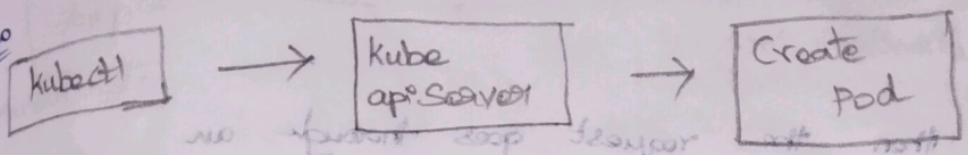
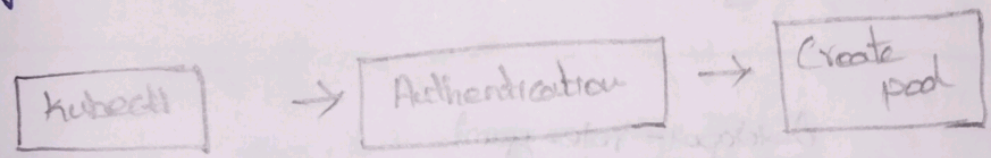
83 (2025 updates) Admission Controllers

1:40



⇒ we've been running commands from our command line using the "kubectl" utility to perform various kinds of operations on our k8s cluster

⇒ Every time when we send a request, say, to create a pod, the request goes to the API server & then the pod is created & the information is finally persisted in the Etcd database.



⇒ When the requests hits the API server, we've learned that it goes through an authentication process & this is usually done through certificates

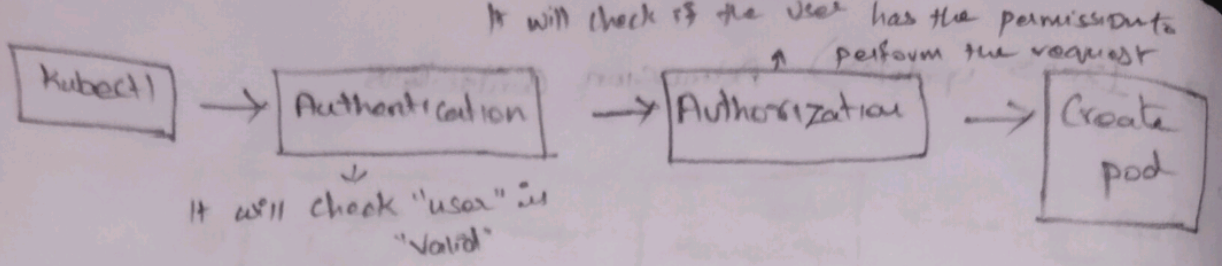
⇒ If the request was sent through kubectl, we know the Kube config file has the certificates configured & the authentication process is responsible for identifying the user who send the request & making sure the user is valid

[apiserver has kubeconfig file which has certificate which contains process to check whether the user is valid]

```
> cat .kube/config

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:

    server: https://10.10.10.15:6443
```



And then the request goes through an "authorization process" this is when we check if the user has permission to perform that operation this is achieved through rbac - role based access control

⇒ So if a user is assigned a particular role of a "developer", the user is allowed to * list * update, * get, * create, * delete pods

developer - role.yaml

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

⇒ And so, if the request that came in matched any of these conditions, in this case it does a request is to create a pod, it is allowed to go through, otherwise it is rejected. So this is "authorization" with "role-based access control"

Authorization - RBAC

⇒ with RBAC, you could place in different kinds of restriction. Such as "to allow" or "to deny" those with a particular role

* To create a list (or)
* To delete different kinds of objects like pods, deployments (or) services

* We could even restrict access to specific resource names, such as a developer can only work on pods named "blue" (or) "Orange" (or) restrict access within a namespace alone

developer-role.yaml

```
apiVersion : rbac.authorization.k8s.io/v1
kind: Role
metadata:
   name : developer
rules:
- apiGroups : [" "]
  resources : ["pods"]
  verbs : ["create"]
  resourceNames : ["blue", "orange"]
```

* Can list POD, Deployments / services / ....
* Can create PODs / Deployments / services / ...
* Can delete PODs / Deployments / Services / ....
* Can create pods named blue or Orange
* Can create pods within a namespace.

