API Access Control

- 1: Authenticating
- 2: Authenticating with Bootstrap Tokens
- 3: Certificate Signing Requests
- 4: Using Admission Controllers
- 5: Dynamic Admission Control
- 6: Managing Service Accounts
- 7: Authorization Overview
- 8: Using RBAC Authorization
- 9: Using ABAC Authorization
- 10: Using Node Authorization
- 11: Webhook Mode

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

1 - Authenticating

This page provides an overview of authenticating.

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. For more details, refer to the normal users topic in certificate request for more details about this.

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, an authenticating proxy, or HTTP basic auth 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.

All values are opaque to the authentication system and only hold significance when interpreted by an authorizer.

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) can be accomplished using an authenticating proxy or the authentication webhook.

X509 Client Certs

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.

Static Token File

The API server reads bearer tokens from a file when given the `--token-auth-file=SOMEFILE` option on the command line. Currently, tokens last indefinitely, and the token list cannot be changed without restarting API server.

The token file is a csv file with a minimum of 3 columns: token, user name, user uid, followed by 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"
```

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 THETOKEN`. 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` flag on the Controller Manager. This is done with something like `--controllers=*,tokencleaner`. `kubeadm` 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` A file containing a PEM encoded key for signing bearer tokens. If unspecified, the API server's TLS private key will be 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: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 3
  template:
    metadata:
    # ...
    spec:
      serviceAccountName: bob-the-bot
      containers:
      - name: nginx
        image: nginx:1.14.2
```

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 and an associated secret.

```
kubectl create serviceaccount jenkins
```

```
serviceaccount "jenkins" created
```

Check an associated secret:

```
kubectl get serviceaccounts jenkins -o yaml
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  # ...
secrets:
- name: jenkins-token-1yvwg
```

The created secret holds the public CA of the API server and a signed JSON Web Token (JWT).

```
kubectl get secret jenkins-token-1yvwg -o yaml
```

```
apiVersion: v1
data:
  ca.crt: (APISERVER'S CA BASE64 ENCODED)
  namespace: ZGVmYXVsdA==
  token: (BEARER TOKEN BASE64 ENCODED)
kind: Secret
metadata:
  # ...
type: kubernetes.io/service-account-token
```

> **Note:** Values are base64 encoded because secrets are always base64 encoded.

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 secrets 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)`.

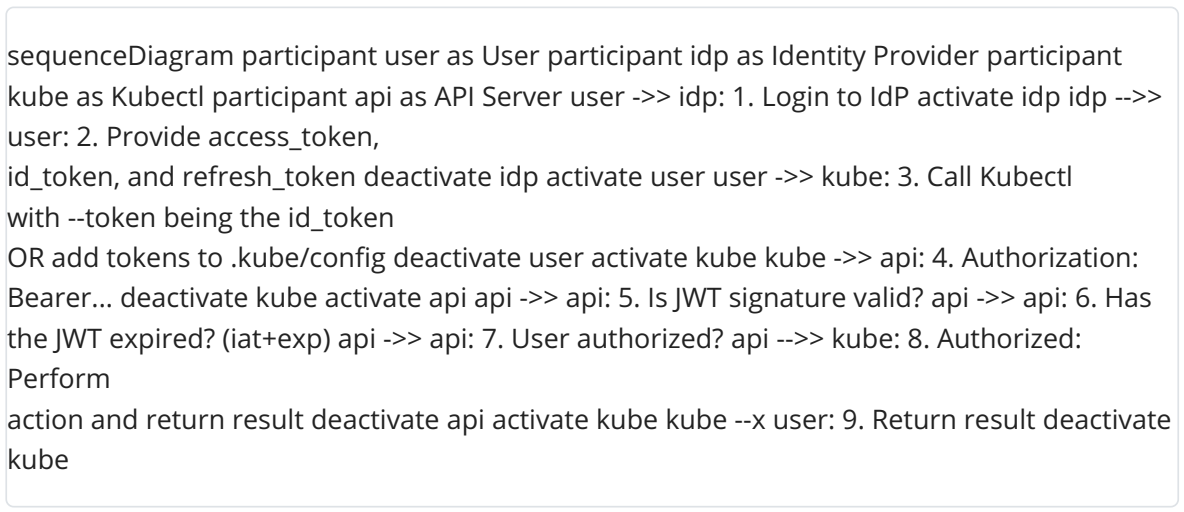
WARNING: Because service account tokens are stored in secrets, any user with read access to those secrets can authenticate as the service account. Be cautious when granting permissions to service accounts and read capabilities for secrets.

OpenID Connect Tokens

OpenID Connect is a flavor of OAuth2 supported by some OAuth2 providers, notably Azure Active Directory, 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.



1. Login 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` flag 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 by checking against the certificate named in the configuration
6. Check to make sure the `id_token` hasn't expired
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

To enable the plugin, configure the following flags on the API server:

Parameter	Description	Example	Required
`--oidc-issuer-url`	URL of the provider which allows the API server to discover public signing keys. Only URLs which use the `https://` scheme are accepted. This is typically the provider's discovery URL without a path, for example "https://accounts.google.com" or "https://login.salesforce.com". This URL should point to the level below .well-known/openid-configuration	If the discovery URL is `https://accounts.google.com/.well-known/openid-configuration`, the value should be `https://accounts.google.com`	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

Parameter	Description	Example	Required
`--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 flag 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 flag 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

Importantly, the API server is not an OAuth2 client, rather it can only be configured to trust a single issuer. This allows the use of public providers, such as Google, without trusting credentials issued to third parties. Admins who wish to utilize multiple OAuth clients should explore providers which support the `azp` (authorized party) claim, a mechanism for allowing one client to issue tokens on behalf of another.

Kubernetes does not provide an OpenID Connect Identity Provider. You can use an existing public OpenID Connect Identity Provider (such as Google, or others). Or, you can run your own Identity Provider, such as dex, Keycloak, CloudFoundry UAA, or Tremolo Security's OpenUnison.

For an identity provider to work with Kubernetes it must:

1. Support OpenID connect discovery; not all do.
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 (as opposed to one of the cloud providers like Google or Microsoft) 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 use this script from the Dex team to create a simple CA and a signed certificate and key pair. Or you can use this similar script that generates SHA256 certs with a longer life and larger key size.

Setup instructions for specific systems:

- UAA
- Dex
- OpenUnison

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" below.

```
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 ) \
   --auth-provider-arg=client-secret=( your client secret ) \
   --auth-provider-arg=refresh-token=( your refresh token ) \
```

```
   --auth-provider-arg=idp-certificate-authority=( path to your ca certificate ) \
   --auth-provider-arg=id-token=( your id_token )
```

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  \
        --auth-provider-arg=client-id=kubernetes  \
        --auth-provider-arg=client-secret=1db158f6-177d-4d9c-8a8b-d36869918ec5  \
        --auth-provider-arg=refresh-token=q1bKLFOyUiosTfawzA93TzZIDzH2TNa2SMm0zEiPKTUwME6BkEo6Sql5y
        --auth-provider-arg=idp-certificate-authority=/root/ca.pem \
        --auth-provider-arg=id-token=eyJraWQiOiJDTj1vaWRjaWRwLnRyZW1vbG8ubGFuLCBPVT1EZW1vLCBPPVRybW
```

Which would produce the below configuration:

```
users:
- name: mmosley
  user:
    auth-provider:
      config:
        client-id: kubernetes
        client-secret: 1db158f6-177d-4d9c-8a8b-d36869918ec5
        id-token: eyJraWQiOiJDTj1vaWRjaWRwLnRyZW1vbG8ubGFuLCBPVT1EZW1vLCBPPVRybWVvbG8gU2VjdXJpdHksI
        idp-certificate-authority: /root/ca.pem
        idp-issuer-url: https://oidcidp.tremolo.lan:8443/auth/idp/OidcIdP
        refresh-token: q1bKLFOyUiosTfawzA93TzZIDzH2TNa2SMm0zEiPKTUwME6BkEo6Sql5yUWVBSWpKUGphaWpxSVA
      name: oidc
```

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` Option

The `kubectl` command lets you pass in a token using the `--token` option. Copy and paste the `id_token` into this option:

```
kubectl --token=eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL21sYi50cmVtb2xvLmxhbjo4MDQzL2F1dGgvaWRwL2
```

Webhook Token Authentication

Webhook authentication is a hook for verifying bearer tokens.

- `--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 object
kind: Config
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authn-service
    cluster:
      certificate-authority: /path/to/ca.pem         # CA for verifying the remote service.
      server: https://authn.example.com/authenticate # URL of remote service to query. 'https' rec

# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webhook plugin to use
      client-key: /path/to/key.pem          # key matching the cert
```

```
# kubeconfig files require a context. Provide one for the API server.
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-authn-service
    user: name-of-api-server
  name: webhook
```

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 res
    # 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 Kuberr
    "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
    # If this is omitted, the token is considered to be valid to authenticate to the Kubernetes
    "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"
  }
}
```

Authenticating Proxy

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, which sets the request header value.

- `--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` 1.6+. Optional, case-insensitive. "X-Remote-Group" is suggested. Header names to check, in order, for the user's groups. All values in all specified headers are used as group names.
- `--requestheader-extra-headers-prefix` 1.6+. Optional, case-insensitive. "X-Remote-Extra-" is suggested. Header prefixes to look for to determine extra information about the user (typically used by the configured authorization plugin). 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.

> **Note:** Prior to 1.11.3 (and 1.10.7, 1.9.11), the extra key could only contain characters which were legal in HTTP header labels.

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
- dachshunds
extra:
  acme.com/project:
  - some-project
  scopes:
  - openid
  - profile
```

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. WARNING: 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.

- `--requestheader-client-ca-file` Required. 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. 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.

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.

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 )` should be lower-case, and any characters which aren't legal in HTTP header labels MUST be utf8 and percent-encoded.

> **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.

An example set of headers:

```
Impersonate-User: jane.doe@example.com
Impersonate-Group: developers
Impersonate-Group: admins
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
```

When using `kubectl` set the `--as` flag to configure the `Impersonate-User` header, 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 mynoc
```

Set the `--as` and `--as-group` flag:

```
kubectl drain mynode --as=superman --as-group=system:masters
```

```
node/mynode cordoned
node/mynode drained
```

To impersonate a user, group, or set extra fields, the impersonating user must have the ability to perform the "impersonate" verb on the kind of attribute being impersonated ("user", "group", etc.). For clusters that enable the RBAC authorization plugin, the following ClusterRole encompasses the rules needed to set user and group impersonation headers:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: impersonator
rules:
- apiGroups: [""]
  resources: ["users", "groups", "serviceaccounts"]
  verbs: ["impersonate"]
```

Extra fields are evaluated as sub-resources of the resource "userextras". To allow a user to use impersonation headers for the extra field "scopes", a user should be granted the following role:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: scopes-impersonator
rules:
# Can set "Impersonate-Extra-scopes" header.
- apiGroups: ["authentication.k8s.io"]
  resources: ["userextras/scopes"]
  verbs: ["impersonate"]
```

The values of impersonation headers can also be restricted by limiting the set of `resourceNames` a resource can take.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: limited-impersonator
rules:
# Can impersonate the user "jane.doe@example.com"
- apiGroups: [""]
  resources: ["users"]
  verbs: ["impersonate"]
  resourceNames: ["jane.doe@example.com"]

# Can impersonate the groups "developers" and "admins"
- apiGroups: [""]
  resources: ["groups"]
  verbs: ["impersonate"]
  resourceNames: ["developers","admins"]

# Can impersonate the extras field "scopes" with the values "view" and "development"
- apiGroups: ["authentication.k8s.io"]
  resources: ["userextras/scopes"]
  verbs: ["impersonate"]
  resourceNames: ["view", "development"]
```

client-go credential plugins

FEATURE STATE: `Kubernetes v1.11 [beta]`

`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.

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.

```yaml
apiVersion: v1
kind: Config
users:
- name: my-user
  user:
    exec:
      # Command to execute. Required.
      command: "example-client-go-exec-plugin"

      # API version to use when decoding the ExecCredentials resource. Required.
      #
      # The API version returned by the plugin MUST match the version listed here.
      #
      # To integrate with tools that support multiple versions (such as client.authentication.k8s.i
      # set an environment variable or pass an argument to the tool that indicates which version th
      apiVersion: "client.authentication.k8s.io/v1beta1"

      # Environment variables to set when executing the plugin. Optional.
      env:
      - name: "FOO"
        value: "bar"

      # Arguments to pass when executing the plugin. Optional.
      args:
      - "arg1"
      - "arg2"

      # Text shown to the user when the executable doesn't seem to be present. Optional.
      installHint: |
        example-client-go-exec-plugin is required to authenticate
        to the current cluster.  It can be installed:

        On macOS: brew install example-client-go-exec-plugin

        On Ubuntu: apt-get install example-client-go-exec-plugin

        On Fedora: dnf install example-client-go-exec-plugin

        ...

      # Whether or not to provide cluster information, which could potentially contain
      # very large CA data, to this exec plugin as a part of the KUBERNETES_EXEC_INFO
      # environment variable.
      provideClusterInfo: true
clusters:
- name: my-cluster
  cluster:
    server: "https://172.17.4.100:6443"
    certificate-authority: "/etc/kubernetes/ca.pem"
    extensions:
    - name: client.authentication.k8s.io/exec # reserved extension name for per cluster exec config
      extension:
        arbitrary: config
        this: can be provided via the KUBERNETES_EXEC_INFO environment variable upon setting provid
        you: ["can", "put", "anything", "here"]
contexts:
- name: my-cluster
  context:
    cluster: my-cluster
    user: my-user
current-context: my-cluster
```

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.

```yaml
- 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/v1beta1"
```

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`.

When run from an interactive session, `stdin` is exposed directly to the plugin. Plugins should use a TTY check to determine if it's appropriate to prompt a user interactively.

To use bearer token credentials, the plugin returns a token in the status of the `ExecCredential`

```
{
  "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.

```
{
  "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 RFC3339 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.

```
{
  "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 with an environment variable, `KUBERNETES_EXEC_INFO`. Information from this environment variable can be used to perform cluster-specific credential acquisition logic. The following `ExecCredential` manifest describes a cluster information sample.

```
{
  "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 pro
        "you": ["can", "put", "anything", "here"]
      }
    }
  }
}
```

What's next

- Read the client authentication reference (v1beta1)

2 - 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.

```yaml
apiVersion: v1
kind: Secret
metadata:
  # Name MUST be of form "bootstrap-token-<token id>"
  name: bootstrap-token-07401b
  namespace: kube-system

# Type MUST be 'bootstrap.kubernetes.io/token'
type: bootstrap.kubernetes.io/token
stringData:
  # Human readable description. Optional.
  description: "The default bootstrap token generated by 'kubeadm init'."

