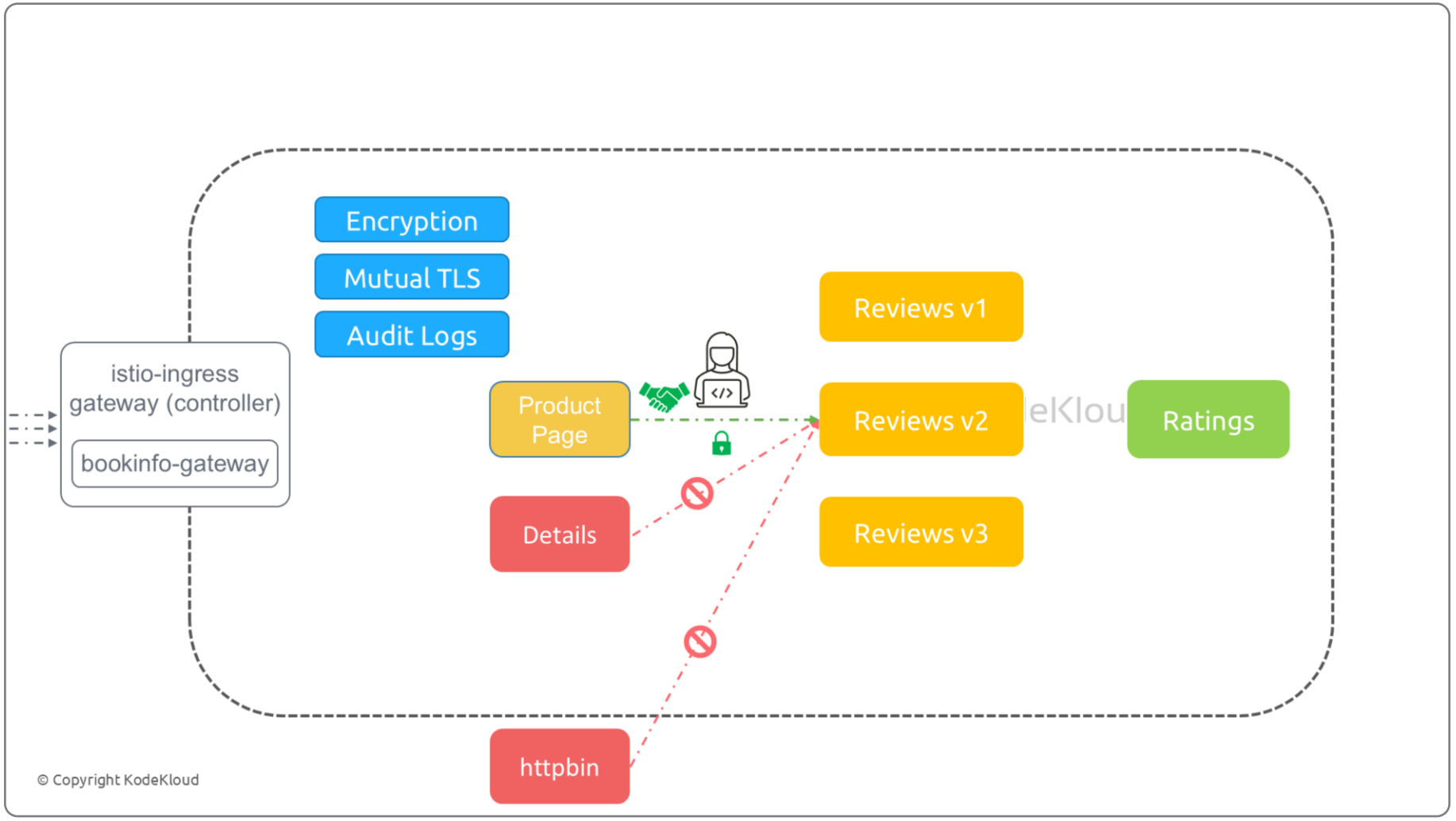
Let us start with a brief introduction of security in Istio.



Microservices have certain security requirements.

1. When a service communicates to another service its possible for an attacker to intercept the traffic and modify it before the request reaches the destination. This is known as man-in-the-middle attack.

2. To prevent this from happening the traffic between services need to be encrypted.

[pause]

Secondly, Certain services may need to implement access control restrictions.

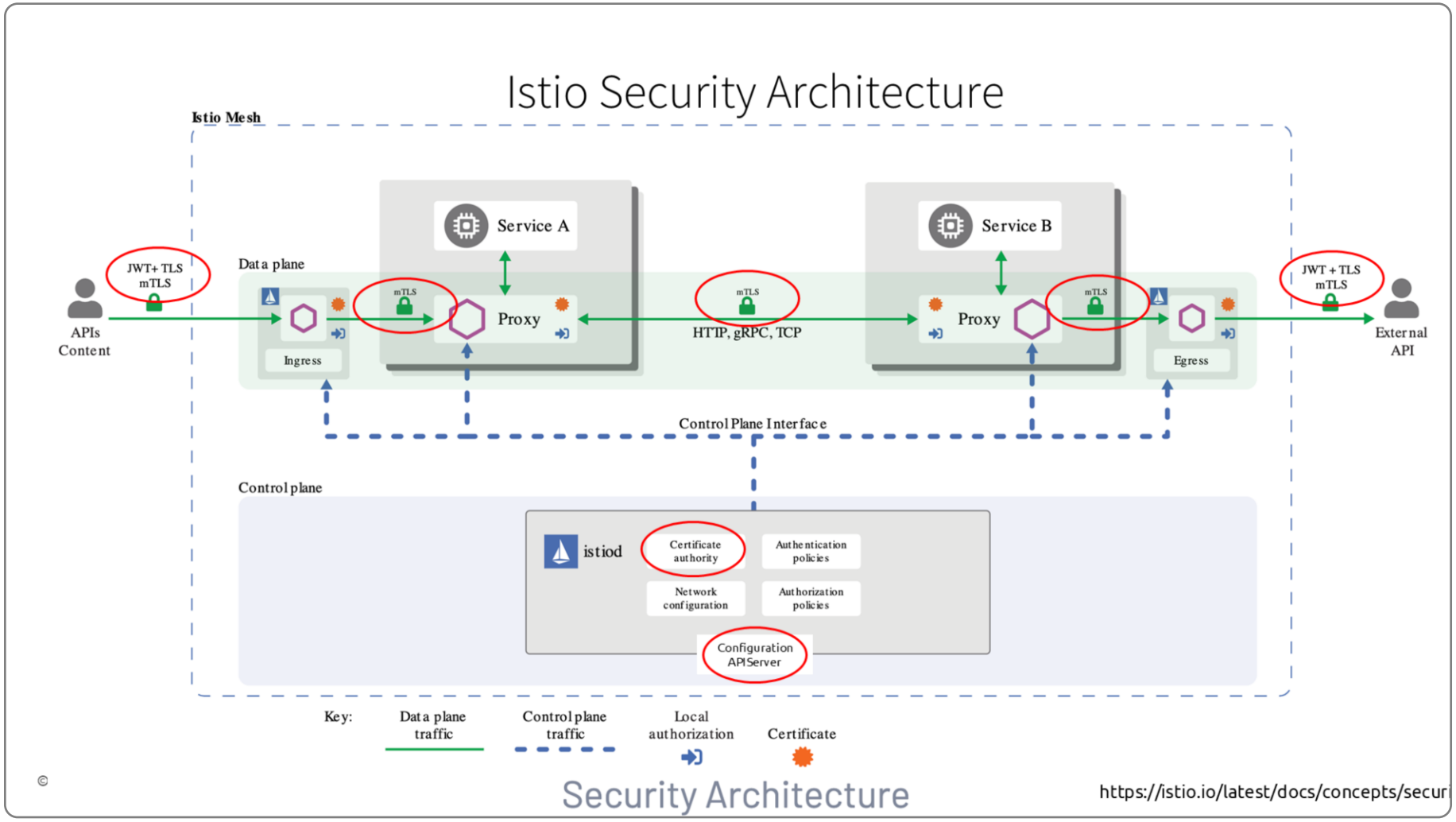
3. For example only the product page service is allowed to access the reviews service and not the details service. Or any other services in the cluster.

