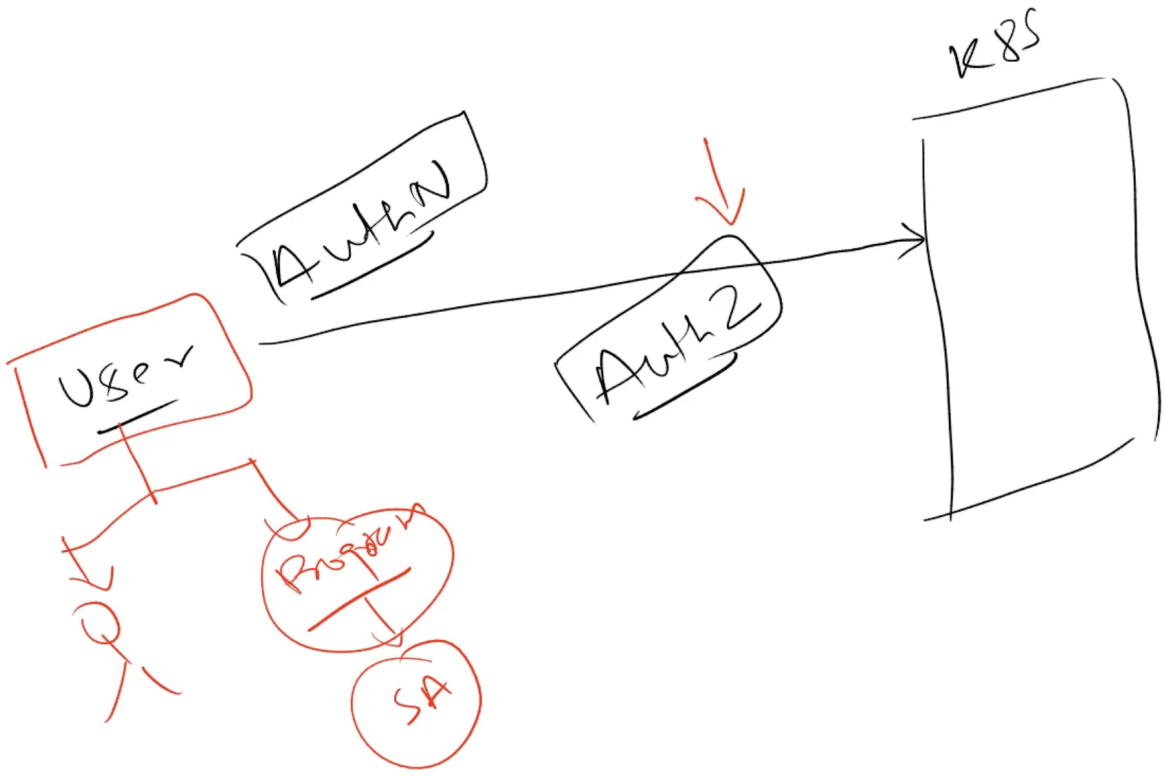
RBAC

After a user is authenticated to Kubernetes cluster, he is then authorized before performing any action to check either is allowed to perform that action or not. This can be done using Kubernetes role based access control (RBAC).

In the case of **applications**, we authenticate and authorize **service accounts** to check whether an application is allowed to perform certain actions inside the kubernetes cluster or not.



The RBAC can be implemented using the following resources that come under the kubernetes **api group** called **rbac.authorization.k8s.io**,

- Role → (For giving access to the resources only at the namespace scope. For e.g, pods, deployments, statefulsets, etc.)
- Role binding → (They give access within a **specific namespace** when a Role or Cluster role is associated with this type of binding)
- Cluster role → (For giving access to the resources beyond the namespace scope. For e.g, storage classes, PV, etc.)
- Cluster role binding → (They give access within **all the namespace** when a Role or Cluster role is associated with this type of binding)

Important Notes:

1. As **Roles** give access to the resources only inside of a namespace, the access given inside of a role is applied to **specific namespaces only** when bound with role binding.
2. As **Cluster roles** can give access to the resources beyond the scope of the namespaces, the accesses given inside of a cluster role is applied on **all the namespaces** when bound using **cluster role binding**.
3. **Cluster roles** can also be bound with **role binding** but in that case the access will be given only in that **namespace in which the role binding was created**. In order to get accesses mentioned inside a cluster role in all the namespaces, we will have to create separate role bindings in **all the namespaces and bind them with the same cluster role**.
4. Binding a **Role with Cluster role binding** means we are giving access to only **namespace scoped resources** for **all the namespaces** to a user or service accounts.
5. A user or service account can have multiple role bindings.

