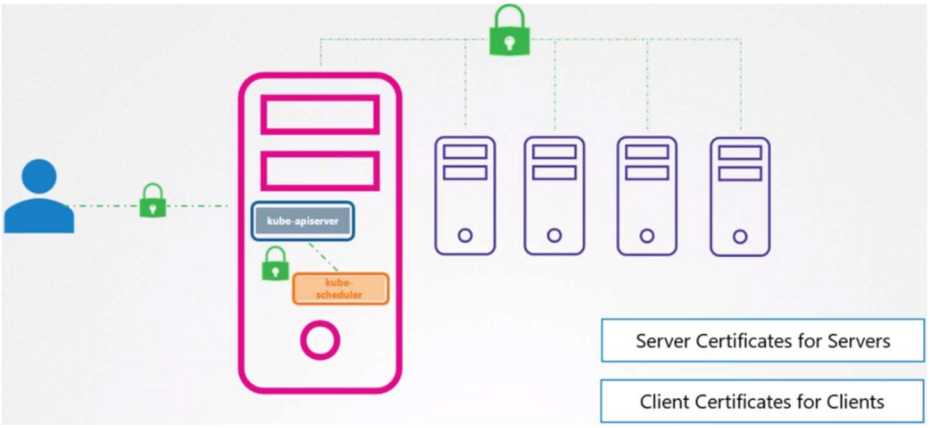
TLS in Kubernetes

So, three types of certificates, server certificates configured on the server's root certificate configured on the CA servers, client certificates configured on the clients. We will now see how these concepts relate to a Kubernetes cluster.

The Kubernetes cluster consists of a set of master and worker nodes. Of course, all communication between these nodes need to be secure and must be encrypted. All interactions between all services and their clients need to be secure. For example, an administrator interacting with the Kubernetes cluster through the kubectl utility or while accessing the Kubernetes API directly must establish secure TLS connection. Communication between all the components within the Kubernetes cluster also need to be secured. So the two primary requirements are to have all the various services within the cluster to use server certificates and all clients to use client certificates to verify they are who they say they are.



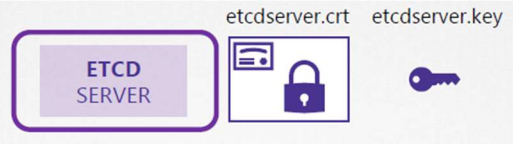
Let's look at the different components within the Kubernetes cluster and identify the various servers and clients and who talks to who.

kube-apiserver

Let's start with a kube-apiserver. As we know already, the API server exposes an HTTPS service that other components, as well as external users, use to manage the Kubernetes cluster. So it is a server and it requires certificates to secure all communication with its clients. So we generate a certificate and key pair. We call it APIserver.cert and APIserver.key. We will try to stick to this naming convention going forward. Anything with a .CRT extension is the certificate and .key extension is the private key. Also remember, these certificate names could be different in different Kubernetes setups depending on who and how the cluster was set up.

ETCD server

Another server in the cluster is the etcd server. The etcd server stores all information about the cluster. So it requires a pair of certificate and key for itself. We will call it etcdserver.crt and etcdserver.key.



kubelet server

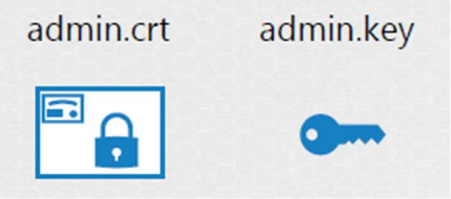
The other server component in the cluster is the worker nodes. They are the kubelet services. They also expose an HTTPS API endpoint that the kube-apiserver talks to interact with the worker nodes. Again that requires a certificate and key pair.

Those are really the server components in the Kubernetes cluster. Let's now look at the client components. Who are the clients who access these services?

The clients who access the **kube-apiserver**

Administrators

The administrators through kubectl Arrest API. the admin user requires a certificate and key pair to authenticate to the Kube-API server. We will call it admin.crt, and admin.key.

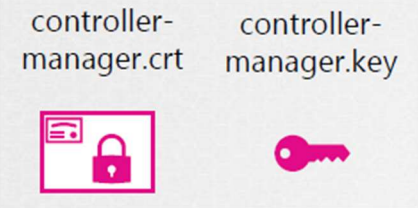


Scheduler

The scheduler talks to the kube-apiserver to look for pods that require scheduling and then get the API server to schedule the pods on the right worker nodes. The scheduler is a client that accesses the kube-apiserver. As far as the kube-apiserver is concerned, the scheduler is just another client, like the admin user. So the scheduler needs to validate its identity using a client TLS certificate. So it needs its own pair of certificate and keys. We will call it scheduler.cert and scheduler.key.

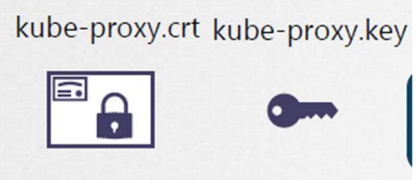
Kube Controller Manager

The Kube Controller Manager is another client that accesses the kube-apiserver, so it also requires a certificate for authentication to the kube-apiserver. So we create a certificate pair for it.

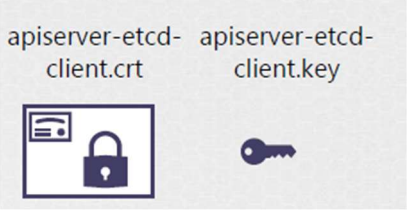


Kube-proxy

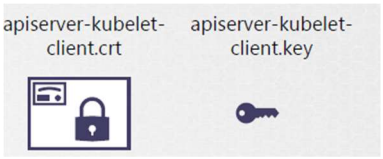
The last client component is the Kube-proxy. The Kube-proxy requires a client certificate to authenticate to the kube-apiserver, and so it requires its own pair of certificate and keys. We will call them kubeproxy.crt, and kubeproxy.key.



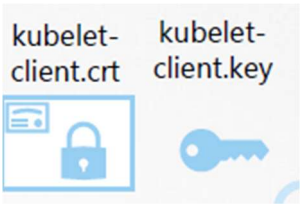
The servers communicate amongst them as well. For example, the kube-apiserver communicates with the etcd server. In fact, among of all the components, the kube-apiserver is the only server that talks to the etcd server. So as far as the etcd server is concerned, the kube-apiserver is a client, so it needs to authenticate. The kube-apiserver can use the same keys that it used earlier for serving its own API service. The APIserver.crt, and the APIserver.key files. Or, you can generate a new pair of certificates specifically for the kube-apiserver to authenticate to the etcd server.



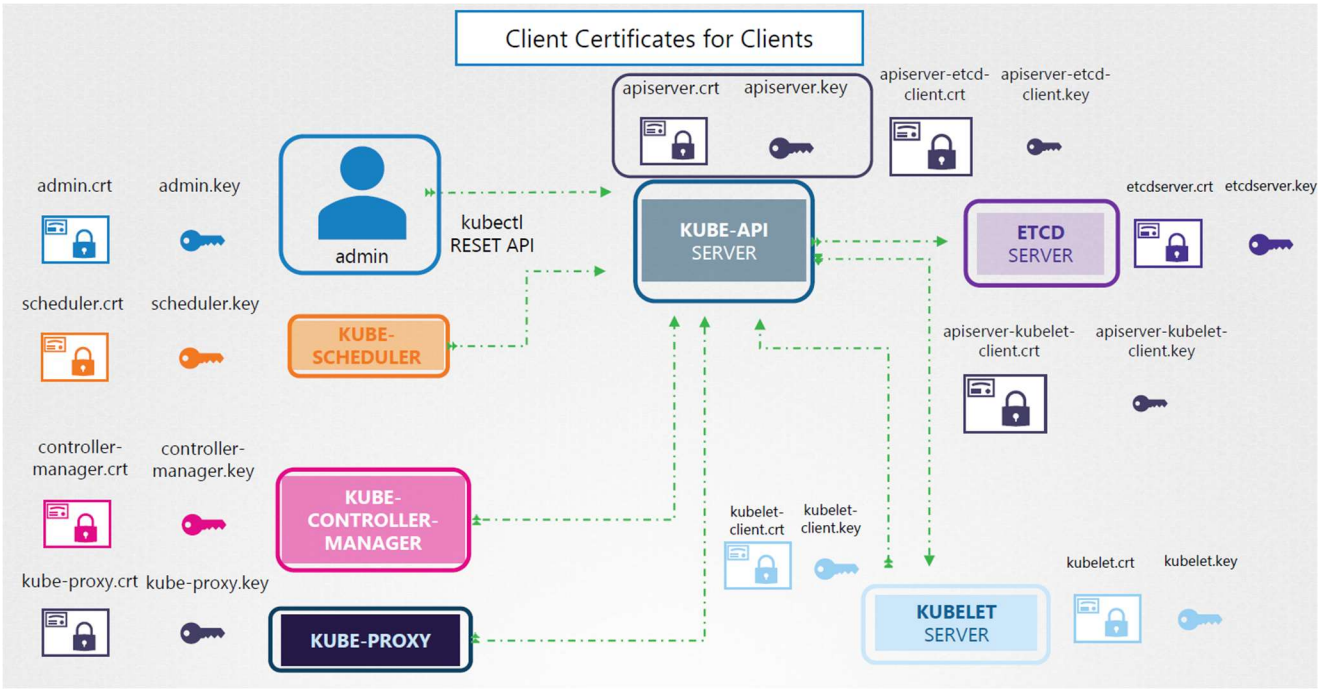
The kube-apiserver also talks to the kubelet server on each of the individual nodes. That's how it monitors the worker nodes for this. Again, it can use the original certificates, or generate new ones specifically for this purpose.



