Service Accounts

Archived: 2026-04-06 03:17:34 UTC

Learn about ServiceAccount objects in Kubernetes.

This page introduces the ServiceAccount object in Kubernetes, providing information about how service accounts work, use cases, limitations, alternatives, and links to resources for additional guidance.

What are service accounts?

A service account is a type of non-human account that, in Kubernetes, provides a distinct identity in a Kubernetes cluster. Application Pods, system components, and entities inside and outside the cluster can use a specific ServiceAccount's credentials to identify as that ServiceAccount. This identity is useful in various situations, including authenticating to the API server or implementing identity-based security policies.

Service accounts exist as ServiceAccount objects in the API server. Service accounts have the following properties:

- **Namespaced:** Each service account is bound to a Kubernetes namespace. Every namespace gets a `default` ServiceAccount upon creation.
- **Lightweight:** Service accounts exist in the cluster and are defined in the Kubernetes API. You can quickly create service accounts to enable specific tasks.
- **Portable:** A configuration bundle for a complex containerized workload might include service account definitions for the system's components. The lightweight nature of service accounts and the namespaced identities make the configurations portable.

Service accounts are different from user accounts, which are authenticated human users in the cluster. By default, user accounts don't exist in the Kubernetes API server; instead, the API server treats user identities as opaque data. You can authenticate as a user account using multiple methods. Some Kubernetes distributions might add custom extension APIs to represent user accounts in the API server.

Description	ServiceAccount	User or group
Location	Kubernetes API (ServiceAccount object)	External
Access control	Kubernetes RBAC or other authorization mechanisms	Kubernetes RBAC or other identity and access management mechanisms
Intended use	Workloads, automation	People

Default service accounts

When you create a cluster, Kubernetes automatically creates a ServiceAccount object named `default` for every namespace in your cluster. The `default` service accounts in each namespace get no permissions by default other than the default API discovery permissions that Kubernetes grants to all authenticated principals if role-based access control (RBAC) is enabled. If you delete the `default` ServiceAccount object in a namespace, the control plane replaces it with a new one.

If you deploy a Pod in a namespace, and you don't manually assign a ServiceAccount to the Pod, Kubernetes assigns the `default` ServiceAccount for that namespace to the Pod.

Use cases for Kubernetes service accounts

As a general guideline, you can use service accounts to provide identities in the following scenarios:

- Your Pods need to communicate with the Kubernetes API server, for example in situations such as the following:
  - Providing read-only access to sensitive information stored in Secrets.
  - Granting cross-namespace access, such as allowing a Pod in namespace `example` to read, list, and watch for Lease objects in the `kube-node-lease` namespace.
- Your Pods need to communicate with an external service. For example, a workload Pod requires an identity for a commercially available cloud API, and the commercial provider allows configuring a suitable trust relationship.
- Authenticating to a private image registry using an `imagePullSecret`.
- An external service needs to communicate with the Kubernetes API server. For example, authenticating to the cluster as part of a CI/CD pipeline.
- You use third-party security software in your cluster that relies on the ServiceAccount identity of different Pods to group those Pods into different contexts.

How to use service accounts

To use a Kubernetes service account, you do the following:

1. Create a ServiceAccount object using a Kubernetes client like `kubectl` or a manifest that defines the object.

2. Grant permissions to the ServiceAccount object using an authorization mechanism such as RBAC.

3. Assign the ServiceAccount object to Pods during Pod creation.

   If you're using the identity from an external service, retrieve the ServiceAccount token and use it from that service instead.

For instructions, refer to Configure Service Accounts for Pods.

Grant permissions to a ServiceAccount

You can use the built-in Kubernetes role-based access control (RBAC) mechanism to grant the minimum permissions required by each service account. You create a *role*, which grants access, and then *bind* the role to your ServiceAccount. RBAC lets you define a minimum set of permissions so that the service account permissions follow the principle of least privilege. Pods that use that service account don't get more permissions than are required to function correctly.

For instructions, refer to ServiceAccount permissions.