Grouping Cluster Role OR Aggregated Rules

If we want to aggregate or combine all the rules of specific cluster roles into one generalized/parent cluster role that will be containing all the rules defined inside its child/specific cluster roles then we can use aggregating rules. Inside the aggregated rules field of a cluster role, we provide some labels and kubernetes then see all other cluster roles and search if they have these labels or not. If labels mentioned inside the aggregated rule of a parent or generalized cluster role found in any other cluster role, then the rules of that child or specific cluster role is then merged/combined/aggregated with the generalized/parent cluster role.
For e.g,
// Yaml of generalized cluster role (See aggregationRule and the label mentioned under it)

```
Every 2.0s: kubectl get clusterrole monitoring -oyaml

aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: "2021-04-04T10:45:40Z"
  managedFields:
  - apiVersion: rbac.authorization.k8s.io/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:aggregationRule:
        .: {}
        f:clusterRoleSelectors: {}
      f:rules: {}
    manager: kubectl
    operation: Update
    time: "2021-04-04T10:45:40Z"
  name: monitoring
  resourceVersion: "4794"
  uid: 92bd2a5b-5a5e-4e20-8427-30c6d50ef360
rules: null
```

Currently, there is no rule aggregated/merged here because **rbac.example.com/aggregate-to-monitoring: “true”** label was not found in other cluster roles. We will create some cluster roles with some rules, having this label in their manifest files and then again we will see how the aggregation works. It is not mandatory to create a parent/aggregated/generalized cluster role first; we can also create it after the creation of a specific/child role.

// Child/Specific cluster roles having the aggregation label and some access rules.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: null
  labels:
    rbac.example.com/aggregate-to-monitoring: "true"
  name: mon-endpoints
rules:
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - list
  - watch

```

Creating this child/specific cluster role,

```
~/yt/rbac » k create -f mon-endpoints.yaml
clusterrole.rbac.authorization.k8s.io/mon-endpoints created
```

Now lets see the parent/generalized/aggregated cluster role again. You can see the access rules mentioned inside the child/specific cluster role are now aggregated here,

```
2: watch kubectl get clusterrole monitoring -oyaml
Every 2.0s: kubectl get clusterrole monitoring -oyaml

aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: "2021-04-04T10:45:40Z"
  managedFields:
  - apiVersion: rbac.authorization.k8s.io/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:aggregationRule:
        .: {}
        f:clusterRoleSelectors: {}
    manager: kubectl
    operation: Update
    time: "2021-04-04T10:45:40Z"
  - apiVersion: rbac.authorization.k8s.io/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:rules: {}
    manager: kube-controller-manager
    operation: Update
    time: "2021-04-04T10:48:06Z"
  name: monitoring
  resourceVersion: "5044"
  uid: 92bd2a5b-5a5e-4e20-8427-30c6d50ef360
rules:
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - list
  - watch
```

We can also add non-kubernetes-resource rules as well in roles and cluster roles. For example, endpoint is something which is not a kubernetes resource so in order to make a rule to make get request to specific endpoint, then we can define a rule something like this,
For e.g, endpoint is pod/log

```
rules:
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - pods/log
  verbs:
  - list
  - update
```

Users and Groups

Kubernetes only care about certificates. There is no user representation object in Kubernetes. Normal users cannot be added to a cluster through an API call. Users need to be created using X. 509 certificates. Any user that presents a valid certificate signed by the cluster's certificate authority (CA) is considered authenticated.

How can a user authenticate with Kubernetes Cluster?

Users must have a certificate issued by the Kubernetes cluster, and then present that certificate to the Kubernetes API to authenticate with the Kubernetes cluster. We can directly give any user the admin KubeConfig file but to limit the accesses and to register users, groups with less privileges we generate a cluster signed certificate and then generate his own less privilege Kubeconfig file from that cluster signed certificate to authenticate and authorize him with the Kubernetes cluster or Kube API server.

Steps to get a cluster signed certificate for a user

1. A user must generate his **private key and csr (certificate signing request)** to make a request to kubernetes cluster to sign his certificate and **return him a cluster signed certificate** so that he can authenticate with the cluster.

   (A)
   Command to generate Private Key
   ```
   openssl genrsa -out <key file name>.key <key bits 2048, 4096, etc>
   ```
   For example,
   ```
   openssl genrsa -out Tashik.key 2048
   ```

   (B)
   Command to generate a Certificate Signing Request (CSR)
   ```
   openssl req -new -key <Private Key Name>.key -out <CSR name>.csr
   -subj "/CN=<name of user here>/O=<name of user group here>"