4. For this we need access control. Istio allows this using mutual TLS and fine-grained access policies.

5. And finally we'd like to know who did what at what time. And for this Istio provides support for Audit logs.

In the upcoming lectures we will dive deeper into each of these.

Let’s now look at the Istio Architecture and see how Istio implements security to achieve the requirements of microservices applications.



0. So this is the Istio architecture diagram from Istio documentation pages. Let's discover its details.
Within Istiod there is a certification authority that manages keys and certificates in Istio. This is where the certificates validated and certificate signing requests are approved. Everytime a workload is started, the envoy proxies requests the certificate and key from the istio agent and that helps secure communication between services.
3. The configuration API server component distributes all the authentication, authorization and secure naming policies to the proxies.

4. Sidecar and ingress / egress proxies work as policy enforcement points. The certificate, keys, authentication, authorization and secure naming policies are sent to these proxies at all times. So, this means, every point has security checks not just the entry points to the network. And this is called security at depth.

In the upcoming lectures we will look into these in more detail.

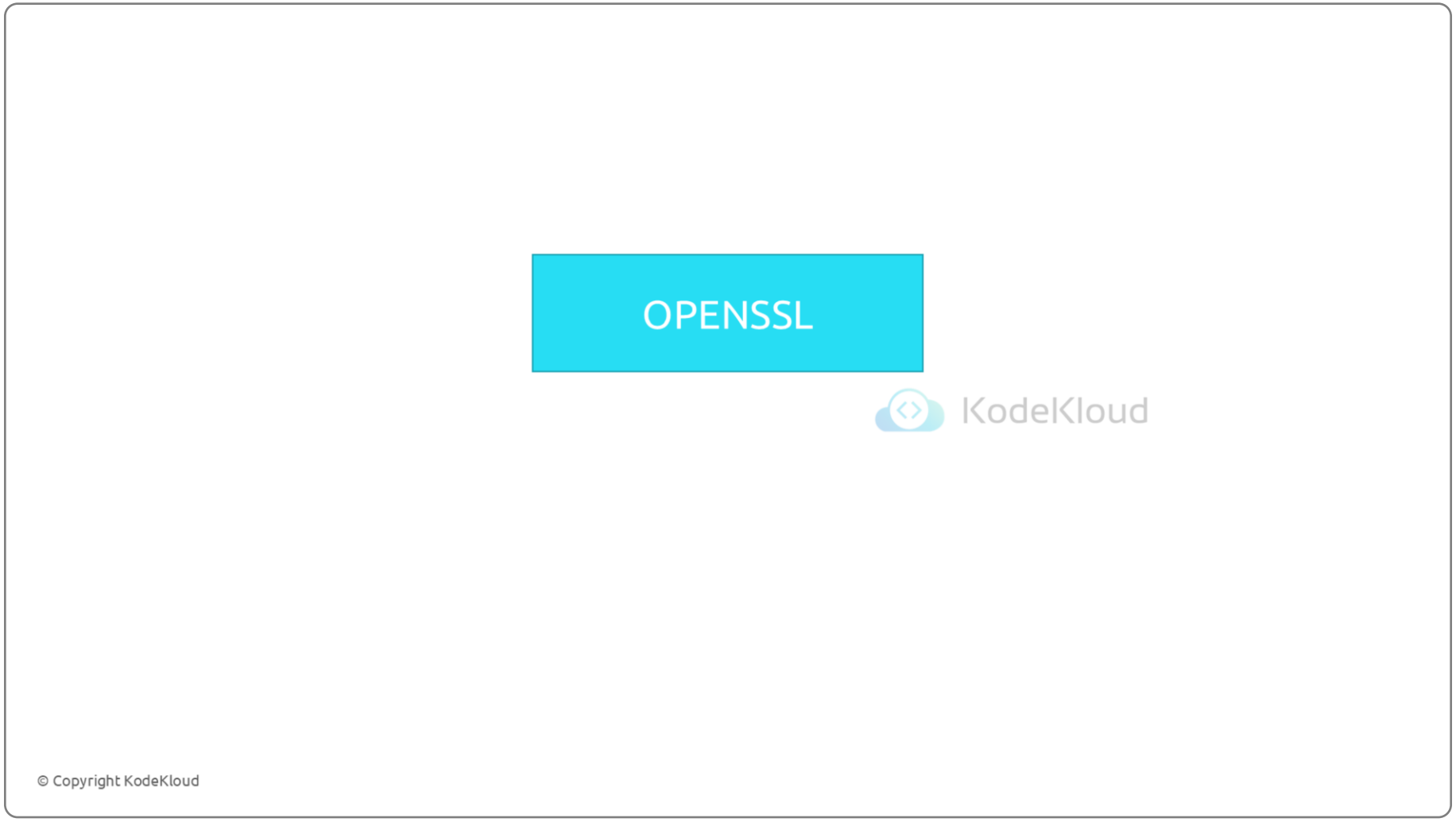
TLS CERTIFICATES

Generate Certificates

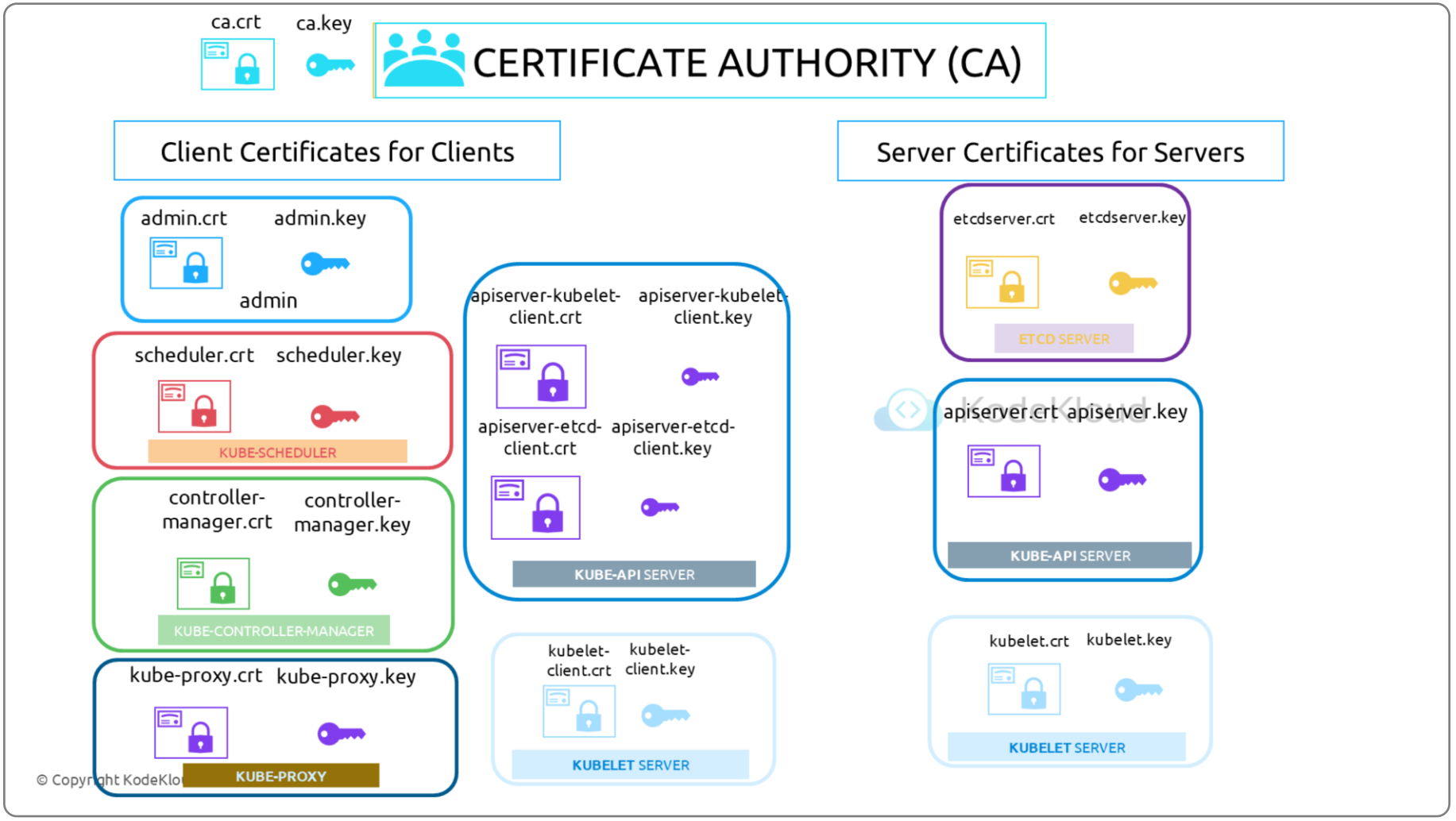
KodeKloud

In this lecture we look at how to generate the certificates for the cluster.

