**Authentication:**

securing access to the k8 cluster with authentication mechanism.

security of end users who access the application deployed on the cluster are managed by the application themselves internally.

our focus is on user access for admin purposes

humans such as admins and developers and robots such as other processes or services or application that require access to the cluster.

k8 does not manages users accounts natively it relies on an external source like a file with the users details or certificates or a third party identity service like LDAP to manage these users, so we can not create and list the user like

* **kubectl create user user1**
* **kubectl get user**

**However, in case of service account k8 can manage them, we can create service account user using**

* **kubectl create serviceaccount sa1**
* **kubectl get serviceaccount**

All user access in k8 is managed by kube-apiserver whether we are accessing the cluster using kubectl tool or API directly, all of these requests go through kube api server.

the kube api server authenticates the requests before processing it.

**So how does the kube api server authenticates the request?**

1. we can have a list of username and password in a static password file or username and tokens in static token file or we can authenticate using certificates or we can connect using third party application like LDAP and kerberos

static authentication using password file

we can create a list of users and their passwords in a csv file like user-details.csv and use that as the source of user information. The file has three column password, user and filename.

we can then pass the file name as an option to the kube-api server.

**--basic-auth-file=user-details.csv**

restart the kube api server for these options to take effect.

If we have set up our cluster using the kubeadm tool then we must modify the kube-apiserver POD definition file. Kubeadm tool we automatically restart the kube-api server once we update the file.

In the user-detail.csv file we can also include the 4th column named as group to assign users to the specific group.

2. instead of static password file we can have static token file here instead of password we can specify the token and pass the token file as an option

**--token-auth-file=user-token-details.csv**

this authentication mechanism is not recommended as it stores username and password in clear text.

**Config file has 3 sections:**

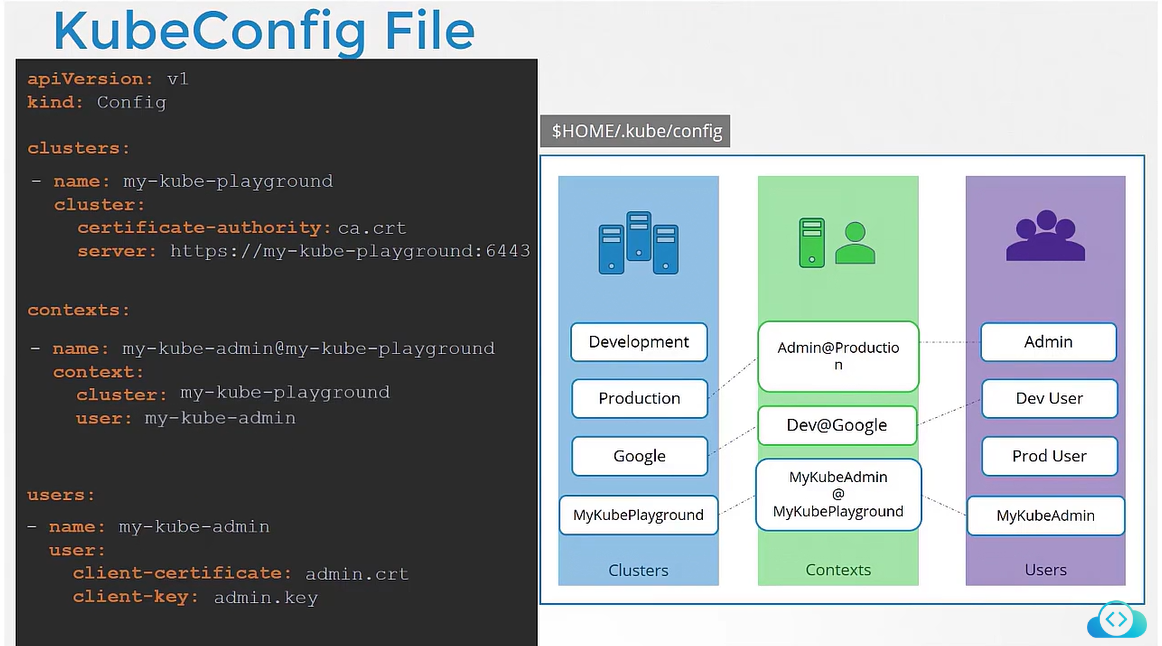
Cluster: Let suppose we have multiple cluster like dev, prod, test environment

Users: Users are user accounts with which we have access to the cluster like admin user or dev user. These users may have different privilages on different cluster.

Contexts: It defines which user account will be used to access which cluster.

Ex: we can create a context named Admin@production that will use admin account to access production cluster.

Remember we are not creating any new user or configuring any kind of user access or authorization in the cluster. With this we are using existing user with existing privileges and defining what user we going to use to access which cluster. That way we don’t have to define the user certificate and server address in each and every kubectl command.



**How does kube config knows which context to use?**

We can specify the default context by adding a filed current context to the kubeconfig file. If we don’t specify the config file then by default it uses the file located at users home directory in the folder.q

CLI option to set kube config:

Check the current context// kubectl config view

**We can set custom kube config file by passing the kubeconfig option**

kubectl config use-context research --kubeconfig=/root/my-kube-config

In this command we are using custom config file names my-kube-config where we specify using the context named research.

**To know the current context, run the command:**

 kubectl config --kubeconfig=/root/my-kube-config current-context

**Update the current context:**

Kubectl config use-context prod-user@production // this will set prod user to access the production cluster

C**an we configure a context to switch to a particular namespace?**

Yes, the context section in the kube config file can take additional filed called namespace where we can specify the namespace so that when we switch to that context then it will automatically switch to that namespace.

