enabled by default: true)

You can define *match conditions* for webhooks if you need fine-grained request filtering. These conditions are useful if you find that match rules, `objectSelectors` and `namespaceSelectors` still doesn't provide the filtering you want over when to call out over HTTP. Match conditions are CEL expressions. All match conditions must evaluate to true for the webhook to be called.

Here is an example illustrating a few different uses for match conditions:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:  - name: my-webhook.example.com    matchPolicy: Equivalent    rules:      - operatio
```

Note:

You can define up to 64 elements in the `matchConditions` field per webhook.

Match conditions have access to the following CEL variables:

- `object` - The object from the incoming request. The value is null for DELETE requests. The object version may be converted based on the matchPolicy.
- `oldObject` - The existing object. The value is null for CREATE requests.

- `request` - The request portion of the AdmissionReview, excluding `object` and `oldObject`.
- `authorizer` - A CEL Authorizer. May be used to perform authorization checks for the principal (authenticated user) of the request. See Authz in the Kubernetes CEL library documentation for more details.
- `authorizer.requestResource` - A shortcut for an authorization check configured with the request resource (group, resource, (subresource), namespace, name).

For more information on CEL expressions, refer to the Common Expression Language in Kubernetes reference.

In the event of an error evaluating a match condition the webhook is never called. Whether to reject the request is determined as follows:

1. If **any** match condition evaluated to `false` (regardless of other errors), the API server skips the webhook.
2. Otherwise:
   - for `failurePolicy: Fail`, reject the request (without calling the webhook).
   - for `failurePolicy: Ignore`, proceed with the request but skip the webhook.

Contacting the webhook

Once the API server has determined a request should be sent to a webhook, it needs to know how to contact the webhook. This is specified in the `clientConfig` stanza of the webhook configuration.

Webhooks can either be called via a URL or a service reference, and can optionally include a custom CA bundle to use to verify the TLS connection.

URL

`url` gives the location of the webhook, in standard URL form (`scheme://host:port/path`).

The `host` should not refer to a service running in the cluster; use a service reference by specifying the `service` field instead. The host might be resolved via external DNS in some API servers (e.g., `kube-apiserver` cannot resolve in-cluster DNS as that would be a layering violation). `host` may also be an IP address.

Please note that using `localhost` or `127.0.0.1` as a `host` is risky unless you take great care to run this webhook on all hosts which run an API server which might need to make calls to this webhook. Such installations are likely to be non-portable or not readily run in a new cluster.

The scheme must be "https"; the URL must begin with "https://".

Attempting to use a user or basic auth (for example `user:password@`) is not allowed. Fragments (`#...`) and query parameters (`?...`) are also not allowed.

Here is an example of a mutating webhook configured to call a URL (and expects the TLS certificate to be verified using system trust roots, so does not specify a caBundle):

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfigurationwebhooks:- name: my-webhook.example.com  clientConfig:    url: "https://my-webhook.example.com:9
```

Service reference

The `service` stanza inside `clientConfig` is a reference to the service for this webhook. If the webhook is running within the cluster, then you should use `service` instead of `url`. The service namespace and name are required. The port is optional and defaults to 443. The path is optional and defaults to "/".

Here is an example of a mutating webhook configured to call a service on port "1234" at the subpath "/my-path", and to verify the TLS connection against the ServerName `my-service-name.my-service-namespace.svc` using a custom CA bundle:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfigurationwebhooks:- name: my-webhook.example.com  clientConfig:    caBundle: <CA_BUNDLE>    service:
```

Note:

You must replace the `<CA_BUNDLE>` in the above example by a valid CA bundle which is a PEM-encoded CA bundle for validating the webhook's server certificate.

Side effects

Webhooks typically operate only on the content of the `AdmissionReview` sent to them. Some webhooks, however, make out-of-band changes as part of processing admission requests.

Webhooks that make out-of-band changes ("side effects") must also have a reconciliation mechanism (like a controller) that periodically determines the actual state of the world, and adjusts the out-of-band data modified by the admission webhook to reflect reality. This is because a call to an admission webhook does not guarantee the admitted object will be persisted as is, or at all. Later webhooks can modify the content of the object, a conflict could be encountered while writing to storage, or the server could power off before persisting the object.

Additionally, webhooks with side effects must skip those side-effects when `dryRun: true` admission requests are handled. A webhook must explicitly indicate that it will not have side-effects when run with `dryRun`, or the dry-run request will not be sent to the webhook and the API request will fail instead.

Webhooks indicate whether they have side effects using the `sideEffects` field in the webhook configuration:

- `None`: calling the webhook will have no side effects.
- `NoneOnDryRun`: calling the webhook will possibly have side effects, but if a request with `dryRun: true` is sent to the webhook, the webhook will suppress the side effects (the webhook is `dryRun`-aware).

Here is an example of a validating webhook indicating it has no side effects on `dryRun: true` requests:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:  - name: my-webhook.example.com    sideEffects: NoneOnDryRun
```

Timeouts

Because webhooks add to API request latency, they should evaluate as quickly as possible. `timeoutSeconds` allows configuring how long the API server should wait for a webhook to respond before treating the call as a failure.

If the timeout expires before the webhook responds, the webhook call will be ignored or the API call will be rejected based on the failure policy.

The timeout value must be between 1 and 30 seconds.

Here is an example of a validating webhook with a custom timeout of 2 seconds:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfigurationwebhooks:  - name: my-webhook.example.com    timeoutSeconds: 2
```

The timeout for an admission webhook defaults to 10 seconds.

Reinvocation policy

A single ordering of mutating admissions plugins (including webhooks) does not work for all cases (see https://issue.k8s.io/64333 as an example). A mutating webhook can add a new sub-structure to the object (like adding a `container` to a `pod`), and other mutating plugins which have already run may have opinions on those new structures (like setting an `imagePullPolicy` on all containers).

To allow mutating admission plugins to observe changes made by other plugins, built-in mutating admission plugins are re-run if a mutating webhook modifies an object, and mutating webhooks can specify a `reinvocationPolicy` to control whether they are reinvoked as well.

`reinvocationPolicy` may be set to `Never` or `IfNeeded`. It defaults to `Never`.

- `Never`: the webhook must not be called more than once in a single admission evaluation.
- `IfNeeded`: the webhook may be called again as part of the admission evaluation if the object being admitted is modified by other admission plugins after the initial webhook call.

The important elements to note are:

- The number of additional invocations is not guaranteed to be exactly one.
- If additional invocations result in further modifications to the object, webhooks are not guaranteed to be invoked again.
- Webhooks that use this option may be reordered to minimize the number of additional invocations.
- To validate an object after all mutations are guaranteed complete, use a validating admission webhook instead (recommended for webhooks with side-effects).

Here is an example of a mutating webhook opting into being re-invoked if later admission plugins modify the object:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfigurationwebhooks:- name: my-webhook.example.com  reinvocationPolicy: IfNeeded
```

Mutating webhooks must be idempotent, able to successfully process an object they have already admitted and potentially modified. This is true for all mutating admission webhooks, since any change they can make in an object could already exist in the user-provided object, but it is essential for webhooks that opt into reinvocation.

Failure policy

`failurePolicy` defines how unrecognized errors and timeout errors from the admission webhook are handled. Allowed values are `Ignore` or `Fail`.

- `Ignore` means that an error calling the webhook is ignored and the API request is allowed to continue.
- `Fail` means that an error calling the webhook causes the admission to fail and the API request to be rejected.

Here is a mutating webhook configured to reject an API request if errors are encountered calling the admission webhook:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfigurationwebhooks:- name: my-webhook.example.com  failurePolicy: Fail
```

The default `failurePolicy` for an admission webhooks is `Fail`.

Monitoring admission webhooks

The API server provides ways to monitor admission webhook behaviors. These monitoring mechanisms help cluster admins to answer questions like:

1. Which mutating webhook mutated the object in a API request?
2. What change did the mutating webhook applied to the object?
3. Which webhooks are frequently rejecting API requests? What's the reason for a rejection?

Mutating webhook auditing annotations

Sometimes it's useful to know which mutating webhook mutated the object in a API request, and what change did the webhook apply.

The Kubernetes API server performs auditing on each mutating webhook invocation. Each invocation generates an auditing annotation capturing if a request object is mutated by the invocation, and optionally generates an annotation capturing the applied patch from the webhook admission response. The annotations are set in the audit event for given request on given stage of its execution, which is then pre-processed according to a certain policy and written to a backend.

The audit level of a event determines which annotations get recorded:

- At `Metadata` audit level or higher, an annotation with key `mutation.webhook.admission.k8s.io/round_{round idx}_index_{order idx}` gets logged with JSON payload indicating a webhook gets invoked for given request and whether it mutated the object or not.

  For example, the following annotation gets recorded for a webhook being reinvoked. The webhook is ordered the third in the mutating webhook chain, and didn't mutated the request object during the invocation.

  ```
  # the audit event recorded
  {    "kind": "Event",    "apiVersion": "audit.k8s.io/v1",    "annotations": {        "mutation.webhook.admission.k8s.io/round

  # the annotation value deserialized
  {    "configuration": "my-mutating-webhook-configuration.example.com",    "webhook": "my-webhook.example.com",    "mutated":
  ```

  The following annotation gets recorded for a webhook being invoked in the first round. The webhook is ordered the first in the mutating webhook chain, and mutated the request object during the invocation.

  ```
  # the audit event recorded
  {    "kind": "Event",    "apiVersion": "audit.k8s.io/v1",    "annotations": {        "mutation.webhook.admission.k8s.io/round

  # the annotation value deserialized
  {    "configuration": "my-mutating-webhook-configuration.example.com",    "webhook": "my-webhook-always-mutate.example.com",
  ```

- At `Request` audit level or higher, an annotation with key `patch.webhook.admission.k8s.io/round_{round idx}_index_{order idx}` gets logged with JSON payload indicating a webhook gets invoked for given request and what patch gets applied to the request object.

  For example, the following annotation gets recorded for a webhook being reinvoked. The webhook is ordered the fourth in the mutating webhook chain, and responded with a JSON patch which got applied to the request object.

  ```
  # the audit event recorded
  {    "kind": "Event",    "apiVersion": "audit.k8s.io/v1",    "annotations": {        "patch.webhook.admission.k8s.io/round_1_

  # the annotation value deserialized
  {    "configuration": "my-other-mutating-webhook-configuration.example.com",    "webhook": "my-webhook-always-mutate.example.
  ```

Admission webhook metrics

The API server exposes Prometheus metrics from the `/metrics` endpoint, which can be used for monitoring and diagnosing API server status. The following metrics record status related to admission webhooks.

API server admission webhook rejection count

Sometimes it's useful to know which admission webhooks are frequently rejecting API requests, and the reason for a rejection.

The API server exposes a Prometheus counter metric recording admission webhook rejections. The metrics are labelled to identify the causes of webhook rejection(s):

- `name`: the name of the webhook that rejected a request.
- `operation`: the operation type of the request, can be one of `CREATE`, `UPDATE`, `DELETE` and `CONNECT`.
- `type`: the admission webhook type, can be one of `admit` and `validating`.
- `error_type`: identifies if an error occurred during the webhook invocation that caused the rejection. Its value can be one of:
  - `calling_webhook_error`: unrecognized errors or timeout errors from the admission webhook happened and the webhook's Failure policy is set to `Fail`.
  - `no_error`: no error occurred. The webhook rejected the request with `allowed: false` in the admission response. The metrics label `rejection_code` records the `.status.code` set in the admission response.
  - `apiserver_internal_error`: an API server internal error happened.
- `rejection_code`: the HTTP status code set in the admission response when a webhook rejected a request.

Example of the rejection count metrics:

```
# HELP apiserver_admission_webhook_rejection_count [ALPHA] Admission webhook rejection count, identified by name and broken out fo
# TYPE apiserver_admission_webhook_rejection_count counter
apiserver_admission_webhook_rejection_count{error_type="calling_webhook_error",name="always-timeout-webhook.example.com",operation=
apiserver_admission_webhook_rejection_count{error_type="calling_webhook_error",name="invalid-admission-response-webhook.example.co
apiserver_admission_webhook_rejection_count{error_type="no_error",name="deny-unwanted-configmap-data.example.com",operation="CREAT
```

Best practices and warnings

For recommendations and considerations when writing mutating admission webhooks, see Admission Webhooks Good Practices.

Webhook Mode

A WebHook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST. A web application implementing WebHooks will POST a message to a URL when certain things happen.

When specified, mode `Webhook` causes Kubernetes to query an outside REST service when determining user privileges.

Configuration File Format

Mode `Webhook` requires a file for HTTP configuration, specify by the `--authorization-webhook-config-file=SOME_FILENAME` flag.

The configuration file uses the kubeconfig file format. Within the file "users" refers to the API Server webhook and "clusters" refers to the remote service.

A configuration example which uses HTTPS client auth:

```
# Kubernetes API version
apiVersion: v1# kind of the API objectkind: Config# clusters refers to the remote service.clusters:  - name: name-of-remote-authz-
```

Request Payloads

When faced with an authorization decision, the API Server POSTs a JSON- serialized `authorization.k8s.io/v1beta1` `SubjectAccessReview` object describing the action. This object contains fields describing the user attempting to make the request, and either details about the resource being accessed or requests attributes.

Note that webhook API objects are subject to the same versioning compatibility rules as other Kubernetes API objects. Implementers should be aware of looser compatibility promises for beta objects and check the "apiVersion" field of the request to ensure correct deserialization. Additionally, the API Server must enable the `authorization.k8s.io/v1beta1` API extensions group (`--runtime-config=authorization.k8s.io/v1beta1=true`).

An example request body:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "kittensandponies",
      "verb": "get",
      "group": "unicorn.example.org",
      "resource": "pods"
    },
    "user": "jane",
    "group": [
      "group1",
      "group2"
    ]
  }
}
```

The remote service is expected to fill the `status` field of the request and respond to either allow or disallow access. The response body's `spec` field is ignored and may be omitted. A permissive response would return:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
  }
}
```

For disallowing access there are two methods.

The first method is preferred in most cases, and indicates the authorization webhook does not allow, or has "no opinion" about the request, but if other authorizers are configured, they are given a chance to allow the request. If there are no other authorizers, or none of them allow the request, the request is forbidden. The webhook would return:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "reason": "user does not have read access to the namespace"
  }
}
```

The second method denies immediately, short-circuiting evaluation by other configured authorizers. This should only be used by webhooks that have detailed knowledge of the full authorizer configuration of the cluster. The webhook would return:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "denied": true,
    "reason": "user does not have read access to the namespace"
  }
}
```

Access to non-resource paths are sent as:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "nonResourceAttributes": {
      "path": "/debug",
      "verb": "get"
    },
    "user": "jane",
    "group": [
      "group1",
```

```
      "group2"
    ]
  }
}
```

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

When calling out to an authorization webhook, Kubernetes passes label and field selectors in the request to the authorization webhook. The authorization webhook can make authorization decisions informed by the scoped field and label selectors, if it wishes.

