API Groups

In k8s, API groups are a way of organizing related resources and operations together. This allows for easier discovery and usage, and also helps to avoid naming conflicts between different resources. When k8s was first introduced, all the resources like Pod, Service, ReplicationController, etc., were all part of a single group, the "core" group, and were accessed at the path /api/v1.As k8s evolved and more resources were added, it became clear that this single group was not scalable. So, the concept of API groups was introduced

Kubernetes uses a versioning scheme to facilitate the evolution of its API. There are three types of versioning in Kubernetes:

Alpha: This is the first stage of the development of a new API. Alpha APIs may be unstable, change significantly after the initial release, and may not even be enabled in your clusters.

Beta: This is the second stage. Beta APIs are well-tested and are enabled by default in your clusters. However, they may still undergo changes, such as in the form of bug fixes or feature enhancements.

Stable: This is the final stage. Stable APIs appear in released software for many subsequent versions

The version of an API group is represented by vXalphaX (e.g., v1alpha1), vXbetaX (e.g., v2beta2), and vX (e.g., v1) for alpha, beta, and stable versions, respectively.

Kubernetes API groups are divided into two categories

Named API Groups

Named API groups are additional API groups introduced to extend the functionality of Kubernetes beyond the core resources. Each named API group focuses on specific features or functionalities and manages specialized resources related to those features. The Named API group is accessed using the /apis endpoint

Core API Group

The core API group, also referred to as the "v1" group, contains the essential resources that are fundamental to the functioning of a Kubernetes cluster. It includes resources such as Pods, Services, Namespaces, ConfigMaps, Secrets, and more. The core API group is accessed using the /api/v1 endpoint

apps: This group contains resources related to running applications on Kubernetes. It includes Deployment, ReplicaSet, StatefulSet, and DaemonSet.

batch: This group includes resources for batch processing and job-like tasks. It includes Job and CronJob.

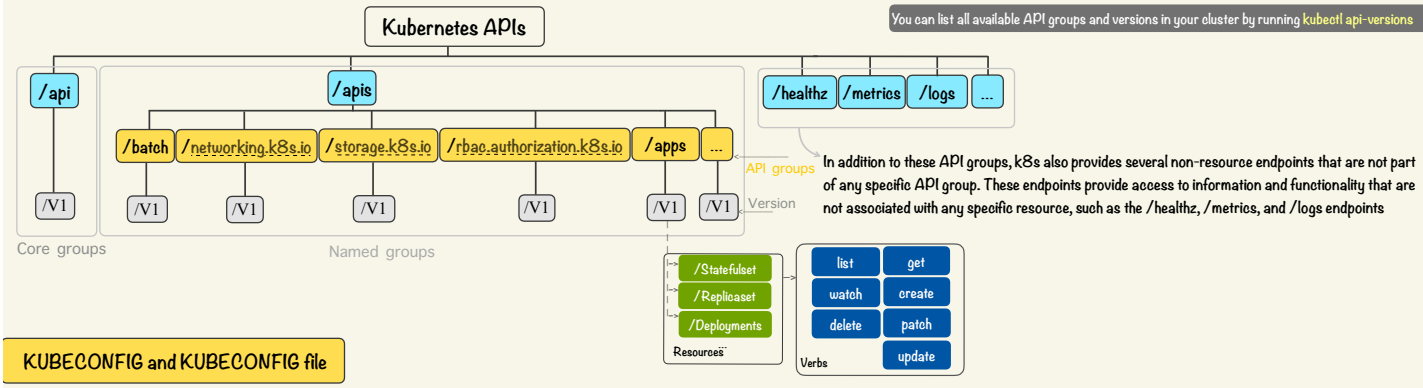
rbac.authorization.k8s.io: This group contains the Role, ClusterRole, RoleBinding, and ClusterRoleBinding resources for handling role-based access control (RBAC) in Kubernetes.

networking.k8s.io: This group contains resources related to networking in k8s, such as NetworkPolicy and Ingress.

storage.k8s.io: This group contains resources related to storage, such as StorageClass, VolumeAttachment, and the CSINode driver

New resources are accessed at the path /apis/{group}/{version}.For example, to access the Deployment resource, which is part of the apps group, you would use the path /apis/apps/v1/deployments.