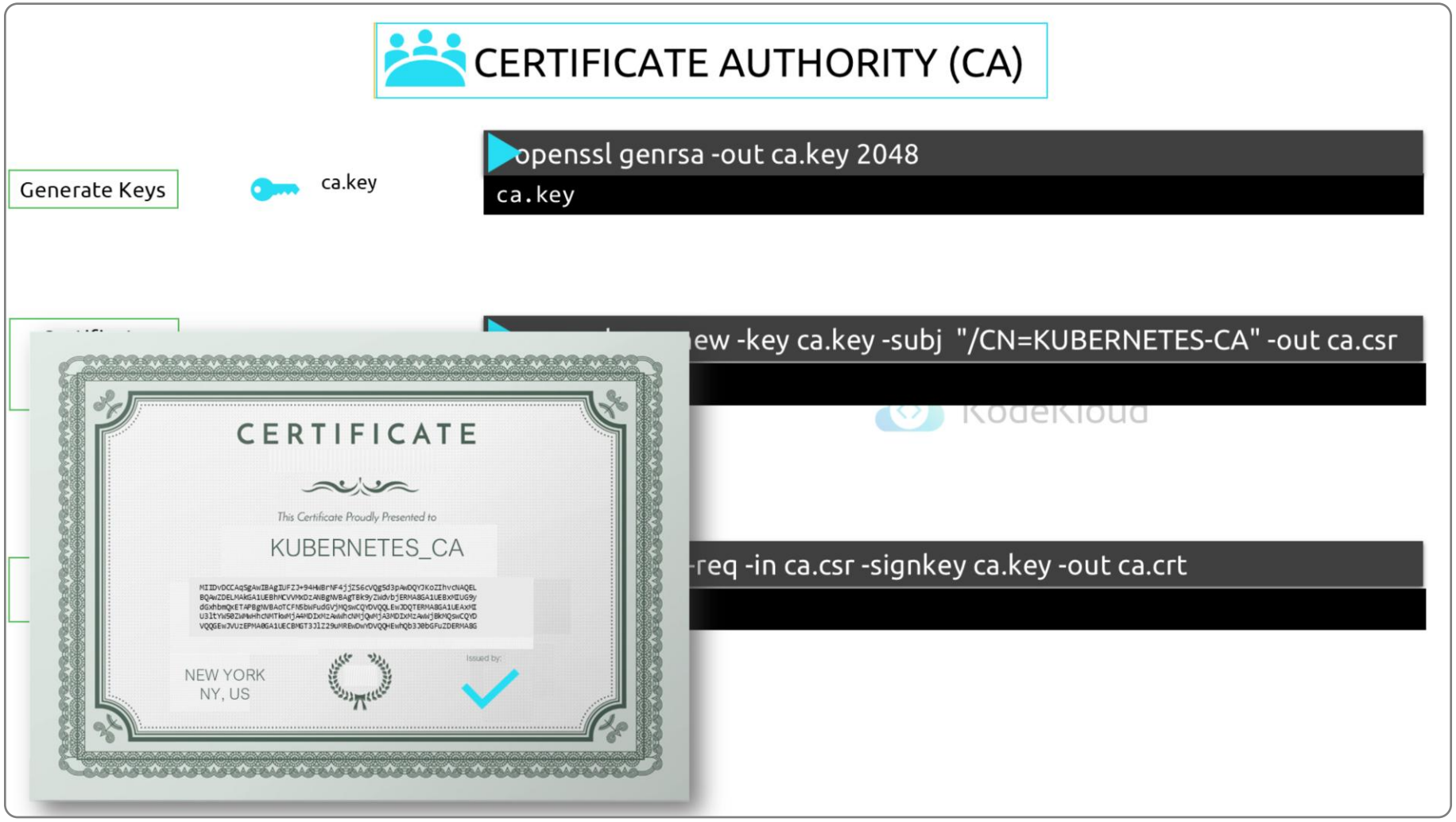
To generate certificates there are different tools available such as easyrsa, openssl or cfssl etc or many others.



In this lecture we will use openssl tool to generate the certificates.

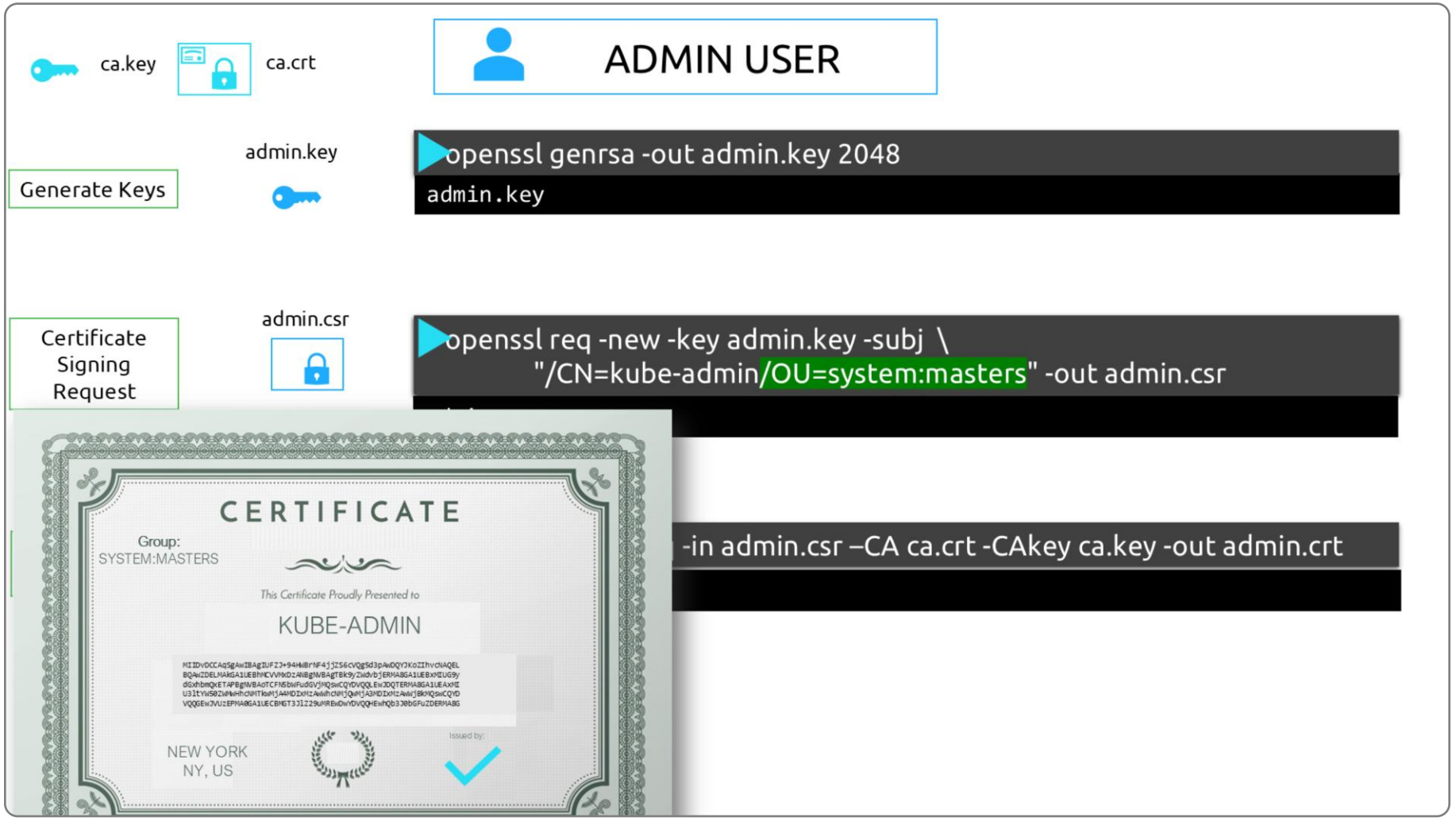


This is where we left off.