The SubjectAccessReview API documentation gives guidelines for how these fields should be interpreted and handled by authorization webhooks, specifically using the parsed requirements rather than the raw selector strings, and how to handle unrecognized operators safely.

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "verb": "list",
      "group": "",
      "resource": "pods",
      "fieldSelector": {
        "requirements": [
          {"key":"spec.nodeName", "operator":"In", "values":["mynode"]}
        ]
      },
      "labelSelector": {
        "requirements": [
          {"key":"example.com/mykey", "operator":"In", "values":["myvalue"]}
        ]
      }
    },
    "user": "jane",
    "group": [
      "group1",
      "group2"
    ]
  }
}
```

Non-resource paths include: `/api`, `/apis`, `/metrics`, `/logs`, `/debug`, `/healthz`, `/livez`, `/openapi/v2`, `/readyz`, and `/version.` Clients require access to `/api`, `/api/*`, `/apis`, `/apis/*`, and `/version` to discover what resources and versions are present on the server. Access to other non-resource paths can be disallowed without restricting access to the REST api.

For further information, refer to the SubjectAccessReview API documentation and webhook.go implementation.

Authenticating

This page provides an overview of authentication in Kubernetes, with a focus on authentication to the Kubernetes API.

Users in Kubernetes

All Kubernetes clusters have two categories of users: service accounts managed by Kubernetes, and normal users.

It is assumed that a cluster-independent service manages normal users in the following ways:

- an administrator distributing private keys
- a user store like Keystone or Google Accounts
- a file with a list of usernames and passwords

In this regard, *Kubernetes does not have objects which represent normal user accounts*. Normal users cannot be added to a cluster through an API call.

Even though a normal user cannot be added via an API call, any user that presents a valid certificate signed by the cluster's certificate authority (CA) is considered authenticated. In this configuration, Kubernetes determines the username from the common name field in the 'subject' of the cert (e.g., "/CN=bob"). From there, the role based access control (RBAC) sub-system would determine whether the user is authorized to perform a specific operation on a resource.

In contrast, service accounts are users managed by the Kubernetes API. They are bound to specific namespaces, and created automatically by the API server or manually through API calls. Service accounts are tied to a set of credentials stored as `Secrets`, which are mounted into pods allowing in-cluster processes to talk to the Kubernetes API.

API requests are tied to either a normal user or a service account, or are treated as anonymous requests. This means every process inside or outside the cluster, from a human user typing `kubectl` on a workstation, to `kubelets` on nodes, to members of the control plane, must authenticate when making requests to the API server, or be treated as an anonymous user.

Authentication strategies

Kubernetes uses client certificates, bearer tokens, or an authenticating proxy to authenticate API requests through authentication plugins. As HTTP requests are made to the API server, plugins attempt to associate the following attributes with the request:

- Username: a string which identifies the end user. Common values might be `kube-admin` or `jane@example.com`.
- UID: a string which identifies the end user and attempts to be more consistent and unique than username.
- Groups: a set of strings, each of which indicates the user's membership in a named logical collection of users. Common values might be `system:masters` or `devops-team`.
- Extra fields: a map of strings to list of strings which holds additional information authorizers may find useful.

Note:

All values are opaque to the authentication system and only hold significance when interpreted by an authorizer.

Authentication methods

You can enable multiple authentication methods at once. You should usually use at least two methods:

- service account tokens for service accounts
- at least one other method for user authentication.

When multiple authenticator modules are enabled, the first module to successfully authenticate the request short-circuits evaluation. The API server does not guarantee the order authenticators run in.

The `system:authenticated` group is included in the list of groups for all authenticated users.

Integrations with other authentication protocols (LDAP, SAML, Kerberos, alternate x509 schemes, etc) are available; for example using an authenticating proxy or the authentication webhook.

X.509 client certificates

Client certificate authentication is enabled by passing the `--client-ca-file=SOMEFILE` option to API server. The referenced file must contain one or more certificate authorities to use to validate client certificates presented to the API server. If a client certificate is presented and verified, the common name of the subject is used as the user name for the request. As of Kubernetes 1.4, client certificates can also indicate a user's group memberships using the certificate's organization fields. To include multiple group memberships for a user, include multiple organization fields in the certificate.

For example, using the `openssl` command line tool to generate a certificate signing request:

```
openssl req -new -key jbeda.pem -out jbeda-csr.pem -subj "/CN=jbeda/O=app1/O=app2"
```

This would create a CSR for the username "jbeda", belonging to two groups, "app1" and "app2".

See Managing Certificates for how to generate a client cert.

Putting a bearer token in a request

When using bearer token authentication from an http client, the API server expects an `Authorization` header with a value of `Bearer <token>`. The bearer token must be a character sequence that can be put in an HTTP header value using no more than the encoding and quoting facilities of HTTP. For example: if the bearer token is `31ada4fd-adec-460c-809a-9e56ceb75269` then it would appear in an HTTP header as shown below.

```
Authorization: Bearer 31ada4fd-adec-460c-809a-9e56ceb75269
```

Bootstrap tokens

FEATURE STATE: `Kubernetes v1.18 [stable]`

To allow for streamlined bootstrapping for new clusters, Kubernetes includes a dynamically-managed Bearer token type called a *Bootstrap Token*. These tokens are stored as Secrets in the `kube-system` namespace, where they can be dynamically managed and created. Controller Manager contains a TokenCleaner controller that deletes bootstrap tokens as they expire.

The tokens are of the form `[a-z0-9]{6}.[a-z0-9]{16}`. The first component is a Token ID and the second component is the Token Secret. You specify the token in an HTTP header as follows:

```
Authorization: Bearer 781292.db7bc3a58fc5f07e
```

You must enable the Bootstrap Token Authenticator with the `--enable-bootstrap-token-auth` flag on the API Server. You must enable the TokenCleaner controller via the `--controllers` command line argument for kube-controller-manager. This is done with something like `--controllers=*,tokencleaner`. The `kubeadm` tool will do this for you if you are using it to bootstrap a cluster.

The authenticator authenticates as `system:bootstrap:<Token ID>`. It is included in the `system:bootstrappers` group. The naming and groups are intentionally limited to discourage users from using these tokens past bootstrapping. The user names and group can be used (and are used by `kubeadm`) to craft the appropriate authorization policies to support bootstrapping a cluster.

Please see Bootstrap Tokens for in depth documentation on the Bootstrap Token authenticator and controllers along with how to manage these tokens with `kubeadm`.

Service account tokens

A service account is an automatically enabled authenticator that uses signed bearer tokens to verify requests. The plugin takes two optional flags:

- `--service-account-key-file` File containing PEM-encoded x509 RSA or ECDSA private or public keys, used to verify ServiceAccount tokens. The specified file can contain multiple keys, and the flag can be specified multiple times with different files. If unspecified, --tls-private-key-file is used.
- `--service-account-lookup` If enabled, tokens which are deleted from the API will be revoked.

Service accounts are usually created automatically by the API server and associated with pods running in the cluster through the `ServiceAccount` Admission Controller. Bearer tokens are mounted into pods at well-known locations, and allow in-cluster processes to talk to the API server. Accounts may be explicitly associated with pods using the `serviceAccountName` field of a `PodSpec`.

Note:

`serviceAccountName` is usually omitted because this is done automatically.

```
apiVersion: apps/v1 # this apiVersion is relevant as of Kubernetes 1.9
kind: Deploymentmetadata:  name: nginx-deployment  namespace: defaultspec:  replicas: 3  template:    metadata:    # ...    spec:
```

Service account bearer tokens are perfectly valid to use outside the cluster and can be used to create identities for long standing jobs that wish to talk to the Kubernetes API. To manually create a service account, use the `kubectl create serviceaccount (NAME)` command. This creates a service account in the current namespace.

```
kubectl create serviceaccount jenkins
```

```
serviceaccount/jenkins created
```

You can manually create an associated token:

```
kubectl create token jenkins
```

```
eyJhbGciOiJSUzI1NiIsImtp...
```

The created token is a signed JSON Web Token (JWT).

The signed JWT can be used as a bearer token to authenticate as the given service account. See above for how the token is included in a request. Normally these tokens are mounted into pods for in-cluster access to the API server, but can be used from outside the cluster as well.

Service accounts authenticate with the username `system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT)`, and are assigned to the groups `system:serviceaccounts` and `system:serviceaccounts:(NAMESPACE)`.

Warning:

Because service account tokens can also be stored in Secret API objects, any user with write access to Secrets can request a token, and any user with read access to those Secrets can authenticate as the service account. Be cautious when granting permissions to service accounts and read or write capabilities for Secrets.

External integrations

Kubernetes has native support for OpenID Connect (OIDC); see OpenID Connect tokens.

Integrations with other authentication protocols (for example: LDAP, SAML, Kerberos, alternate X.509 schemes) can be accomplished using an authenticating proxy or by integrating with an authentication webhook.

You can also use any custom method that issues client X.509 certificates to clients, provided that the API server will trust the valid certificates. Read X.509 client certificates to learn about how to generate a certificate.

If you do issue certificates to clients, it is up to you (as a cloud platform administrator) to make sure that the certificate validity period, and other design choices you make, provide a suitable level of security.

OpenID Connect tokens

OpenID Connect is a flavor of OAuth2 supported by some OAuth2 providers, notably Microsoft Entra ID, Salesforce, and Google. The protocol's main extension of OAuth2 is an additional field returned with the access token called an ID Token. This token is a JSON Web Token (JWT) with well known fields, such as a user's email, signed by the server.

To identify the user, the authenticator uses the `id_token` (not the `access_token`) from the OAuth2 token response as a bearer token. See above for how the token is included in a request.

> sequenceDiagram participant user as User participant idp as Identity Provider participant kube as kubectl participant api as API Server user ->> idp: 1. Log in to IdP activate idp idp -->> user: 2. Provide access_token, id_token, and refresh_token deactivate idp activate user user ->> kube: 3. Call kubectl with --token being the id_token OR add tokens to .kube/config deactivate user activate kube kube ->> api: 4. Authorization: Bearer... deactivate kube activate api api ->> api: 5. Is JWT signature valid? api ->> api: 6. Has the JWT expired? (iat+exp) api ->> api: 7. User authorized? api -->> kube: 8. Authorized: Perform action and return result deactivate api activate kube kube --x user: 9. Return result deactivate kube

1. Log in to your identity provider
2. Your identity provider will provide you with an `access_token`, `id_token` and a `refresh_token`
3. When using `kubectl`, use your `id_token` with the `--token` command line argument or add it directly to your `kubeconfig`
4. `kubectl` sends your `id_token` in a header called Authorization to the API server
5. The API server will make sure the JWT signature is valid
6. Check to make sure the `id_token` hasn't expired

   Perform claim and/or user validation if CEL expressions are configured with `AuthenticationConfiguration`.
7. Make sure the user is authorized
8. Once authorized the API server returns a response to `kubectl`
9. `kubectl` provides feedback to the user

Since all of the data needed to validate who you are is in the `id_token`, Kubernetes doesn't need to "phone home" to the identity provider. In a model where every request is stateless this provides a very scalable solution for authentication. It does offer a few challenges:

1. Kubernetes has no "web interface" to trigger the authentication process. There is no browser or interface to collect credentials which is why you need to authenticate to your identity provider first.
2. The `id_token` can't be revoked, it's like a certificate so it should be short-lived (only a few minutes) so it can be very annoying to have to get a new token every few minutes.
3. To authenticate to the Kubernetes dashboard, you must use the `kubectl proxy` command or a reverse proxy that injects the `id_token`.

Configuring the API Server

Using command line arguments

To enable the plugin, configure the following command line arguments for the API server:

Parameter	Description	Example	Required
`--oidc-issuer-url`	URL of the provider that allows the API server to discover public signing keys. Only URLs that use the `https://` scheme are accepted. This is typically the provider's discovery URL, changed to have an empty path.	If the issuer's OIDC discovery URL is `https://accounts.provider.example/.well-known/openid-configuration`, the value should be `https://accounts.provider.example`	Yes
`--oidc-client-id`	A client id that all tokens must be issued for.	kubernetes	Yes
`--oidc-username-claim`	JWT claim to use as the user name. By default `sub`, which is expected to be a unique identifier of the end user. Admins can choose other claims, such as `email` or `name`, depending on their provider. However, claims other than `email` will be prefixed with the issuer URL to prevent naming clashes with other plugins.	sub	No
`--oidc-username-prefix`	Prefix prepended to username claims to prevent clashes with existing names (such as `system:` users). For example, the value `oidc:` will create usernames like `oidc:jane.doe`. If this argument isn't provided and `--oidc-username-claim` is a value other than `email` the prefix defaults to `( Issuer URL )#` where `( Issuer URL )` is the value of `--oidc-issuer-url`. The value `-` can be used to disable all prefixing.	`oidc:`	No
`--oidc-groups-claim`	JWT claim to use as the user's group. If the claim is present it must be an array of strings.	groups	No
`--oidc-groups-prefix`	Prefix prepended to group claims to prevent clashes with existing names (such as `system:` groups). For example, the value `oidc:` will create group names like `oidc:engineering` and `oidc:infra`.	`oidc:`	No
`--oidc-required-claim`	A key=value pair that describes a required claim in the ID Token. If set, the claim is verified to be present in the ID Token with a matching value. Repeat this argument to specify multiple claims.	`claim=value`	No
`--oidc-ca-file`	The path to the certificate for the CA that signed your identity provider's web certificate. Defaults to the host's root CAs.	`/etc/kubernetes/ssl/kc-ca.pem`	No
`--oidc-signing-algs`	The signing algorithms accepted. Default is RS256. Allowed values are: RS256, RS384, RS512, ES256, ES384, ES512, PS256, PS384, PS512. Values are defined by RFC 7518 https://tools.ietf.org/html/rfc7518#section-3.1.	`RS512`	No

Authentication configuration from a file

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

JWT Authenticator is an authenticator to authenticate Kubernetes users using JWT compliant tokens. The authenticator will attempt to parse a raw ID token, verify it's been signed by the configured issuer. The public key to verify the signature is discovered from the issuer's public endpoint using OIDC discovery.

The minimum valid JWT payload must contain the following claims:

```
{
  "iss": "https://example.com",   // must match the issuer.url
  "aud": ["my-app"],              // at least one of the entries in issuer.audiences must match the "aud" claim in presented JWTs.
```

The configuration file approach allows you to configure multiple JWT authenticators, each with a unique `issuer.url` and `issuer.discoveryURL`. The configuration file even allows you to specify CEL expressions to map claims to user attributes, and to validate claims and user information. The API server also automatically reloads the authenticators when the configuration file is modified. You can use `apiserver_authentication_config_controller_automatic_reload_last_timestamp_seconds` metric to monitor the last time the configuration was reloaded by the API server.

You must specify the path to the authentication configuration using the `--authentication-config` command line argument to the API server. If you want to use command line arguments instead of the configuration file, those will continue to work as-is. To access the new capabilities like configuring multiple authenticators, setting multiple audiences for an issuer, switch to using the configuration file.

To use structured authentication, specify the `--authentication-config` command line argument to the kube-apiserver. An example of the structured authentication configuration file is shown below.

Note:

If you specify `--authentication-config` along with any of the `--oidc-*` command line arguments, this is a misconfiguration. In this situation, the API server reports an error and then immediately exits. If you want to switch to using structured authentication configuration, you have to remove the `--oidc-*` command line arguments, and use the configuration file instead.

```
---
## CAUTION: this is an example configuration.#         Do not use this for your own cluster!#apiVersion: apiserver.config.k8s.io/
```

- Claim validation rule expression

  `jwt.claimValidationRules[i].expression` represents the expression which will be evaluated by CEL. CEL expressions have access to the contents of the token payload, organized into `claims` CEL variable. `claims` is a map of claim names (as strings) to claim values (of any type).

- User validation rule expression

  `jwt.userValidationRules[i].expression` represents the expression which will be evaluated by CEL. CEL expressions have access to the contents of `userInfo`, organized into `user` CEL variable. Refer to the UserInfo API documentation for the schema of `user`.

- Claim mapping expression

  `jwt.claimMappings.username.expression`, `jwt.claimMappings.groups.expression`, `jwt.claimMappings.uid.expression` `jwt.claimMappings.extra[i].valueExpression` represents the expression which will be evaluated by CEL. CEL expressions have access to the contents of the token payload, organized into `claims` CEL variable. `claims` is a map of claim names (as strings) to claim values (of any type).

  To learn more, see the Documentation on CEL

  Here are examples of the `AuthenticationConfiguration` with different token payloads.

  - Valid token
  - Fails claim validation
  - Fails user validation

```
apiVersion: apiserver.config.k8s.io/v1
kind: AuthenticationConfigurationjwt:- issuer:    url: https://example.com    audiences:    - my-app  claimMappings:    usern

TOKEN=eyJhbGciOiJSUzI1NiIsImtpZCI6ImY3dF9tOEROWmFTQk1oWGw5QXZTWGhBUC04Y0JmZ0JVbFVpTG5oQkgxdXMiLCJ0eXAiOiJKV1QifQ.eyJhdWQiOiJr
```

where the token payload is:

```
{
  "aud": "kubernetes",
  "exp": 1703232949,
  "iat": 1701107233,
  "iss": "https://example.com",
  "jti": "7c337942807e73caa2c30c868ac0ce910bce02ddcbfebe8c23b8b5f27ad62873",
  "nbf": 1701107233,
  "roles": "user,admin",
  "sub": "auth",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db4a",
  "username": "foo"
}
```

The token with the above `AuthenticationConfiguration` will produce the following `UserInfo` object and successfully authenticate the user.

```
{
    "username": "foo:external-user",
    "uid": "auth",
    "groups": [
        "user",
        "admin"
    ],
    "extra": {
        "example.com/tenant": ["72f988bf-86f1-41af-91ab-2d7cd011db4a"]
    }
}
```

```
apiVersion: apiserver.config.k8s.io/v1
kind: AuthenticationConfigurationjwt:- issuer:    url: https://example.com    audiences:    - my-app  claimValidationRules:

TOKEN=eyJhbGciOiJSUzI1NiIsImtpZCI6ImY3dF9tOEROWmFTQk1oWGw5QXZTWGhBUC04Y0JmZ0JVbFVpTG5oQkgxdXMiLCJ0eXAiOiJKV1QifQ.eyJhdWQiOiJr
```

where the token payload is:

```
{
  "aud": "kubernetes",
  "exp": 1703232949,
  "iat": 1701107233,
  "iss": "https://example.com",
  "jti": "7c337942807e73caa2c30c868ac0ce910bce02ddcbfebe8c23b8b5f27ad62873",
  "nbf": 1701107233,
  "roles": "user,admin",
  "sub": "auth",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db4a",
  "username": "foo"
}
```

The token with the above `AuthenticationConfiguration` will fail to authenticate because the `hd` claim is not set to `example.com`. The API server will return `401 Unauthorized` error.

```
apiVersion: apiserver.config.k8s.io/v1
kind: AuthenticationConfigurationjwt:- issuer:    url: https://example.com    audiences:    - my-app  claimValidationRules:

TOKEN=eyJhbGciOiJSUzI1NiIsImtpZCI6ImY3dF9tOEROWmFTQk1oWGw5QXZTWGhBUC04Y0JmZ0JVbFVpTG5oQkgxdXMiLCJ0eXAiOiJKV1QifQ.eyJhdWQiOiJr
```

where the token payload is:

```
{
  "aud": "kubernetes",
  "exp": 1703232949,
  "hd": "example.com",
```

```
  "iat": 1701113101,
  "iss": "https://example.com",
  "jti": "b5b0652372cd20e345b6fdffcdc2181f4afd6f259aab4b7e35881237d29220bc",
  "nbf": 1701113101,
  "roles": "user,admin",
  "sub": "auth",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db4a",
  "username": "foo"
}
```


The token with the above `AuthenticationConfiguration` will produce the following `UserInfo` object:


```
{
    "username": "system:foo",
    "uid": "auth",
    "groups": [
        "user",
        "admin"
    ],
    "extra": {
        "example.com/tenant": ["72f988bf-86f1-41af-91ab-2d7cd011db4a"]
    }
}
```


which will fail user validation because the username starts with `system:`. The API server will return `401 Unauthorized` error.

JWT egress selector type

FEATURE STATE: `Kubernetes v1.34 [beta]` (enabled by default: true)

The *egressSelectorType* field in the JWT issuer configuration allows you to specify which egress selector should be used for sending all traffic related to the issuer (discovery, JWKS, distributed claims, etc). This feature requires the `StructuredAuthenticationConfigurationEgressSelector` feature gate to be enabled.

Limitations

1. Distributed claims do not work via CEL expressions.

Kubernetes does not provide an OpenID Connect Identity Provider. You can use an existing public OpenID Connect Identity Provider or run your own Identity Provider that supports the OpenID Connect protocol.

For an identity provider to work with Kubernetes it must:

1. Support OpenID connect discovery

   The public key to verify the signature is discovered from the issuer's public endpoint using OIDC discovery. If you're using the authentication configuration file, the identity provider doesn't need to publicly expose the discovery endpoint. You can host the discovery endpoint at a different location than the issuer (such as locally in the cluster) and specify the `issuer.discoveryURL` in the configuration file.

2. Run in TLS with non-obsolete ciphers

3. Have a CA signed certificate (even if the CA is not a commercial CA or is self signed)

A note about requirement #3 above, requiring a CA signed certificate. If you deploy your own identity provider you MUST have your identity provider's web server certificate signed by a certificate with the `CA` flag set to `TRUE`, even if it is self signed. This is due to GoLang's TLS client implementation being very strict to the standards around certificate validation. If you don't have a CA handy, you can create a simple CA and a signed certificate and key pair using standard certificate generation tools.

Using kubectl

Option 1 - OIDC authenticator

The first option is to use the kubectl `oidc` authenticator, which sets the `id_token` as a bearer token for all requests and refreshes the token once it expires. After you've logged into your provider, use kubectl to add your `id_token`, `refresh_token`, `client_id`, and `client_secret` to configure the plugin.

Providers that don't return an `id_token` as part of their refresh token response aren't supported by this plugin and should use Option 2 (specifying `--token`).

```
kubectl config set-credentials USER_NAME \
   --auth-provider=oidc \   --auth-provider-arg=idp-issuer-url=( issuer url ) \   --auth-provider-arg=client-id=( your client id )
```

As an example, running the below command after authenticating to your identity provider:

```
kubectl config set-credentials mmosley  \
        --auth-provider=oidc  \        --auth-provider-arg=idp-issuer-url=https://oidcidp.tremolo.lan:8443/auth/idp/OidcIdP  \
```

Which would produce the below configuration:

```
users:
- name: mmosley  user:    auth-provider:      config:        client-id: kubernetes        client-secret: 1db158f6-177d-4d9c-8a8b-d
```

Once your `id_token` expires, `kubectl` will attempt to refresh your `id_token` using your `refresh_token` and `client_secret` storing the new values for the `refresh_token` and `id_token` in your `.kube/config`.

Option 2 - Use the `--token` command line argument

The `kubectl` command lets you pass in a token using the `--token` command line argument. Copy and paste the `id_token` into this option:

```
kubectl --token=eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL21sYi50cmVtb2xvLmxhbjo4MDQzL2F1dGgvaWRwL29pZGMiLCJhdWQiOiJrdWJlcm5ldGVzI:
```

Webhook token authentication

Kubernetes *webhook authentication* is a mechanism to make an HTTP callout for verifying bearer tokens.

In terms of how you configure the API server:

- `--authentication-token-webhook-config-file` a configuration file describing how to access the remote webhook service.
- `--authentication-token-webhook-cache-ttl` how long to cache authentication decisions. Defaults to two minutes.
- `--authentication-token-webhook-version` determines whether to use `authentication.k8s.io/v1beta1` or `authentication.k8s.io/v1` `TokenReview` objects to send/receive information from the webhook. Defaults to `v1beta1`.

The configuration file uses the kubeconfig file format. Within the file, `clusters` refers to the remote service and `users` refers to the API server webhook. An example would be:

```
# Kubernetes API version
apiVersion: v1# kind of the API objectkind: Config# clusters refers to the remote service.clusters:  - name: name-of-remote-authn-
```

When a client attempts to authenticate with the API server using a bearer token as discussed above, the authentication webhook POSTs a JSON-serialized `TokenReview` object containing the token to the remote service.

Note that webhook API objects are subject to the same versioning compatibility rules as other Kubernetes API objects. Implementers should check the `apiVersion` field of the request to ensure correct deserialization, and **must** respond with a `TokenReview` object of the same version as the request.

- authentication.k8s.io/v1
- authentication.k8s.io/v1beta1

Note:

The Kubernetes API server defaults to sending `authentication.k8s.io/v1beta1` token reviews for backwards compatibility. To opt into receiving `authentication.k8s.io/v1` token reviews, the API server must be started with `--authentication-token-webhook-version=v1`.

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "TokenReview",
  "spec": {
    # Opaque bearer token sent to the API server
    "token": "014fbff9a07c...",

    # Optional list of the audience identifiers for the server the token was presented to.
    # Audience-aware token authenticators (for example, OIDC token authenticators)
    # should verify the token was intended for at least one of the audiences in this list,
    # and return the intersection of this list and the valid audiences for the token in the response status.
    # This ensures the token is valid to authenticate to the server it was presented to.
    # If no audiences are provided, the token should be validated to authenticate to the Kubernetes API server.
    "audiences": ["https://myserver.example.com", "https://myserver.internal.example.com"]
  }
}

{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    # Opaque bearer token sent to the API server
    "token": "014fbff9a07c...",

    # Optional list of the audience identifiers for the server the token was presented to.
    # Audience-aware token authenticators (for example, OIDC token authenticators)
    # should verify the token was intended for at least one of the audiences in this list,
    # and return the intersection of this list and the valid audiences for the token in the response status.
    # This ensures the token is valid to authenticate to the server it was presented to.
    # If no audiences are provided, the token should be validated to authenticate to the Kubernetes API server.
    "audiences": ["https://myserver.example.com", "https://myserver.internal.example.com"]
  }
}
```

The remote service is expected to fill the `status` field of the request to indicate the success of the login. The response body's `spec` field is ignored and may be omitted. The remote service must return a response using the same `TokenReview` API version that it received. A successful validation of the bearer token would return:

- authentication.k8s.io/v1
- authentication.k8s.io/v1beta1

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "TokenReview",
  "status": {
    "authenticated": true,
    "user": {
      # Required
      "username": "janedoe@example.com",
      # Optional
      "uid": "42",
      # Optional group memberships
      "groups": ["developers", "qa"],
      # Optional additional information provided by the authenticator.
      # This should not contain confidential data, as it can be recorded in logs
      # or API objects, and is made available to admission webhooks.
      "extra": {
```

```
        "extrafield1": [
          "extravalue1",
          "extravalue2"
        ]
      }
    },
    # Optional list audience-aware token authenticators can return,
    # containing the audiences from the `spec.audiences` list for which the provided token was valid.
    # If this is omitted, the token is considered to be valid to authenticate to the Kubernetes API server.
    "audiences": ["https://myserver.example.com"]
  }
}

{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": true,
    "user": {
      # Required
      "username": "janedoe@example.com",
      # Optional
      "uid": "42",
      # Optional group memberships
      "groups": ["developers", "qa"],
      # Optional additional information provided by the authenticator.
      # This should not contain confidential data, as it can be recorded in logs
      # or API objects, and is made available to admission webhooks.
      "extra": {
        "extrafield1": [
          "extravalue1",
          "extravalue2"
        ]
      }
    },
    # Optional list audience-aware token authenticators can return,
    # containing the audiences from the `spec.audiences` list for which the provided token was valid.
    # If this is omitted, the token is considered to be valid to authenticate to the Kubernetes API server.
    "audiences": ["https://myserver.example.com"]
  }
}
```

An unsuccessful request would return:

- authentication.k8s.io/v1
- authentication.k8s.io/v1beta1

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "TokenReview",
  "status": {
    "authenticated": false,
    # Optionally include details about why authentication failed.
    # If no error is provided, the API will return a generic Unauthorized message.
    # The error field is ignored when authenticated=true.
    "error": "Credentials are expired"
  }
}

{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": false,
    # Optionally include details about why authentication failed.
    # If no error is provided, the API will return a generic Unauthorized message.
    # The error field is ignored when authenticated=true.
    "error": "Credentials are expired"
  }
}
```

Authenticating reverse proxy

Warning:

If you have a certificate authority (CA) that is also used in a different context, **do not** trust that certificate authority to identify authenticating proxy clients, unless you understand the risks and the mechanisms to protect that CA's usage.

The API server can be configured to identify users from request header values, such as `X-Remote-User`. It is designed for use in combination with an *authenticating proxy* that sets these headers.

The command line arguments to configure this mode are:

`--requestheader-client-ca-file`
: *Required*. Path to a PEM-encoded certificate bundle.
A valid client certificate must be presented and validated against the certificate authorities in the specified file before the request headers are checked for user names.

`--requestheader-allowed-names`
: *Optional*. Comma-separated list of Common Name values (CNs).
If set, a valid client certificate with a CN in the specified list must be presented before the request headers are checked for user names. If empty, any CN is allowed.

`--requestheader-username-headers`

Required; case-insensitive. Header names to check, in order, for the user identity.
The first header containing a value is used as the username.

`--requestheader-group-headers`
Optional; case-insensitive. Header names to check, in order, for the user's groups.
`X-Remote-Group` is suggested. All values in all specified headers are used as group names.

`--requestheader-extra-headers-prefix`
Optional; case-insensitive. Header prefixes to look for to determine extra information about the user.
`X-Remote-Extra-` is suggested. Extra data is typically used by the configured authorization plugin(s). Any headers beginning with any of the specified prefixes have the prefix removed. The remainder of the header name is lowercased and percent-decoded and becomes the extra key, and the header value is the extra value.

For example, with this configuration:

```
--requestheader-username-headers=X-Remote-User
--requestheader-group-headers=X-Remote-Group
--requestheader-extra-headers-prefix=X-Remote-Extra-
```

this request:

```
GET / HTTP/1.1
X-Remote-User: fido
X-Remote-Group: dogs
X-Remote-Group: dachshunds
X-Remote-Extra-Acme.com%2Fproject: some-project
X-Remote-Extra-Scopes: openid
X-Remote-Extra-Scopes: profile
```

would result in this user info:

```
name: fido
groups:- dogs- dachshundsextra:  acme.com/project:  - some-project  scopes:  - openid  - profile
```

Note:

Prior to Kubernetes 1.11.3 (and 1.10.7, 1.9.11), the `extra` key could only contain characters that were legal in HTTP header labels.

Client certificate

In order to prevent header spoofing, the authenticating proxy is required to present a valid client certificate to the API server for validation against the specified CA before the request headers are checked.

Do **not** reuse a CA that is used in a different context unless you understand the risks and the mechanisms to protect the CA's usage.

Static token file integration

The API server reads static bearer tokens from a file when given the `--token-auth-file=<SOMEFILE>` option on the command line. In Kubernetes 1.34, tokens last indefinitely, and the token list cannot be changed without restarting the API server.

The token file is a CSV file with a minimum of 3 columns: token, user name, user uid, followed by a comma-separated list of optional group names.

Note:

If you have more than one group, the column must be double quoted e.g.

```
token,user,uid,"group1,group2,group3"
```

Using a static token file is appropriate for tokens that by their nature are long-lived, static, and perhaps may never be rotated. It is also useful when the client is local to a particular API server within the control plane, such as a monitoring agent.

If you use this method during cluster provisioning, and then transition to a different authentication method that will be used longer term, you should deactivate the token that was used for bootstrapping (this requires a restart of each API server.

For other circumstances, and especially where very prompt token rotation is important, the Kubernetes project recommends using a webhook token authenticator instead of this mechanism.

Anonymous requests

When enabled, requests that are not rejected by other configured authentication methods are treated as anonymous requests, and given a username of `system:anonymous` and a group of `system:unauthenticated`.

For example, on a server with token authentication configured, and anonymous access enabled, a request providing an invalid bearer token would receive a `401 Unauthorized` error. A request providing no bearer token would be treated as an anonymous request.

In 1.5.1-1.5.x, anonymous access is disabled by default, and can be enabled by passing the `--anonymous-auth=true` option to the API server.

In 1.6+, anonymous access is enabled by default if an authorization mode other than `AlwaysAllow` is used, and can be disabled by passing the `--anonymous-auth=false` option to the API server. Starting in 1.6, the ABAC and RBAC authorizers require explicit authorization of the `system:anonymous` user or the `system:unauthenticated` group, so legacy policy rules that grant access to the `*` user or `*` group do not include anonymous users.

Anonymous Authenticator Configuration

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

The `AuthenticationConfiguration` can be used to configure the anonymous authenticator. If you set the anonymous field in the `AuthenticationConfiguration` file then you cannot set the `--anonymous-auth` flag.

The main advantage of configuring anonymous authenticator using the authentication configuration file is that in addition to enabling and disabling anonymous authentication you can also configure which endpoints support anonymous authentication.

A sample authentication configuration file is below:

```
---
## CAUTION: this is an example configuration.#          Do not use this for your own cluster!#apiVersion: apiserver.config.k8s.io/
```

In the configuration above only the `/livez`, `/readyz` and `/healthz` endpoints are reachable by anonymous requests. Any other endpoints will not be reachable even if it is allowed by RBAC configuration.

User impersonation

A user can act as another user through impersonation headers. These let requests manually override the user info a request authenticates as. For example, an admin could use this feature to debug an authorization policy by temporarily impersonating another user and seeing if a request was denied.

Impersonation requests first authenticate as the requesting user, then switch to the impersonated user info.

- A user makes an API call with their credentials *and* impersonation headers.
- API server authenticates the user.
- API server ensures the authenticated users have impersonation privileges.
- Request user info is replaced with impersonation values.
- Request is evaluated, authorization acts on impersonated user info.

The following HTTP headers can be used to performing an impersonation request:

- `Impersonate-User`: The username to act as.
- `Impersonate-Group`: A group name to act as. Can be provided multiple times to set multiple groups. Optional. Requires "Impersonate-User".
- `Impersonate-Extra-( extra name )`: A dynamic header used to associate extra fields with the user. Optional. Requires "Impersonate-User". In order to be preserved consistently, `( extra name )` must be lower-case, and any characters which aren't legal in HTTP header labels MUST be utf8 and percent-encoded.
- `Impersonate-Uid`: A unique identifier that represents the user being impersonated. Optional. Requires "Impersonate-User". Kubernetes does not impose any format requirements on this string.

Note:

Prior to 1.11.3 (and 1.10.7, 1.9.11), `( extra name )` could only contain characters which were legal in HTTP header labels.

Note:

`Impersonate-Uid` is only available in versions 1.22.0 and higher.

An example of the impersonation headers used when impersonating a user with groups:

```
Impersonate-User: jane.doe@example.com
Impersonate-Group: developers
Impersonate-Group: admins
```

An example of the impersonation headers used when impersonating a user with a UID and extra fields:

```
Impersonate-User: jane.doe@example.com
Impersonate-Extra-dn: cn=jane,ou=engineers,dc=example,dc=com
Impersonate-Extra-acme.com%2Fproject: some-project
Impersonate-Extra-scopes: view
Impersonate-Extra-scopes: development
Impersonate-Uid: 06f6ce97-e2c5-4ab8-7ba5-7654dd08d52b
```

When using `kubectl` set the `--as` command line argument to configure the `Impersonate-User` header, you can also set the `--as-group` flag to configure the `Impersonate-Group` header.

```
kubectl drain mynode
```

```
Error from server (Forbidden): User "clark" cannot get nodes at the cluster scope. (get nodes mynode)
```

Set the `--as` and `--as-group` flag:

```
kubectl drain mynode --as=superman --as-group=system:masters
```

```
node/mynode cordoned
node/mynode drained
```

Note:

`kubectl` cannot impersonate extra fields or UIDs.

To impersonate a user, group, user identifier (UID) or extra fields, the impersonating user must have the ability to perform the **impersonate** verb on the kind of attribute being impersonated ("user", "group", "uid", etc.). For clusters that enable the RBAC authorization plugin, the following ClusterRole encompasses the rules needed to set user and group impersonation headers:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: impersonatorrules:- apiGroups: [""]  resources: ["users", "groups", "serviceaccounts"]  verbs: [
```

For impersonation, extra fields and impersonated UIDs are both under the "authentication.k8s.io" `apiGroup`. Extra fields are evaluated as sub-resources of the resource "userextras". To allow a user to use impersonation headers for the extra field `scopes` and for UIDs, a user should be granted the following role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: scopes-and-uid-impersonatorrules:# Can set "Impersonate-Extra-scopes" header and the "Impersonat
```

The values of impersonation headers can also be restricted by limiting the set of `resourceNames` a resource can take.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: limited-impersonatorrules:# Can impersonate the user "jane.doe@example.com"- apiGroups: [""]  re
```

Note:

Impersonating a user or group allows you to perform any action as if you were that user or group; for that reason, impersonation is not namespace scoped. If you want to allow impersonation using Kubernetes RBAC, this requires using a ClusterRole and a ClusterRoleBinding, not a Role and RoleBinding.

client-go credential plugins

FEATURE STATE: `Kubernetes v1.22 [stable]`

`k8s.io/client-go` and tools using it such as `kubectl` and `kubelet` are able to execute an external command to receive user credentials.

This feature is intended for client side integrations with authentication protocols not natively supported by `k8s.io/client-go` (LDAP, Kerberos, OAuth2, SAML, etc.). The plugin implements the protocol specific logic, then returns opaque credentials to use. Almost all credential plugin use cases require a server side component with support for the webhook token authenticator to interpret the credential format produced by the client plugin.

Note:

Earlier versions of `kubectl` included built-in support for authenticating to AKS and GKE, but this is no longer present.

Example use case

In a hypothetical use case, an organization would run an external service that exchanges LDAP credentials for user specific, signed tokens. The service would also be capable of responding to webhook token authenticator requests to validate the tokens. Users would be required to install a credential plugin on their workstation.

To authenticate against the API:

- The user issues a `kubectl` command.
- Credential plugin prompts the user for LDAP credentials, exchanges credentials with external service for a token.
- Credential plugin returns token to client-go, which uses it as a bearer token against the API server.
- API server uses the webhook token authenticator to submit a `TokenReview` to the external service.
- External service verifies the signature on the token and returns the user's username and groups.

Configuration

Credential plugins are configured through kubectl config files as part of the user fields.

- client.authentication.k8s.io/v1
- client.authentication.k8s.io/v1beta1

```
apiVersion: v1
kind: Configusers:- name: my-user  user:    exec:      # Command to execute. Required.      command: "example-client-go-exec-plugi
```

```
apiVersion: v1
kind: Configusers:- name: my-user  user:    exec:      # Command to execute. Required.      command: "example-client-go-exec-plugi
```

Relative command paths are interpreted as relative to the directory of the config file. If KUBECONFIG is set to `/home/jane/kubeconfig` and the exec command is `./bin/example-client-go-exec-plugin`, the binary `/home/jane/bin/example-client-go-exec-plugin` is executed.