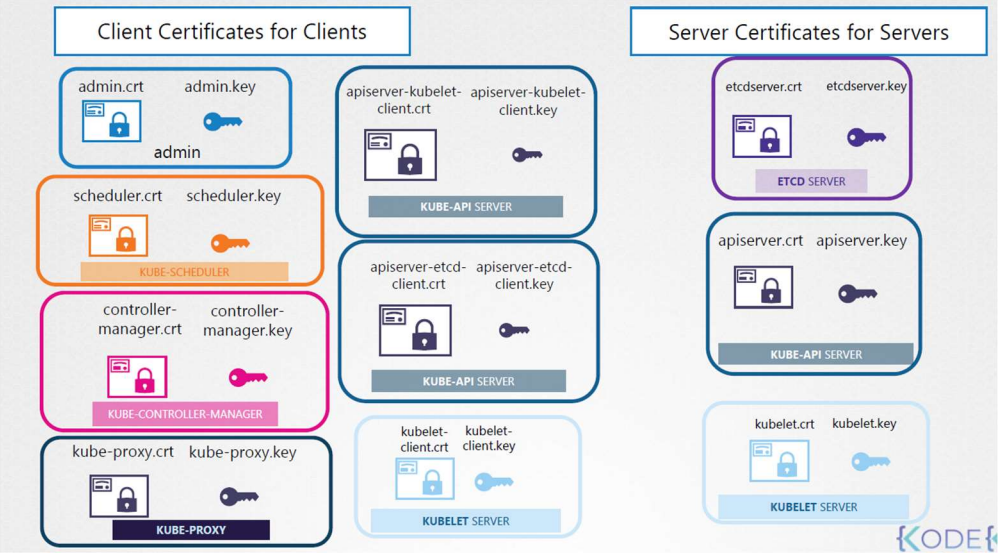
The kubelet server also talk to api server so it need client key pairs



So that's too many certificates.

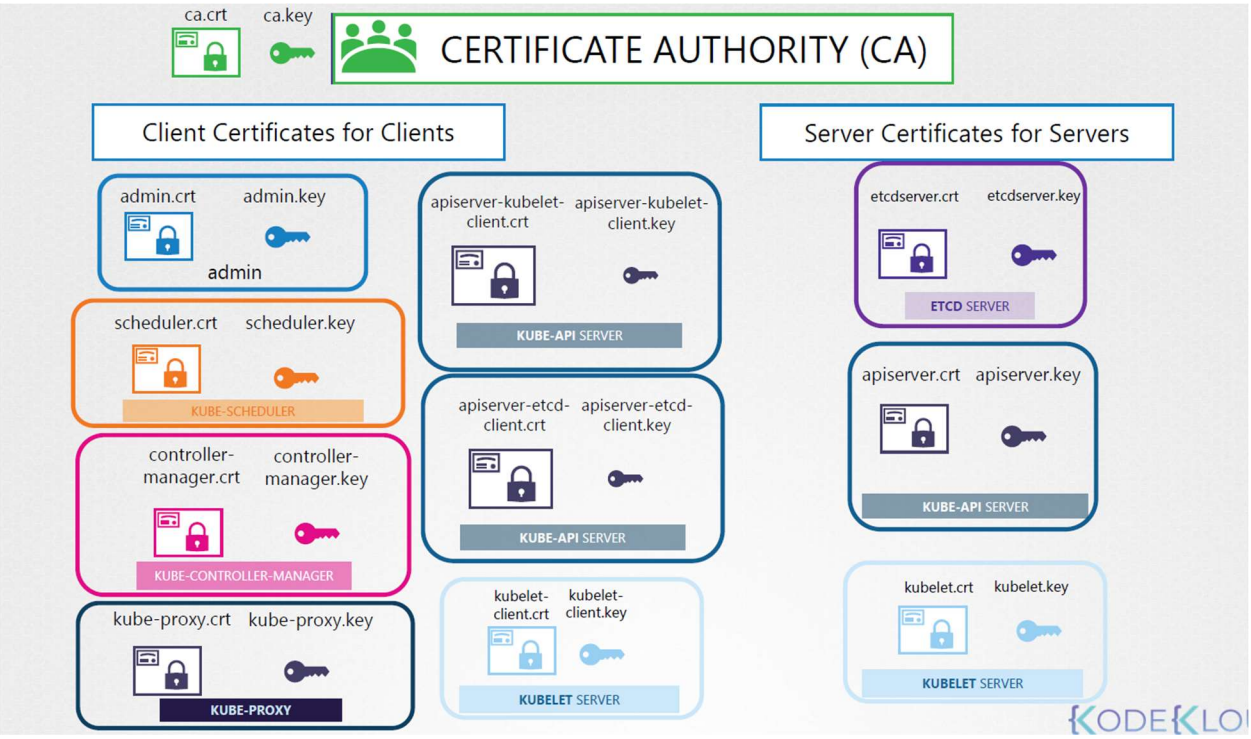


Let's try and group them. There are a set of client certificates, mostly used by clients to connect to the kube-apiserver and there are a set of server site certificates used by the kube-apiserver, etcd server, and kubelet to authenticate their clients.



We will now see how to generate these certificates. As we know already, we need a certificate authority to sign all of these certificates. Kubernetes requires you to have at least one certificate authority for your cluster. In fact, you can have more than one. One for all the components in the cluster and another one specifically for etcd. In that case, the etcd server's certificates and the etcd server's client certificates, which in this case is the API server client certificate, will be all signed by the etcd server CA. For now, we will stick to just one CA for our cluster.

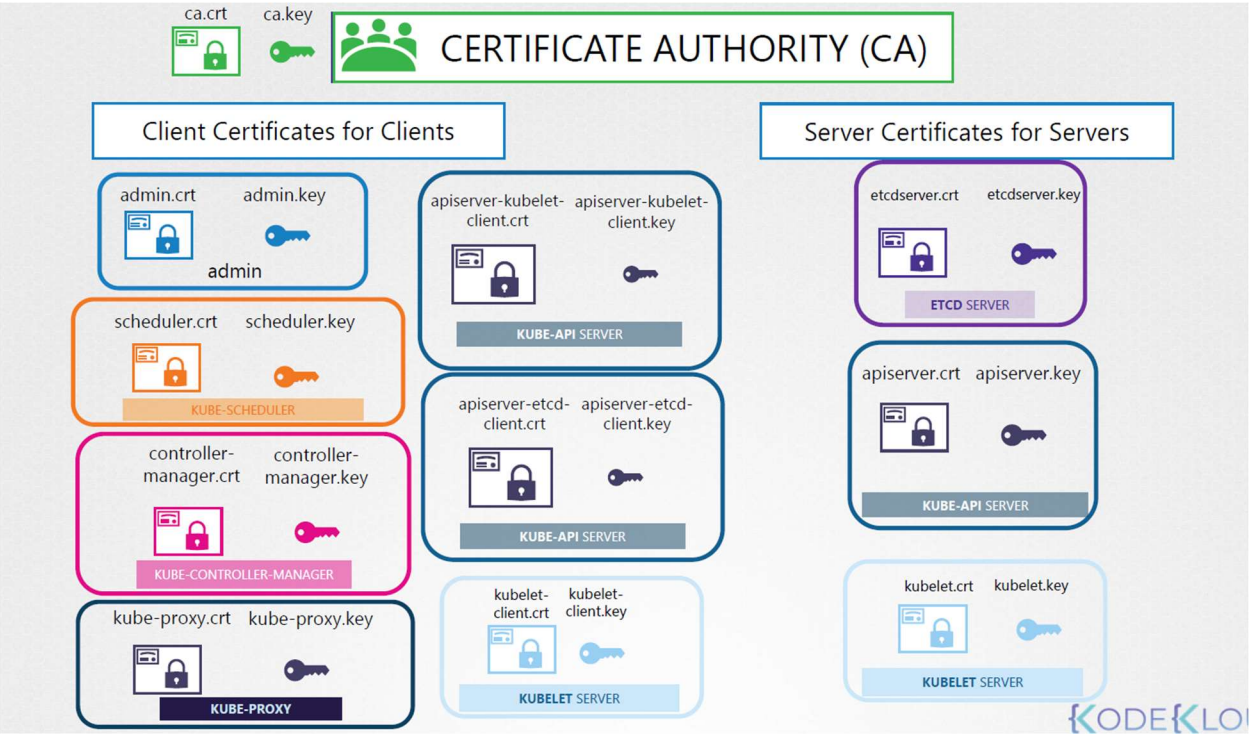
The CA, as we know, has its own pair of certificate and key. We will call it CA.crt and CA.key. That should sum up all the certificates used in the cluster.