We will start with the CA certificates. First we create a private key using the openssl command, openssl genrsa –out ca.key. Then we use the openssl request command along with the key we just created, to generate a certificate signing request. 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 this 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-CA. Finally, we sign the certificate using the openssl x509 command and by specifying the certificate signing request we generated in the previous command. Since

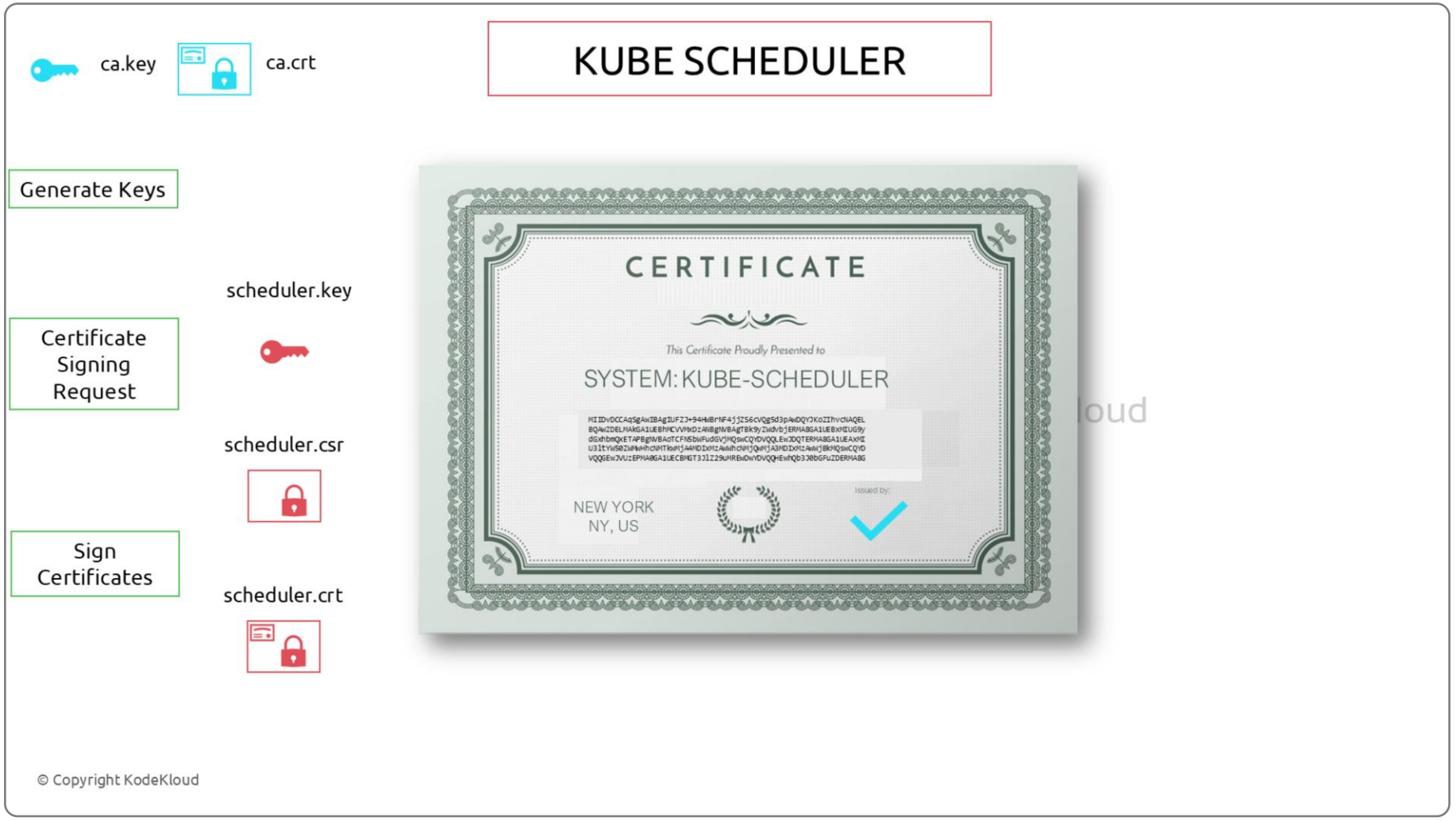
this is for the CA itself, it is self-signed by the CA using its own private key that it generated in the first step. Going forward for all other certificates we will use this ca key pair to sign them. The ca now has its private key and root certificate file. Great!



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 the kubectl client authenticates with when you run kubectl commands. 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 command. But this time, you specify the CA certificate and the

CA key. You are signing your certificate with the CA key pair. That makes this a valid certificate. 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. The certificate is the validated user id and the key is like the password. It's just that its much more secure than a simple username and password.

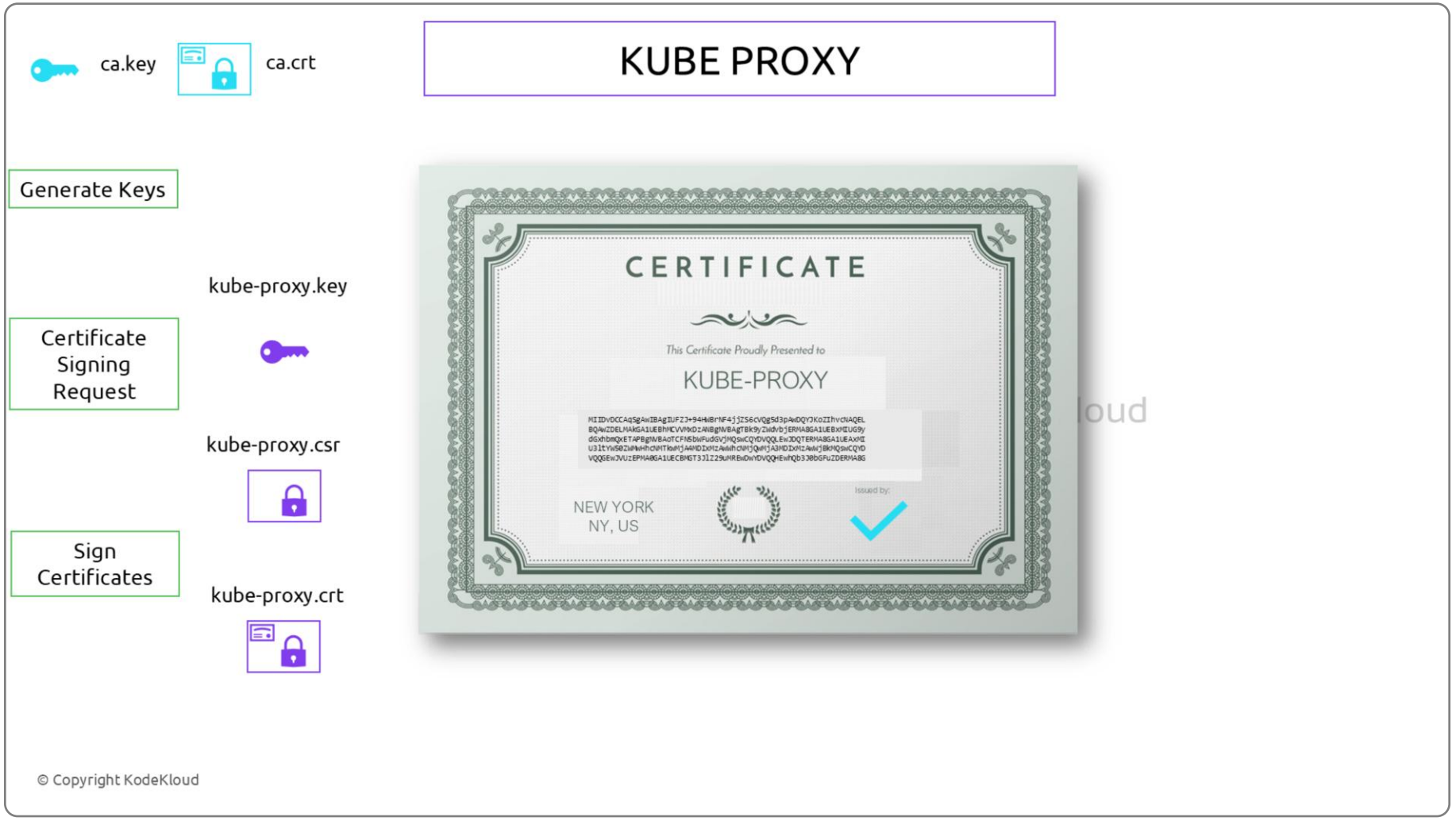
So this is for the admin user. How do you different this user from any other users. The user account needs to be identified as an admin user and not just another basic user. You do that by adding the group details for your user in the certificate. In this case a group named SYSTEM:MASTERS exists on kubernetes with administrative privileges. We will discuss about groups later. But for now its important to know 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 signing request. Once its signed, we now have our certificate for the admin user with admin privileges.