→ Most of the rules that you can create with "role-based access control" is @ the K8S API level $ what user is allowed, access to what kind of API operations & it doesnot go beyond that.

what if we want to do more than just define what kind of access a user has to an object

web-pod.yaml

```
apiVersion : v1
kind: Pod
metadata:
    name: web pod ✗
spec:
  containers:
    -name: ubuntu
     image: ubuntu-latest ✗
     command: ["sleep", "3600"]
     securityContext:
       runAsUser: 0 ✗
       Capabilities:
         add: ["MAC_ADMIN"]
```

For eg:-
When a pod creation request comes in, you'd like to review the configuration file & look at the image name & say you do not want to allow images from public docker

hub registry, only allow images from a specific internal registry. (or) to enforce that we must never use the latest tag (or) only permit images from certain registry. for any images. For-eg:

⇒ If you say the container is running as the root user, then you do not want to allow that request. (or) runAsUser: 0 [do not permit runAs root user]

⇒ Allow certain Capabilities only permit certain capabilities (or) to enforce that the metadata always contains labels

add: ["MAC_ADMIN"] {pods always has labels}

* Only permit images from certain registry.
* Do not permit runAS root user
* Only permit certain capabilities.
* Pod always has labels.

These are the some of the things that you cannot achieve with the existing rbac & this is where "admission controllers" comes in.
Authorization

Kubectl → Authentication → Authorization → Create pod

Kubectl → Authentication → Authorization → Admission Controllers → Create pods

⇒ Admission controllers help us implement better security measures to enforce how a cluster is used.

⇒ Apart from simply validating configuration, admission controllers can do a lot more suggest change the request itself (or) perform additional operations before the pod gets created

⇒ There are number of "admission controllers" that come pre-built with k8s such as always pull images that ensures that everytime a pod is created the images are always pulled

DefaultStorageClass :-

It observes the creation of PVCs & automatically adds a default storage class to them, if one is not specified,

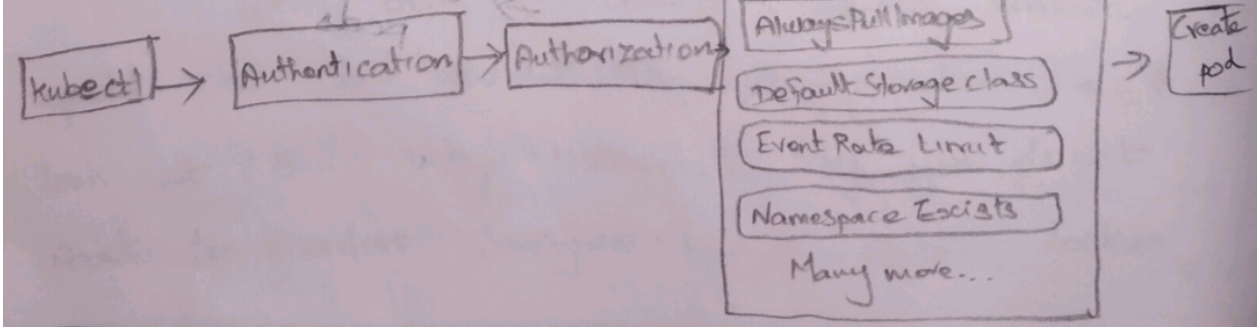
Event Rate Limit :-

It can help set a limit on the request with the API server can handle at a time to prevent the API server from flooding with requests