TLS certificate creation

To generate certificates, there are different tools available such as Easy-RSA, OpenSSL, or CFSSL, et cetera, or many others. In this lecture we will use OpenSSL tool to generate the certificates.

This is where we left off.



We will start with the CA certificates. First we create a private key using the OpenSSL command:

openssl genrsa -out ca.key 2048

ca.key

Then we use the OpenSSL Request command along with the key we just created to generate a certificate signing request.

openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr

ca.csr

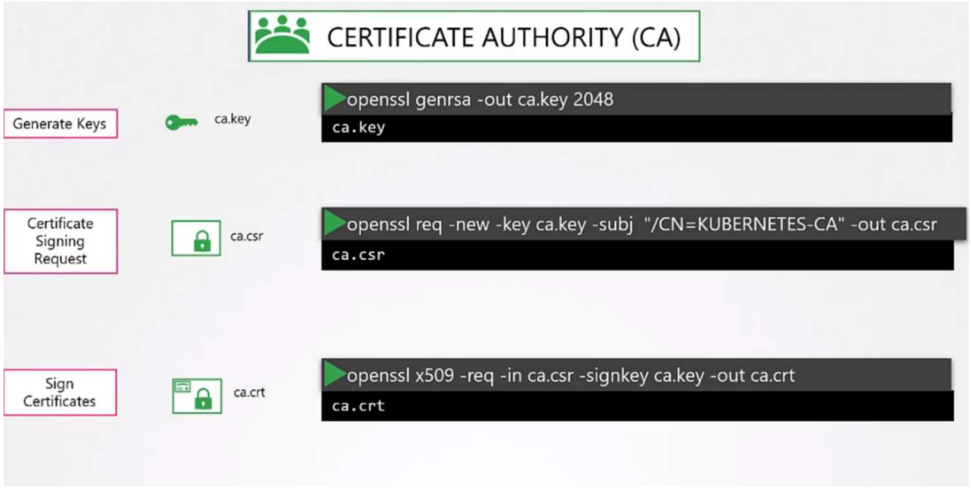
The certificate signing request is like a certificate with all of your details, but with no signature. In the certificate signing request we specify the name of the component the certificate is for in the Common Name, or CN field. In this case, since we are creating a certificate for the Kubernetes CA, we name it Kubernetes dash CA.

Finally, we sign the certificate using the below command, and by specifying the certificate signing request we generated in the previous command. Since this is for the CA itself it is self-signed by the CA using its own private key that it generated in the first step.

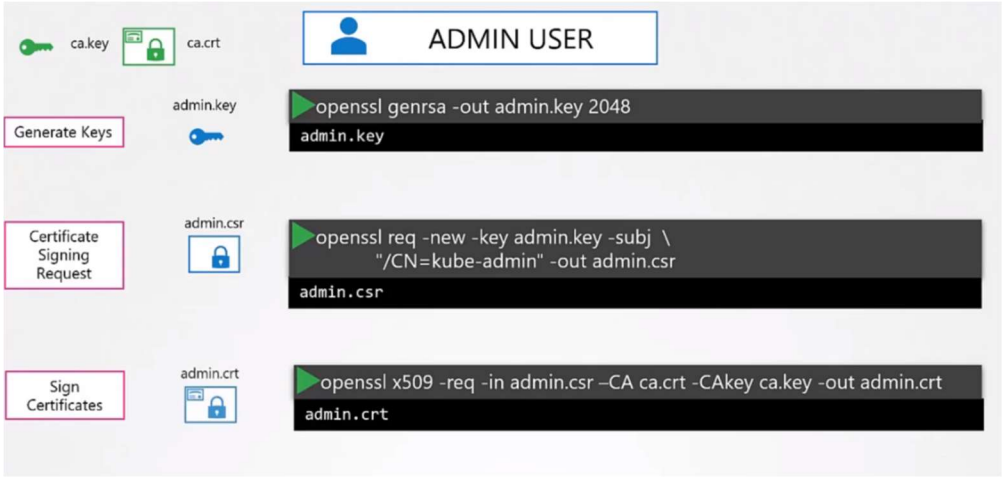
openssl x509 -req -in ca.csr –signkey ca.key -out ca.crt

ca.crt

Going forward for all other certificates we will use the CA key pair to sign them. The CA now has its private key and route certificate file.



Let's now look at generating the client certificates. We start with the admin user. We follow the same process where we create a private key for the admin user using the OpenSSL command. We then generate a CSR and that is where we specify the name of the admin user, which is Kube Admin. A quick note about the name, it doesn't really have to be Kube Admin. It could be anything, but remember, this is the name that Kube Control client authenticates with and when you run the Kube Control command. So in the audit logs and elsewhere, this is the name that you will see. So provide a relevant name in this field. Finally, generate a signed certificate using the openssl x509 –req -in admin.csr –CA ca.crt –Cakey ca.key-out admin.crt. But this time you specify the CA certificate and the CA key. You're signing your certificate with the CA key pair. That makes this a valid certificate within your cluster. The signed certificate is then output to admin.crt file. That is the certificate that the admin user will use to authenticate to Kubernetes cluster.



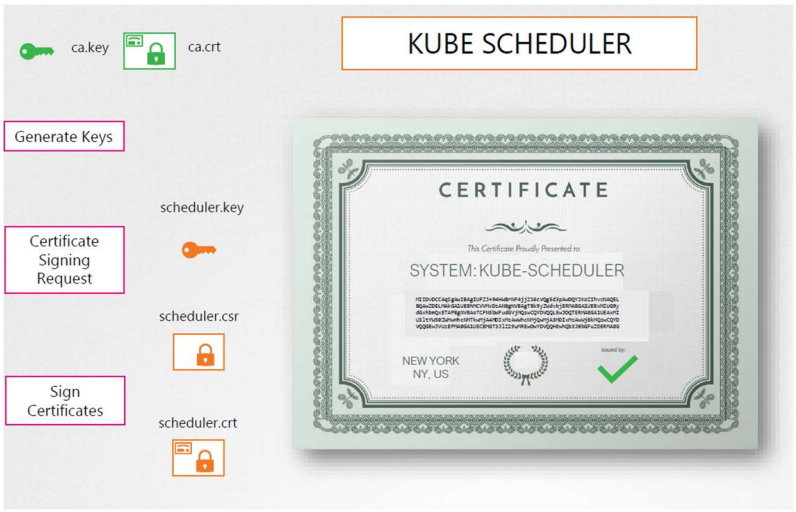
If you look at it, this whole process of generating a key and a certificate pair is similar to creating a user account for a new user. It's just that it's much more secure than a simple username and password.

So this is for the admin user. How do you differentiate this user from any other users? The user account needs to be identified as an admin user and not just another basic user. You do that by adding the group details for the user in the certificate. In this case, a group named SYSTEM:MASTERS exists on Kubernetes with administrative privileges. We will discuss about groups later, but for now it's important to note that you must mention this information in your certificate signing request. You can do this by adding group details with the OU parameter while generating a certificate signing request. Once it's signed, we now have our certificate for the admin user with admin privileges.

```
openssl req -new -key admin.key -subj \
        "/CN=kube-admin/OU=system:masters" -out admin.csr
```

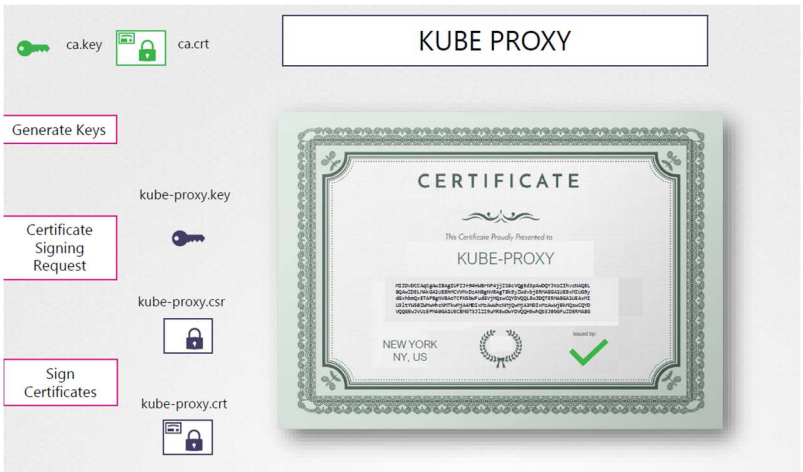
We follow the same process to generate client certificates for all other components that access the Kube API server.

The kube-scheduler. Now, the kube-scheduler is a system component, part of the Kubernetes control pane, so its name must be prefixed with the keyword system.