   CN means Common Name
   O for specifying the user group.
   ```

   For example,
   ```
   openssl req -new -key Tashik.key -out Tashik.csr  -subj
   "/CN=Tashik/O=DevOps"
   ```

2. Give this certificate signing request (CSR) file to your cluster admin then he will generate a signed certificate for you so that you can authenticate with the kubernetes API server or the cluster. A cluster admin can generate a signed certificate in two ways,

   (A)
   If he has access to the master node of the cluster (he is using Minikube, Kubeadm, etc tools for cluster provisioning).
   Inside the master node, there are two files ca.key and ca.crt which are used to generate cluster **signed certificates** for users. The **ca.key** is the certificate authority key of the cluster and **ca.crt** is the cluster certificate. These files can be found on the following paths,

   ```
   /etc/kubernetes/pki/ca.key
   /etc/kubernetes/pki/ca.crt
   ```

   The cluster admin can copy these files from the master node of the cluster to his machine using the scp command and then he can generate a cluster signed certificate for a user using these two files + user's csr (certificate signing request) file with the following command,

   ```
   openssl x509 -req -in <user's csr file path> -CA <path to cluster's
   CA certificate crt file> -CAkey <path to cluster's private key ca
   key file> -CAcreateserial -out <name of output cluster signed
   certificate>.crt -day <total days of expiry of this signed
   certificate>
   ```

   For example,
   ```
   openssl x509 -req -in Tashik.csr -CA ca.crt -CAkey ca.key
   -CAcreateserial -out Tashik_Signed.crt -day 365
   ```

   (B)
   If he does not have access to the master node of the cluster **(he is using AKS, EKS, etc tools for cluster provisioning)**.
   Make a CertificateSigningRequest and submit it to a Kubernetes Cluster using the following Kubectl command.

   ```
   cat <<EOF | kubectl apply -f -

   apiVersion: certificates.k8s.io/v1
   ```

```
kind: CertificateSigningRequest

metadata:

 name: <requesting user name>

spec:

 request: <convert csr file content in base64 encoding and paste it
here. Note: ignore the % character at the end of the base64 while
pasting the content here.
The command to convert csr content to base64 encoding is,
cat <name of csr file>.csr | base64 | tr -d "\n"
>

 signerName: kubernetes.io/kube-apiserver-client

 expirationSeconds: 31536000
# signed certificate expiry time
# 1 day = 86400, 1 year = 31536000

 usages:

 - client auth
EOF
```

For example,

Reference:
https://kubernetes.io/docs/reference/access-authn-authz/certificate-signing-requests/#normal-user

Now Kubectl will use the same ca.key and ca.crt files from the master node on your behalf to generate a cluster signed certificate.

The cluster admin will execute the above cat command inside the kubernetes cluster to generate a signed cluster certificate for a user

```
tashik [ ~ ]$ kubectl get csr
No resources found
tashik [ ~ ]$ cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: Tashik
spec:
  request: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ1p6Q0NBVThDQVFBd0lqRVBNQTBHQTFVRUF3d0dWR0Z6YUdsclROHdEUVlEVlFRS0RBWkVaWFpQY0hNdwpnZ0VpTUEwR0
NTcUdTSWIzRFFFQkFRVUFBNElCRHdBd2dnRUtBb0lCQVFER0QxTmJJL3c0RE5UdFpUbmgramFoCktWOUFmUzBMamNzbFJteHdvY3VELzljQ3R6eGtya2lxcFRQVS9tUlh2Zk13R1lKU2NWVE8zM0ExS0kzVHQ
0YmoKcGduOU9mNlJrMWVnUDd2QTZhVDhPdlBscmVFaUhwWFNHZmJaazRHQ0JWcjg3WlRrS1BEOTFoa3pTTU9qMlNDSApvZ0FrSEhKNXlPUFVoakptWnBQSnpHVnRjZGglWmhSVjhNb1NoaWRGbHlSdHlnUjNO
dWJCeVYrVUhhbURscUttCjhsS0M4WUJ5bUdRcmFqaFZYUzd0Sm1OdkZ0OWZibVZvanNZNUFUa2FsdGxneE0rLzVtTk12elVLVUMrbGlzYXAKSjBDZFFJNDl2YkhZMVFSd3NhQ3RLNUdrZWJHS203eXV2Uk5TT
ms3Y1BySzc0OUIxRW8zWC9SQnpZVzZ6RXNYZgpBZ01CQUFHZ0FEQU5CZ2txaGtpRzl3MEJBUXNGQUFPQ0FRRUFEdUNXWGQ1eklrROHdEUVlEVlFRS0RBWkVaWFpQY0hNdwpnZ0VpTUEwR0
lreFBUWWNqZFF2VjlEVk94aHR3MlZNMmVPSEVjdjE5ei9DbWlVMGYKTmp5dHZQOXkvc2wvS2FHWXZDeXR3UlhkKzkvOTlwSVkvSDFHTmtDQzA4MzBFK1RFNnZ2MGtxNnk0UGtDMWpwVgpYZXNNL3kydDlDNzF
HV25DdGJZVi9MdzFoOVgrSXg2R3FQTzNaNUJ6KzJxOXlkUUtEbjAzeXFqT0hialdqVUMvCm5tZGVacVVRSEFvQ1BMWmlmM3FzRDRxNzE0dzJ5Y0prRlBHcTd4Q1RHNUN4Z0pFaW5rMDlCOWFndVVCZEhoLzkK
SmJJRW9nQkg5dFlyWVJKYUVwMDAzM09HUjVLRWxCWmxMSkxqcHlGYVRDdkxWdUdRZWc1OHdreHUvdz09Ci0tLS0tRU5EIENFUlRJRklDQVRFIFJFUVVFU1QtLS0tLQo=
  signerName: kubernetes.io/kube-apiserver-client
  expirationSeconds: 31536000
  usages:
  - client auth
EOF
certificatesigningrequest.certificates.k8s.io/Tashik created
tashik [ ~ ]$ kubectl get csr
NAME     AGE   SIGNERNAME                            REQUESTOR      REQUESTEDDURATION   CONDITION
Tashik   46s   kubernetes.io/kube-apiserver-client   masterclient   365d                Pending
tashik [ ~ ]$
```

Run the following command to approve the certificate,

`kubectl certificate approve <username>`

For example,

`kubectl certificate approve Tashik`