  # Token ID and secret. Required.
  token-id: 07401b
  token-secret: f395accd246ae52d

  # Expiration. Optional.
  expiration: 2017-03-10T03:22:11Z

  # Allowed usages.
```

```
  usage-bootstrap-authentication: "true"
  usage-bootstrap-signing: "true"

  # Extra groups to authenticate the token as. Must start with "system:bootstrappers:"
  auth-extra-groups: system:bootstrappers:worker,system:bootstrappers:ingress
```

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: ConfigMap
metadata:
  name: cluster-info
  namespace: kube-public
data:
  jws-kubeconfig-07401b: eyJhbGciOiJIUzI1NiIsImtpZCI6IjA3NDAxYiJ9..tYEfbo6zDNo40MQE07aZcQX2m3EB2rO3
  kubeconfig: |
    apiVersion: v1
    clusters:
    - cluster:
        certificate-authority-data: <really long certificate data>
        server: https://10.138.0.2:6443
      name: ""
    contexts: []
    current-context: ""
    kind: Config
    preferences: {}
    users: []
```

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.

3 - Certificate Signing Requests

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

The Certificates API enables 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).

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`.

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
- Pending requests: automatically deleted after 1 hour

Signers

All signers should provide information about how they work so that clients can predict what will happen to their CSRs. This includes:

1. **Trust distribution**: how trust (CA 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 object etc and the behavior when an expiration is different than the signer-determined expiration that is specified in the CSR.
6. **CA bit allowed/disallowed**: and behavior if a CSR contains a request a for a CA certificate when the signer does not permit it.

Commonly, the `status.certificate` field 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.

The PKCS#10 signing request format doesn't allow to specify a certificate expiration or lifetime. The expiration or lifetime therefore has to be set through e.g. an annotation on the CSR object. While it's theoretically possible for a signer to use that expiration date, there is currently no known implementation that does. (The built-in signers all use the same `ClusterSigningDuration` configuration option, which defaults to 1 year, and can be changed with the `--cluster-signing-duration` command-line flag of the kube-controller-manager.)

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 - set by the `--cluster-signing-duration` option for the kube-controller-manager implementation of this signer.
    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 starts with "`system:node:`".
    3. Permitted x509 extensions - honors key usage extensions, forbids subjectAltName extensions and drops other extensions.
    4. Permitted key usages - exactly `["key encipherment", "digital signature", "client auth"]`.
    5. Expiration/certificate lifetime - set by the `--cluster-signing-duration` option for the kube-controller-manager implementation of this signer.
    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 starts with "`system:node:`".
    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 - exactly `["key encipherment", "digital signature", "server auth"]`.
    5. Expiration/certificate lifetime - set by the `--cluster-signing-duration` option for the kube-controller-manager implementation of this signer.
    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 - set by the `--cluster-signing-duration` option for the kube-controller-manager implementation of this signer.
    6. CA bit allowed/disallowed - not allowed.

> **Note:** Failures for all of these are only reported in kube-controller-manager logs.

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`).

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

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csr-creator
rules:
- apiGroups:
  - certificates.k8s.io
  resources:
  - certificatesigningrequests
  verbs:
  - create
  - get
  - list
  - watch
```

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

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csr-approver
rules:
- apiGroups:
  - certificates.k8s.io
  resources:
  - certificatesigningrequests
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - certificates.k8s.io
  resources:
  - certificatesigningrequests/approval
  verbs:
  - update
- apiGroups:
  - certificates.k8s.io
  resources:
  - signers
  resourceNames:
  - example.com/my-signer-name # example.com/* can be used to authorize for all signers in the 'ex
  verbs:
  - approve
```

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

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
  name: csr-signer
rules:
- apiGroups:
  - certificates.k8s.io
  resources:
  - certificatesigningrequests
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - certificates.k8s.io
  resources:
  - certificatesigningrequests/status
  verbs:
  - update
- apiGroups:
  - certificates.k8s.io
  resources:
  - signers
  resourceNames:
  - example.com/my-signer-name # example.com/* can be used to authorize for all signers in the 'ex
  verbs:
  - sign
```

Normal user

A few steps are required in order to get a normal user to be able to authenticate and invoke an API. First, this user must have certificate issued by the Kubernetes cluster, and then present that Certificate to the API call as the Certificate Header or through the kubectl.

Create private key

The following scripts show how to generate PKI private key and CSR. It is important to set CN and O attribute of the CSR. CN is the name of the user and O is the group that this user will belong to. You can refer to RBAC for standard groups.

```
openssl genrsa -out myuser.key 2048
openssl req -new -key myuser.key -out myuser.csr
```

Create CertificateSigningRequest

Create a CertificateSigningRequest and submit it to a Kubernetes Cluster via kubectl. Below is a script to generate the CertificateSigningRequest.

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: myuser
spec:
  groups:
  - system:authenticated
  request: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ1ZqQ0NBVDRDQVFBd0VURVBNQTBHQTFVRUF3
  signerName: kubernetes.io/kube-apiserver-client
  usages:
  - client auth
EOF
```

Some points to note:

- `usages` has to be '`client auth`'
- `request` is the base64 encoded value of the CSR file content. You can get the content using this command:
  `cat myuser.csr | base64 | tr -d "\n"`

Approve certificate signing request

Use kubectl to create a CSR and approve it.

Get the list of CSRs:

```
kubectl get csr
```

Approve the CSR:

```
kubectl certificate approve myuser
```

Get the certificate

Retrieve the certificate from the CSR:

```
kubectl get csr/myuser -o yaml
```

The certificate value is in Base64-encoded format under `status.certificate`.

Export the issued certificate from the CertificateSigningRequest.

```
kubectl get csr myuser -o jsonpath='{.status.certificate}'| base64 -d > myuser.crt
```

Create Role and RoleBinding

With the certificate created. it is time to define the Role and RoleBinding for this user to access Kubernetes cluster resources.

This is a sample script to create a Role for this new user:

```
kubectl create role developer --verb=create --verb=get --verb=list --verb=update --verb=delete --re
```

This is a sample command to create a RoleBinding for this new user:

```
kubectl create rolebinding developer-binding-myuser --role=developer --user=myuser
```

Add to kubeconfig

The last step is to add this user into the kubeconfig file.

First, you need to add new credentials:

```
kubectl config set-credentials myuser --client-key=myuser.key --client-certificate=myuser.crt --emb
```

Then, you need to add the context:

```
kubectl config set-context myuser --cluster=kubernetes --user=myuser
```

To test it, change the context to `myuser`:

```
kubectl config use-context myuser
```

Approval or rejection

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:

```yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
...
status:
  conditions:
  - lastUpdateTime: "2020-02-08T11:37:35Z"
    lastTransitionTime: "2020-02-08T11:37:35Z"
    message: Approved by my custom approver controller
    reason: ApprovedByMyPolicy # You can set this to any string
    type: Approved
```

For `Denied` CSRs:

```yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
...
status:
  conditions:
  - lastUpdateTime: "2020-02-08T11:37:35Z"
    lastTransitionTime: "2020-02-08T11:37:35Z"
    message: Denied by my custom approver controller
    reason: DeniedByMyPolicy # You can set this to any string
    type: Denied
```

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.

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.

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
Q7Et13wrvTJqBMJo1GTwQuF+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:

```yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
...
status:
  certificate: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JS..."
```

What's next

- Read Manage TLS Certificates in a Cluster
- 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

4 - Using Admission Controllers

This page provides an overview of Admission Controllers.