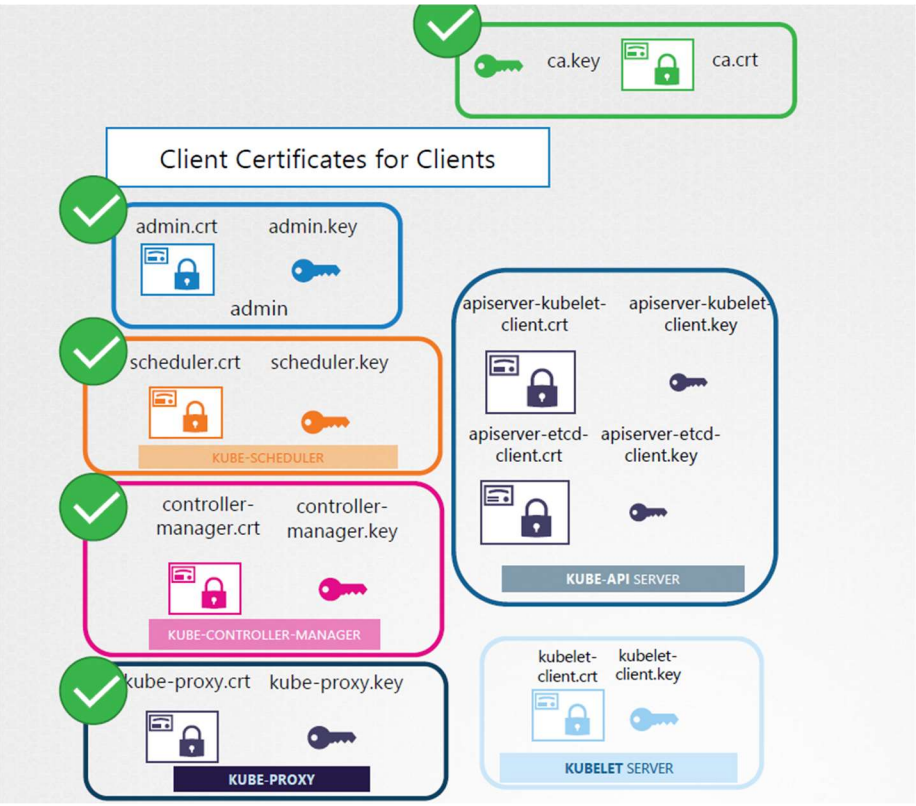
The same with kube-controller-manager. It is again a system component so its name must be prefixed with the keyword system.

And finally, kube-proxy.



So far we have created CA certificates, then all of the client certificates including the admin user, scheduler, controller-manager, and kube-proxy.

We will follow the same procedure to create the remaining three client certificates for API servers and kubelets when we create the server certificates for them. So we will set them aside for now.



Now, what do you do with these certificates? Take the admin certificate, for instance, to manage the cluster. You can use this certificate instead of a username and password in a REST API call you make to the Kube API server. You specify the key, the certificate, and the CA certificate as options. That's one simple way.

```
curl https://kube-apiserver:6443/api/v1/pods \
  --key admin.key --cert admin.crt
  --cacert ca.crt
```

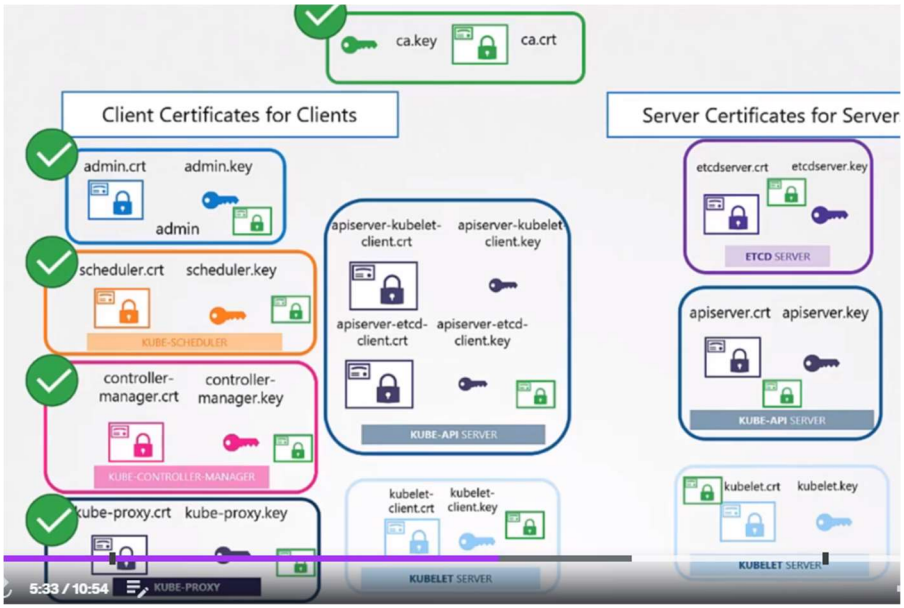
```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
  },
  "items": []
}
```

The other way is to move all of these parameters into a configuration file called kubeconfig. Within that, specify the API server endpoint details, the certificates to use, et cetera. That is what most of the Kubernetes clients use.

kube-config.yaml

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority: ca.crt
    server: https://kube-apiserver:6443
  name: kubernetes
kind: Config
users:
- name: kubernetes-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```

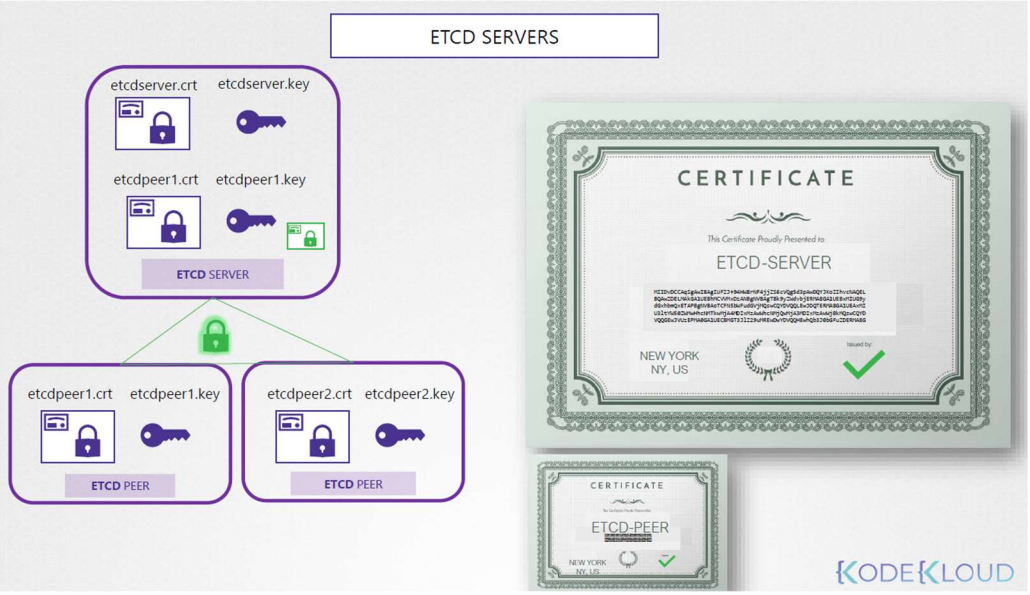
Okay, so we are now left with the server site certificates but before we proceed, one more thing. Remember in the prerequisite lecture we mentioned that for clients to validate the certificates sent by the server and vice versa, they all need a copy of the certificate authorities public certificate. The one that we said is already installed within the user's browsers in case of a web application. Similarly, in Kubernetes for these various components to verify each other, they all need a copy of the CA's root certificate. So whenever you configure a server or a client with certificates, you will need to specify the CA root certificate as well.



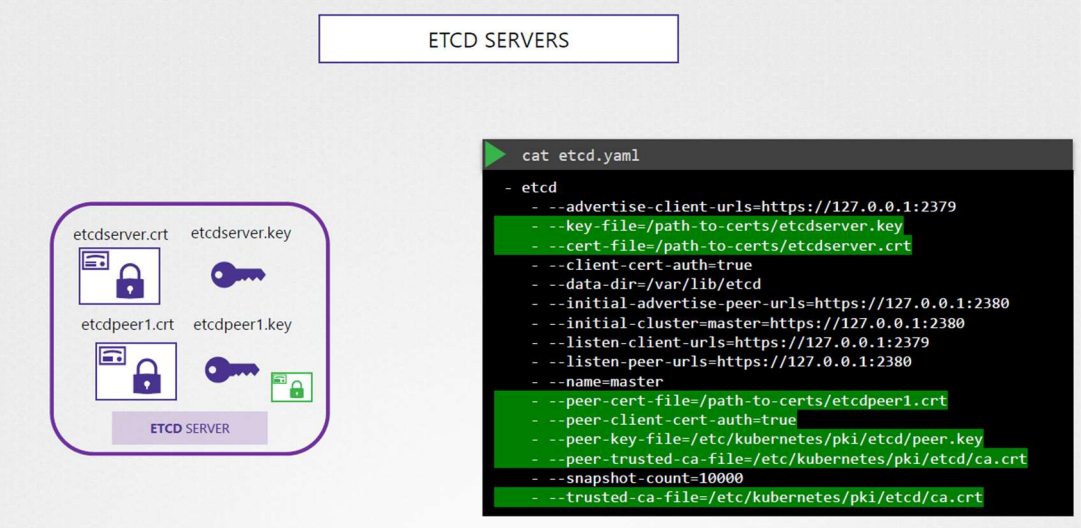
Let's look at the server-side certificates now.

ETCD server

Let's start with the **ETCD server**. We follow the same procedure as before to generate a certificate for ETCD. We will name it ETCD dash server. ETCD server can be deployed as a cluster across multiple servers as in a high availability environment. In that case, to secure communication between the different members in the cluster, we must generate additional peer certificates.