```
tashik [ ~ ]$ kubectl get csr
NAME     AGE     SIGNERNAME                            REQUESTOR      REQUESTEDDURATION   CONDITION
Tashik   3m36s   kubernetes.io/kube-apiserver-client   masterclient   365d                Pending
tashik [ ~ ]$ kubectl certificate approve Tashik
certificatesigningrequest.certificates.k8s.io/Tashik approved
tashik [ ~ ]$ kubectl get csr
NAME     AGE     SIGNERNAME                            REQUESTOR      REQUESTEDDURATION   CONDITION
Tashik   3m53s   kubernetes.io/kube-apiserver-client   masterclient   365d                Approved,Issued
tashik [ ~ ]$
```

To get the signed certificate in yaml format,

`kubectl get csr/Tashik -o yaml`

```
tashik [ ~ ]$ kubectl get csr/Tashik -o yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"certificates.k8s.io/v1","kind":"CertificateSigningRequest","metadata":{"annotations":{},"name":"Tashik"},"spec":{"expirationSeconds":315
36000,"request":"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ1p6Q0NBVThDQVFBd0lqRVBNQTBHQTFVRUF3d0dWR0Z6YUdsclROHdEUVlEVlFRS0RBWkVaWFpQY0hNdwpnZ0VpTUEwR0NTcUdTSWIzRFFFQkFRVUFBNElCRHdBd2dnRUtBb0lCQVFER0QxTmJJL3c0RE5UdFpUbmgramFoCktWOUFmUzBMamNzbFJteHdvY3VELzljQ3R6eGtya2lxcFRQVS9tUlh2Zk13R1lKU2NWVE8zM0ExS0kzVHQ
0YmoKcGduOU9mNlJrMWVnUDd2QTZhVDhPdlBscmVFaUhwWFNHZmJaazRHQ0JWcjg3WlRrS1BEOTFoa3pTTU9qMlNDSApvZ0FrSEhKNXlPUFVoakptWnBQSnpHVnRjZGglWmhSVjhNb1NoaWRGbHlSdHlnUjNOdWJCeVYrVUhhbURscUttCjhsS0M4WUJ5bUdRcmFqaFZYUzd0Sm1OdkZ0OWZibVZvanNZNUFUa2FsdGxneE0rLzVtTk12elVLVUMrbGlzYXAKSjBDZFFJNDl2YkhZMVFSd3NhQ3RLNUdrZWJHS203eXV
2Uk5TTms3Y1BySzc0OUIxRW8zWC9SQnpZVzZ6RXNYZgpBZ01CQUFHZ0FEQU5CZ2txaGtpRzl3MEJBUXNGQUFPQ0FRRUFEdUNXWGQ1eklrROHdEUVlEVlFRS0RBWkVaWFpQY0hNdwpnZ0VpTUEwR0lreFBUWWNqZFF2VjlEVk94aHR3MlZNMmVPSEVjdjE5ei9DbWlVMGYKTmp5dHZQOXkvc2wvS2FHWXZDeXR3UlhkKzkvOTlwSVkvSDFHTmtDQzA4MzBFK1RFNnZ2MGtxNnk0UGtDMWpwVgpYZXNNL3kyd
DlDNzFHV25DdGJZVi9MdzFoOVgrSXg2R3FQTzNaNUJ6KzJxOXlkUUtEbjAzeXFqT0hialdqVUMvCm5tZGVacVVRSEFvQ1BMWmlmM3FzRDRxNzE0dzJ5Y0prRlBHcTd4Q1RHNUN4Z0pFaW5rMDlCOWFndVVCZE
hOLzkKSmJJRW9nQkg5dFlyWVJKYUVwMDAzM09HUjVLRWxCWmxMSkxqcHlGYVRDdkxWdUdRZWc1OHdreHUvdz09Ci0tLS0tRU5EIENFUlRJRklDQVRFIFJFUVVFU1QtLS0tLQo=","signerName":"kuberne
tes.io/kube-apiserver-client","usages":["client auth"]}}
  creationTimestamp: "2022-12-18T12:24:29Z"
  name: Tashik
  resourceVersion: "35380"
  uid: 5c27b2f2-4528-4a8b-9c24-6f14fe35d4fa
spec:
  expirationSeconds: 31536000
  groups:
  - system:masters
  - system:authenticated