What are they?

An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized. The controllers consist of the list below, are compiled into the `kube-apiserver` binary, and may only be configured by the cluster administrator. In that list, there are two special controllers: MutatingAdmissionWebhook and ValidatingAdmissionWebhook. These execute the mutating and validating (respectively) admission control webhooks which are configured in the API.

Admission controllers may be "validating", "mutating", or both. Mutating controllers may modify the objects they admit; validating controllers may not.

Admission controllers limit requests to create, delete, modify or connect to (proxy). They do not support read requests.

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.

Why do I need them?

Many advanced features in 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 the current version, the default ones are:

```
CertificateApproval, CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass, Defaul
```

What does each admission controller do?

AlwaysAdmit

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

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]`

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

AlwaysPullImages

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

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

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.

CertificateSubjectRestrictions

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

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

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 not do anything when no default storage class is configured. When more than one storage class is marked as default, it rejects any creation of `PersistentVolumeClaim` with an error and an administrator must revisit their `StorageClass` objects and mark only one as default. 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

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.

DenyEscalatingExec

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

This admission controller will deny exec and attach commands to pods that run with escalated privileges that allow host access. This includes pods that run as privileged, have access to the host IPC namespace, and have access to the host PID namespace.

The DenyEscalatingExec admission plugin is deprecated.

Use of a policy-based admission plugin (like PodSecurityPolicy or a custom admission plugin) which can be targeted at specific users or Namespaces and also protects against creation of overly privileged Pods is recommended instead.

DenyExecOnPrivileged

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

This admission controller will intercept all requests to exec a command in a pod if that pod has a privileged container.

This functionality has been merged into DenyEscalatingExec. The DenyExecOnPrivileged admission plugin is deprecated.

Use of a policy-based admission plugin (like PodSecurityPolicy or a custom admission plugin) which can be targeted at specific users or Namespaces and also protects against creation of overly privileged Pods is recommended instead.

DenyServiceExternalIPs

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.

EventRateLimit

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

This admission controller mitigates the problem where the API server gets flooded by event requests. 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`:

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

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

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

- `Server`: All event requests 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:

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

See the EventRateLimit proposal for more details.

ExtendedResourceToleration

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.

ImagePolicyWebhook

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

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`:

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

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

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

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

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: ImagePolicyWebhook
  configuration:
    imagePolicy:
      kubeConfigFile: <path-to-kubeconfig-file>
      allowTTL: 50
      denyTTL: 50
      retryBackoff: 500
      defaultAllow: true
```

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.

```
# clusters refers to the remote service.
clusters:
- name: name-of-remote-imagepolicy-service
  cluster:
    certificate-authority: /path/to/ca.pem    # CA for verifying the remote service.
    server: https://images.example.com/policy # URL of remote service to query. Must use 'https'.

# users refers to the API server's webhook configuration.
users:
- name: name-of-api-server
  user:
    client-certificate: /path/to/cert.pem # cert for the webhook admission controller to use
    client-key: /path/to/key.pem          # key matching the cert
```

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 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 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:

```
{
  "apiVersion":"imagepolicy.k8s.io/v1alpha1",
  "kind":"ImageReview",
  "spec":{
    "containers":[
      {
        "image":"myrepo/myimage:v1"
      },
      {
        "image":"myrepo/myimage@sha256:beb6bd6a68f114c1dc2ea4b28db81bdf91de202a9014972bec5e4d9171d9
      }
    ],
    "annotations":{
      "mycluster.image-policy.k8s.io/ticket-1234": "break-glass"
    },
    "namespace":"mynamespace"
  }
}
```

The remote service is expected to fill the `ImageReviewStatus` 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": "imagepolicy.k8s.io/v1alpha1",
  "kind": "ImageReview",
  "status": {
    "allowed": true
  }
}
```

To disallow access, the service would return:

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

For further documentation refer to the `imagepolicy.v1alpha1` API objects and `plugin/pkg/admission/imagepolicy/admission.go`.

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. In the future, if an annotation is determined to be widely useful, it may be promoted to a named field of `ImageReviewSpec`.

LimitPodHardAntiAffinityTopology

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

LimitRanger

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 design doc and the example of Limit Range for more details.

MutatingAdmissionWebhook

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 in versions >= 1.9).

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

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

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

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

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. In Kubernetes 1.11+, kubelets are not allowed to update or remove taints from their `Node` API object.

In Kubernetes 1.13+, 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

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

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

> **Note:** Support for volume resizing is available as an alpha feature. Admins must set the feature gate `ExpandPersistentVolumes` to `true` to enable resizing.

After enabling the `ExpandPersistentVolumes` feature gate, enabling the `PersistentVolumeClaimResize` admission controller is recommended, too. 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:

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gluster-vol-default
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://192.168.10.100:8080"
  restuser: ""
  secretNamespace: ""
  secretName: ""
allowVolumeExpansion: true
```

For more information about persistent volume claims, see PersistentVolumeClaims.

PersistentVolumeLabel

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

This admission controller automatically attaches region or zone labels to PersistentVolumes as defined by the cloud provider (for example, GCE or AWS). It helps ensure the Pods and the PersistentVolumes mounted are in the same region and/or zone. If the admission controller doesn't support automatic labelling your PersistentVolumes, you may need to add the labels manually to prevent pods from mounting volumes from a different zone. PersistentVolumeLabel is DEPRECATED and labeling persistent volumes has been taken over by the cloud-controller-manager. Starting from 1.11, this admission controller is disabled by default.

PodNodeSelector

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

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

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`:

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

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- 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: Namespace
metadata:
  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.

PodSecurityPolicy

This admission controller acts on creation and modification of the pod and determines if it should be admitted based on the requested security context and the available Pod Security Policies.

See also Pod Security Policy documentation for more information.

PodTolerationRestriction

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

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: Namespace
metadata:
  name: apps-that-need-nodes-exclusively
  annotations:
    scheduler.alpha.kubernetes.io/defaultTolerations: '[{"operator": "Exists", "effect": "NoSchedul
    scheduler.alpha.kubernetes.io/tolerationsWhitelist: '[{"operator": "Exists", "effect": "NoSched
```

</>

Priority

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

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 design doc and the example of Resource Quota for more details.

RuntimeClass

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

If you enable the `PodOverhead` feature gate, and 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 is configured and selected in their `.spec`, this admission controller sets `.spec.overhead` in the Pod based on the value defined in the corresponding RuntimeClass.

> **Note:** The `.spec.overhead` field for Pod and the `.overhead` field for RuntimeClass are both in beta. If you do not enable the `PodOverhead` feature gate, all Pods are treated as if `.spec.overhead` is unset.

See also Pod Overhead for more information.

SecurityContextDeny

This admission controller will deny any pod that attempts to set certain escalating SecurityContext fields, as shown in the Configure a Security Context for a Pod or Container task. This should be enabled if a cluster doesn't utilize pod security policies to restrict the set of values a security context can take.

ServiceAccount

This admission controller implements automation for serviceAccounts. We strongly recommend using this admission controller if you intend to make use of Kubernetes `ServiceAccount` objects.

StorageObjectInUseProtection

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

FEATURE STATE: `Kubernetes v1.17 [stable]`

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.

ValidatingAdmissionWebhook

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 (both are on by default in versions 1.9 and later).

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**).

> **Note:** `--admission-control` was deprecated in 1.10 and replaced with `--enable-admission-plugins`.

