Service Accounts in Kubernetes

A Service Account in Kubernetes is an identity that Pods can use to interact with the Kubernetes API. It's often used to give a Pod permissions to access Kubernetes resources, like reading or modifying resources (e.g., ConfigMaps, Secrets).

📄 1. Service Account Example

By default, Pods run with the default Service Account, but you can create custom Service Accounts to provide more control over the permissions.

Creating a Service Account YAML:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
```

Creating the Service Account using kubectl:

```
kubectl apply -f service-account.yaml
```

📄 2. Assigning a Service Account to a Pod

Once the Service Account is created, you can assign it to a Pod using the serviceAccountName field in the Pod specification:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  serviceAccountName: my-service-account
  containers:
    - name: mycontainer
      image: nginx
```

Security Context

Security Context in Kubernetes

A Security Context in Kubernetes defines privilege and access control settings for Pod and Container. This includes settings like running containers as a non-root user, setting resource limits, or configuring security-related options (e.g., SELinux).

📄 1. Example of a Pod with Security Context

Here’s how you can define a Security Context for a Pod to run the container as a non-root user:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  securityContext:
    runAsUser: 1000  # Non-root user
    runAsGroup: 3000  # Group ID
    fsGroup: 2000     # Group ID for file system permissions
  containers:
    - name: mycontainer
      image: nginx
      securityContext:
        allowPrivilegeEscalation: false  # Prevents privilege escalation
```

💡 Explanation:

runAsUser: Runs the container as a specific user (in this case, user 1000).

runAsGroup: Runs the container under a specific group (group 3000).

fsGroup: Grants the specified group permissions for the container’s filesystem.

allowPrivilegeEscalation: Prevents the container from escalating privileges.

📄 2. Container-Level Security Context

A Security Context can also be applied specifically at the container level inside a Pod. For example, you can run a container as a privileged container or set capabilities:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mycontainer
      image: nginx
      securityContext:
        privileged: true  # Allows the container to run with elevated privileges
        capabilities:
          add:
            - NET_ADMIN  # Add network admin capability
```

```
# Service Account commands

# 1. Create a Service Account from a YAML file
kubectl apply -f service-account.yaml

# 2. List all Service Accounts in the current namespace
kubectl get serviceaccounts

# 3. Get detailed information about a specific Service Account
kubectl describe serviceaccount <service-account-name>

# 4. Delete a Service Account
kubectl delete serviceaccount <service-account-name>

# 5. Create a Pod with a specific Service Account
kubectl apply -f pod-with-service-account.yaml

# 6. Assign a Service Account to a Pod (using `serviceAccountName` in YAML)
kubectl apply -f pod-with-service-account.yaml

# Security Context commands
# 7. Create a Pod with a Security Context (Pod level)
kubectl apply -f pod-with-security-context.yaml

# 8. Create a Pod with a container-level Security Context
kubectl apply -f pod-with-container-security-context.yaml

# 9. Get detailed information about a Pod's Security Context
kubectl describe pod <pod-name>

# 10. Edit a Pod's Security Context
kubectl edit pod <pod-name>

# 11. Delete a Pod
kubectl delete pod <pod-name>
```