  request: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ1p6Q0NBVThDQVFBd0lqRVBNQTBHQTFVRUF3d0dWR0Z6YUdsclROHdEUVlEVlFRS0RBWkVaWFpQY0hNdwpnZ0VpTUEwR0
NTcUdTSWIzRFFFQkFRVUFBNElCRHdBd2dnRUtBb0lCQVFER0QxTmJJL3c0RE5UdFpUbmgramFoCktWOUFmUzBMamNzbFJteHdvY3VELzljQ3R6eGtya2lxcFRQVS9tUlh2Zk13R1lKU2NWVE8zM0ExS0kzVHQ
```

To get the signed certificate in base64 encoded json format file,

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

Important notes:
- Send this Signed_certificate <.crt file>, along with **cluster's ca.crt file** to the user so that he can generate his kubeconfig himself **using ca.crt, user.key and signed_certificate.crt** OR the cluster admin himself can generate his kubeconfig file for him. This kubeconfig will contain the **Authentication logic** (This signed certificate), and the user's information.

Generating Kubeconfig

- If you are using AKS, EKS, etc clusters and you do not have access to the master node then the process of generating the Kubeconfig file is different but if you are using kubeadm, minikube or other cluster and you have access to the master node then the process of generating the kubeconfig file is different.

(A) **If you do not have access to the master node (AKS, EKS, etc) , follow this process to generate the Kubeconfig file for new users.**

1. Create a copy/backup of your existing/admin Kubeconfig file because the commands that we will be running here will append the existing Kubeconfig file and add the new user's certificate, context, and other information in the same admin's kubeconfig file. Once the user's information is added, we will delete the admin's context, and other admin information from the kubeconfig and then send it to the user so that he cannot use the admin context from the kubeconfig. We will then replace the backup/copy of kubeconfig to our .kube folder again to use admin kubectl commands.

Important Note: kubectl config view command does not show the certificate content (it will show DATA+OMITTED, REDACTED) but they are present in the kubeconfig file. If you run the "cat ~/.kube/config" command, you will see the certificate content as well. Do not share the content of "kubectl config view" to users because it does not contain certificate content. Always share the kubeconfig file output generated from the cat command.

2. Add new credentials using the following command,

```
kubectl config set-credentials <username here> --client-key=<user key file>.key --client-certificate=<user cluster signed certificate>.crt --embed-certs=true
```