5 - 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 the Kubernetes cluster is at least as new as v1.16 (to use `admissionregistration.k8s.io/v1`), or v1.9 (to use `admissionregistration.k8s.io/v1beta1`).
- 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` or `admissionregistration.k8s.io/v1beta1` 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 apiservers, 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 apiservers. If you need mutual TLS or other ways to authenticate the clients, see how to authenticate apiservers.

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.

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
```

```
  rules:
  - apiGroups:   [""]
    apiVersions: ["v1"]
    operations:  ["CREATE"]
    resources:   ["pods"]
    scope:       "Namespaced"
  clientConfig:
    service:
      namespace: "example-namespace"
      name: "example-service"
    caBundle: "Ci0tLS0tQk...<`caBundle` is a PEM encoded CA bundle which will be used to valida
  admissionReviewVersions: ["v1", "v1beta1"]
  sideEffects: None
  timeoutSeconds: 5
```

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 for webhooks registered created using `admissionregistration.k8s.io/v1`, and 30 seconds for webhooks created using `admissionregistration.k8s.io/v1beta1`. Starting in kubernetes 1.14 you can set the timeout and it is encouraged to use a small timeout for webhooks. If the webhook call times out, the request is handled according to the webhook's failure policy.

When an apiserver receives a request that matches one of the `rules`, the apiserver 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 apiservers

If your admission webhooks require authentication, you can configure the apiservers 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 apiserver, 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: AdmissionConfiguration
plugins:
- name: ValidatingAdmissionWebhook
  configuration:
    apiVersion: apiserver.config.k8s.io/v1
    kind: WebhookAdmissionConfiguration
    kubeConfigFile: "<path-to-kubeconfig-file>"
- name: MutatingAdmissionWebhook
  configuration:
    apiVersion: apiserver.config.k8s.io/v1
    kind: WebhookAdmissionConfiguration
    kubeConfigFile: "<path-to-kubeconfig-file>"
```

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: Config
users:
# name should be set to the DNS name of the service or the host (including port) of the URL th
# If a non-443 port is used for services, it must be included in the name when configuring 1.1
#
# For a webhook configured to speak to a service on the default port (443), specify the DNS na
# - name: webhook1.ns1.svc
#   user: ...
#
# For a webhook configured to speak to a service on non-default port (e.g. 8443), specify the
# - name: webhook1.ns1.svc:8443
#   user: ...
```

```
# and optionally create a second stanza using only the DNS name of the service for compatibili
# - name: webhook1.ns1.svc
#   user: ...
#
# For webhooks configured to speak to a URL, match the host (and port) specified in the webhoo
# A webhook with `url: https://www.example.com`:
# - name: www.example.com
#   user: ...
#
# A webhook with `url: https://www.example.com:443`:
# - name: www.example.com:443
#   user: ...
#
# A webhook with `url: https://www.example.com:8443`:
# - name: www.example.com:8443
#   user: ...
#
- name: 'webhook1.ns1.svc'
  user:
    client-certificate-data: "<pem encoded certificate>"
    client-key-data: "<pem encoded key>"
# The `name` supports using * to wildcard-match prefixing segments.
- name: '*.webhook-company.org'
  user:
    password: "<password>"
    username: "<name>"
# '*' is the default match.
- name: '*'
  user:
    token: "<token>"
```

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

Webhook request and response

Request

Webhooks are sent a POST request, 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:

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

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

`admissionReviewVersions` is a required field when creating `admissionregistration.k8s.io/v1` 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`:

admission.k8s.io/v1 | admission.k8s.io/v1beta1

```
{
  "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
    # 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
    # This only differs from `resource` if the webhook specified `matchPolicy: Equivalent` and
    # 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` a
    # 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 Namespac
    "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 and should be included
      # if any SubjectAccessReview checks are performed by the webhook.
      "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 contains the options for the operation being admitted, like meta.k8s.io/v1 Create
    # 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.
    # See http://k8s.io/docs/reference/using-api/api-concepts/#make-a-dry-run-request for more
    "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:

admission.k8s.io/v1 admission.k8s.io/v1beta1

```
{
  "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:

admission.k8s.io/v1 admission.k8s.io/v1beta1

```
{
  "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:

admission.k8s.io/v1 admission.k8s.io/v1beta1

```
{
  "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:

admission.k8s.io/v1 admission.k8s.io/v1beta1

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true,
    "patchType": "JSONPatch",
    "patch": "W3sib3AiOiAiYWRkIiwgInBhdGgiOiAiL3NwZWMvcmVwbGljYXMiLCAidmFsdWUiOiAzfV0="
  }
}
```

Starting in v1.19, 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.

admission.k8s.io/v1 admission.k8s.io/v1beta1

```
{
  "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 s
    ]
  }
}
```

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 should be given a unique name. This is required in `admissionregistration.k8s.io/v1`, but strongly recommended when using `admissionregistration.k8s.io/v1beta1`, 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. Supported in v1.14+. Default is `"*"`, matching pre-1.14 behavior.
  - `"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`:

admissionregistration.k8s.io/v1 admissionregistration.k8s.io/v1beta1

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  rules:
  - operations: ["CREATE", "UPDATE"]
    apiGroups: ["apps"]
    apiVersions: ["v1", "v1beta1"]
    resources: ["deployments", "replicasets"]
    scope: "Namespaced"
  ...
```

Match create requests for all resources (but not subresources) in all API groups and versions:

admissionregistration.k8s.io/v1 admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  rules:
  - operations: ["CREATE"]
    apiGroups: ["*"]
    apiVersions: ["*"]
    resources: ["*"]
    scope: "*"
  ...
```

Match update requests for all `status` subresources in all API groups and versions:

admissionregistration.k8s.io/v1 admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  rules:
  - operations: ["UPDATE"]
    apiGroups: ["*"]
    apiVersions: ["*"]
    resources: ["*/status"]
    scope: "*"
  ...
```

Matching requests: objectSelector

In v1.15+, 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 with the label `foo: bar`:

admissionregistration.k8s.io/v1 admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  objectSelector:
    matchLabels:
      foo: bar
  rules:
  - operations: ["CREATE"]
    apiGroups: ["*"]
    apiVersions: ["*"]
    resources: ["*"]
    scope: "*"
  ...
```

See https://kubernetes.io/docs/concepts/overview/working-with-objects/labels 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":

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  namespaceSelector:
    matchExpressions:
    - key: runlevel
      operator: NotIn
      values: ["0","1"]
  rules:
  - operations: ["CREATE"]
    apiGroups: ["*"]
    apiVersions: ["*"]
    resources: ["*"]
    scope: "Namespaced"
  ...
```

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":

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  namespaceSelector:
    matchExpressions:
    - key: environment
      operator: In
      values: ["prod","staging"]
  rules:
  - operations: ["CREATE"]
    apiGroups: ["*"]
    apiVersions: ["*"]
    resources: ["*"]
    scope: "Namespaced"
  ...
```

See https://kubernetes.io/docs/concepts/overview/working-with-objects/labels for more examples of label selectors.

Matching requests: matchPolicy

API servers can make objects available via multiple API groups or versions. For example, the Kubernetes API server may allow creating and modifying `Deployment` objects via `extensions/v1beta1`, `apps/v1beta1`, `apps/v1beta2`, and `apps/v1` APIs.

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.