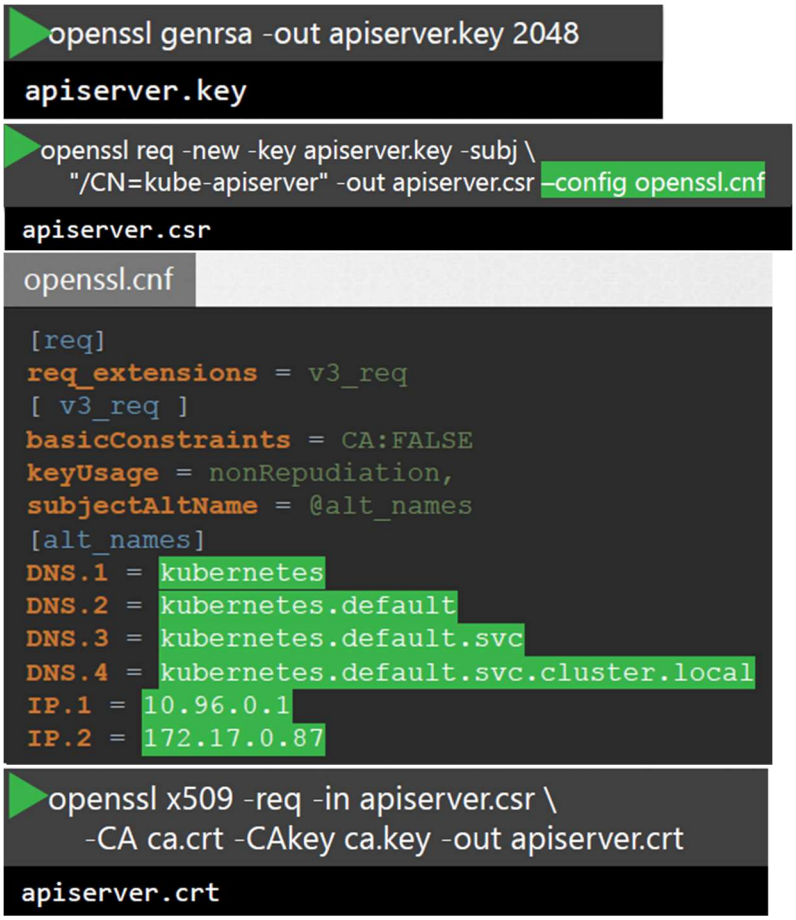
Once the certificates are generated specify them while starting the ETCD server. There are key and cert file options where you specify the ETCD server keys. There are other options available for specifying the peer certificates. And finally, as we discussed earlier, it requires the CA root certificate to verify that the clients connecting to the ETCD server are valid.



Kube API server

Let's talk about the **Kube API server now**. We generate a certificate for the API server like before. But wait, the API server is the most popular of all components within the cluster. Everyone talks to the Kube API server. Every operation goes through the Kube API server. Anything moves within the cluster, the API server knows about it. You need information, you talk to the API server. And so it goes by many names and aliases within the cluster. Its real name is Kube API server but some call it Kubernetes because for a lot of people who don't really know what goes under the hoods of Kubernetes, the Kube API server is Kubernetes. Others like to call it **Kubernetes.default**. Well, some refer to it as **Kubernetes.default.svc** and some like to call it by its full name **Kubernetes.default.svc.cluster.local**. Finally, it is also referred to in some places simply by its IP address. The IP address of the host running the Kube API server or the pod running it. So all of these names must be present in the certificate generated for the Kube API server. Only then those referring to the Kube API server by these names will be able to establish a valid connection.

So we use the same set of commands as earlier to generate a key. In the certificate signing request you specify the name Kube API server. But how do you specify all the alternate names? For that, you must create an OpenSSL config file. Create an OpenSSL.CNF file and specify the alternate names in the Alt Name section of the file. Include all the DNS names the API server goes by as well as the IP address. Pass this config file as an option while generating the certificate signing request. Finally, sign the certificate using the CA certificate end key. You then have the Kube API server certificate.

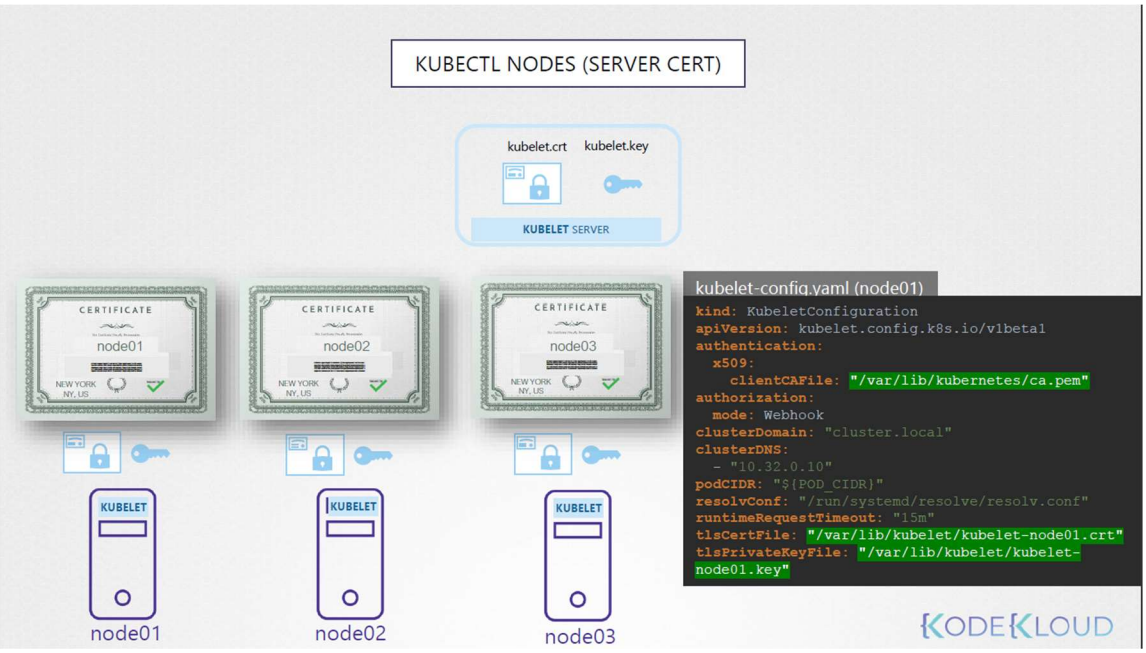


It is time to look at where we are going to specify these keys. Remember to consider the API client certificates that are used by the API server while communicating as a client to the ETCD and kubelet servers. The location of these certificates are passed in to the Kube API servers executable or service configuration file. First, the CA file needs to be passed in, to remember every component needs the CA certificate to verify its clients. Then we provide the API server certificates under the TLS cert options. We then specify the client certificates used by Kube API server to connect to the ETCD server again with the CA file. And finally, the Kube API server client certificates to connect to the kubelets.

kubelets server

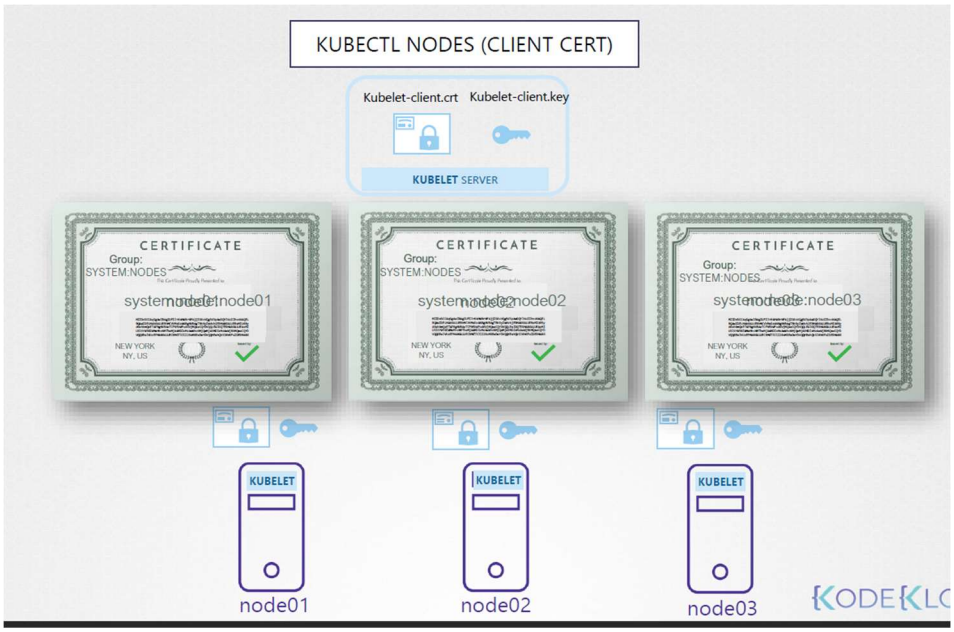
Next comes the **kubelets server**. The kubelets server is an https API server that runs on each node, responsible for managing the node. That's who the API server talks to, to monitor the node as well as send information regarding what pods to schedule on this node. As such, you need a key certificate pair for each node in the cluster.