```
tashik.moin@M-23VHCW aks % cat Tashik.key
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAxg9TWyP8OAzU7WU54fo2oSlfQH0tC43LJUZscKHLg//XArc8
ZK5IqqUz1P5kV73zMBmCUnFUzt9wNSiN07eG46YJ/Tn+kZNXoD+7wOmk/DsD5a3h
Ih6V0hn22ZOBggVa/O2U5Cjw/dYZM0jDo9kgh6IAJBxyecjj1IYyZmaTycxlbXHY
eWYUVfDKEoYnRZckbcoEdzbmwclflB2pg5aipvJSgvGAcphkK2o4VV0u7SZjbxbf
X25laI7GOQE5GpbZYMTPv+ZjTL81ClAvpYrGqSdAnUCOPb2x2NUEcLGgrSuRpHmx
ipu8rr0TUjZO3D6yoQPQdRKN1/0Qc2FusxLF3wIDAQABAoIBAQC5a+Gi4tae9sxl
D66X8/MvABePvWDtfyrfU8Xz0FVC18LIZRliPJQFRjJ3Rk5Uop1Iu6j5W+aE4kQf
UD8vJZNgDrMtYh0E1sdjXCd/XH28MdfdL+MkWTP0BaDHY0QMoMzlG9uXmwi667Se
AZI/E+hTcoPlSsiVYVCCR7xph8D2WwI2KRBL99cphUEQUAnwP351sZtPnqYbD6Lk
+m735Ud07uPNsSTiyI9+rWd4ougw6ZtlYdUebD5CKiNNo6wmw2/82SG8Pf4lDSIE
We2YLKuKfWcufLIH7uO3uEQdqmu9Yv9sOx6HC6jYS6nXZ6alneTKR1y3C1FqoWuU
J293NFgBAoGBAOQ5f1mOsbo/VRJV7GXxagIQAhu4sr/88Bqxaiy72X1sDzRnZ+JZ
NqDNCBQqmrg0ltlRpYEuFL8BwBEk1vXIit7nLZSfW5XiXZx80/6CYSiFaAlfNp3A
AL8vAEeGfE2U3UmxsPxZSAsgTWNOfpUDRwjepJDo8gY6Zghm9GiihoHfAoGBAN4q
BieGYs8z984guegubjzvhh9OZhYDfZi7eYL4g3079lQgLoSvZEg7DTPK6shieczA
yejABFMwXjQzjwHkAj7CiXjU7i4Q1Mi+CS3LeCTr2EbeW2Id1960PtgArLV6Scvl
idammdQc2y+KaQmoJcvL/CJzhaWaZhNJZLwRZDwBAoGBALn3SBgXuCaFWLK6JQjk
cHcByFYCaFUcrCF0E4oj/5K9rJ4M7I96FpNsvnqARNZ4s7qdVvkuLWV+7ZyKokZg
hRgg0/lY6O33KERNSwkGAFlea2nHld9xBqrCh73AjiVBeheEnSzjK1s1lRmoamjN
5OILWZl112PPzjhmfB2O4X+NAoGBANygCSqZR2kT58S5NUYBThKPn7dZSQIVh1X0
39AT8aidspuKtgeokkM0+YB0xEIpoWGYFytf/SuL/hGDHkK3Zw9ia9kmRD5VEqsk
JcNnq7Gm0UpXqZK/plAQ2dPyn5E9lym4kvEq09sUqQsGPheL97TypzqjVJbHxRTk
LvihoNQBAoGATx3Y7oLTuG1kvuPJZnnSweGG2ymxMeTG7JSIvxQxhGdrgkkfZJLj
/3NEKWA24QlhJ68Ag2+IXjuclnawvih2fWVSdfwpROgN05lEKV1AxjP6ZyimGkIk
DMLoLL/GV0uUzqtYT54ajG4zg3phek1aJI5bz+CQd3NRNsSvaCtOlaQ=
-----END RSA PRIVATE KEY-----
```

```
tashik [ ~ ]$ ls
clouddrive  Tashik.key  Tashik_Signed.crt
tashik [ ~ ]$ kubectl config set-credentials Tashik --client-key=Tashik.key --client-certificate=Tashik_Signed.crt --embed-certs=true
User "Tashik" set.
tashik [ ~ ]$
```

Now add new context,

```
kubectl config set-context <context name> --cluster=<cluster name here>
--user=<user name here>
```

For example,

```
kubectl config set-context Tashik --cluster=K8 --user=Tashik
```

Now switch context to check if the user has permissions or not,

```
kubectl config use-context <context name here>
```

For example,

```
kubectl config use-context Tashik
```

```
tashik [ ~ ]$ ls
clouddrive  Tashik.key  Tashik_Signed.crt
tashik [ ~ ]$ kubectl config set-credentials Tashik --client-key=Tashik.key --client-certificate=Tashik_Signed.crt --embed-certs=true
User "Tashik" set.
tashik [ ~ ]$ kubectl config set-context Tashik --cluster=K8 --user=Tashik
Context "Tashik" created.
tashik [ ~ ]$ kubectl config use-context Tashik
Switched to context "Tashik".
tashik [ ~ ]$
```

```
tashik [ ~ ]$ kubectl get pods
Error from server (Forbidden): pods is forbidden: User "Tashik" cannot list resource "pods" in API group "" in the namespace "default"
```

Run “kubectl config view” to view the kubeconfig file. As you can see the admin user (clusterUser_K8_K8) certificate and his context is also present here in this kubeconfig file.

```
tashik [ ~ ]$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://k8-dns-85abbf42.hcp.westus.azmk8s.io:443
  name: K8
contexts:
- context:
    cluster: K8
    user: clusterUser_K8_K8
  name: K8
- context:
    cluster: K8
    user: Tashik
  name: Tashik
current-context: Tashik
kind: Config
preferences: {}
users:
- name: Tashik
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
- name: clusterUser_K8_K8
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: REDACTED
tashik [ ~ ]$
```

We will now delete this information from the kubeconfig file manually **OR** using the following commands.

// command to **delete user** from config file

`kubectl config delete-user <username>`

```
tashik [ ~ ]$ kubectl config delete-user clusterUser_K8_K8
deleted user clusterUser_K8_K8 from /home/tashik/.kube/config
tashik [ ~ ]$
```

// command to **delete user context** from config file

`kubectl config delete-context <context name>`