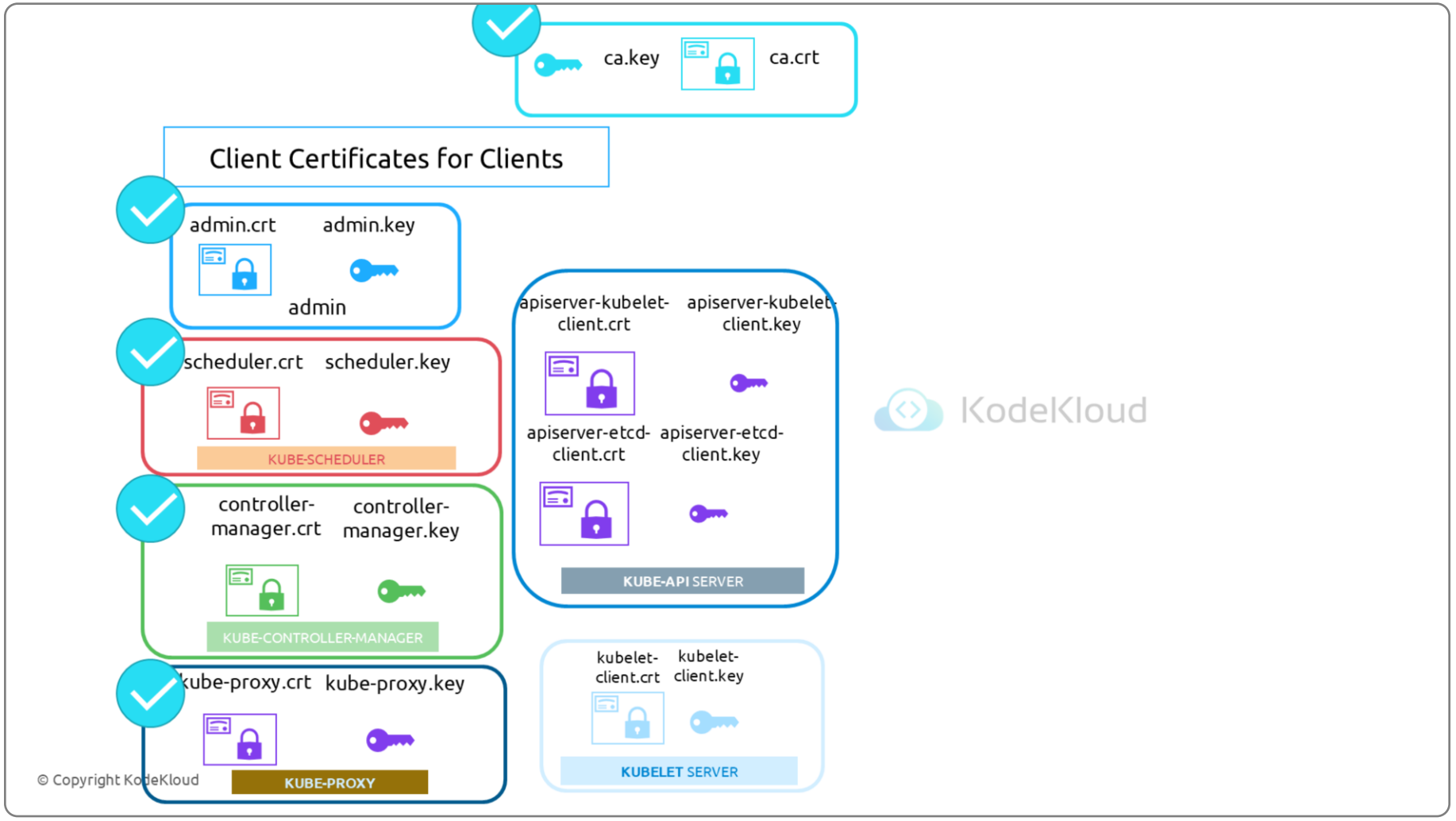
We follow the same process to generate client certificates for all other components that access the kube-api server.

The kube-scheduler. Now the kube-scheduler is a system component part of the kubernetes control plane. So it's name must be pre-fixed with the key word "system".

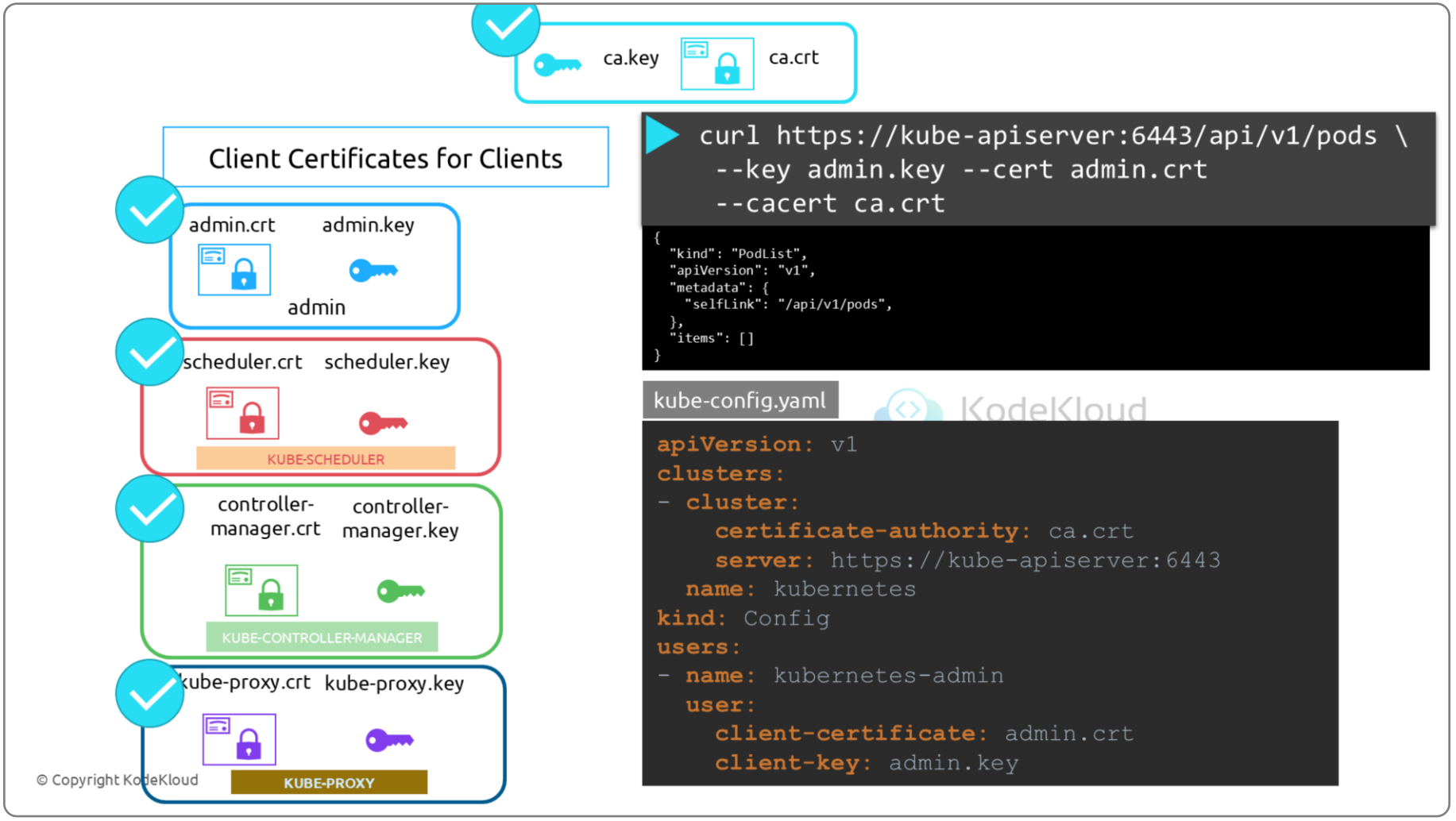
The same with Kube-controller-manager. It is again a system component. So its name must be prefixed with the keyword system.