You can list all available API groups and versions in your cluster by running kubectl api-versions



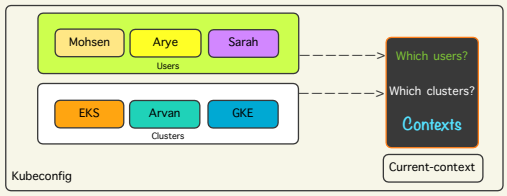
In addition to these API groups, k8s also provides several non-resource endpoints that are not part of any specific API group. These endpoints provide access to information and functionality that are not associated with any specific resource, such as the /healthz, /metrics, and /logs endpoints

KUBECONFIG and KUBECONFIG file

The KUBECONFIG environment variable is used to specify the path to the Kubernetes configuration file, which contains information about the cluster, user, and context used by kubectl and other Kubernetes command-line tools. The KUBECONFIG file can contain multiple contexts, each representing a different cluster and namespace. The KUBECONFIG file is typically stored in the user's home directory at the path ~/.kube/config on Unix-based systems

```
kubectl --kubeconfig=/path/to/my-kubeconfig/my-kubeconfig.yaml get pods
```

This command uses the specified KUBECONFIG file instead of the default ~/.kube/config file



Contexts section defines the mapping between the Kubernetes cluster(s) and the users who can access them. A context includes the cluster and user information, as well as a reference to a default namespace and a name for the context

```
apiVersion: v1
Kind: Config
Clusters:
- name: GKE
  cluster:
    certificate-authority: ca.cert
    servrer: https://k8s-endpoint:6443
contexts:
- name: Arye@GKE
  context:
    cluster: GKE
    user: Arye
    namespace: dev
users:
- name: Arye
  user:
    client-certificate: arye.crt
    client-key: arye.key

current-context: Arye@GKE
```

new.kubeconfig

The default namespace to use for this context.

To create a kubeconfig file using kubectl, you can follow these steps:

Set the cluster details | **Set the user credentials** | **Set the context** | **Use the context**

```
kubectl config set-cluster GKE --server=https://k8s-endpoint:6443 --certificate-authority=ca.crt --embed-certs --kubeconfig new.kubeconfig
```

If embed-certs=false

- Set a cluster entry in kubeconfig
- Cluster name
- Server Address
- K8s CA certificate for TLS verification
- Embeds the certificate data directly in the kubeconfig instead of linking to a file
- kubeconfig file that will be created with this new entry.

```
kubectl config set-credentials Arye --client-key=/path/to/arye.key --client-certificate=/path/to/arye.crt --embed-certs --kubeconfig new.kubeconfig
```

If you use a token

- User name
- user key file
- user certificate file

```
kubectl config set-credentials Arye --token= eyJhbGciOiJSUzIlNiIsIm...Lz9APOb2rsWHr9HWA --kubeconfig new.kubeconfig
```

If you use a client certificate

```
kubectl config set-context Arye@GKE --cluster=GKE --user=Arye namespace=dev --kubeconfig new.kubeconfig
```

- context name
- default namespace

```
kubectl config use-context Arye@GKE
```

you can specify a different kubeconfig file by setting the KUBECONFIG environment variable.

```
export KUBECONFIG=new.kubeconfig
```

Cluster Scope in Kubernetes

In Kubernetes, resources are divided into two categories based on their scope: Namespaced and Cluster-scoped

Namespaced resources: These resources exist and operate within a namespace. They can have different configurations and states in different namespaces

configmaps | Role | PVC | Rolebinding | Deployment | Replicaset | Pods | Services | Jobs

```
kubectl api-resources —namespace=true
```