```
- name: my-user
  user:
    exec:
      # Path relative to the directory of the kubeconfig
      command: "./bin/example-client-go-exec-plugin"
      apiVersion: "client.authentication.k8s.io/v1"
      interactiveMode: Never
```

Input and output formats

The executed command prints an `ExecCredential` object to `stdout`. `k8s.io/client-go` authenticates against the Kubernetes API using the returned credentials in the `status`. The executed command is passed an `ExecCredential` object as input via the `KUBERNETES_EXEC_INFO` environment variable. This input contains helpful information like the expected API version of the returned `ExecCredential` object and whether or not the plugin can use `stdin` to interact with the user.

When run from an interactive session (i.e., a terminal), `stdin` can be exposed directly to the plugin. Plugins should use the `spec.interactive` field of the input `ExecCredential` object from the `KUBERNETES_EXEC_INFO` environment variable in order to determine if `stdin` has been provided. A plugin's `stdin` requirements (i.e., whether `stdin` is optional, strictly required, or never used in order for the plugin to run successfully) is declared via the `user.exec.interactiveMode` field in the kubeconfig (see table below for valid values). The `user.exec.interactiveMode` field is optional in `client.authentication.k8s.io/v1beta1` and required in `client.authentication.k8s.io/v1`.

`interactiveMode` Value	Meaning
`Never`	This exec plugin never needs to use standard input, and therefore the exec plugin will be run regardless of whether standard input is available for user input.
`IfAvailable`	This exec plugin would like to use standard input if it is available, but can still operate if standard input is not available. Therefore, the exec plugin will be run regardless of whether stdin is available for user input. If standard input is available for user input, then it will be provided to this exec plugin.
`Always`	This exec plugin requires standard input in order to run, and therefore the exec plugin will only be run if standard input is available for user input. If standard input is not available for user input, then the exec plugin will not be run and an error will be returned by the exec plugin runner.

To use bearer token credentials, the plugin returns a token in the status of the `ExecCredential`

- client.authentication.k8s.io/v1
- client.authentication.k8s.io/v1beta1

```
{
  "apiVersion": "client.authentication.k8s.io/v1",
  "kind": "ExecCredential",
  "status": {
    "token": "my-bearer-token"
  }
}
```

```
{
  "apiVersion": "client.authentication.k8s.io/v1beta1",
  "kind": "ExecCredential",
  "status": {
    "token": "my-bearer-token"
  }
}
```

Alternatively, a PEM-encoded client certificate and key can be returned to use TLS client auth. If the plugin returns a different certificate and key on a subsequent call, `k8s.io/client-go` will close existing connections with the server to force a new TLS handshake.

If specified, `clientKeyData` and `clientCertificateData` must both must be present.

`clientCertificateData` may contain additional intermediate certificates to send to the server.

- client.authentication.k8s.io/v1
- client.authentication.k8s.io/v1beta1

```
{
  "apiVersion": "client.authentication.k8s.io/v1",
  "kind": "ExecCredential",
  "status": {
    "clientCertificateData": "-----BEGIN CERTIFICATE-----\n...\n-----END CERTIFICATE-----",
    "clientKeyData": "-----BEGIN RSA PRIVATE KEY-----\n...\n-----END RSA PRIVATE KEY-----"
  }
}
```

```
{
  "apiVersion": "client.authentication.k8s.io/v1beta1",
  "kind": "ExecCredential",
  "status": {
    "clientCertificateData": "-----BEGIN CERTIFICATE-----\n...\n-----END CERTIFICATE-----",
    "clientKeyData": "-----BEGIN RSA PRIVATE KEY-----\n...\n-----END RSA PRIVATE KEY-----"
  }
}
```

Optionally, the response can include the expiry of the credential formatted as a RFC 3339 timestamp.

Presence or absence of an expiry has the following impact:

- If an expiry is included, the bearer token and TLS credentials are cached until the expiry time is reached, or if the server responds with a 401 HTTP status code, or when the process exits.
- If an expiry is omitted, the bearer token and TLS credentials are cached until the server responds with a 401 HTTP status code or until the process exits.

- client.authentication.k8s.io/v1
- client.authentication.k8s.io/v1beta1

```
{
  "apiVersion": "client.authentication.k8s.io/v1",
  "kind": "ExecCredential",
  "status": {
    "token": "my-bearer-token",
    "expirationTimestamp": "2018-03-05T17:30:20-08:00"
  }
}
```

```
{
  "apiVersion": "client.authentication.k8s.io/v1beta1",
  "kind": "ExecCredential",
  "status": {
    "token": "my-bearer-token",
    "expirationTimestamp": "2018-03-05T17:30:20-08:00"
  }
}
```

To enable the exec plugin to obtain cluster-specific information, set `provideClusterInfo` on the `user.exec` field in the kubeconfig. The plugin will then be supplied this cluster-specific information in the `KUBERNETES_EXEC_INFO` environment variable. Information from this environment variable can be used to perform cluster-specific credential acquisition logic. The following `ExecCredential` manifest describes a cluster information sample.

- client.authentication.k8s.io/v1
- client.authentication.k8s.io/v1beta1

```
{
  "apiVersion": "client.authentication.k8s.io/v1",
  "kind": "ExecCredential",
  "spec": {
    "cluster": {
      "server": "https://172.17.4.100:6443",
      "certificate-authority-data": "LS0t...",
      "config": {
        "arbitrary": "config",
        "this": "can be provided via the KUBERNETES_EXEC_INFO environment variable upon setting provideClusterInfo",
        "you": ["can", "put", "anything", "here"]
      }
    },
    "interactive": true
  }
}
```

```
{
  "apiVersion": "client.authentication.k8s.io/v1beta1",
  "kind": "ExecCredential",
  "spec": {
    "cluster": {
      "server": "https://172.17.4.100:6443",
      "certificate-authority-data": "LS0t...",
      "config": {
        "arbitrary": "config",
        "this": "can be provided via the KUBERNETES_EXEC_INFO environment variable upon setting provideClusterInfo",
        "you": ["can", "put", "anything", "here"]
      }
    },
    "interactive": true
  }
}
```

API access to authentication information for a client

FEATURE STATE: `Kubernetes v1.28 [stable]`

If your cluster has the API enabled, you can use the SelfSubjectReview API to find out how your Kubernetes cluster maps your authentication information to identify you as a client. This works whether you are authenticating as a user (typically representing a real person) or as a ServiceAccount.

SelfSubjectReview objects do not have any configurable fields. On receiving a request, the Kubernetes API server fills the status with the user attributes and returns it to the user.

Request example (the body would be a SelfSubjectReview):

```
POST /apis/authentication.k8s.io/v1/selfsubjectreviews
```

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "SelfSubjectReview"
}
```

Response example:

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "SelfSubjectReview",
  "status": {
    "userInfo": {
      "name": "jane.doe",
      "uid": "b6c7cfd4-f166-11ec-8ea0-0242ac120002",
      "groups": [
        "viewers",
        "editors",
        "system:authenticated"
      ],
      "extra": {
        "provider_id": ["token.company.example"]
      }
    }
  }
}
```

For convenience, the `kubectl auth whoami` command is present. Executing this command will produce the following output (yet different user attributes will be shown):

- Simple output example

```
ATTRIBUTE          VALUE
Username           jane.doe
Groups             [system:authenticated]
```

- Complex example including extra attributes

```
ATTRIBUTE         VALUE
Username          jane.doe
UID               b79dbf30-0c6a-11ed-861d-0242ac120002
Groups            [students teachers system:authenticated]
Extra: skills     [reading learning]
Extra: subjects   [math sports]
```

By providing the output flag, it is also possible to print the JSON or YAML representation of the result:

- JSON
- YAML

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "SelfSubjectReview",
  "status": {
    "userInfo": {
      "username": "jane.doe",
      "uid": "b79dbf30-0c6a-11ed-861d-0242ac120002",
      "groups": [
        "students",
        "teachers",
        "system:authenticated"
      ],
      "extra": {
        "skills": [
          "reading",
          "learning"
        ],
        "subjects": [
          "math",
          "sports"
        ]
      }
    }
  }
}
```

```
apiVersion: authentication.k8s.io/v1
kind: SelfSubjectReviewstatus:  userInfo:    username: jane.doe    uid: b79dbf30-0c6a-11ed-861d-0242ac120002    groups:    - stude
```

This feature is extremely useful when a complicated authentication flow is used in a Kubernetes cluster, for example, if you use webhook token authentication or authenticating proxy.

Note:

The Kubernetes API server fills the `userInfo` after all authentication mechanisms are applied, including impersonation. If you, or an authentication proxy, make a SelfSubjectReview using impersonation, you see the user details and properties for the user that was impersonated.

By default, all authenticated users can create `SelfSubjectReview` objects when the `APISelfSubjectReview` feature is enabled. It is allowed by the `system:basic-user` cluster role.

Note:

You can only make `SelfSubjectReview` requests if:

- the `APISelfSubjectReview` feature gate is enabled for your cluster (not needed for Kubernetes 1.34, but older Kubernetes versions might not offer this feature gate, or might default it to be off)
- (if you are running a version of Kubernetes older than v1.28) the API server for your cluster has the `authentication.k8s.io/v1alpha1` or `authentication.k8s.io/v1beta1` API group enabled.

What's next

- To learn about issuing certificates for users, read Issue a Certificate for a Kubernetes API Client Using A CertificateSigningRequest
- Read the client authentication reference (v1)
- Read the client authentication reference (v1beta1)

Certificates and Certificate Signing Requests

Kubernetes certificate and trust bundle APIs enable automation of X.509 credential provisioning by providing a programmatic interface for clients of the Kubernetes API to request and obtain X.509 certificates from a Certificate Authority (CA).

There is also experimental (alpha) support for distributing trust bundles.

Certificate signing requests

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

A CertificateSigningRequest (CSR) resource is used to request that a certificate be signed by a denoted signer, after which the request may be approved or denied before finally being signed.

Request signing process

The CertificateSigningRequest resource type allows a client to ask for an X.509 certificate be issued, based on a signing request. The CertificateSigningRequest object includes a PEM-encoded PKCS#10 signing request in the `spec.request` field. The CertificateSigningRequest denotes the signer (the recipient that the request is being made to) using the `spec.signerName` field. Note that `spec.signerName` is a required key after API version `certificates.k8s.io/v1`. In Kubernetes v1.22 and later, clients may optionally set the `spec.expirationSeconds` field to request a particular lifetime for the issued certificate. The minimum valid value for this field is `600`, i.e. ten minutes.

Once created, a CertificateSigningRequest must be approved before it can be signed. Depending on the signer selected, a CertificateSigningRequest may be automatically approved by a controller. Otherwise, a CertificateSigningRequest must be manually approved either via the REST API (or client-go) or by running `kubectl certificate approve`. Likewise, a CertificateSigningRequest may also be denied, which tells the configured signer that it must not sign the request.

For certificates that have been approved, the next step is signing. The relevant signing controller first validates that the signing conditions are met and then creates a certificate. The signing controller then updates the CertificateSigningRequest, storing the new certificate into the `status.certificate` field of the existing CertificateSigningRequest object. The `status.certificate` field is either empty or contains a X.509 certificate, encoded in PEM format. The CertificateSigningRequest `status.certificate` field is empty until the signer does this.

Once the `status.certificate` field has been populated, the request has been completed and clients can now fetch the signed certificate PEM data from the CertificateSigningRequest resource. The signers can instead deny certificate signing if the approval conditions are not met.

In order to reduce the number of old CertificateSigningRequest resources left in a cluster, a garbage collection controller runs periodically. The garbage collection removes CertificateSigningRequests that have not changed state for some duration:

- Approved requests: automatically deleted after 1 hour
- Denied requests: automatically deleted after 1 hour
- Failed requests: automatically deleted after 1 hour
- Pending requests: automatically deleted after 24 hours
- All requests: automatically deleted after the issued certificate has expired

Certificate signing authorization

To allow creating a CertificateSigningRequest and retrieving any CertificateSigningRequest:

- Verbs: `create`, `get`, `list`, `watch`, group: `certificates.k8s.io`, resource: `certificatesigningrequests`

For example:

access/certificate-signing-request/clusterrole-create.yaml Copy access/certificate-signing-request/clusterrole-create.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: csr-creatorrules:- apiGroups:  - certificates.k8s.io  resources:  - certificatesigningrequests
```

To allow approving a CertificateSigningRequest:

- Verbs: `get`, `list`, `watch`, group: `certificates.k8s.io`, resource: `certificatesigningrequests`
- Verbs: `update`, group: `certificates.k8s.io`, resource: `certificatesigningrequests/approval`
- Verbs: `approve`, group: `certificates.k8s.io`, resource: `signers`, resourceName: `<signerNameDomain>/<signerNamePath>` or `<signerNameDomain>/*`

For example:

access/certificate-signing-request/clusterrole-approve.yaml Copy access/certificate-signing-request/clusterrole-approve.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: csr-approverrules:- apiGroups:  - certificates.k8s.io  resources:  - certificatesigningrequests
```

To allow signing a CertificateSigningRequest:

- Verbs: `get`, `list`, `watch`, group: `certificates.k8s.io`, resource: `certificatesigningrequests`
- Verbs: `update`, group: `certificates.k8s.io`, resource: `certificatesigningrequests/status`
- Verbs: `sign`, group: `certificates.k8s.io`, resource: `signers`, resourceName: `<signerNameDomain>/<signerNamePath>` or `<signerNameDomain>/*`

access/certificate-signing-request/clusterrole-sign.yaml Copy access/certificate-signing-request/clusterrole-sign.yaml to clipboard