In v1.15+, `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 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:

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  matchPolicy: Equivalent
  rules:
  - operations: ["CREATE","UPDATE","DELETE"]
    apiGroups: ["apps"]
    apiVersions: ["v1"]
    resources: ["deployments"]
    scope: "Namespaced"
  ...
```

Admission webhooks created using `admissionregistration.k8s.io/v1` default to `Equivalent`.

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 apiservers (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 apiserver 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):

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  clientConfig:
    url: "https://my-webhook.example.com:9443/my-webhook-path"
  ...
```

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:

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  clientConfig:
    caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle containing the CA that signed the webhoo
    service:
      namespace: my-service-namespace
      name: my-service-name
      path: /my-path
      port: 1234
  ...
```

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:

- `Unknown`: no information is known about the side effects of calling the webhook. If a request with `dryRun: true` would trigger a call to this webhook, the request will instead fail, and the webhook will not be called.
- `None`: calling the webhook will have no side effects.
- `Some`: calling the webhook will possibly have side effects. If a request with the dry-run attribute would trigger a call to this webhook, the request will instead fail, and the webhook will not be called.
- `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).

Allowed values:

- In `admissionregistration.k8s.io/v1beta1`, `sideEffects` may be set to `Unknown`, `None`, `Some`, or `NoneOnDryRun`, and defaults to `Unknown`.
- In `admissionregistration.k8s.io/v1`, `sideEffects` must be set to `None` or `NoneOnDryRun`.

Here is an example of a validating webhook indicating it has no side effects on `dryRun: true` requests:

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
webhooks:
- 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:

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
```

```
webhooks:
- name: my-webhook.example.com
  timeoutSeconds: 2
  ...
```

Admission webhooks created using `admissionregistration.k8s.io/v1` default timeouts 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).

In v1.15+, 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:

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
...
webhooks:
- 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:

admissionregistration.k8s.io/v1 | admissionregistration.k8s.io/v1beta1

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
...
webhooks:
- name: my-webhook.example.com
  failurePolicy: Fail
  ...
```

Admission webhooks created using `admissionregistration.k8s.io/v1` default `failurePolicy` to `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.

In v1.16+, kube-apiserver 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_1_index_2": "{\"configuration\":\"my-mutating-webh
        # other annotations
        ...
    }
    # other fields
    ...
}
```

```
# the annotation value deserialized
{
    "configuration": "my-mutating-webhook-configuration.example.com",
    "webhook": "my-webhook.example.com",
    "mutated": false
}
```

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_0_index_0": "{\"configuration\":\"my-mutating-webh
        # other annotations
        ...
    }
    # other fields
    ...
}
```

```
# the annotation value deserialized
{
    "configuration": "my-mutating-webhook-configuration.example.com",
    "webhook": "my-webhook-always-mutate.example.com",
    "mutated": true
}
```

- 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_index_3": "{\"configuration\":\"my-other-mutating-w
        # other annotations
        ...
    }
    # other fields
    ...
}
```

```
# the annotation value deserialized
{
    "configuration": "my-other-mutating-webhook-configuration.example.com",
    "webhook": "my-webhook-always-mutate.example.com",
    "patchType": "JSONPatch",
    "patch": [
        {
            "op": "add",
            "path": "/data/mutation-stage",
            "value": "yes"
        }
    ]
}
```

Admission webhook metrics

Kube-apiserver 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.

In v1.16+, kube-apiserver 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, ident
# TYPE apiserver_admission_webhook_rejection_count counter
apiserver_admission_webhook_rejection_count{error_type="calling_webhook_error",name="always-timeout
apiserver_admission_webhook_rejection_count{error_type="calling_webhook_error",name="invalid-admiss
apiserver_admission_webhook_rejection_count{error_type="no_error",name="deny-unwanted-configmap-dat
```

Best practices and warnings

Idempotence

An idempotent mutating admission webhook is able to successfully process an object it has already admitted and potentially modified. The admission can be applied multiple times without changing the result beyond the initial application.

Example of idempotent mutating admission webhooks:

1. For a `CREATE` pod request, set the field `.spec.securityContext.runAsNonRoot` of the pod to true, to enforce security best practices.
2. For a `CREATE` pod request, if the field `.spec.containers[].resources.limits` of a container is not set, set default resource limits.
3. For a `CREATE` pod request, inject a sidecar container with name `foo-sidecar` if no container with the name `foo-sidecar` already exists.

In the cases above, the webhook can be safely reinvoked, or admit an object that already has the fields set.

Example of non-idempotent mutating admission webhooks:

1. For a `CREATE` pod request, inject a sidecar container with name `foo-sidecar` suffixed with the current timestamp (e.g. `foo-sidecar-19700101-000000`).
2. For a `CREATE` / `UPDATE` pod request, reject if the pod has label `"env"` set, otherwise add an `"env": "prod"` label to the pod.
3. For a `CREATE` pod request, blindly append a sidecar container named `foo-sidecar` without looking to see if there is already a `foo-sidecar` container in the pod.

In the first case above, reinvoking the webhook can result in the same sidecar being injected multiple times to a pod, each time with a different container name. Similarly the webhook can inject duplicated containers if the sidecar already exists in a user-provided pod.

In the second case above, reinvoking the webhook will result in the webhook failing on its own output.

In the third case above, reinvoking the webhook will result in duplicated containers in the pod spec, which makes the request invalid and rejected by the API server.

Intercepting all versions of an object

It is recommended that admission webhooks should always intercept all versions of an object by setting `.webhooks[].matchPolicy` to `Equivalent`. It is also recommended that admission webhooks should prefer registering for stable versions of resources. Failure to intercept all versions of an object can result in admission policies not being enforced for requests in certain versions. See Matching requests: matchPolicy for examples.

Availability

It is recommended that admission webhooks should evaluate as quickly as possible (typically in milliseconds), since they add to API request latency. It is encouraged to use a small timeout for webhooks. See Timeouts for more detail.

It is recommended that admission webhooks should leverage some format of load-balancing, to provide high availability and performance benefits. If a webhook is running within the cluster, you can run multiple webhook backends behind a service to leverage the load-balancing that service supports.

Guaranteeing the final state of the object is seen

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.

For example, a mutating admission webhook is configured to inject a sidecar container with name "foo-sidecar" on every `CREATE` pod request. If the sidecar *must* be present, a validating admisson webhook should also be configured to intercept `CREATE` pod requests, and validate that a container with name "foo-sidecar" with the expected configuration exists in the to-be-created object.

Avoiding deadlocks in self-hosted webhooks

A webhook running inside the cluster might cause deadlocks for its own deployment if it is configured to intercept resources required to start its own pods.

For example, a mutating admission webhook is configured to admit `CREATE` pod requests only if a certain label is set in the pod (e.g. `"env": "prod"`). The webhook server runs in a deployment which doesn't set the `"env"` label. When a node that runs the webhook server pods becomes unhealthy, the webhook deployment will try to reschedule the pods to another node. However the requests will get rejected by the existing webhook server since the `"env"` label is unset, and the migration cannot happen.

It is recommended to exclude the namespace where your webhook is running with a namespaceSelector.

Side effects

It is recommended that admission webhooks should avoid side effects if possible, which means the webhooks operate only on the content of the `AdmissionReview` sent to them, and do not make out-of-band changes. The `.webhooks[].sideEffects` field should be set to `None` if a webhook doesn't have any side effect.

If side effects are required during the admission evaluation, they must be suppressed when processing an `AdmissionReview` object with `dryRun` set to `true`, and the `.webhooks[].sideEffects` field should be set to `NoneOnDryRun`. See Side effects for more detail.

Avoiding operating on the kube-system namespace

The `kube-system` namespace contains objects created by the Kubernetes system, e.g. service accounts for the control plane components, pods like `kube-dns`. Accidentally mutating or rejecting requests in the `kube-system` namespace may cause the control plane components to stop functioning or introduce unknown behavior. If your admission webhooks don't intend to modify the behavior of the Kubernetes control plane, exclude the `kube-system` namespace from being intercepted using a `namespaceSelector`.

6 - Managing Service Accounts

This is a Cluster Administrator guide to service accounts. You should be familiar with configuring Kubernetes service accounts.

Support for authorization and user accounts is planned but incomplete. Sometimes incomplete features are referred to in order to better describe service accounts.

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 processes, which run in pods.
- User accounts are intended to be global. Names must be unique across all namespaces of a cluster. Service accounts are namespaced.
- Typically, a cluster's user accounts might be synced from a corporate database, where new user account creation requires special privileges and is tied to complex business processes. Service account creation is intended to be more lightweight, allowing cluster users to create service accounts for specific tasks by following the principle of least privilege.
- Auditing considerations for humans and service accounts may differ.
- A config 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 config is portable.

Service account automation

Three separate components cooperate to implement the automation around service accounts:

- A `ServiceAccount` admission controller
- A Token controller
- A `ServiceAccount` controller

ServiceAccount Admission Controller

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

1. If the pod does not have a `ServiceAccount` set, it sets the `ServiceAccount` to `default`.
2. It ensures that the `ServiceAccount` referenced by the pod exists, and otherwise rejects it.
3. It adds a `volume` to the pod which contains a token for API access if neither the ServiceAccount `automountServiceAccountToken` nor the Pod's `automountServiceAccountToken` is set to `false`.
4. It adds a `volumeSource` to each container of the pod mounted at `/var/run/secrets/kubernetes.io/serviceaccount`, if the previous step has created a volume for ServiceAccount token.
5. If the pod does not contain any `imagePullSecrets`, then `imagePullSecrets` of the `ServiceAccount` are added to the pod.

Bound Service Account Token Volume

FEATURE STATE: `Kubernetes v1.21 [beta]`

When the `BoundServiceAccountTokenVolume` feature gate is enabled, the service account admission controller will add the following projected volume instead of a Secret-based volume for the non-expiring service account token created by Token Controller.