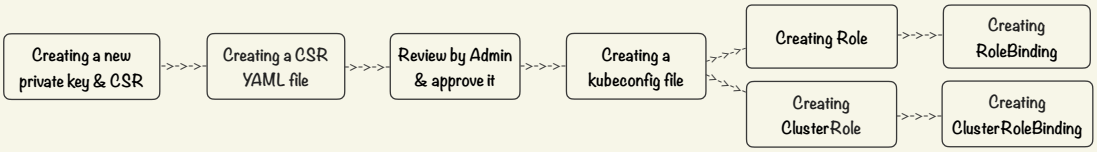
Cluster-scoped resources: These resources exist and operate across the entire cluster. They are not confined to any particular namespace

nodes | PV | Clusterroles | Clusterrolebinding | Namespace

```
kubectl api-resources —namespace=false
```

```
kubectl api-resources
NAME.                SHORTNAMES.   APIVERSION.                NAMESPACED.   KIND
...
endpoint             no            v1                         true          Endpoints
Pods                 po            v1                         false         Pod
Service              svc           v1                         true          Service
deployment           deploy        apps/v1                    true          Deployment
Ingresses            ing           extensions/v1beta1         true          Ingress
...
```

How to create a new admin or developer user account for accessing to a k8s cluster with X.509 ?



1 Creating a new private key & a csr file by new user

Generate a private key for the user using OpenSSL. The private key is used as part of the user's credentials to authenticate with the Kubernetes API server

```
openssl genrsa -out mojtaba.key 2048
```

Create a CSR for the user using the private key

```
openssl req -new -key mojtaba.key -subj "/CN=mojtaba" -out mojtaba.csr
```

The CSR includes the user's identifying information and the public key associated with the private key

2 Creating a new CSR yaml file and Sign the CSR using the Kubernetes CA

Create a CertificateSigningRequest object in Kubernetes that includes the user's CSR and submit the CSR to the Kubernetes cluster

csr-mojtaba.yml

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: mojtaba
spec:
  groups:
    - system:authenticated
  request: LS0tLS1CRUdJTjkKNUlEdC9BWT.....OKLS0tLS1FTkQ...
gQ0VSVElGSUNBVEUgUkVRUVTVC0tLS0tCg==
  signerName: kubernetes.io/kube-apiserver-client
  usages:
    - client auth
```

```
request: $(cat user-name.csr | base64 -w 0)
Or
cat mojtaba.csr | base64 -w 0
```

The signerName specifies the Kubernetes CA that will sign the certificate.
The usages field specifies that the certificate will be used for client authentication.

```
kuectl apply -f csr-mojtaba.yml
```

3 submit the CSR to the Kubernetes cluster and approve it

Once the CSR is submitted, it needs to be approved by a cluster administrator.

```
k get csr
NAME      AGE   SIGNERNAME                            REQUESTOR           CONDITION
mojtaba   33m   kubernetes.io/kube-apiserver-client   kubernetes-admin    Pending
```

```
k describe csr mojtaba
Name:         mojtaba
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"certificates.k8s.io/
v1","kind":"CertificateSigningRequest","metadata":{"annotations":{},"name":"mojtaba"},"spec":{"groups":
["system:authenticated"],"request":"LS0tLS1CRUdJTjkKNUlEdC9BWT0KLS0tLS1FTkQgQ0VSVElGSUNBVEUgUkVRVU
VTVC0tLS0tCg==","signerName":"kubernetes.io/kube-apiserver-client","usages":["client auth"]}}

CreationTimestamp:  Sat, 11 Jun 2022 17:51:33 +0000
Requesting User:    kubernetes-admin
Signer:             kubernetes.io/kube-apiserver-client
Status:             Pending
Subject:
         Common Name:          mojtaba
         Serial Number:
         Organization:         StarkWare
         Organizational Unit:  blockchain
         Country:              IL
         Locality:             haifa
         Province:             haifa
Events:  <none>
```