And finally kube-proxy



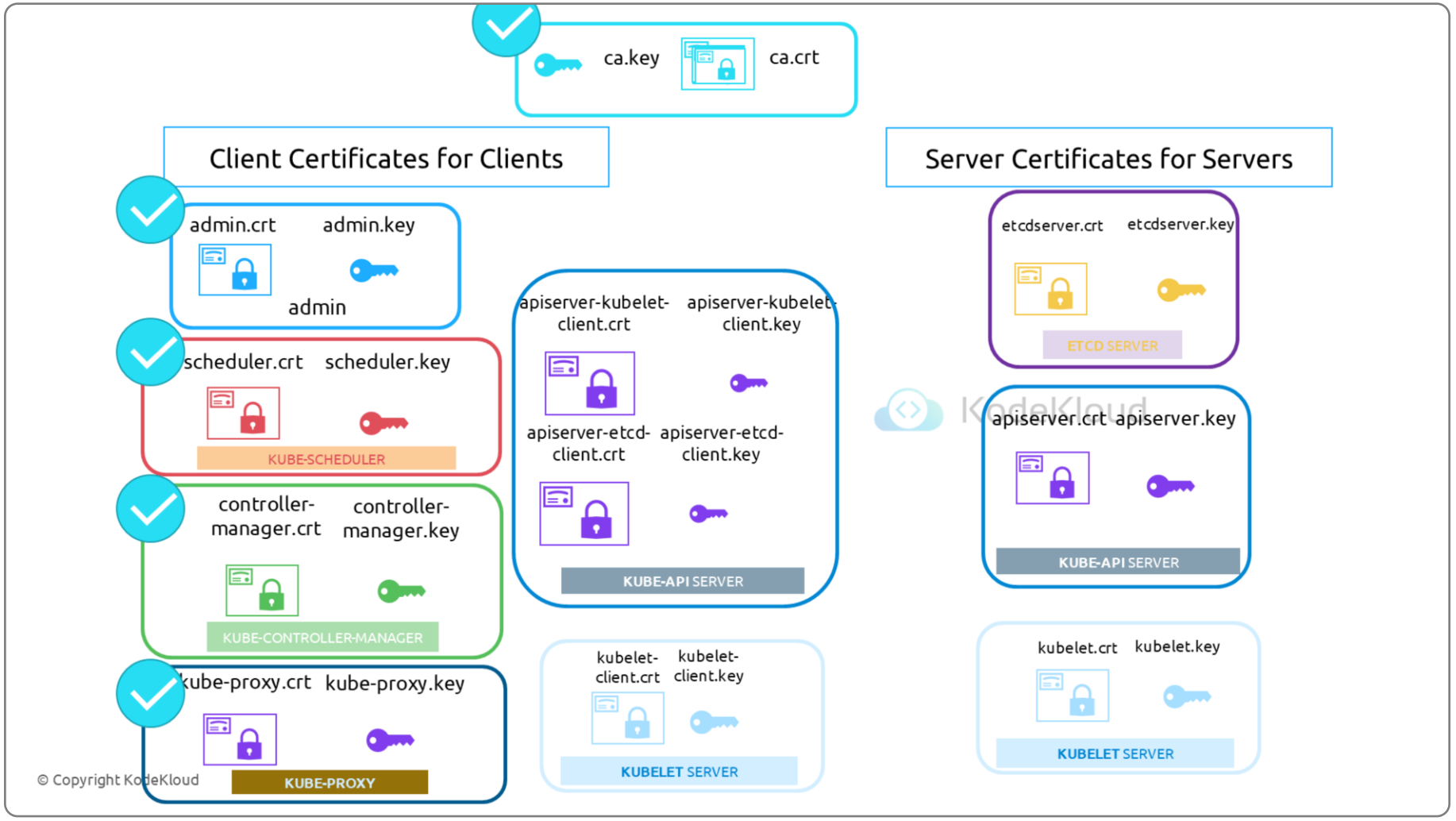
So, so far we have created CA certificates, then all of the client certificates including the admin user, scheduler, controller-manager, kube-proxy. We will follow the same procedure to create the remaining 3 client certificates for the apiservers and kubelets when we create the server certificates for 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 the certificate instead of a user 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.

The other way is to move all of these parameters into a configuration file called kube-config. Within that specify the api server endpoint details, the certificates to use etc. This is what most of the kubernetes clients use. We will look at kube-

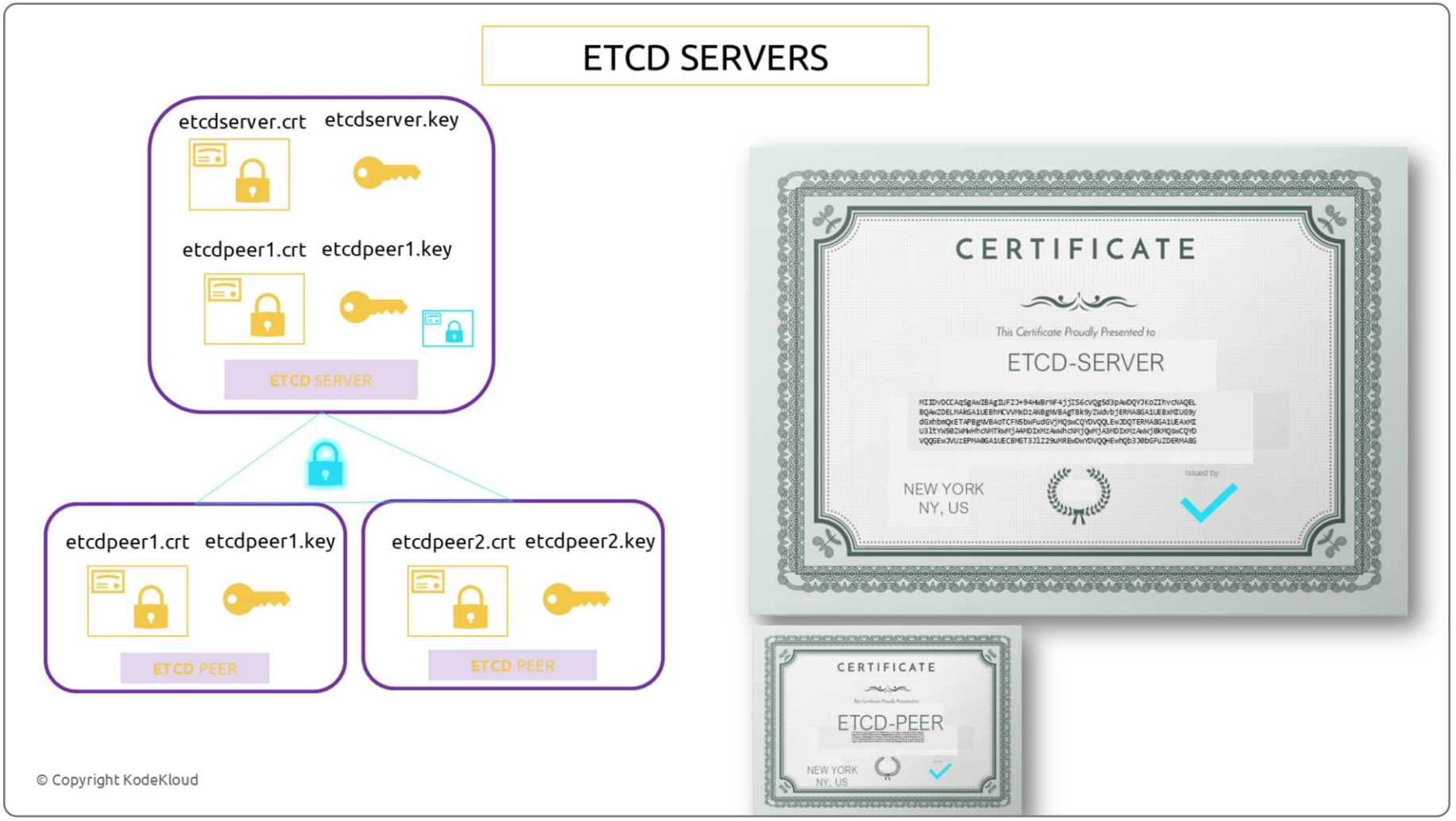
config in depth in one of the upcoming lectures.