```
- name: kube-api-access-<random-suffix>
  projected:
    defaultMode: 420 # 0644
    sources:
      - serviceAccountToken:
          expirationSeconds: 3600
          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
```

This projected volume consists of three sources:

1. A ServiceAccountToken acquired from kube-apiserver via TokenRequest API. It will expire after 1 hour by default or when the pod is deleted. It is bound to the pod and has kube-apiserver as the audience.
2. A ConfigMap containing a CA bundle used for verifying connections to the kube-apiserver. This feature depends on the `RootCAConfigMap` feature gate, which publishes a "kube-root-ca.crt" ConfigMap to every namespace. `RootCAConfigMap` feature gate is graduated to GA in 1.21 and default to true. (This flag will be removed from --feature-gate arg in 1.22)
3. A DownwardAPI that references the namespace of the pod.

See more details about projected volumes.

You can manually migrate a Secret-based service account volume to a projected volume when the `BoundServiceAccountTokenVolume` feature gate is not enabled by adding the above projected volume to the pod spec.

Token Controller

TokenController runs as part of `kube-controller-manager`. It acts asynchronously. It:

- watches ServiceAccount creation and creates a corresponding ServiceAccount token Secret to allow API access.
- watches ServiceAccount deletion and deletes all corresponding ServiceAccount token Secrets.
- watches ServiceAccount token Secret addition, and ensures the referenced ServiceAccount exists, and adds a token to the Secret if needed.
- watches 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.

To create additional API tokens

A controller loop ensures a Secret with an API token exists for each ServiceAccount. To create additional API tokens for a ServiceAccount, create a Secret of type `kubernetes.io/service-account-token` with an annotation referencing the ServiceAccount, and the controller will update it with a generated token:

Below is a sample configuration for such a Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecretname
  annotations:
    kubernetes.io/service-account.name: myserviceaccount
type: kubernetes.io/service-account-token
```

```
kubectl create -f ./secret.yaml
kubectl describe secret mysecretname
```

To delete/invalidate a ServiceAccount token Secret

```
kubectl delete secret mysecretname
```

ServiceAccount controller

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

7 - Authorization Overview

Learn more about Kubernetes authorization, including details about creating policies using the supported authorization modules.

In Kubernetes, you must be authenticated (logged in) before your request can be authorized (granted permission to access). For information about authentication, see Controlling Access to the Kubernetes API.

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.

Determine Whether a Request is Allowed or Denied

Kubernetes authorizes API requests using the API server. It evaluates all of the request attributes against all policies and allows or denies the request. All parts of an API request must be allowed by some policy in order to proceed. This means that permissions are denied by default.

(Although Kubernetes uses the API server, access controls and policies that depend on specific fields of specific kinds of objects are handled by Admission Controllers.)

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. A deny returns an HTTP status code 403.

Review Your Request Attributes

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 Determine the request verb.
- **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.

Determine the Request Verb

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, a `GET` request to endpoints like `/api` or `/healthz` would use `get` as the verb.

Resource requests To determine the request verb for a resource API endpoint, review the HTTP verb used and whether or not the request acts on an individual resource or 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)

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

- PodSecurityPolicy
  - `use` verb on `podsecuritypolicies` resources in the `policy` API group.
- RBAC
  - `bind` and `escalate` verbs on `roles` and `clusterroles` resources in the `rbac.authorization.k8s.io` API group.
- Authentication
  - `impersonate` verb on `users`, `groups`, and `serviceaccounts` in the core API group, and the `userextras` in the `authentication.k8s.io` API group.

Authorization Modes

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

- **Node** - A special-purpose authorization mode that grants permissions to kubelets based on the pods they are scheduled to run. To learn more about using the Node authorization mode, see Node Authorization.
- **ABAC** - 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. The policies can use any type of attributes (user attributes, resource attributes, object, environment attributes, etc). To learn more about using the ABAC mode, see ABAC Mode.
- **RBAC** - Role-based access control (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. To learn more about using the RBAC mode, see RBAC Mode
  - When specified RBAC (Role-Based Access Control) uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing admins to dynamically configure permission policies through the Kubernetes API.
  - To enable RBAC, start the apiserver with `--authorization-mode=RBAC`.
- **Webhook** - 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. To learn more about using the Webhook mode, see Webhook Mode.

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
```

`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.

```
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:

```
apiVersion: authorization.k8s.io/v1
kind: SelfSubjectAccessReview
metadata:
  creationTimestamp: null
spec:
  resourceAttributes:
    group: apps
    resource: deployments
    namespace: dev
    verb: create
status:
  allowed: true
  denied: false
```

Using Flags for Your Authorization Module

You must include a flag in your policy to indicate which authorization module your policies include:

The following flags can be used:

- `--authorization-mode=ABAC` Attribute-Based Access Control (ABAC) mode allows you to configure policies using local files.
- `--authorization-mode=RBAC` Role-based access control (RBAC) mode allows you to create and store policies using the Kubernetes API.
- `--authorization-mode=Webhook` WebHook is an HTTP callback mode that allows you to manage authorization using a remote REST endpoint.
- `--authorization-mode=Node` Node authorization is a special-purpose authorization mode that specifically authorizes API requests made by kubelets.
- `--authorization-mode=AlwaysDeny` This flag blocks all requests. Use this flag only for testing.
- `--authorization-mode=AlwaysAllow` This flag allows all requests. Use this flag only if you do not require authorization for your API requests.