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:  name: csr-signerrules:- apiGroups:  - certificates.k8s.io  resources:  - certificatesigningrequests  v
```

Signers

Signers abstractly represent the entity or entities that might sign, or have signed, a security certificate.

Any signer that is made available for outside a particular cluster should provide information about how the signer works, so that consumers can understand what that means for CertificateSigningRequests and (if enabled) ClusterTrustBundles. This includes:

1. **Trust distribution**: how trust anchors (CA certificates or certificate bundles) are distributed.
2. **Permitted subjects**: any restrictions on and behavior when a disallowed subject is requested.
3. **Permitted x509 extensions**: including IP subjectAltNames, DNS subjectAltNames, Email subjectAltNames, URI subjectAltNames etc, and behavior when a disallowed extension is requested.
4. **Permitted key usages / extended key usages**: any restrictions on and behavior when usages different than the signer-determined usages are specified in the CSR.
5. **Expiration/certificate lifetime**: whether it is fixed by the signer, configurable by the admin, determined by the CSR `spec.expirationSeconds` field, etc and the behavior when the signer-determined expiration is different from the CSR `spec.expirationSeconds` field.

6. **CA bit allowed/disallowed**: and behavior if a CSR contains a request for a CA certificate when the signer does not permit it.

Commonly, the `status.certificate` field of a CertificateSigningRequest contains a single PEM-encoded X.509 certificate once the CSR is approved and the certificate is issued. Some signers store multiple certificates into the `status.certificate` field. In that case, the documentation for the signer should specify the meaning of additional certificates; for example, this might be the certificate plus intermediates to be presented during TLS handshakes.

If you want to make the *trust anchor* (root certificate) available, this should be done separately from a CertificateSigningRequest and its `status.certificate` field. For example, you could use a ClusterTrustBundle.

The PKCS#10 signing request format does not have a standard mechanism to specify a certificate expiration or lifetime. The expiration or lifetime therefore has to be set through the `spec.expirationSeconds` field of the CSR object. The built-in signers use the `ClusterSigningDuration` configuration option, which defaults to 1 year, (the `--cluster-signing-duration` command-line flag of the kube-controller-manager) as the default when no `spec.expirationSeconds` is specified. When `spec.expirationSeconds` is specified, the minimum of `spec.expirationSeconds` and `ClusterSigningDuration` is used.

Note:

The `spec.expirationSeconds` field was added in Kubernetes v1.22. Earlier versions of Kubernetes do not honor this field. Kubernetes API servers prior to v1.22 will silently drop this field when the object is created.

Kubernetes signers

Kubernetes provides built-in signers that each have a well-known `signerName`:

1. `kubernetes.io/kube-apiserver-client`: signs certificates that will be honored as client certificates by the API server. Never auto-approved by kube-controller-manager.
    1. Trust distribution: signed certificates must be honored as client certificates by the API server. The CA bundle is not distributed by any other means.
    2. Permitted subjects - no subject restrictions, but approvers and signers may choose not to approve or sign. Certain subjects like cluster-admin level users or groups vary between distributions and installations, but deserve additional scrutiny before approval and signing. The `CertificateSubjectRestriction` admission plugin is enabled by default to restrict `system:masters`, but it is often not the only cluster-admin subject in a cluster.
    3. Permitted x509 extensions - honors subjectAltName and key usage extensions and discards other extensions.
    4. Permitted key usages - must include `["client auth"]`. Must not include key usages beyond `["digital signature", "key encipherment", "client auth"]`.
    5. Expiration/certificate lifetime - for the kube-controller-manager implementation of this signer, set to the minimum of the `--cluster-signing-duration` option or, if specified, the `spec.expirationSeconds` field of the CSR object.
    6. CA bit allowed/disallowed - not allowed.
2. `kubernetes.io/kube-apiserver-client-kubelet`: signs client certificates that will be honored as client certificates by the API server. May be auto-approved by kube-controller-manager.
    1. Trust distribution: signed certificates must be honored as client certificates by the API server. The CA bundle is not distributed by any other means.
    2. Permitted subjects - organizations are exactly `["system:nodes"]`, common name is `"system:node:${NODE_NAME}"`.
    3. Permitted x509 extensions - honors key usage extensions, forbids subjectAltName extensions and drops other extensions.
    4. Permitted key usages - `["key encipherment", "digital signature", "client auth"]` or `["digital signature", "client auth"]`.
    5. Expiration/certificate lifetime - for the kube-controller-manager implementation of this signer, set to the minimum of the `--cluster-signing-duration` option or, if specified, the `spec.expirationSeconds` field of the CSR object.
    6. CA bit allowed/disallowed - not allowed.
3. `kubernetes.io/kubelet-serving`: signs serving certificates that are honored as a valid kubelet serving certificate by the API server, but has no other guarantees. Never auto-approved by kube-controller-manager.
    1. Trust distribution: signed certificates must be honored by the API server as valid to terminate connections to a kubelet. The CA bundle is not distributed by any other means.
    2. Permitted subjects - organizations are exactly `["system:nodes"]`, common name is `"system:node:${NODE_NAME}"`.
    3. Permitted x509 extensions - honors key usage and DNSName/IPAddress subjectAltName extensions, forbids EmailAddress and URI subjectAltName extensions, drops other extensions. At least one DNS or IP subjectAltName must be present.
    4. Permitted key usages - `["key encipherment", "digital signature", "server auth"]` or `["digital signature", "server auth"]`.
    5. Expiration/certificate lifetime - for the kube-controller-manager implementation of this signer, set to the minimum of the `--cluster-signing-duration` option or, if specified, the `spec.expirationSeconds` field of the CSR object.
    6. CA bit allowed/disallowed - not allowed.
4. `kubernetes.io/legacy-unknown`: has no guarantees for trust at all. Some third-party distributions of Kubernetes may honor client certificates signed by it. The stable CertificateSigningRequest API (version `certificates.k8s.io/v1` and later) does not allow to set the `signerName` as `kubernetes.io/legacy-unknown`. Never auto-approved by kube-controller-manager.
    1. Trust distribution: None. There is no standard trust or distribution for this signer in a Kubernetes cluster.
    2. Permitted subjects - any
    3. Permitted x509 extensions - honors subjectAltName and key usage extensions and discards other extensions.
    4. Permitted key usages - any
    5. Expiration/certificate lifetime - for the kube-controller-manager implementation of this signer, set to the minimum of the `--cluster-signing-duration` option or, if specified, the `spec.expirationSeconds` field of the CSR object.
    6. CA bit allowed/disallowed - not allowed.

The kube-controller-manager implements control plane signing for each of the built in signers. Failures for all of these are only reported in kube-controller-manager logs.

Note:

The `spec.expirationSeconds` field was added in Kubernetes v1.22. Earlier versions of Kubernetes do not honor this field. Kubernetes API servers prior to v1.22 will silently drop this field when the object is created.

Distribution of trust happens out of band for these signers. Any trust outside of those described above are strictly coincidental. For instance, some distributions may honor `kubernetes.io/legacy-unknown` as client certificates for the kube-apiserver, but this is not a standard. None of these usages are related to ServiceAccount token secrets `.data[ca.crt]` in any way. That CA bundle is only guaranteed to verify a connection to the API server using the default service (`kubernetes.default.svc`).

Custom signers

You can also introduce your own custom signer, which should have a similar prefixed name but using your own domain name. For example, if you represent an open source project that uses the domain `open-fictional.example` then you might use `issuer.open-fictional.example/service-mesh` as a signer name.

A custom signer uses the Kubernetes API to issue a certificate. See API-based signers.

Signing

Control plane signer

The Kubernetes control plane implements each of the Kubernetes signers, as part of the kube-controller-manager.

Note:

Prior to Kubernetes v1.18, the kube-controller-manager would sign any CSRs that were marked as approved.

Note:

The `spec.expirationSeconds` field was added in Kubernetes v1.22. Earlier versions of Kubernetes do not honor this field. Kubernetes API servers prior to v1.22 will silently drop this field when the object is created.

API-based signers

Users of the REST API can sign CSRs by submitting an UPDATE request to the `status` subresource of the CSR to be signed.

As part of this request, the `status.certificate` field should be set to contain the signed certificate. This field contains one or more PEM-encoded certificates.

All PEM blocks must have the "CERTIFICATE" label, contain no headers, and the encoded data must be a BER-encoded ASN.1 Certificate structure as described in section 4 of RFC5280.

Example certificate content:

```
-----BEGIN CERTIFICATE-----
MIIDgjCCAmqgAwIBAgIUC1N1EJ4Qnsd322BhDPRwmg3b/oAwDQYJKoZIhvcNAQEL
BQAwXDELMAkGA1UEBhMCeHgxCjAIBgNVBAgMAXgxCjAIBgNVBAcMAXgxCjAIBgNV
BAoMAXgxCjAIBgNVBAsMAXgxCzAJBgNVBAMMAmNhMRAwDgYJKoZIhvcNAQkBFgF4
MB4XDTIwMDcwNjIyMDcwMFoXDTI1MDcwNTIyMDcwMFowNzEVMBMGA1UEChMMc3lz
dGVtOm5vZGVzMR4wHAYDVQQDExVzeXN0ZW06bm9kZToxMjcuMC4wLjEwggEiMA0G
CSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDne5X2eQ1JcLZkKvhzCR4Hxl9+ZmU3
+e1zfOywLdoQxrPi+o4hVsUH3q0y52BMa7u1yehHDRSaq9u62cmi5ekgXhXHzGmm
kmW5n0itRECv3SFsSm2DSghRKf0mm6iTYHWDHzUXKdm9lPPWoSOxoR5oqOsm3JEh
Q7Et13wrvTJqBMJolGTwQuF+HYOku0NF/DLqbZIcpI08yQKyrBgYz2uO51/oNp8a
sTCsV4OUfyHhx2BBLUo4g4SptHFySTBwlpRWBnSjZPOhmN74JcpTLB4J5f4iEeA7
2QytZfADckG4wVkhH3C2EJUmRtFIBVirwDn39GXkSGlnvnMgF3uLZ6zNAgMBAAGj
YTBfMA4GA1UdDwEB/wQEAwIFoDATBgNVHSUEDDAKBggrBgEFBQcDAjAMBgNVHRMB
Af8EAjAAMB0GA1UdDgQWBBTREl2hW54lkQBDeVCcd2f2VSlB1DALBgNVHREEBDAC
ggAwDQYJKoZIhvcNAQELBQADggEBABpZjuIKTq8pCaX8dMEGPWtAykgLsTcD2jYr
L0/TCrqmuaaliUa42jQTt2OVsVP/L8ofFunj/KjpQU0bvKJPLMRKtmxbhXuQCQi1
qCRkp8o93mHvEz3mTUN+D1cfQ2fpsBENLnpS0F4G/JyY2Vrh19/X8+mImMEK5eOy
o0BMby7byUj98WmcUvNCiXbC6F45QTmkwEhMqWns0JZQY+/XeDhEcg+lJvz9Eyo2
aGgPsye1o3DpyXnyfJWAWMhOz7cikS5X2adesbgI86PhEHBXPIJ1v13ZdfCExmdd
M1fLPhLyR54fGaY+7/X8P9AZzPefAkwizeXwe9ii6/a08vWoiE4=
-----END CERTIFICATE-----
```

Non-PEM content may appear before or after the CERTIFICATE PEM blocks and is unvalidated, to allow for explanatory text as described in section 5.2 of RFC7468.

When encoded in JSON or YAML, this field is base-64 encoded. A CertificateSigningRequest containing the example certificate above would look like this:

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest...status:  certificate: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JS..."
```

Approval or rejection

Before a signer issues a certificate based on a CertificateSigningRequest, the signer typically checks that the issuance for that CSR has been *approved*.

Control plane automated approval

The kube-controller-manager ships with a built-in approver for certificates with a signerName of `kubernetes.io/kube-apiserver-client-kubelet` that delegates various permissions on CSRs for node credentials to authorization. The kube-controller-manager POSTs SubjectAccessReview resources to the API server in order to check authorization for certificate approval.

Approval or rejection using `kubectl`

A Kubernetes administrator (with appropriate permissions) can manually approve (or deny) CertificateSigningRequests by using the `kubectl certificate approve` and `kubectl certificate deny` commands.

To approve a CSR with kubectl:

```
kubectl certificate approve <certificate-signing-request-name>
```

Likewise, to deny a CSR:

```
kubectl certificate deny <certificate-signing-request-name>
```

Approval or rejection using the Kubernetes API

Users of the REST API can approve CSRs by submitting an UPDATE request to the `approval` subresource of the CSR to be approved. For example, you could write an operator that watches for a particular kind of CSR and then sends an UPDATE to approve them.

When you make an approval or rejection request, set either the `Approved` or `Denied` status condition based on the state you determine:

For `Approved` CSRs:

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest...status:  conditions:  - lastUpdateTime: "2020-02-08T11:37:35Z"    lastTransitionTime: "2020-02-0
```

For `Denied` CSRs:

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest...status:  conditions:  - lastUpdateTime: "2020-02-08T11:37:35Z"    lastTransitionTime: "2020-02-0
```

It's usual to set `status.conditions.reason` to a machine-friendly reason code using TitleCase; this is a convention but you can set it to anything you like. If you want to add a note for human consumption, use the `status.conditions.message` field.

PodCertificateRequests

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

Note:

In Kubernetes 1.34, you must enable support for Pod Certificates using the `PodCertificateRequest` feature gate and the `--runtime-config=certificates.k8s.io/v1alpha1/podcertificaterequests=true` kube-apiserver flag.

PodCertificateRequests are API objects tailored to provisioning certificates to workloads running as Pods within a cluster. The user typically does not interact with PodCertificateRequests directly, but uses podCertificate projected volume sources, which are a `kubelet` feature that handles secure key provisioning and automatic certificate refresh. The application inside the pod only needs to know how to read the certificates from the filesystem.

PodCertificateRequests are similar to CertificateSigningRequests, but have a simpler format enabled by their narrower use case.

A PodCertificateRequest has the following spec fields:

- `signerName`: The signer to which this request is addressed.
- `podName` and `podUID`: The Pod that Kubelet is requesting a certificate for.
- `serviceAccountName` and `serviceAccountUID`: The ServiceAccount corresponding to the Pod.
- `nodeName` and `nodeUID`: The Node corresponding to the Pod.
- `maxExpirationSeconds`: The maximum lifetime that the workload author will accept for this certificate. Defaults to 24 hours if not specified.
- `pkixPublicKey`: The public key for which the certificate should be issued.
- `proofOfPossession`: A signature demonstrating that the requester controls the private key corresponding to `pkixPublicKey`.

Nodes automatically receive permissions to create PodCertificateRequests and read PodCertificateRequests related to them (as determined by the `spec.nodeName` field). The `NodeRestriction` admission plugin, if enabled, ensures that nodes can only create PodCertificateRequests that correspond to a real pod that is currently running on the node.

After creation, the `spec` of a PodCertificateRequest is immutable.

Unlike CSRs, PodCertificateRequests do not have an approval phase. Once the PodCertificateRequest is created, the signer's controller directly decides to issue or deny the request. It also has the option to mark the request as failed, if it encountered a permanent error when attempting to issue the request.