Now, what do you name these certificates? Are they all going to be named kubelets? No. They will be named after their nodes. Node01, node02, and node03. Once the certificates are created, use them in the kubelet config file. As always, you specify the root CS certificate and then provide the kubelet node certificates. You must do this for each node in the cluster.



We also talked about a set of client certificates that will be used by the kubelet to communicate with the Kube API server. These are used by the kubelet to authenticate into the Kube API server. They need to be generated as well. What do you name these certificates? The API server needs to know which node is authenticating and give it the right set of permissions so it requires the nodes to have the right names in the right formats.

Since the nodes are system components like the kube-scheduler and the controller-manager we talked about earlier, the format starts with the system keyword, followed by node, and then the node name. In this case, node01 to node03. And how would the API server give it the right set of permissions? Remember we specified a group name for the admin user so the admin user gets administrative privileges? Similarly, the nodes must be added to a group named System Nodes. Once the certificates are generated, they go into the kubeconfig files as we discussed earlier.



View certificate details

In this lecture, we see how we can view certificates in an existing cluster. So you join a new team to help them manage their Kubernetes environment. You're a new administrator to this team. You've been told that there are multiple issues related to certificates in the environment. So you're asked to perform a health check of all the certificates in the entire cluster.

What do you do?

First of all, it's important to know how the cluster was set up. There are different solutions available for deploying a Kubernetes cluster, and they use different methods to generate and manage certificates.

If you were to deploy a Kubernetes cluster from scratch you generate all the certificates by yourself, as we did in the previous lecture. Or else, if you were to rely on an automated provisioning tool like Kubeadm, it takes care of automatically generating and configuring the cluster for you. While you deploy all the components as native services on the nodes in the hard way, the Kubeadm tool deploys these as pods.

So it's important to know where to look at, to view the right information. In this lecture, we're going to look, at a cluster provision by KubeADM, as an example. In order to perform a health check, start by identifying all the certificates used in the system.

So the idea is to create a list of certificate files used. There are paths, the names configured on them. The alternate names configured, if any, the organization the certificate account belongs to, the issue of the certificate and the expiration date on the certificate.

kubeadm

Component	Type	Certificate Path	CN Name	ALT Names	Organization	Issuer	Expiration
kube-apiserver	Server						
kube-apiserver	Server						
kube-apiserver	Server						
kube-apiserver	Client (Kubelet)						
kube-apiserver	Client (Kubelet)						
kube-apiserver	Client (Etcd)						
kube-apiserver	Client (Etcd)						
kube-apiserver	Client (Etcd)						

KODEKLOUD

So how do you get these? Start with the certificate files used. For this, in an environment set up by Kube ADM, look for the Kube API server definition file under /etc/kubernetes/manifests folder. The command used to start the API server has information about all the certificates it uses. Identify the certificate file used for each purpose and note it down.

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.32
    - --allow-privileged=true
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --disable-admission-plugins=PersistentVolumeLabel
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
    - --secure-port=6443
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-cluster-ip-range=10.96.0.0/12
    - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
    - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
```

Next, take each certificate and look inside it to find more details about that certificate. For example, we will start with the API server certificate file. Run the open SSL X509 command and provide the certificate file as input to decode the certificate and view details.

Start with a name on the certificate under the subject section. In this case, it's Kube-API Server. Then the alternate names the kube API server has many, so you must ensure all of them are there. And then check the validity section of the certificate to, identify the expiry date, and then the issuer of the certificate. This should be the CA who issued the certificate. Kube ADM names the Kubernetes CA, as Kubernetes itself.

```
openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 3147495682089747350 (0x2bae26a58f090396)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Feb 11 05:39:19 2019 GMT
            Not After : Feb 11 05:39:20 2020 GMT
        Subject: CN=kube-apiserver
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:d9:69:38:80:68:3b:b7:2e:9e:25:00:e8:fd:01:

                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Subject Alternative Name:
                DNS:master, DNS:kubernetes, DNS:kubernetes.default,
DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster.local, IP
Address:10.96.0.1, IP Address:172.17.0.27
```

Follow the same procedure to identify information about all the other certificates. Things to look for: Check to make sure you have the right names the right alternate names, make sure the certificates are part of the correct organization, and most importantly, they're issued by the right issuer and that the certificates are not expired.

kubeadm

Certificate Path	CN Name	ALT Names	Organization	Issuer	Expiration
/etc/kubernetes/pki/apiserver.crt /etc/kubernetes/pki/apiserver.key	kube-apiserver	DNS:master DNS:kubernetes DNS:kubernetes.default DNS:kubernetes.default.svc IP Address:10.96.0.1 IP Address:172.17.0.27		kubernetes	Feb 11 05:39:20 2020
/etc/kubernetes/pki/ca.crt	kubernetes			kubernetes	Feb 8 05:39:19 2029
/etc/kubernetes/pki/apiserver-kubelet-client.crt /etc/kubernetes/pki/apiserver-kubelet-client.key	kube-apiserver-kubelet-client		system:masters	kubernetes	Feb 11 05:39:20 2020
/etc/kubernetes/pki/apiserver-etcd-client.crt /etc/kubernetes/pki/apiserver-etcd-client.key	kube-apiserver-etcd-client		system:masters	self	Feb 11 05:39:22 2020
/etc/kubernetes/pki/etcd/ca.crt	kubernetes			kubernetes	Feb 8 05:39:21 2017

The certificate requirements are listed in detail in the Kubernetes documentation page.

Default CN	Parent CA	O (in Subject)	kind	hosts (SAN)
kube-etcd	etcd-ca		server, client [1][etcdbug]	`localhost`, `127.0.0.1`
kube-etcd-peer	etcd-ca		server, client	`<hostname>`, `<Host_IP>`, `localhost`, `127.0.0.1`
kube-etcd-healthcheck-client	etcd-ca		client	
kube-apiserver-etcd-client	etcd-ca	system:masters	client	
kube-apiserver	kubernetes-ca		server	`<hostname>`, `<Host_IP>`, `<advertise_IP>`, `[1]`
kube-apiserver-kubelet-client	kubernetes-ca	system:masters	client	
front-proxy-client	kubernetes-front-proxy-ca		client	

Default CN	recommend key path	recommended cert path	command	key argument	cert argument
etcd-ca		etcd/ca.crt	kube-apiserver		--etcd-cafile
etcd-client	apiserver-etcd-client.key	apiserver-etcd-client.crt	kube-apiserver	--etcd-keyfile	--etcd-certfile
kubernetes-ca		ca.crt	kube-apiserver		--client-ca-file
kube-apiserver	apiserver.key	apiserver.crt	kube-apiserver	--tls-private-key-file	--tls-cert-file
apiserver-kubelet-client		apiserver-kubelet-client.crt	kube-apiserver		--kubelet-client-certificate
front-proxy-ca		front-proxy-ca.crt	kube-apiserver		--requestheader-client-ca-file
front-proxy-client	front-proxy-client.key	front-proxy-client.crt	kube-apiserver	--proxy-client-key-file	--proxy-client-cert-file
etcd-ca		etcd/ca.crt	etcd		--trusted-ca-file, --peer-trusted-ca-file
kube-etcd	etcd/server.key	etcd/server.crt	etcd	--key-file	--cert-file
kube-etcd-peer	etcd/peer.key	etcd/peer.crt	etcd	--peer-key-file	--peer-cert-file
etcd-ca		etcd/ca.crt	etcdctl[2]		--cacert
kube-etcd-healthcheck-client	etcd/healthcheck-client.key	etcd/healthcheck-client.crt	etcdctl[2]	--key	--cert

When you're running to issues you want to start looking at logs. If you set up the cluster from scratch by yourself and the services are configured as "native services" in the OS, you want to start looking at the service logs using the operating systems logging functionality.

```
journalctl -u etcd.service -l
2019-02-13 02:53:28.144631 I | etcdmain: etcd Version: 3.2.18
2019-02-13 02:53:28.144680 I | etcdmain: Git SHA: eddf599c6
2019-02-13 02:53:28.144684 I | etcdmain: Go Version: go1.8.7
2019-02-13 02:53:28.144688 I | etcdmain: Go OS/Arch: linux/amd64
2019-02-13 02:53:28.144692 I | etcdmain: setting maximum number of CPUs to 4, total number of available CPUs is 4
2019-02-13 02:53:28.144734 N | etcdmain: the server is already initialized as member before, starting as etcd
member...
2019-02-13 02:53:28.146625 I | etcdserver: name = master
2019-02-13 02:53:28.146637 I | etcdserver: data dir = /var/lib/etcd
2019-02-13 02:53:28.146642 I | etcdserver: member dir = /var/lib/etcd/member
2019-02-13 02:53:28.146645 I | etcdserver: heartbeat = 100ms
2019-02-13 02:53:28.146648 I | etcdserver: election = 1000ms
2019-02-13 02:53:28.146651 I | etcdserver: snapshot count = 10000
2019-02-13 02:53:28.146677 I | etcdserver: advertise client URLs = 2019-02-13 02:53:28.185353 I | etcdserver/api:
enabled capabilities for version 3.2
2019-02-13 02:53:28.185588 I | embed: ClientTLS: cert = /etc/kubernetes/pki/etcd/server.crt, key =
/etc/kubernetes/pki/etcd/server.key, ca = , trusted-ca = /etc/kubernetes/pki/etcd/old-ca.crt, client-cert-auth =
true
2019-02-13 02:53:30.080017 I | embed: ready to serve client requests
2019-02-13 02:53:30.080130 I | etcdserver: published {Name:master ClientURLs:[https://127.0.0.1:2379]} to cluster
c9be114fc2da2776
2019-02-13 02:53:30.080281 I | embed: serving client requests on 127.0.0.1:2379
WARNING: 2019/02/13 02:53:30 Failed to dial 127.0.0.1:2379: connection error: desc = "transport: authentication
handshake failed: remote error: tls: bad certificate"; please retry.
```

In case you set up the cluster with Cube ADM then the various components are deployed as pods. So you can look at the logs, using the kubectl logs command followed by the pod name.