Cross-namespace access using a ServiceAccount

You can use RBAC to allow service accounts in one namespace to perform actions on resources in a different namespace in the cluster. For example, consider a scenario where you have a service account and Pod in the `dev` namespace and you want your Pod to see Jobs running in the `maintenance` namespace. You could create a Role object that grants permissions to list Job objects. Then, you'd create a RoleBinding object in the `maintenance` namespace to bind the Role to the ServiceAccount object. Now, Pods in the `dev` namespace can list Job objects in the `maintenance` namespace using that service account.

Assign a ServiceAccount to a Pod

To assign a ServiceAccount to a Pod, you set the `spec.serviceAccountName` field in the Pod specification. Kubernetes then automatically provides the credentials for that ServiceAccount to the Pod. In v1.22 and later, Kubernetes gets a short-lived, **automatically rotating** token using the `TokenRequest` API and mounts the token as a projected volume.

By default, Kubernetes provides the Pod with the credentials for an assigned ServiceAccount, whether that is the `default` ServiceAccount or a custom ServiceAccount that you specify.

To prevent Kubernetes from automatically injecting credentials for a specified ServiceAccount or the `default` ServiceAccount, set the `automountServiceAccountToken` field in your Pod specification to `false`.

In versions earlier than 1.22, Kubernetes provides a long-lived, static token to the Pod as a Secret.

Manually retrieve ServiceAccount credentials

If you need the credentials for a ServiceAccount to mount in a non-standard location, or for an audience that isn't the API server, use one of the following methods:

- TokenRequest API (recommended): Request a short-lived service account token from within your own *application code*. The token expires automatically and can rotate upon expiration. If you have a legacy application that is not aware of Kubernetes, you could use a sidecar container within the same pod to fetch these tokens and make them available to the application workload.
- Token Volume Projection (also recommended): In Kubernetes v1.20 and later, use the Pod specification to tell the kubelet to add the service account token to the Pod as a *projected volume*. Projected tokens expire automatically, and the kubelet rotates the token before it expires.

- Service Account Token Secrets (not recommended): You can mount service account tokens as Kubernetes Secrets in Pods. These tokens don't expire and don't rotate. In versions prior to v1.24, a permanent token was automatically created for each service account. This method is not recommended anymore, especially at scale, because of the risks associated with static, long-lived credentials. The LegacyServiceAccountTokenNoAutoGeneration feature gate (which was enabled by default from Kubernetes v1.24 to v1.26), prevented Kubernetes from automatically creating these tokens for ServiceAccounts. The feature gate is removed in v1.27, because it was elevated to GA status; you can still create indefinite service account tokens manually, but should take into account the security implications.

Note:

For applications running outside your Kubernetes cluster, you might be considering creating a long-lived ServiceAccount token that is stored in a Secret. This allows authentication, but the Kubernetes project recommends you avoid this approach. Long-lived bearer tokens represent a security risk as, once disclosed, the token can be misused. Instead, consider using an alternative. For example, your external application can authenticate using a well-protected private key `and` a certificate, or using a custom mechanism such as an authentication webhook that you implement yourself.

You can also use TokenRequest to obtain short-lived tokens for your external application.

Restricting access to Secrets (deprecated)

FEATURE STATE: `Kubernetes v1.32 [deprecated]`

Note:

`kubernetes.io/enforce-mountable-secrets` is deprecated since Kubernetes v1.32. Use separate namespaces to isolate access to mounted secrets.

Kubernetes provides an annotation called `kubernetes.io/enforce-mountable-secrets` that you can add to your ServiceAccounts. When this annotation is applied, the ServiceAccount's secrets can only be mounted on specified types of resources, enhancing the security posture of your cluster.

You can add the annotation to a ServiceAccount using a manifest:

```
 apiVersion: v1
kind: ServiceAccount
metadata:
  annotations:
    kubernetes.io/enforce-mountable-secrets: "true"
  name: my-serviceaccount
  namespace: my-namespace
```

When this annotation is set to "true", the Kubernetes control plane ensures that the Secrets from this ServiceAccount are subject to certain mounting restrictions.