To take any of these actions, the signing controller needs to have the appropriate permissions on both the PodCertificateRequest type, as well as on the signer name:

- Verbs: **update**, group: `certificates.k8s.io`, resource: `podcertificaterequests/status`
- Verbs: **sign**, group: `certificates.k8s.io`, resource: `signers`, resourceName: `<signerNameDomain>/<signerNamePath>` or `<signerNameDomain>/*`

The signing controller is free to consider other information beyond what's contained in the request, but it can rely on the information in the request to be accurate. For example, the signing controller might load the Pod and read annotations set on it, or perform a SubjectAccessReview on the ServiceAccount.

To issue a certificate in response to a request, the signing controller:

- Adds an `Issued` condition to `status.conditions`.
- Puts the issued certificate in `status.certificateChain`
- Puts the `NotBefore` and `NotAfter` fields of the certificate in the `status.notBefore` and `status.notAfter` fields — these fields are denormalized into the Kubernetes API in order to aid debugging

- Suggests a time to begin attempting to refresh the certificate using `status.beginRefreshAt`.

To deny a request, the signing controller adds a "Denied" condition to `status.conditions[]`.

To mark a request failed, the signing controller adds a "Failed" condition to `status.conditions[]`.

All of these conditions are mutually-exclusive, and must have status "True". No other condition types are permitted on PodCertificateRequests. In addition, once any of these conditions are set, the `status` field becomes immutable.

Like all conditions, the `status.conditions[].reason` field is meant to contain a machine-readable code describing the condition in TitleCase. The `status.conditions[].message` field is meant for a free-form explanation for human consumption.

To ensure that terminal PodCertificateRequests do not build up in the cluster, a `kube-controller-manager` controller deletes all PodCertificateRequests older than 15 minutes. All certificate issuance flows are expected to complete within this 15-minute limit.

Cluster trust bundles

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: false)

Note:

In Kubernetes 1.34, you must enable the `ClusterTrustBundle` feature gate *and* the `certificates.k8s.io/v1alpha1` API group in order to use this API.

A ClusterTrustBundles is a cluster-scoped object for distributing X.509 trust anchors (root certificates) to workloads within the cluster. They're designed to work well with the signer concept from CertificateSigningRequests.

ClusterTrustBundles can be used in two modes: signer-linked and signer-unlinked.

Common properties and validation

All ClusterTrustBundle objects have strong validation on the contents of their `trustBundle` field. That field must contain one or more X.509 certificates, DER-serialized, each wrapped in a PEM `CERTIFICATE` block. The certificates must parse as valid X.509 certificates.

Esoteric PEM features like inter-block data and intra-block headers are either rejected during object validation, or can be ignored by consumers of the object. Additionally, consumers are allowed to reorder the certificates in the bundle with their own arbitrary but stable ordering.

ClusterTrustBundle objects should be considered world-readable within the cluster. If your cluster uses RBAC authorization, all ServiceAccounts have a default grant that allows them to **get**, **list**, and **watch** all ClusterTrustBundle objects. If you use your own authorization mechanism and you have enabled ClusterTrustBundles in your cluster, you should set up an equivalent rule to make these objects public within the cluster, so that they work as intended.

If you do not have permission to list cluster trust bundles by default in your cluster, you can impersonate a service account you have access to in order to see available ClusterTrustBundles:

```
kubectl get clustertrustbundles --as='system:serviceaccount:mynamespace:default'
```

Signer-linked ClusterTrustBundles

Signer-linked ClusterTrustBundles are associated with a *signer name*, like this:

```
apiVersion: certificates.k8s.io/v1alpha1
kind: ClusterTrustBundlemetadata:  name: example.com:mysigner:foospec:  signerName: example.com/mysigner  trustBundle: "<... PEM da
```

These ClusterTrustBundles are intended to be maintained by a signer-specific controller in the cluster, so they have several security features:

- To create or update a signer-linked ClusterTrustBundle, you must be permitted to **attest** on the signer (custom authorization verb `attest`, API group `certificates.k8s.io`; resource path `signers`). You can configure authorization for the specific resource name `<signerNameDomain>/<signerNamePath>` or match a pattern such as `<signerNameDomain>/*`.
- Signer-linked ClusterTrustBundles **must** be named with a prefix derived from their `spec.signerName` field. Slashes (`/`) are replaced with colons (`:`), and a final colon is appended. This is followed by an arbitrary name. For example, the signer `example.com/mysigner` can be linked to a ClusterTrustBundle `example.com:mysigner:<arbitrary-name>`.

Signer-linked ClusterTrustBundles will typically be consumed in workloads by a combination of a field selector on the signer name, and a separate label selector.

Signer-unlinked ClusterTrustBundles

Signer-unlinked ClusterTrustBundles have an empty `spec.signerName` field, like this:

```
apiVersion: certificates.k8s.io/v1alpha1
kind: ClusterTrustBundlemetadata:  name: foospec:  # no signerName specified, so the field is blank  trustBundle: "<... PEM data .
```

They are primarily intended for cluster configuration use cases. Each signer-unlinked ClusterTrustBundle is an independent object, in contrast to the customary grouping behavior of signer-linked ClusterTrustBundles.

Signer-unlinked ClusterTrustBundles have no `attest` verb requirement. Instead, you control access to them directly using the usual mechanisms, such as role-based access control.

To distinguish them from signer-linked ClusterTrustBundles, the names of signer-unlinked ClusterTrustBundles **must not** contain a colon (`:`).

Accessing ClusterTrustBundles from pods

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: false)

The contents of ClusterTrustBundles can be injected into the container filesystem, similar to ConfigMaps and Secrets. See the clusterTrustBundle projected volume source for more details.

What's next

- Read Manage TLS Certificates in a Cluster
- Read Issue a Certificate for a Kubernetes API Client Using A CertificateSigningRequest
- View the source code for the kube-controller-manager built in signer
- View the source code for the kube-controller-manager built in approver
- For details of X.509 itself, refer to RFC 5280 section 3.1
- For information on the syntax of PKCS#10 certificate signing requests, refer to RFC 2986
- Read about the ClusterTrustBundle API:
  - %!s()

Using ABAC Authorization

Attribute-based access control (ABAC) defines an access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together.

Policy File Format

To enable `ABAC` mode, specify `--authorization-policy-file=SOME_FILENAME` and `--authorization-mode=ABAC` on startup.

The file format is one JSON object per line. There should be no enclosing list or map, only one map per line.

Each line is a "policy object", where each such object is a map with the following properties:

- Versioning properties:
  - `apiVersion`, type string; valid values are "abac.authorization.kubernetes.io/v1beta1". Allows versioning and conversion of the policy format.
  - `kind`, type string: valid values are "Policy". Allows versioning and conversion of the policy format.
- `spec` property set to a map with the following properties:
  - Subject-matching properties:
    - `user`, type string; the user-string from `--token-auth-file`. If you specify `user`, it must match the username of the authenticated user.
    - `group`, type string; if you specify `group`, it must match one of the groups of the authenticated user. `system:authenticated` matches all authenticated requests. `system:unauthenticated` matches all unauthenticated requests.
  - Resource-matching properties:
    - `apiGroup`, type string; an API group.
      - Ex: `apps`, `networking.k8s.io`
      - Wildcard: `*` matches all API groups.
    - `namespace`, type string; a namespace.
      - Ex: `kube-system`
      - Wildcard: `*` matches all resource requests.
    - `resource`, type string; a resource type
      - Ex: `pods`, `deployments`
      - Wildcard: `*` matches all resource requests.
  - Non-resource-matching properties:
    - `nonResourcePath`, type string; non-resource request paths.
      - Ex: `/version` or `/apis`
      - Wildcard:
        - `*` matches all non-resource requests.
        - `/foo/*` matches all subpaths of `/foo/`.
  - `readonly`, type boolean, when true, means that the Resource-matching policy only applies to get, list, and watch operations, Non-resource-matching policy only applies to get operation.

Note:

An unset property is the same as a property set to the zero value for its type (e.g. empty string, 0, false). However, unset should be preferred for readability.

In the future, policies may be expressed in a JSON format, and managed via a REST interface.

Authorization Algorithm

A request has attributes which correspond to the properties of a policy object.

When a request is received, the attributes are determined. Unknown attributes are set to the zero value of its type (e.g. empty string, 0, false).

A property set to `"*"` will match any value of the corresponding attribute.

The tuple of attributes is checked for a match against every policy in the policy file. If at least one line matches the request attributes, then the request is authorized (but may fail later validation).

To permit any authenticated user to do something, write a policy with the group property set to `"system:authenticated"`.

To permit any unauthenticated user to do something, write a policy with the group property set to `"system:unauthenticated"`.

To permit a user to do anything, write a policy with the apiGroup, namespace, resource, and nonResourcePath properties set to `"*"`.

Kubectl

Kubectl uses the `/api` and `/apis` endpoints of apiserver to discover served resource types, and validates objects sent to the API by create/update operations using schema information located at `/openapi/v2`.

When using ABAC authorization, those special resources have to be explicitly exposed via the `nonResourcePath` property in a policy (see examples below):

- `/api`, `/api/*`, `/apis`, and `/apis/*` for API version negotiation.
- `/version` for retrieving the server version via `kubectl version`.
- `/swaggerapi/*` for create/update operations.

To inspect the HTTP calls involved in a specific kubectl operation you can turn up the verbosity:

```
kubectl --v=8 version
```

Examples

1. Alice can do anything to all resources:

   ```
   {"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user": "alice", "namespace": "*", "res
   ```

2. The kubelet can read any pods:

   ```
   {"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user": "kubelet", "namespace": "*", "r
   ```

3. The kubelet can read and write events:

   ```
   {"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user": "kubelet", "namespace": "*", "r
   ```

4. Bob can just read pods in namespace "projectCaribou":

   ```
   {"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user": "bob", "namespace": "projectCar
   ```

5. Anyone can make read-only requests to all non-resource paths:

   ```
   {"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"group": "system:authenticated", "readon
    {"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"group": "system:unauthenticated", "re
   ```

Complete file example

A quick note on service accounts

Every service account has a corresponding ABAC username, and that service account's username is generated according to the naming convention:

```
system:serviceaccount:<namespace>:<serviceaccountname>
```

Creating a new namespace leads to the creation of a new service account in the following format:

```
system:serviceaccount:<namespace>:default
```

For example, if you wanted to grant the default service account (in the `kube-system` namespace) full privilege to the API using ABAC, you would add this line to your policy file:

```
{"apiVersion":"abac.authorization.kubernetes.io/v1beta1","kind":"Policy","spec":{"user":"system:serviceaccount:kube-system:default
```

The apiserver will need to be restarted to pick up the new policy lines.

Using Node Authorization

Node authorization is a special-purpose authorization mode that specifically authorizes API requests made by kubelets.

Overview

The Node authorizer allows a kubelet to perform API operations. This includes:

Read operations:

- services
- endpoints
- nodes
- pods
- secrets, configmaps, persistent volume claims and persistent volumes related to pods bound to the kubelet's node

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

Kubelets are limited to reading their own Node objects, and only reading pods bound to their node.

Write operations:

- nodes and node status (enable the `NodeRestriction` admission plugin to limit a kubelet to modify its own node)
- pods and pod status (enable the `NodeRestriction` admission plugin to limit a kubelet to modify pods bound to itself)
- events

Auth-related operations:

- read/write access to the CertificateSigningRequests API for TLS bootstrapping
- the ability to create TokenReviews and SubjectAccessReviews for delegated authentication/authorization checks

In future releases, the node authorizer may add or remove permissions to ensure kubelets have the minimal set of permissions required to operate correctly.

In order to be authorized by the Node authorizer, kubelets must use a credential that identifies them as being in the `system:nodes` group, with a username of `system:node:<nodeName>`. This group and user name format match the identity created for each kubelet as part of kubelet TLS bootstrapping.

The value of `<nodeName>` **must** match precisely the name of the node as registered by the kubelet. By default, this is the host name as provided by `hostname`, or overridden via the kubelet option `--hostname-override`. However, when using the `--cloud-provider` kubelet option, the specific hostname may be determined by the cloud provider, ignoring the local `hostname` and the `--hostname-override` option. For specifics about how the kubelet determines the hostname, see the kubelet options reference.

To enable the Node authorizer, start the API server with the `--authorization-config` flag set to a file that includes the `Node` authorizer; for example:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AuthorizationConfigurationauthorizers:  ...  - type: Node  ...
```

Or, start the API server with the `--authorization-mode` flag set to a comma-separated list that includes `Node`; for example:

```
kube-apiserver --authorization-mode=...,Node --other-options --more-options
```

To limit the API objects kubelets are able to write, enable the NodeRestriction admission plugin by starting the apiserver with `--enable-admission-plugins=...,NodeRestriction,...`

Migration considerations

Kubelets outside the `system:nodes` group

Kubelets outside the `system:nodes` group would not be authorized by the `Node` authorization mode, and would need to continue to be authorized via whatever mechanism currently authorizes them. The node admission plugin would not restrict requests from these kubelets.

Kubelets with undifferentiated usernames

In some deployments, kubelets have credentials that place them in the `system:nodes` group, but do not identify the particular node they are associated with, because they do not have a username in the `system:node:...` format. These kubelets would not be authorized by the `Node` authorization mode, and would need to continue to be authorized via whatever mechanism currently authorizes them.

The `NodeRestriction` admission plugin would ignore requests from these kubelets, since the default node identifier implementation would not consider that a node identity.

Authenticating with Bootstrap Tokens

FEATURE STATE: `Kubernetes v1.18 [stable]`

Bootstrap tokens are a simple bearer token that is meant to be used when creating new clusters or joining new nodes to an existing cluster. It was built to support kubeadm, but can be used in other contexts for users that wish to start clusters without `kubeadm`. It is also built to work, via RBAC policy, with the kubelet TLS Bootstrapping system.

Bootstrap Tokens Overview

Bootstrap Tokens are defined with a specific type (`bootstrap.kubernetes.io/token`) of secrets that lives in the `kube-system` namespace. These Secrets are then read by the Bootstrap Authenticator in the API Server. Expired tokens are removed with the TokenCleaner controller in the Controller Manager. The tokens are also used to create a signature for a specific ConfigMap used in a "discovery" process through a BootstrapSigner controller.

Token Format

Bootstrap Tokens take the form of `abcdef.0123456789abcdef`. More formally, they must match the regular expression `[a-z0-9]{6}\.[a-z0-9]{16}`.

The first part of the token is the "Token ID" and is considered public information. It is used when referring to a token without leaking the secret part used for authentication. The second part is the "Token Secret" and should only be shared with trusted parties.

Enabling Bootstrap Token Authentication

The Bootstrap Token authenticator can be enabled using the following flag on the API server:

```
--enable-bootstrap-token-auth
```

When enabled, bootstrapping tokens can be used as bearer token credentials to authenticate requests against the API server.