NamespaceExists

It rejects requests to namespace that do not exist

And there are many more "admission controllers" available



Eg:-
The NamespaceExists admission Controller

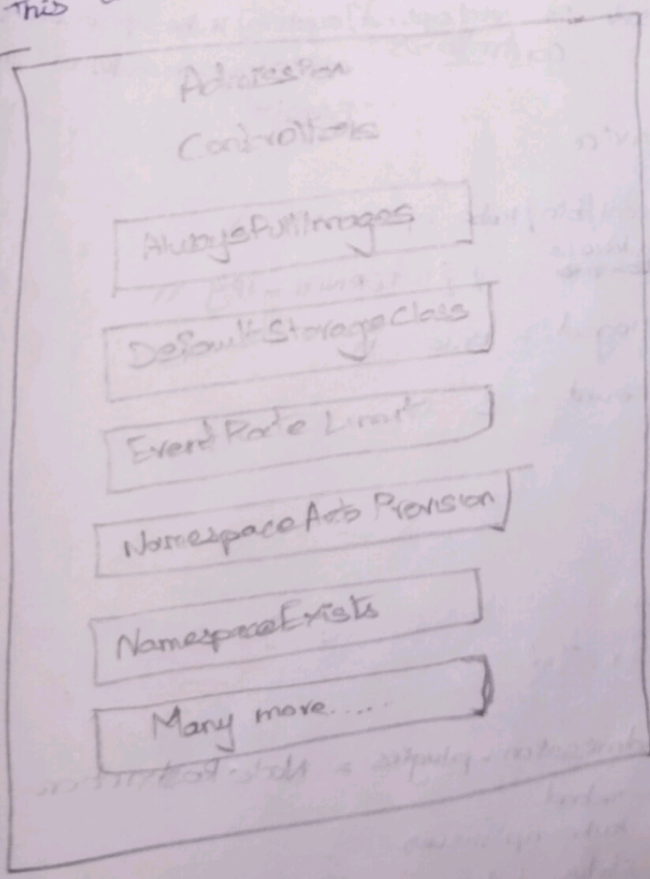
⇒ Say we want to create a pod in a namespace called "Blue" that doesn't exist

```
kubectl run nginx --image nginx --namespace blue.
Error from server (NotFound): namespace "blue" not found.
```

what happening here is, that my requests gets authenticated, then authorized & it then goes through the admission controllers, "NamespaceExists" admission controller handles the request & checks if the "blue" namespace is available. If it is not, the request is rejected

The "namespace exists" is a built-in admission Controller that is enabled by default

There is another admission controller that is not enabled by default, & that is called as NamespaceAutoProvision Admission Controller. This will automatically create the namespace, if it doesn't exist



View Enabled Admission Controllers

> To see list of admission controllers enabled by default run kube-apiserver -h command

```
kube-apiserver -h | grep enable-admission-plugins
```

The command will list admission controllers that are enabled by default, the ones that are highlighted in "green"

NOTE:- If you're running this in a kubeadm based setup then you must run this command within the kube API server control plane pod using the kubectl exec command first like this

```
>kubectl exec kube-apiserver-controlplane -n kube-system
-- kube-apiserver -h | grep enable-admission-plugins
```

Enable Admission Controllers

⇒ Sudo systemctl edit kube-apiserver (or)

⇒ sudo vim /etc/systemd/system/kube-apiserver.service add "admission controller" under "ExecStart"

This approach is for kube API server service

```
kube-apiserver.service
ExecStart = /usr/local/bin/kube-apiserver \\
--advertise-address = ${INTERNAL-IP} \\
--allow-privileged = true \\
--apiserver-count = 3 \\
--enable-admission-plugins = NodeRestriction
```

⇒ Sudo systemctl daemon-reload
⇒ Sudo systemctl restart kube-apiserver
⇒ Sudo systemctl status kube-apiserver.

/etc/kubernetes/manifests/kube-apiserver.yaml

This approach is for the apiserver is running as pod in kubeadm based setup

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.107
    - --allow-privileged=true
    - --enable-bootstrap-token-auth=true
    - --enabe-admission-plugins=NodeRestriction,
    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
```

To add an admission controller, update the enable admission plugins flag on the kube API server service to add the new admission controller

kube-apiserver.service

--enable-admission-plugins=NodeRestricten,NamespaceProvision

From kubeadm setup

Update the flag within the kube API server manifest file as shown below.

/etc/kubernetes/manifests/kube-apiserver.yaml

- --enable-admission-plugins=NodeRestriction,NamespaceAutoprovisio

To disable admission controller plugins

use the "disable admission plugins flag"

kube-apiserver.service.