Ok.
So we are now left with the server side certificates. But before we proceed one more thing. Remember in the pre-requisite lecture we mentioned that for the clients to validate the certificate 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 users 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

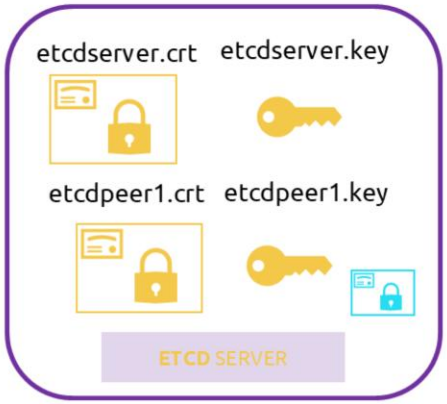
root certificate as well.

Let’s look at the server side certificates now.



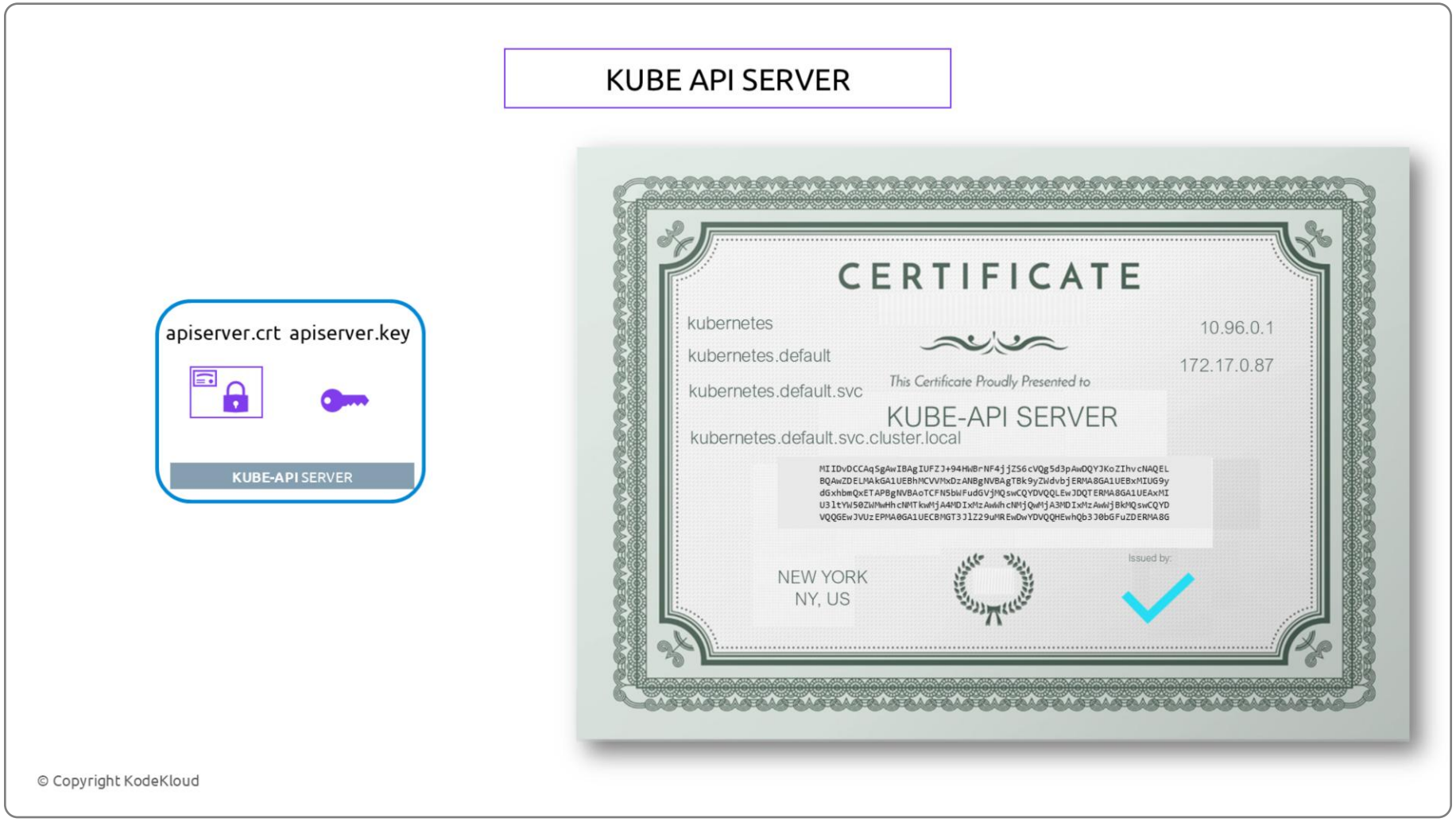
Let's start with the ETCD server. We follow the same procedure as before to generate a certificate for ETCD. We will name it ETCD-SERVER. ETCD Server can be deployed as a cluster across multiple servers as in a High Availability environment. In that case to secure communication between the different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