**Kubectl proxy and Kube proxy**

Kubeproxy it is used to enable the communication b/w parts and services across different nodes in the cluster.

Kubectl proxy it is an HTTP proxy service which is created by kubectl utility to access the kubepai server.

**Authorization Mechanism (Defined in kube pai server)**

1. **Node** **Authoriser**: It is responsible for authorizing the Api request from kubelet (an agent run on each node and responsible for managing the pods) by interacting with kube-Api server.

The node authorize is responsible for authorizing the requests originated from the kubelet**.**

2. **Attribute based authorization**: It makes the use of attributes to provide access. Attributes can be property or characteristics associated with user, group or resources. for every user we need to add the authorization policy manually.

3. **Role Base authorization**: We create a role for users and associate with right set of permission.

4. Webhook: outsource all the authorisation mechanism to the third party tool. We can make k8 to make a call to the third party tool with the information about the user and his access requirements and let the 3rd party agent decide if the user should be permitted or not. Based on that user is granted access.

We can define more than one authorization mechanism one by one. So if the request is denied by first mechanism then it will go to the next, if the request is approved by the next mechanism then user is provided access.

RBAC:

We create role by creating a role object. We specify rules in role yaml. Each rule had 3 options:

rules:

* apiGroups: [‘“”]
* resources: [“pods”] // pod is the resource on which we want tp apply rule//
* verb: [“list”, “get”, “create”, “update”, “delete”] // list of action that can be done//

resource name: we can also restrict which out of all pos we want to the action to be taken by defining resoursename under verb

* **resourceNames: [“red”] //** it will perform list of action on red pods**//**

**link the user to the role:**

We can an object named as rolebinding. It links user object to the role. It has 2 section.

* Subject: here we specify the user details.
* roleRef: here we specify the details of the role that we created

Note: we need to specify the namespace for role and role binding.

**Roles and role bindings are used to authorize a user for the namespace wide resources like pod, replicaset, deployment etc**

**Check whether we haveaccess to resources In the cluster**

***Kubectl auth can-I create pod*** // check if I can create pod//

***Kubectl auth can-I create pod –as dev-user*** // check if dev user can create pod//

***Kubectl auth can-I create pod –as dev-user –namespace development*** // check if dev user can create pod in development namespace//

**Cluster Roles**:

To authorize users to cluster wide resources like node, PV, we use cluster roles and cluster role binding. These are roles just like normal roles except they are for cluster wide resouces.

***Note: We can also create cluster roles for the resources which are limited to namespace so that user can have access to all the namespace within the cluster.***

**Admission Controller:**

It can not only validate or reject the request from users, it can also perform the operations in the backend or change the request itself.

If a request was sent through kubectl, we know that the kubeconfig file has the certificate configured and the authentication process is responsible for identifying the user who sent the request and make sure the request is valid.

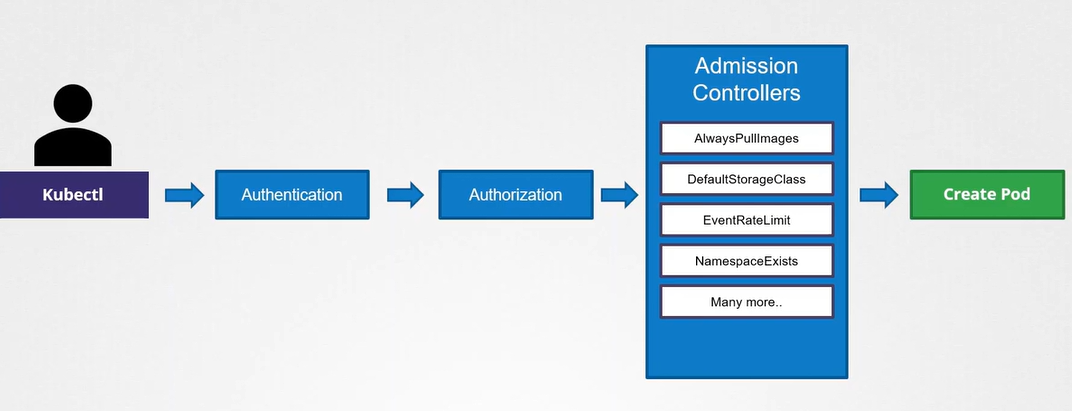
Then the request goes through the authorization process where it is checked if the user has permission to perform the task and this is achieved through role-based access control.

**AC help us to implement better security measures to enforce how a cluster is used. Like only permist images from certain link, do not permit run as root user, only permit certain capabilities, pod always has label.**

**Apart from simply validating the configuration, AC can do a lot more such as change the request itself or perform additional operations before a pod is created.**

There are number of inbuild AC like:

* AlwaysPullImages: ensures that every time a pod is created the images are always pulled.
* DefaultStorageClass: observes the creation of PVCs and automatically adds a default storage class to them if one is not specified.
* EventRateLimit: Helps to set a limit on the request where the API server can handle at a time to prevent an API server from flooding with requests.



**Note:** The NamespaceExists and NamespaceAutoProvision admission controllers are deprecated and now replaced by NamespaceLifecycle admission controller.

The NamespaceLifecycle admission controller will make sure that requests  
to a non-existent namespace is rejected and that the default namespaces such as  
default, kube-system and kube-public cannot be deleted.