```
kubectl logs etcd-master
2019-02-13 02:53:28.144631 I | etcdmain: etcd Version: 3.2.18
2019-02-13 02:53:28.144680 I | etcdmain: Git SHA: eddf599c6
2019-02-13 02:53:28.144684 I | etcdmain: Go Version: go1.8.7
2019-02-13 02:53:28.144688 I | etcdmain: Go OS/Arch: linux/amd64
2019-02-13 02:53:28.144692 I | etcdmain: setting maximum number of CPUs to 4, total number of available CPUs is 4
2019-02-13 02:53:28.144734 N | etcdmain: the server is already initialized as member before, starting as etcd
member...
2019-02-13 02:53:28.146625 I | etcdserver: name = master
2019-02-13 02:53:28.146637 I | etcdserver: data dir = /var/lib/etcd
2019-02-13 02:53:28.146642 I | etcdserver: member dir = /var/lib/etcd/member
2019-02-13 02:53:28.146645 I | etcdserver: heartbeat = 100ms
2019-02-13 02:53:28.146648 I | etcdserver: election = 1000ms
2019-02-13 02:53:28.146651 I | etcdserver: snapshot count = 10000
2019-02-13 02:53:28.146677 I | etcdserver: advertise client URLs = 2019-02-13 02:53:28.185353 I | etcdserver/api:
enabled capabilities for version 3.2
2019-02-13 02:53:28.185588 I | embed: ClientTLS: cert = /etc/kubernetes/pki/etcd/server.crt, key =
/etc/kubernetes/pki/etcd/server.key, ca = , trusted-ca = /etc/kubernetes/pki/etcd/old-ca.crt, client-cert-auth =
true
2019-02-13 02:53:30.080017 I | embed: ready to serve client requests
2019-02-13 02:53:30.080130 I | etcdserver: published {Name:master ClientURLs:[https://127.0.0.1:2379]} to cluster
c9be114fc2da2776
2019-02-13 02:53:30.080281 I | embed: serving client requests on 127.0.0.1:2379
WARNING: 2019/02/13 02:53:30 Failed to dial 127.0.0.1:2379: connection error: desc = "transport: authentication
handshake failed: remote error: tls: bad certificate"; please retry.
```

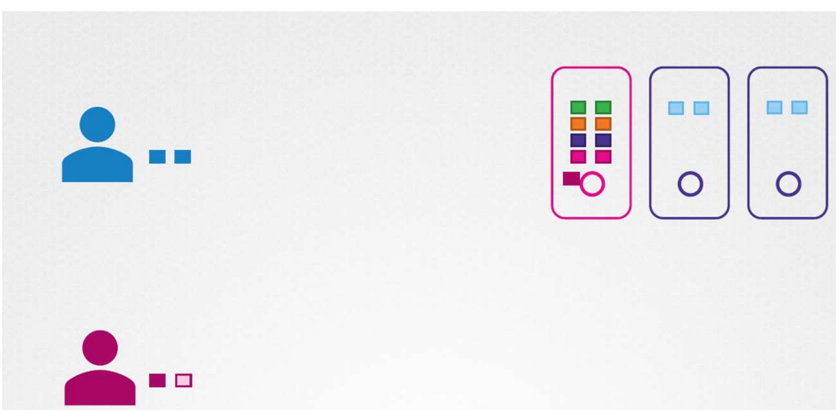
Sometimes if the core components, such as, the Kubernetes API server or the ETCD server are down, the kube control commands won't function. In that case, you have to go one level down to Docker to fetch the logs. List all the containers using the Docker PS-A command and then view the logs using the Docker logs command followed by the container ID.

```
docker ps -a
CONTAINER ID                               STATUS            NAMES
23482a09f25b   Up 12 minutes                k8s_kube-apiserver_kube-apiserver-master_kube-system_8758a3d10776bb527e043
b9bf77348c96   Up 18 minutes                k8s_etcd_etcd-master_kube-system_2cc1c8a24b68ab9b46bca47e153e74c6_0
87fc69913973   Up 18 minutes                k8s_POD_etcd-master_kube-system_2cc1c8a24b68ab9b46bca47e153e74c6_0
fda322157b86   Exited (255) 18 minutes ago  k8s_kube-apiserver_kube-apiserver-master_kube-system_8758a3d10776bb527e043
0794bdfd57d8   Up 40 minutes                k8s_kube-scheduler_kube-scheduler-master_kube-system_009228e74aef4d7babd79
00f3f95d2102   Up 40 minutes                k8s_kube-controller-manager_kube-controller-manager-master_kube-system_ac1
b8e6a0e173dd   Up About an hour             k8s_weave_weave-net-8dzwb_kube-system_22cd7993-2f2d-11e9-a2a6-0242ac110021
18e47bad320e   Up About an hour             k8s_weave-npc_weave-net-8dzwb_kube-system_22cd7993-2f2d-11e9-a2a6-0242ac11
4d087daf0380   Exited (1) About an hour ago k8s_weave_weave-net-8dzwb_kube-system_22cd7993-2f2d-11e9-a2a6-0242ac110021
e923140101a3   Up About an hour             k8s_kube-proxy_kube-proxy-cdmlz_kube-system_22cd267f-2f2d-11e9-a2a6-0242ac
e0db7e63d18e   Up About an hour             k8s_POD_weave-net-8dzwb_kube-system_22cd7993-2f2d-11e9-a2a6-0242ac110021_0
74c257366f65   Up About an hour             k8s_POD_kube-proxy-cdmlz_kube-system_22cd267f-2f2d-11e9-a2a6-0242ac110021_
8f514eac9d04   Exited (255) 40 minutes ago  k8s_kube-controller-manager_kube-controller-manager-master_kube-system_ac1
b39c5c594913   Exited (1) 40 minutes ago    k8s_kube-scheduler_kube-scheduler-master_kube-system_009228e74aef4d7babd79
3aefcb20ed30   Up 2 hours                   k8s_POD_kube-apiserver-master_kube-system_8758a3d10776bb527e043fccfc835986
576c8a273b50   Up 2 hours                   k8s_POD_kube-controller-manager-master_kube-system_ac1d4c5ae0fbe553b664a6c
4b3c5f34efde   Up 2 hours                   k8s_POD_kube-scheduler-master_kube-system_009228e74aef4d7babd7968782118d5e
```

Certificate Workflow and API

In this lecture, we look at how to manage certificates and what the certificate API is in Kubernetes. So what have we done so far? I, as an administrator of the cluster, in the process of setting up the whole cluster, have set up a CA server, and bunch of certificates for various components. We then started the services using the right certificates and it's all up and working. I am the only administrator and user of the cluster and I have my own admin certificate and key.

A new admin comes into my team. She needs access to the cluster. We need to get her a pair of certificate and key pair for her to access the cluster. She creates her own private key, generates a certificate signing request, and sends it to me. Since I'm the only admin, I then take the certificate signing request to my CA server, gets it signed by the CA server using the CA server's private key and route certificate, thereby generating a certificate and then sends the certificate back to her. She now has her own valid pair of certificate and key that she can use to access the cluster.



The certificates have a validity period. It ends after a period of time. Every time it expires, we follow the same process of generating a new CSR and getting it signed by the CA. So we keep rotating the certificate files.

We keep talking about the CA server, what is the CA server and where is it located in the Kubernetes setup? The CA is really just the pair of key and certificate files we have generated. Whoever gains access to these pair of files can sign any certificate for the Kubernetes environment. They can create as many users as they want put whatever privileges they want. So these files need to be protected and stored in a safe environment. Say we place them on a server that is fully secure, now that server becomes your CA server. The certificate key file is safely stored in that server and only on that server. Every time you want sign a certificate you can only do it by logging into that server. As of now, we have the certificates place on the Kubernetes master node itself, so the master node is also our CA server.

The Kubeadm tool does the same thing. It creates a CA pair of files and stores that on the master node itself. So far we have been signing requests manually, but as, and when the users increase and your team grows you need a better automated way to manage the certificates, signing requests, as well as to rotate certificates when they expire. Kubernetes has a built in certificates API that can do this for you.

With the certificates API, you now send a certificate signing request directly to Kubernetes through an API call. This time when the administrator receives a certificate signing request, instead of logging onto the master node and signing the certificate by himself, he creates a Kubernetes API object called certificate signing request. Once the object is created all certificate signing requests can be seen by administrators of the cluster. The request can be reviewed and approved easily using Kube control commands. This certificate can then be extracted and shared with the user.

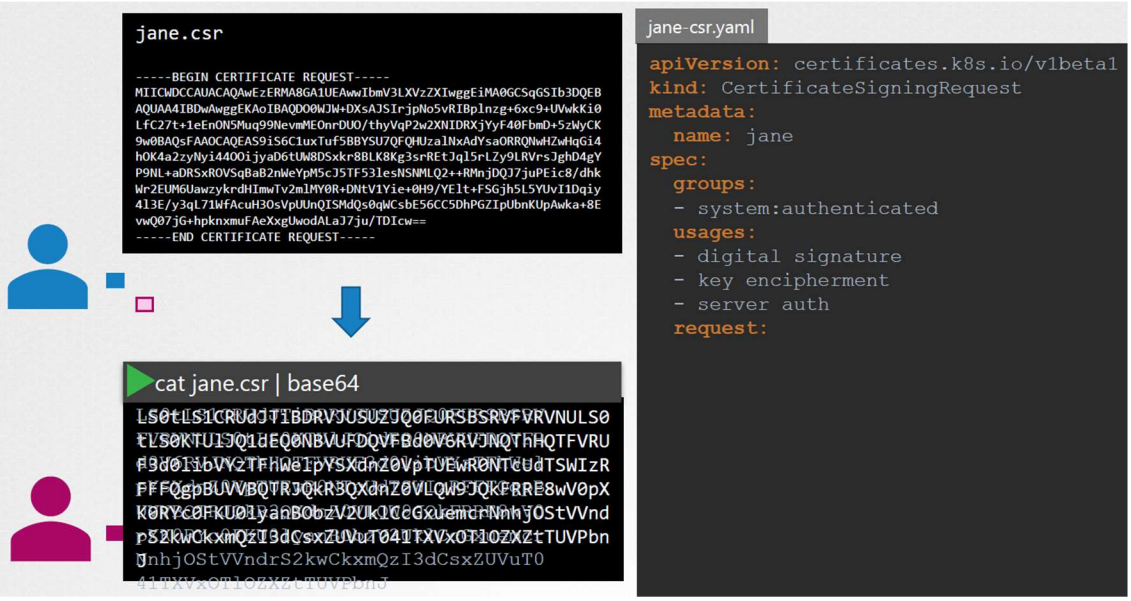
Let's see how it is done. A user first creates a key then generates a certificate signing request, using the key with her name on it, then sends the request to the administrator.

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

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

-----BEGIN CERTIFICATE REQUEST-----
MIICWDCCAUACAQAwEzERMA8GA1UEAwwIbmV3LXVzZXIwggEiMA0GCSqGSIb3DQEB
AQUAA4IBDwAwggEKAoIBAQDO0WJW+DXsAJSIrjpNo5vRIBplnzg+6xc9+UVwkKi0
LfC27t+1eEnON5Muq99NevmMEOnrDUO/thyVqP2w2XNIDRXjYyF40FbmD+5zWyCK
9w0BAQsFAAOCAQEAS9iS6C1uxTuf5BBYSU7QFQHUzalNxAdYsaORRQNwHZwHqGi4
hOK4a2zyNyi44OOijyaD6tUW8DSxkr8BLK8Kg3srREtJql5rLZy9LRVrsJghD4gY
P9NL+aDRSxROVSqBaB2nWeYpM5cJ5TF53lesNSNMLQ2++RMnjDQJ7juPEic8/dhk
Wr2EUM6UawzykrdHImwTv2mlMY0R+DNtV1Yie+0H9/YElt+FSGjh5L5YUvI1Dqiy
4l3E/y3qL71WfAcuH3OsVpUUnQISMdQs0qWCsbE56CC5DhPGZIpUbnKUpAwka+8E
vwQ07jG+hpknxmuFAeXxgUwodALaJ7ju/TDIcw==
-----END CERTIFICATE REQUEST-----
```