```
ExecStart = /usr/local/bin/kube-apiserver \
  --advertise-address = ${INTERNAL_IP} \
  --allow-privileged = true \
  --apiserver-count = 3 \
  --authorization-mode = Node, RBAC \
  .
  .
  .
  --enable-admission-plugins = NodeRestriction, NamespaceAutoProvision
  --disable-admission-plugins = DefaultStorageClass
image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
name: kube-apiserver.
```

⇒ Once updated, the next time we run the command to provision a pod in a namespace that does not exist yet

⇒ The request goes through authentication, then authorization & then the Namespace Auto Provision Controller at which point it realizes that the namespace doesn't exist & so it creates namespace automatically & the request goes through successfully to create the pod

⇒ If you list the namespaces now, you'll see the blue namespace is automatically created

```
kubectl run nginx --image nginx --namespace blue
pod/nginx created
```

kubectl get namespaces

NAME	STATUS	AGE
blue	Active	3m
default	Active	23m
kube-public	Active	24m
kube-system	Active	24m

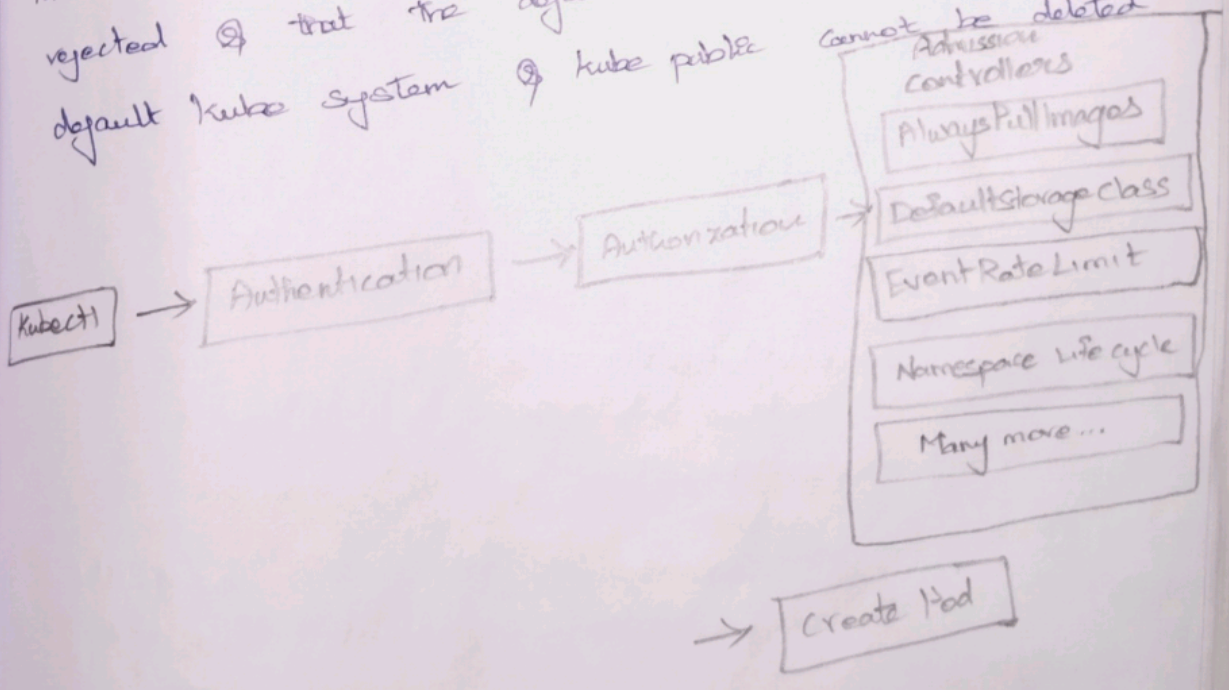
⇒ So, this is one eg. of how an "admission controller" works

⇒ It cannot only validates and rejects request from users, it can also perform operations in backend (or) change the request itself

NOTE:-

NamespaceAutoProvision and NamespaceExists admission Controllers are deprecated & is now replaced by the Namespace Lifecycle Admission Controller

⇒ The "Namespace Lifecycle" admission controller will make sure that requests to a non-existant namespace is rejected & that the default namespace, such as default, kube system & kube public cannot be deleted



Since the kube-apiserver is running as pod you can check the process to see enable & disabled plugins

```
ps -ef | grep kube-apiserver | grep admission-plugins
```