1. The name of each Secret that is mounted as a volume in a Pod must appear in the `secrets` field of the Pod's ServiceAccount.
2. The name of each Secret referenced using `envFrom` in a Pod must also appear in the `secrets` field of the Pod's ServiceAccount.
3. The name of each Secret referenced using `imagePullSecrets` in a Pod must also appear in the `secrets` field of the Pod's ServiceAccount.

By understanding and enforcing these restrictions, cluster administrators can maintain a tighter security profile and ensure that secrets are accessed only by the appropriate resources.

Authenticating service account credentials

ServiceAccounts use signed JSON Web Tokens (JWTs) to authenticate to the Kubernetes API server, and to any other system where a trust relationship exists. Depending on how the token was issued (either time-limited using a `TokenRequest` or using a legacy mechanism with a Secret), a ServiceAccount token might also have an expiry time, an audience, and a time after which the token *starts* being valid. When a client that is acting as a ServiceAccount tries to communicate with the Kubernetes API server, the client includes an `Authorization: Bearer <token>` header with the HTTP request. The API server checks the validity of that bearer token as follows:

1. Checks the token signature.
2. Checks whether the token has expired.
3. Checks whether object references in the token claims are currently valid.
4. Checks whether the token is currently valid.
5. Checks the audience claims.

The TokenRequest API produces *bound tokens* for a ServiceAccount. This binding is linked to the lifetime of the client, such as a Pod, that is acting as that ServiceAccount. See Token Volume Projection for an example of a bound pod service account token's JWT schema and payload.

For tokens issued using the `TokenRequest` API, the API server also checks that the specific object reference that is using the ServiceAccount still exists, matching by the unique ID of that object. For legacy tokens that are mounted as Secrets in Pods, the API server checks the token against the Secret.

For more information about the authentication process, refer to Authentication.

Authenticating service account credentials in your own code

If you have services of your own that need to validate Kubernetes service account credentials, you can use the following methods:

- TokenReview API (recommended)
- OIDC discovery

The Kubernetes project recommends that you use the TokenReview API, because this method invalidates tokens that are bound to API objects such as Secrets, ServiceAccounts, Pods or Nodes when those objects are deleted. For

example, if you delete the Pod that contains a projected ServiceAccount token, the cluster invalidates that token immediately and a TokenReview immediately fails. If you use OIDC validation instead, your clients continue to treat the token as valid until the token reaches its expiration timestamp.

Your application should always define the audience that it accepts, and should check that the token's audiences match the audiences that the application expects. This helps to minimize the scope of the token so that it can only be used in your application and nowhere else.

Alternatives

- Issue your own tokens using another mechanism, and then use Webhook Token Authentication to validate bearer tokens using your own validation service.
- Provide your own identities to Pods.
    - Use the SPIFFE CSI driver plugin to provide SPIFFE SVIDs as X.509 certificate pairs to Pods.

      🚫 This item links to a third party project or product that is not part of Kubernetes itself. More information
    - Use a service mesh such as Istio to provide certificates to Pods.
- Authenticate from outside the cluster to the API server without using service account tokens:
    - Configure the API server to accept OpenID Connect (OIDC) tokens from your identity provider.
    - Use service accounts or user accounts created using an external Identity and Access Management (IAM) service, such as from a cloud provider, to authenticate to your cluster.
    - Use the CertificateSigningRequest API with client certificates.
- Configure the kubelet to retrieve credentials from an image registry.
- Use a Device Plugin to access a virtual Trusted Platform Module (TPM), which then allows authentication using a private key.

What's next

- Learn how to manage your ServiceAccounts as a cluster administrator.
- Learn how to assign a ServiceAccount to a Pod.
- Read the ServiceAccount API reference.

Last modified November 19, 2024 at 10:53 PM PST: Address comments (3b8c927a3b)

Source: https://kubernetes.io/docs/concepts/security/service-accounts/