```
Authorization: Bearer 07401b.f395accd246ae52d
```

Tokens authenticate as the username `system:bootstrap:<token id>` and are members of the group `system:bootstrappers`. Additional groups may be specified in the token's Secret.

Expired tokens can be deleted automatically by enabling the `tokencleaner` controller on the controller manager.

```
--controllers=*,tokencleaner
```

Bootstrap Token Secret Format

Each valid token is backed by a secret in the `kube-system` namespace. You can find the full design doc here.

Here is what the secret looks like.

```
apiVersion: v1
kind: Secretmetadata:  # Name MUST be of form "bootstrap-token-<token id>"  name: bootstrap-token-07401b  namespace: kube-system# 
```

The type of the secret must be `bootstrap.kubernetes.io/token` and the name must be `bootstrap-token-<token id>`. It must also exist in the `kube-system` namespace.

The `usage-bootstrap-*` members indicate what this secret is intended to be used for. A value must be set to `true` to be enabled.

- `usage-bootstrap-authentication` indicates that the token can be used to authenticate to the API server as a bearer token.
- `usage-bootstrap-signing` indicates that the token may be used to sign the `cluster-info` ConfigMap as described below.

The `expiration` field controls the expiry of the token. Expired tokens are rejected when used for authentication and ignored during ConfigMap signing. The expiry value is encoded as an absolute UTC time using RFC3339. Enable the `tokencleaner` controller to automatically delete expired tokens.

Token Management with kubeadm

You can use the `kubeadm` tool to manage tokens on a running cluster. See the kubeadm token docs for details.

ConfigMap Signing

In addition to authentication, the tokens can be used to sign a ConfigMap. This is used early in a cluster bootstrap process before the client trusts the API server. The signed ConfigMap can be authenticated by the shared token.

Enable ConfigMap signing by enabling the `bootstrapsigner` controller on the Controller Manager.

```
--controllers=*,bootstrapsigner
```

The ConfigMap that is signed is `cluster-info` in the `kube-public` namespace. The typical flow is that a client reads this ConfigMap while unauthenticated and ignoring TLS errors. It then validates the payload of the ConfigMap by looking at a signature embedded in the ConfigMap.

The ConfigMap may look like this:

```
apiVersion: v1
kind: ConfigMapmetadata:  name: cluster-info  namespace: kube-publicdata:  jws-kubeconfig-07401b: eyJhbGciOiJIUzI1NiIsImtpZCI6IjA3
```

The `kubeconfig` member of the ConfigMap is a config file with only the cluster information filled out. The key thing being communicated here is the `certificate-authority-data`. This may be expanded in the future.

The signature is a JWS signature using the "detached" mode. To validate the signature, the user should encode the `kubeconfig` payload according to JWS rules (base64 encoded while discarding any trailing `=`). That encoded payload is then used to form a whole JWS by inserting it between the 2 dots. You can verify the JWS using the `HS256` scheme (HMAC-SHA256) with the full token (e.g. `07401b.f395accd246ae52d`) as the shared secret. Users *must* verify that HS256 is used.

Warning:

Any party with a bootstrapping token can create a valid signature for that token. When using ConfigMap signing it's discouraged to share the same token with many clients, since a compromised client can potentially man-in-the middle another client relying on the signature to bootstrap TLS trust.

Consult the kubeadm implementation details section for more information.

Mutating Admission Policy

FEATURE STATE: `Kubernetes v1.34 [beta]`

This page provides an overview of *MutatingAdmissionPolicies*. MutatingAdmissionPolicies allow you change what happens when someone writes a change to the Kubernetes API. If you want to use declarative policies just to prevent a particular kind of change to resources (for example: protecting platform namespaces from deletion), ValidatingAdmissionPolicy is a simpler and more effective alternative.

To use the feature, enable the `MutatingAdmissionPolicy` feature gate (which is off by default) and set `--runtime-config=admissionregistration.k8s.io/v1beta1=true` on the kube-apiserver.

What are MutatingAdmissionPolicies?

Mutating admission policies offer a declarative, in-process alternative to mutating admission webhooks.

Mutating admission policies use the Common Expression Language (CEL) to declare mutations to resources. Mutations can be defined either with an *apply configuration* that is merged using the server side apply merge strategy, or a JSON patch.

Mutating admission policies are highly configurable, enabling policy authors to define policies that can be parameterized and scoped to resources as needed by cluster administrators.

What resources make a policy

A policy is generally made up of three resources:

- The MutatingAdmissionPolicy describes the abstract logic of a policy (think: "this policy sets a particular label to a particular value").
- A *parameter resource* provides information to a MutatingAdmissionPolicy to make it a concrete statement (think "set the `owner` label to something like `company.example.com`"). Parameter resources refer to Kubernetes resources, available in the Kubernetes API. They can be built-in types or extensions, such as a CustomResourceDefinition (CRD). For example, you can use a ConfigMap as a parameter.
- A MutatingAdmissionPolicyBinding links the above (MutatingAdmissionPolicy and parameter) resources together and provides scoping. If you only want to set an `owner` label for `Pods`, and not other API kinds, the binding is where you specify this mutation.

At least a MutatingAdmissionPolicy and a corresponding MutatingAdmissionPolicyBinding must be defined for a policy to have an effect.

If a MutatingAdmissionPolicy does not need to be configured via parameters, simply leave `spec.paramKind` in MutatingAdmissionPolicy not specified.

Getting Started with MutatingAdmissionPolicies

Mutating admission policy is part of the cluster control-plane. You should write and deploy them with great caution. The following describes how to quickly experiment with Mutating admission policy.

Create a MutatingAdmissionPolicy

The following is an example of a MutatingAdmissionPolicy. This policy mutates newly created Pods to have a sidecar container if it does not exist.

mutatingadmissionpolicy/applyconfiguration-example.yaml Copy mutatingadmissionpolicy/applyconfiguration-example.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingAdmissionPolicymetadata:  name: "sidecar-policy.example.com"spec:  paramKind:    kind: Sidecar    apiVersion: mutati
```

The `.spec.mutations` field consists of a list of expressions that evaluate to resource patches. The emitted patches may be either apply configurations or JSON Patch patches. You cannot specify an empty list of mutations. After evaluating all the expressions, the API server applies those changes to the resource that is passing through admission.

To configure a mutating admission policy for use in a cluster, a binding is required. The MutatingAdmissionPolicy will only be active if a corresponding binding exists with the referenced `spec.policyName` matching the `spec.name` of a policy.

Once the binding and policy are created, any resource request that matches the `spec.matchConditions` of a policy will trigger the set of mutations defined.

In the example above, creating a Pod will add the `mesh-proxy` initContainer mutation:

```
apiVersion: v1
kind: Podmetadata:  name: myapp  namespace: defaultspec:  ...  initContainers:  - name: mesh-proxy    image: mesh/proxy:v1.0.0    
```

Parameter resources

Parameter resources allow a policy configuration to be separate from its definition. A policy can define `paramKind`, which outlines GVK of the parameter resource, and then a policy binding ties a policy by name (via `policyName`) to a particular parameter resource via `paramRef`.

Please refer to parameter resources for more information.

`ApplyConfiguration`

MutatingAdmissionPolicy expressions are always CEL. Each apply configuration `expression` must evaluate to a CEL object (declared using `Object()` initialization).

Apply configurations may not modify atomic structs, maps or arrays due to the risk of accidental deletion of values not included in the apply configuration.

CEL expressions have access to the object types needed to create apply configurations:

- `Object` - CEL type of the resource object.
- `Object.<fieldName>` - CEL type of object field (such as `Object.spec`)
- `Object.<fieldName1>.<fieldName2>...<fieldNameN>` - CEL type of nested field (such as `Object.spec.containers`)

CEL expressions have access to the contents of the API request, organized into CEL variables as well as some other useful variables:

- `object` - The object from the incoming request. The value is null for DELETE requests.
- `oldObject` - The existing object. The value is null for CREATE requests.
- `request` - Attributes of the API request.
- `params` - Parameter resource referred to by the policy binding being evaluated. Only populated if the policy has a ParamKind.
- `namespaceObject` - The namespace object that the incoming object belongs to. The value is null for cluster-scoped resources.
- `variables` - Map of composited variables, from its name to its lazily evaluated value. For example, a variable named `foo` can be accessed as `variables.foo`.

- `authorizer` - A CEL Authorizer. May be used to perform authorization checks for the principal (user or service account) of the request. See https://pkg.go.dev/k8s.io/apiserver/pkg/cel/library#Authz
- `authorizer.requestResource` - A CEL ResourceCheck constructed from the `authorizer` and configured with the request resource.

The `apiVersion`, `kind`, `metadata.name`, `metadata.generateName` and `metadata.labels` are always accessible from the root of the object. No other metadata properties are accessible.

`JSONPatch`

The same mutation can be written as a JSON Patch as follows:

mutatingadmissionpolicy/json-patch-example.yaml Copy mutatingadmissionpolicy/json-patch-example.yaml to clipboard

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingAdmissionPolicymetadata:  name: "sidecar-policy.example.com"spec:  paramKind:    kind: Sidecar    apiVersion: mutati
```

The expression will be evaluated by CEL to create a JSON patch. ref: https://github.com/google/cel-spec

Each evaluated `expression` must return an array of `JSONPatch` values. The `JSONPatch` type represents one operation from a JSON patch.

For example, this CEL expression returns a JSON patch to conditionally modify a value:

```
[
  JSONPatch{op: "test", path: "/spec/example", value: "Red"},
  JSONPatch{op: "replace", path: "/spec/example", value: "Green"}
]
```

To define a JSON object for the patch operation `value`, use CEL `Object` types. For example:

```
[
  JSONPatch{
    op: "add",
    path: "/spec/selector",
    value: Object.spec.selector{matchLabels: {"environment": "test"}}
  }
]
```

To use strings containing '/' and '~' as JSONPatch path keys, use `jsonpatch.escapeKey()`. For example:

```
[
  JSONPatch{
    op: "add",
    path: "/metadata/labels/" + jsonpatch.escapeKey("example.com/environment"),
    value: "test"
  },
]
```

CEL expressions have access to the types needed to create JSON patches and objects:

- `JSONPatch` - CEL type of JSON Patch operations. JSONPatch has the fields `op`, `from`, `path` and `value`. See JSON patch for more details. The `value` field may be set to any of: string, integer, array, map or object. If set, the `path` and `from` fields must be set to a JSON pointer string, where the `jsonpatch.escapeKey()` CEL function may be used to escape path keys containing `/` and `~`.
- `Object` - CEL type of the resource object.
- `Object.<fieldName>` - CEL type of object field (such as `Object.spec`)
- `Object.<fieldName1>.<fieldName2>...<fieldNameN>` - CEL type of nested field (such as `Object.spec.containers`)

CEL expressions have access to the contents of the API request, organized into CEL variables as well as some other useful variables:

- `object` - The object from the incoming request. The value is null for DELETE requests.
- `oldObject` - The existing object. The value is null for CREATE requests.
- `request` - Attributes of the API request.
- `params` - Parameter resource referred to by the policy binding being evaluated. Only populated if the policy has a ParamKind.
- `namespaceObject` - The namespace object that the incoming object belongs to. The value is null for cluster-scoped resources.
- `variables` - Map of composited variables, from its name to its lazily evaluated value. For example, a variable named `foo` can be accessed as `variables.foo`.
- `authorizer` - A CEL Authorizer. May be used to perform authorization checks for the principal (user or service account) of the request. See https://pkg.go.dev/k8s.io/apiserver/pkg/cel/library#Authz
- `authorizer.requestResource` - A CEL ResourceCheck constructed from the `authorizer` and configured with the request resource.

CEL expressions have access to Kubernetes CEL function libraries as well as:

- `jsonpatch.escapeKey` - Performs JSONPatch key escaping. ~ and / are escaped as ~0 and ~1 respectively.

Only property names of the form `[a-zA-Z_.-/][a-zA-Z0-9_.-/]*` are accessible.

API kinds exempt from mutating admission

There are certain API kinds that are exempt from admission-time mutation. For example, you can't create a MutatingAdmissionPolicy that changes a MutatingAdmissionPolicy.

The list of exempt API kinds is:

- ValidatingAdmissionPolicies
- ValidatingAdmissionPolicyBindings

- MutatingAdmissionPolicies
- MutatingAdmissionPolicyBindings
- TokenReviews
- LocalSubjectAccessReviews
- SelfSubjectAccessReviews
- SelfSubjectReviews

API Access Control

For an introduction to how Kubernetes implements and controls API access, read Controlling Access to the Kubernetes API.

Reference documentation:

- Authenticating
  - Authenticating with Bootstrap Tokens
- Admission Controllers
  - Dynamic Admission Control
- Authorization
  - Role Based Access Control
  - Attribute Based Access Control
  - Node Authorization
  - Webhook Authorization
- Certificate Signing Requests
  - including CSR approval and certificate signing
- Service accounts
  - Developer guide
  - Administration
- Kubelet Authentication & Authorization
  - including kubelet TLS bootstrapping