```
tashik [ ~ ]$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://k8-dns-85abbf42.hcp.westus.azmk8s.io:443
  name: K8
contexts:
- context:
    cluster: K8
    user: clusterUser_K8_K8
  name: K8
- context:
    cluster: K8
    user: Tashik
  name: Tashik
current-context: Tashik
kind: Config
preferences: {}
users:
- name: Tashik
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
tashik [ ~ ]$ kubectl config delete-context K8
deleted context K8 from /home/tashik/.kube/config
tashik [ ~ ]$
```

Important Note: kubectl config view command does not show the certificate content (it will show DATA+OMITTED, REDACTED) but they are present in the kubeconfig file. If you run the “cat ~/.kube/config” command, you will see the certificate content as well. Do not share the content of “kubectl config view” to users because it does not contain certificate content. Always share the kubeconfig file output generated from the cat command.

Run the cat command and then share the content of the kubeconfig file with the user.

(B) if you have access to the master node, cluster ca.key and ca.crt file (kubeadm, minikube, etc) then you/user can do the following to generate the kubeconfig file.
- Required files for generating the kubeconfig file for a user are ca.crt, signed_certificate.crt, and user.key.
- Run following commands,
kubectl --kubeconfig <name of config file> config set-credentials <username> --client-certificate <path to user's signed certificate> --client-key <path to user's key>

kubectl --kubeconfig <name of config file> config set-context <context name> --cluster <cluster name> --namespace <default namespace name> --user <username>

Currently, the user can **only successfully authenticate** with the cluster using the kubeconfig file but he **cannot execute any kubectl commands** because he is **not authorized to perform any action yet** because we did not bind any role to this newly created/registered user.

Let's switch to admin's context (admin kubeconfig file replacement) so that we can execute kubectl commands to create a role, cluster role, role binding and cluster role binding.

Creating a Kubernetes **Role and Role binding** to give user the access of only namespace scoped resources in a specific namespace

Let's create a role using the kubectl apply command,

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: DevOps
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
tashik.moin@M-23VHCW rbac %
```

Lets bind this role to a group using role binding. A role binding can be done on users as well as groups. Since our user is part of DevOps group, we will bind this role with the **group instead of binding with a user** but we can also bind it with user directly but in that case, if we want to assign the same role to multiple users, then we will have to make separate role bindings for each user.

For role and role bindings we have to **tell the namespace** because they are applied on namespace scoped resources of a specific namespace only.

// To apply role binding on **group**, use Group as kind
// To apply role binding on a **specific user** only, use User as kind
// To apply role binding on a **service account** only, use ServiceAccount as kind
Lets apply role binding on a group,

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: DevOps-RB
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: Group
  name: DevOps # "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: DevOps # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

Role successfully bound with the DevOps group using rolebinding and since the user “Tashik” was part of DevOps group (by providing his subject CN, O while generating his key, certificate) is now part of this group having this role therefore Tashik will have only those access that are defined in this role.

See yaml files in the same rbac directory, modify them and try yourself!

Service Accounts

- Service accounts are used to manage kubernetes resources from inside of a pod.

- **Use cases?**

1. CI/CD pipelines may create a pod in your kubernetes cluster to manage your kubernetes resources like deployments, services, pods etc.
2. You may deploy some pods in your cluster that maintain your cluster nodes and perform other cluster operations.

- **Important Note**: We can assign Role, ClusterRole to give permissions to a serviceaccount to perform certain accounts from inside of the pod for e.g during CICD, other cluster operations, etc

- When a namespace is created, automatically a service account with name “default” is also created inside that namespace. If you do not specify any service account in your pod/deployment manifest files, then by default this default service account is used by every pod running in that namespace.

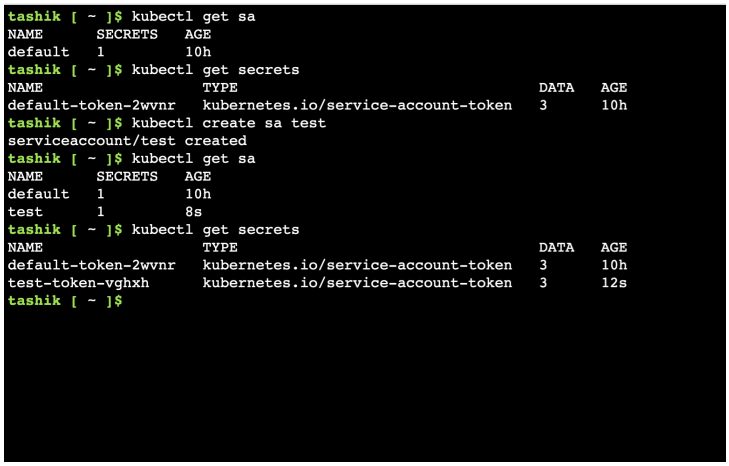
Use the default service account to access the API server

When Pods contact the API server, Pods authenticate as a particular ServiceAccount (for example, `default`). There is always at least one ServiceAccount in each namespace.