```
k certificate approve mojtaba
```

```
k get csr
NAME      AGE   SIGNERNAME                            REQUESTOR           CONDITION
mojtaba   39m   kubernetes.io/kube-apiserver-client   kubernetes-admin    Approved,Issued
```

This command notifies the Kubernetes CA that the CSR has been approved and requests a signed certificate for the user. The signed certificate is then stored in the status.certificate field of the CertificateSigningRequest object

4 Export the issued certificate from the CertificateSigningRequest.

you retrieve the signed certificate for the user

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

5 Create a kubeconfig File for the User

Create a kubeconfig file for the user that includes the cluster details, user credentials, and context. The certificate-authority-data field contains the base64-encoded CA certificate for the Kubernetes cluster.

```
apiVersion: v1
kind: Config

current-context: mojtaba@cka

clusters:
  - name: cka
    cluster:
      server: https://kubemaster-1:6443
      certificate-authority: ca.crt

users:
  - name: mojtaba
    user:
      client-certificate: mojtaba.crt
      client-key: mojtaba.key

contexts:
  - name: mojtaba@cka
    context:
      cluster: cka
      user: mojtaba
      namespace: dev
```

```
apiVersion: v1
kind: Config

current-context: mojtaba@cka

clusters:
  - name: cka
    cluster:
      server: https://kubemaster-1:6443
      certificate-authority-data: <base64-encoded CA certificate data>

users:
  - name: mojtaba
    user:
      client-certificate-data: <base64-encoded client certificate data>
      client-key-data: <base64-encoded client key data>

contexts:
  - name: mojtaba@cka
    context:
      cluster: cka
      user: mojtaba
      namespace: dev
```

To become independent from external files in the configuration, you can use the data field directly within the configuration file

certificate-authority-data	client-certificate-data	client-key-data
cat /etc/kubernetes/pki/ca.crt | base64 -w 0	cat mojtaba.csr | base64 -w 0	cat mojtaba.csr | base64 -w 0

5.1 If you don't want to create a kubeconfig manually, you can create a kubeconfig using kubectl

```
kubectl config set-cluster cka --server=https://kubemaster-1:6443 --certificate-authority=ca.crt --embed-certs --kubeconfig devuser.kubeconfig
kubectl config set-credentials mojtaba --client-key=/path/to/mojtaba.key --client-certificate=/path/to/mojtaba.crt --embed-certs --kubeconfig devuser.kubeconfig
kubectl config set-context mojtaba@cka --cluster=cka --user=mojtaba namespace=dev --kubeconfig devuser.kubeconfig
kubectl config use-context mojtaba@cka
```

6 Set Up Role-Based Access Control (RBAC) for the User

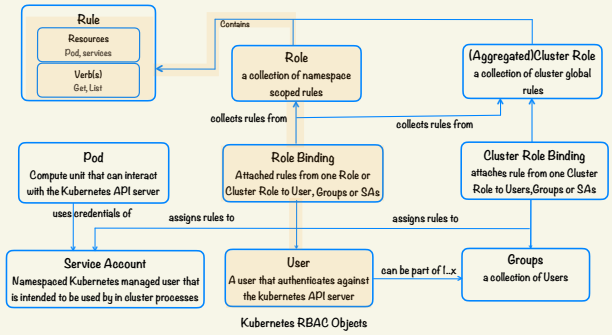
In this final step, you create a role and role binding to grant the user permissions in the Kubernetes cluster

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
  namespace: dev
rules:
  - apiGroups: [ "" ]
    resources: ["pods"]
    verbs: ["list" , "get" , "create" , "update" , "delete"]

  - apiGroups: [ "" ]
    resources: ["configMap"]
    verbs: ["create"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: mojtaba-developer
  namespace: dev
roleRef:
  apiGroup: "rbac.authorization.k8s.io"
  kind: "Role"
  name: "developer"
subjects:
  - apiGroup: "rbac.authorization.k8s.io"
    kind: "User"
    name: "mojtaba"
```

The reason for having two separate rules in the Role definition is that the two resources, "pods" and "configMap", have different permissions requirements



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