The administrator takes a key and creates a certificate signing request object. The certificate signing request object is created like any other Kubernetes object using a manifest file with the usual fields. The kind is certificate signing request. Under the spec section, specify the groups the user should be part of, and lists the usages of the account as a list of strings. The request field is where you specify the certificate signing request sent by the user but you don't specify it as plain text. Instead, it must be encoded using the base 64 command. Then, move the encoded text into the request field and then submit the request.



Once the object is created all certificate signing requests can be seen by administrators by running the Kubectl, get CSR command. Identify the new request and approve the request by running the Kubectl certificate approved command. Kubernetes signs the certificate using the CA key pairs and generates a certificate for the user.

```
kubectl get csr
NAME                  AGE       REQUESTOR                 CONDITION
jane                  10m       admin@example.com         Pending

kubectl certificate approve jane
jane approved!
```

This certificate can then be extracted and shared with the user. View the certificate by viewing it in a YAML format. The generated certificate is part of the output, but as before, it is in a base 64 and coded format. To decode it, take the text and use the base 64 utilities decode option. This gives the certificate in a plain text format. This can then be shared with the end user.

```
kubectl get csr jane -o yaml
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  creationTimestamp: 2019-02-13T16:36:43Z
  name: new-user
spec:
  groups:
  - system:masters
  - system:authenticated
usages:
  - digital signature
  - key encipherment
  - server auth
  username: kubernetes-admin
status:
  certificate:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURDakNDQWZLZ0F3SUJBZ0lVRmwy
Q2wxYXoxaWl5M3JNVisreFRYQUowU3dnd0RRWUpLb1pJaHZjTkFRRUwKQlFBd0ZURVRN
QkVHQTFVRUF4TUthM1ZpWlhKdVpYUmxjekFlRncweE9UQXlNVE14TmpNeU1EQmFGd1dn
Y0ZFeDl2ajNuSXY3eFdDS1NIRm5sU041c0t5Z0VxUkwzTFM5V29GelhHZDdWCmlEZ2FO
MVVRMFBXTVhjN09FVnVjSWc1Yk4weEVHTkVwRU5tdUlBNlZWeHVjS1h6aG9ldDY0MEd1
MGU0YXFKWVIKWmVMbjBvRTFCY3dod2xic0I1ND0KLS0tLS1FTkQgQ0VSVElGSUNBVEUt
LS0tLQo=
  conditions:
  - lastUpdateTime: 2019-02-13T16:37:21Z
    message: This CSR was approved by kubectl certificate approve.
    reason: KubectlApprove
    type: Approved
```

```
echo "LS0…Qo=" | base64 --decode
-----BEGIN CERTIFICATE -----
MIICWDCCAUACAQAwEzERMA8GA1UEAwwIbmV3LXVzZXIwgg
AQUAA4IBDwAwggEKAoIBAQDO0WJW+DXsAJSIrjpNo5vRIB
LfC27t+1eEnON5Muq99NevmMEOnrDUO/thyVqP2w2XNIDR
y3BihhB93MJ7Oql3UTvZ8TELqyaDknRl/jv/SxgXkok0AB
IF5nxAttMVkDPQ7NbeZRG43b+QWlVGR/z6DWOfJnbfezOt
EcCXAwqChjBLkz2BHPR4J89D6Xb8k39pu6jpyngV6uP0tI
j2qEL+hZEWkkFz80lNNtyT5LxMqENDCnIgwC4GZiRGbrAg
9w0BAQsFAAOCAQEAS9iS6C1uxTuf5BBYSU7QFQHUzalNxA
hOK4a2zyNyi44OOijyaD6tUW8DSxkr8BLK8Kg3srREtJql
P9NL+aDRSxROVSqBaB2nWeYpM5cJ5TF53lesNSNMLQ2++R
Wr2EUM6UawzykrdHImwTv2mlMY0R+DNtV1Yie+0H9/YElt
4l3E/y3qL71WfAcuH3OsVpUUnQISMdQs0qWCsbE56CC5Dh
vwQ07jG+hpknxmuFAeXxgUwodALaJ7ju/TDIcw==
-----END CERTIFICATE -----
```


Now that we have seen how it works Let's see who does all of this for us. If you look at the Kubernetes control plane you see the Kube API server, the scheduler, the control manager, and CD server, et cetera. Which of these components is actually responsible for all the certificate related operations? All the certificate related operations are carried out by the controller manager. If you look closely at the controller manager, you will see that it has controllers in it called as CSR approving, CSR signing, et cetera, that are responsible for carrying out these specific tasks.

We know that if anyone has to sign certificates they need the CA servers route certificate and private key. The Controller Manager service configuration has two options where you can specify this.

```
cat /etc/kubernetes/manifests/kube-controller-manager.yaml
spec:
  containers:
  - command:
    - kube-controller-manager
    - --address=127.0.0.1
    - --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
    - --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
    - --controllers=*,bootstrapsigner,tokencleaner
    - --kubeconfig=/etc/kubernetes/controller-manager.conf
    - --leader-elect=true
    - --root-ca-file=/etc/kubernetes/pki/ca.crt
    - --service-account-private-key-file=/etc/kubernetes/pki/sa.key
    - --use-service-account-credentials=true
```