You can choose more than one authorization module. Modules are checked in order so an earlier module has higher priority to allow or deny a request.

Privilege escalation via pod creation

Users who have the ability to create pods in a namespace can potentially escalate their privileges within that namespace. They can create pods that access their privileges within that namespace. They can create pods that access secrets the user cannot themselves read, or that run under a service account with different/greater permissions.

> **Caution:** System administrators, use care when granting access to pod creation. A user granted permission to create pods (or controllers that create pods) in the namespace can: read all secrets in the namespace; read all config maps in the namespace; and impersonate any service account in the namespace and take any action the account could take. This applies regardless of authorization mode.

What's next

- To learn more about Authentication, see **Authentication** in Controlling Access to the Kubernetes API.

- To learn more about Admission Control, see Using Admission Controllers.

</>

8 - 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-mode` flag set to a comma-separated list that includes `RBAC`; for example:

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

API objects

The RBAC API declares four kinds of Kubernetes object: *Role*, *ClusterRole*, *RoleBinding* and *ClusterRoleBinding*. You can describe objects, or amend them, 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 within individual namespace(s)
2. define permissions on namespaced resources and be granted 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:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

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):

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

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.

```yaml
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 that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

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".

```yaml
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 "secret-reader".
kind: RoleBinding
metadata:
  name: read-secrets
  #
  # The namespace of the RoleBinding determines where the permissions are granted.
  # This only grants permissions within the "development" namespace.
  namespace: development
subjects:
- kind: User
  name: dave # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

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.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

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:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-and-pod-logs-reader
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
```

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`:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: configmap-updater
rules:
- apiGroups: [""]
```

```
  #
  # at the HTTP level, the name of the resource for accessing ConfigMap
  # objects is "configmaps"
  resources: ["configmaps"]
  resourceNames: ["my-configmap"]
  verbs: ["update", "get"]
```

Note: You cannot restrict `create` or `deletecollection` requests by resourceName. For `create`, this limitation is because the object name is not known at authorization time.

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.

Here is an example aggregated ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: monitoring
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
rules: [] # The control plane automatically fills in the rules
```

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: ClusterRole
metadata:
  name: monitoring-endpoints
  labels:
    rbac.example.com/aggregate-to-monitoring: "true"
# When you create the "monitoring-endpoints" ClusterRole,
# the rules below will be added to the "monitoring" ClusterRole.
rules:
- apiGroups: [""]
  resources: ["services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
```

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: ClusterRole
metadata:
  name: aggregate-cron-tabs-edit
  labels:
    # Add these permissions to the "admin" and "edit" default roles.
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
rules:
- apiGroups: ["stable.example.com"]
  resources: ["crontabs"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true"
```

```yaml
rules:
- apiGroups: ["stable.example.com"]
  resources: ["crontabs"]
  verbs: ["get", "list", "watch"]
```

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:

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

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

```yaml
rules:
- apiGroups: ["extensions", "apps"]
  #
  # at the HTTP level, the name of the resource for accessing Deployment
  # objects is "deployments"
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

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

```yaml
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Pod
  # objects is "pods"
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["batch", "extensions"]
  #
  # at the HTTP level, the name of the resource for accessing Job
  # objects is "jobs"
  resources: ["jobs"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

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

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

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):

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

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):

```yaml
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`:

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

For a group named `frontend-admins`:

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

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

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

For all service accounts in the "qa" group in any namespace:

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

For all service accounts in the "dev" group in the "development" namespace:

```yaml
subjects:
- kind: Group
  name: system:serviceaccounts:dev
```

```
  apiGroup: rbac.authorization.k8s.io
  namespace: development
```

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
  apiGroup: rbac.authorization.k8s.io
```

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 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 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` 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.

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.

Kubernetes RBAC API discovery roles

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.
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.

Default ClusterRole	Default ClusterRoleBinding	Description
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**. 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.

Default ClusterRole	Default ClusterRoleBinding	Description
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/*`), and `/metrics`). 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`:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: role-grantor
rules:
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["rolebindings"]
  verbs: ["create"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["clusterroles"]
  verbs: ["bind"]
  # omit resourceNames to allow binding any ClusterRole
  resourceNames: ["admin","edit","view"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: role-grantor-binding
  namespace: user-1-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: role-grantor
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: user-1
```

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.
- If your API server runs with the insecure port enabled (`--insecure-port`), you can also make API calls via that port, which does not enforce authentication or authorization.

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 --resour
```

- 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 --
```

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 -
```

- 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-monito
```

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 -
```

- 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 --serviceaccou
```

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 --use
```

- 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=
```

- 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:
```

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-permission
```

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 discovery permissions given to all authenticated users).

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
```

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 discovery permissions given to all authenticated users).

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.

9 - 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: `extensions`
      - 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`
      - 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 api-server 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":
```

2. The Kubelet can read any pods:

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user":
```

3. The Kubelet can read and write events:

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user":
```

4. Bob can just read pods in namespace "projectCaribou":

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user":
```

5. Anyone can make read-only requests to all non-resource paths:

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"group":
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"group":
```

Complete file example

A quick note on service accounts

Every service account has a corresponding ABAC username, and that service account's user name 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:se
```

The apiserver will need to be restarted to pickup the new policy lines.

</>

</>

</>

</>

</>

</>

</>

10 - 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

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 certificationsigningrequests 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 apiserver with `--authorization-mode=Node`.

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.

Upgrades from previous versions using RBAC

Upgraded pre-1.7 clusters using RBAC will continue functioning as-is because the `system:nodes` group binding will already exist.

If a cluster admin wishes to start using the `Node` authorizer and `NodeRestriction` admission plugin to limit node access to the API, that can be done non-disruptively:

1. Enable the `Node` authorization mode (`--authorization-mode=Node,RBAC`) and the `NodeRestriction` admission plugin
2. Ensure all kubelets' credentials conform to the group/username requirements
3. Audit apiserver logs to ensure the `Node` authorizer is not rejecting requests from kubelets (no persistent `NODE DENY` messages logged)
4. Delete the `system:node` cluster role binding

RBAC Node Permissions

In 1.6, the `system:node` cluster role was automatically bound to the `system:nodes` group when using the RBAC Authorization mode.

In 1.7, the automatic binding of the `system:nodes` group to the `system:node` role is deprecated because the node authorizer accomplishes the same purpose with the benefit of additional restrictions on secret and configmap access. If the `Node` and `RBAC` authorization modes are both enabled, the automatic binding of the `system:nodes` group to the `system:node` role is not created in 1.7.

In 1.8, the binding will not be created at all.

When using RBAC, the `system:node` cluster role will continue to be created, for compatibility with deployment methods that bind other users or groups to that role.

11 - 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 object
kind: Config
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authz-service
    cluster:
      # CA for verifying the remote service.
      certificate-authority: /path/to/ca.pem
      # URL of remote service to query. Must use 'https'. May not include parameters.
      server: https://authz.example.com/authorize

# users refers to the API Server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webhook plugin to use
      client-key: /path/to/key.pem          # key matching the cert

# kubeconfig files require a context. Provide one for the API Server.
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-authz-service
    user: name-of-api-server
  name: webhook
```

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"
    ]
  }
}
```

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 documentation refer to the authorization.v1beta1 API objects and webhook.go.