Every Kubernetes namespace contains at least one ServiceAccount: the default ServiceAccount for that namespace, named `default`. If you do not specify a ServiceAccount when you create a Pod, Kubernetes automatically assigns the ServiceAccount named `default` in that namespace.

Secret mounting into a service account

Whenever a service account is created, a secret with 3 data fields (token, namespace, ca certificate) is also created and mounted into the service account.



How to link pods with your custom service account?

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
```

When a pod is linked with the service account, the same service account secret is also mounted inside the pod at the following directory,
/var/run/secrets/kubernetes.io/serviceaccount

```
root@ip-172-31-30-83:/home/ubuntu/sa# kubectl exec -it test-pod -- /bin/bash
root@test-pod:/# ls /var/run/secrets/kubernetes.io/serviceaccount
ca.crt  namespace  token
```

Let say, a service account is linked to a role via rolebinding to get some permissions like for example “listing all the pods” then from inside of a pod running in our cluster, we can list the pods using kubernetes api pod url. A kubernetes service called “Kubernetes” inside the default namespace is used when a pod tries to communicate with the kubernetes API server. This service exposes all the resource endpoints like for .eg, the endpoint for listing the pods from this service is something like this,
https://kubernetes/api/v1/namespaces/default/pods

If a service account has a role bound to it having permissions to list the pods, then if we try to curl the pod kubernetes api endpoint we will get list of pods from inside of the pods,

curl https://kubernetes/api/v1/namespaces/default/pods -k --header Authorization: Bearer <Secret token value mounted at **/var/run/secrets/kubernetes.io/serviceaccount** directory**>**

```
}root@test-pod:/# curl https://kubernetes/api/v1/namespaces/default/pods -k --header "Authorization: Beare
 $TOKEN" | jq '.items[].metadata.name'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 20036    0 20036    0     0   416k      0 --:--:-- --:--:-- --:--:--  416k
"some-deployment-6dfcc5b79f-cvjtc"
"some-deployment-6dfcc5b79f-hvbqp"
"test-pod"
```

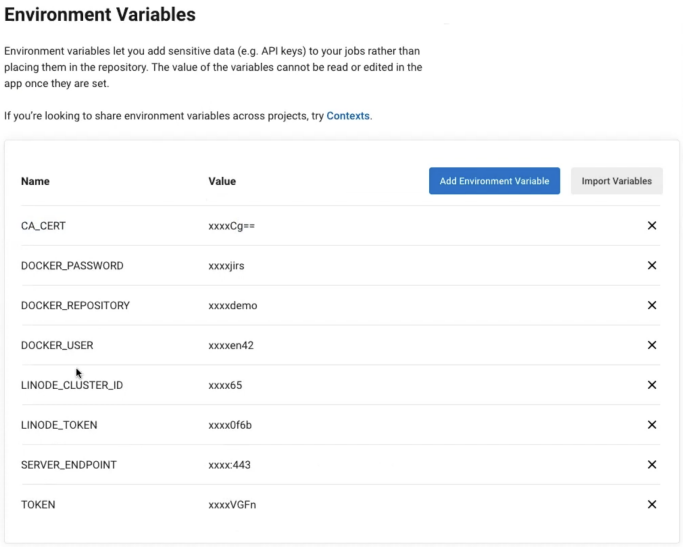
Few important points

- Created in a namespace
- Used to allow a process in a pod access to the API Server
- Default service account = default (no access to API Server)
- Create your own service account
  - use it in a RoleBinding or ClusterRoleBinding
  - use the service account secret to obtain the authentication token and CA certificate

Configuring kubeconfig using service accounts to run kubectl commands from inside of a pod during the CI/CD process,

```
20  jobs:
21  > test: …
28  > build_docker: …
39    deploy_linode:
40      executor: python
41      steps:
42        - checkout
43        - kubernetes/install-kubectl
44        - run:
45            name: Set up LKE kubeconfig
46            command: |
47              export DECODED_TOKEN=$(echo ${TOKEN} | base64 -d)
48              echo $CA_CERT | base64 -d > ca.crt
49              kubectl config set-cluster sa-demo --server=${SERVER_ENDPOINT} --certificate-authority=ca.crt
50              kubectl config set-credentials sa-demo --token=${DECODED_TOKEN}
51              kubectl config set-context cci --user=sa-demo --cluster=sa-demo
52              kubectl config use-context cci
53  >     - helm/install-helm-client: …
55  >     - helm/upgrade-helm-chart: …
60  >     - run: …
65
66  > executors: …
```

Here,
${TOKEN} → Path to service account secret token
${SERVER_ENDPOINT} → kubernetes api server endpoint/url
${CA_CERT} → The service account ca certificate in non-encoded form



Running the CICD pipeline to test if we can execute kubectl commands from the pod using the service account or not,\