ETCD SERVERS




```
cat etcd.yaml
- etcd
    - --advertise-client-urls=https://127.0.0.1:2379
    - --key-file=/path-to-certs/etcdserver.key
    - --cert-file=/path-to-certs/etcdserver.crt
    - --client-cert-auth=true
    - --data-dir=/var/lib/etcd
    - --initial-advertise-peer-urls=https://127.0.0.1:2380
    - --initial-cluster=master=https://127.0.0.1:2380
    - --listen-client-urls=https://127.0.0.1:2379
    - --listen-peer-urls=https://127.0.0.1:2380
    - --name=master
    - --peer-cert-file=/path-to-certs/etcdpeer1.crt
    - --peer-client-cert-auth=true
    - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
    - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    - --snapshot-count=10000
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

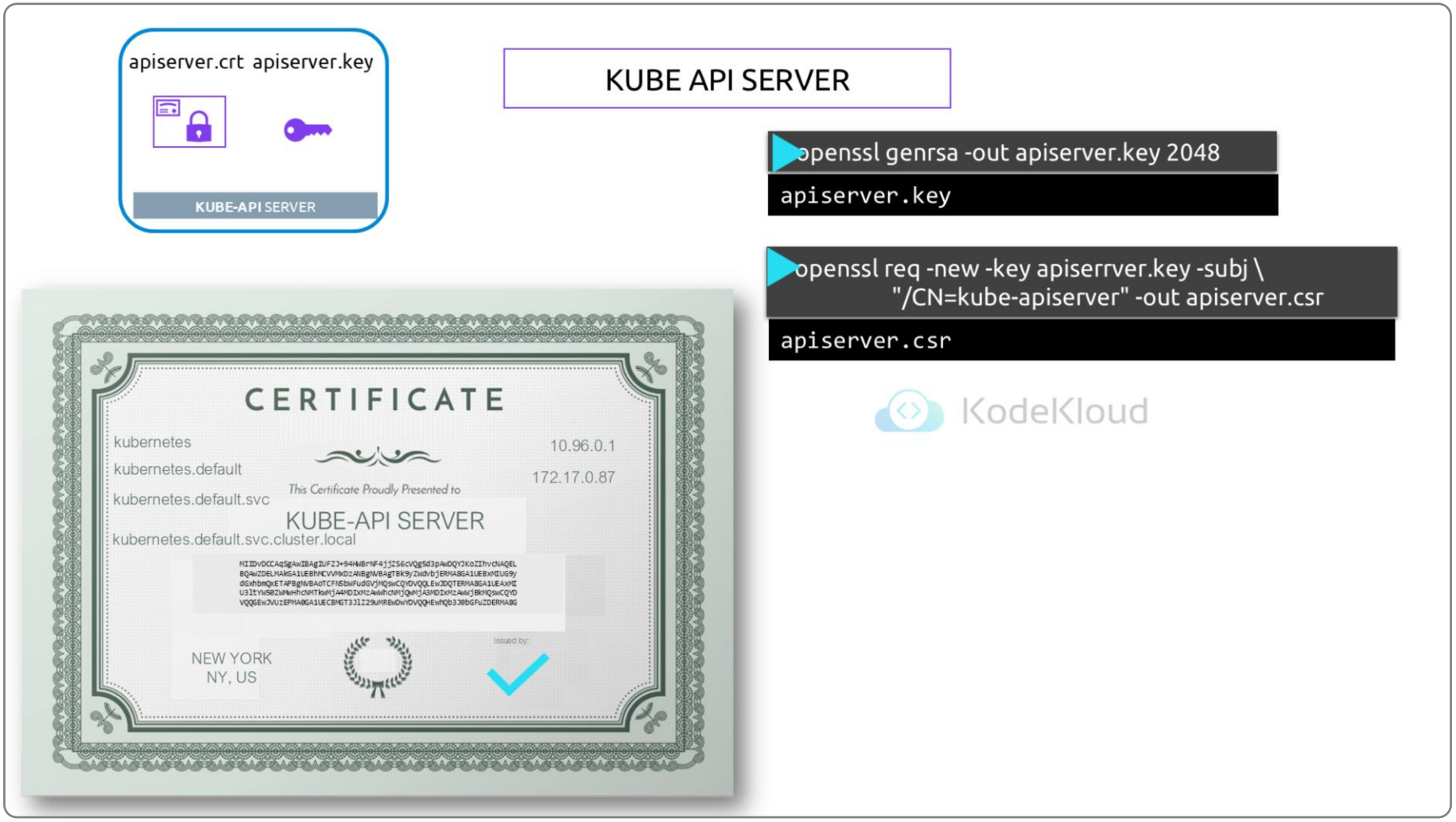

Once the certificates are generated specify them while starting the ETCD server. There are key and cert file options where you specify the etcdserver keys. There are other options available for specifying the peer certificates. And finally as we discussed earlier it requires the CA root certificate to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



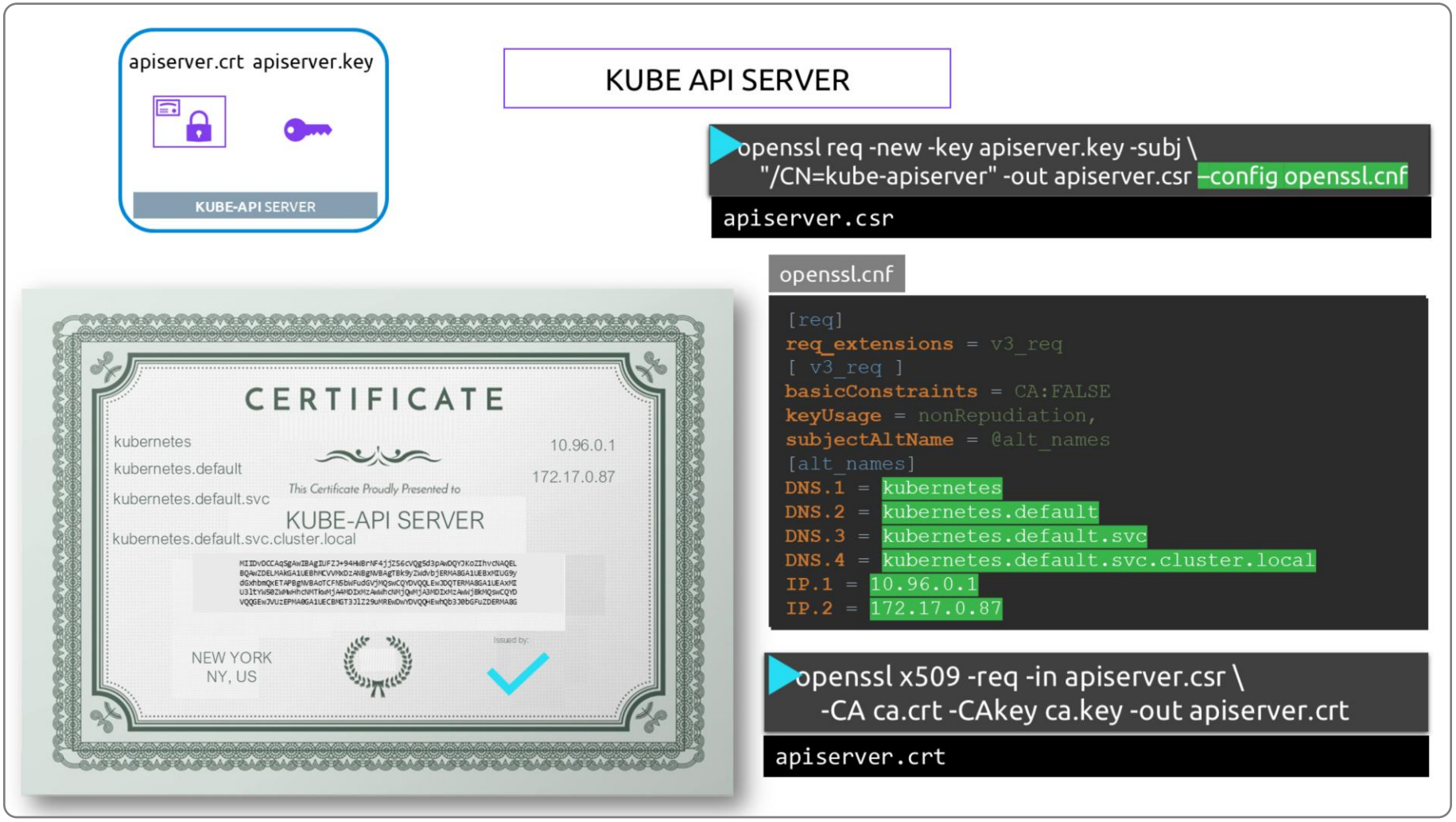
Let’s talk about the kube-api server now. We generate a certificate for the API server like before. But wait, the API server is the most popular of all the components within the cluster. Everyone talks to the kube-api server. Every operation goes through the kube-api server. Anything moves within the 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. It’s 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. While some refer to it as kubernete.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 kubernetes by these names will be able to establish a valid connection.

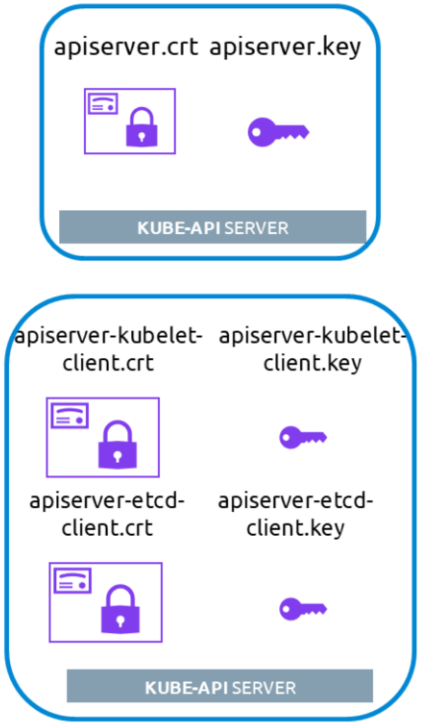


So we use the same set of commands as earlier to generate a key. In the certificate signing request you specify the name KUBE-APIserver. But how do you specify all the alternate names?



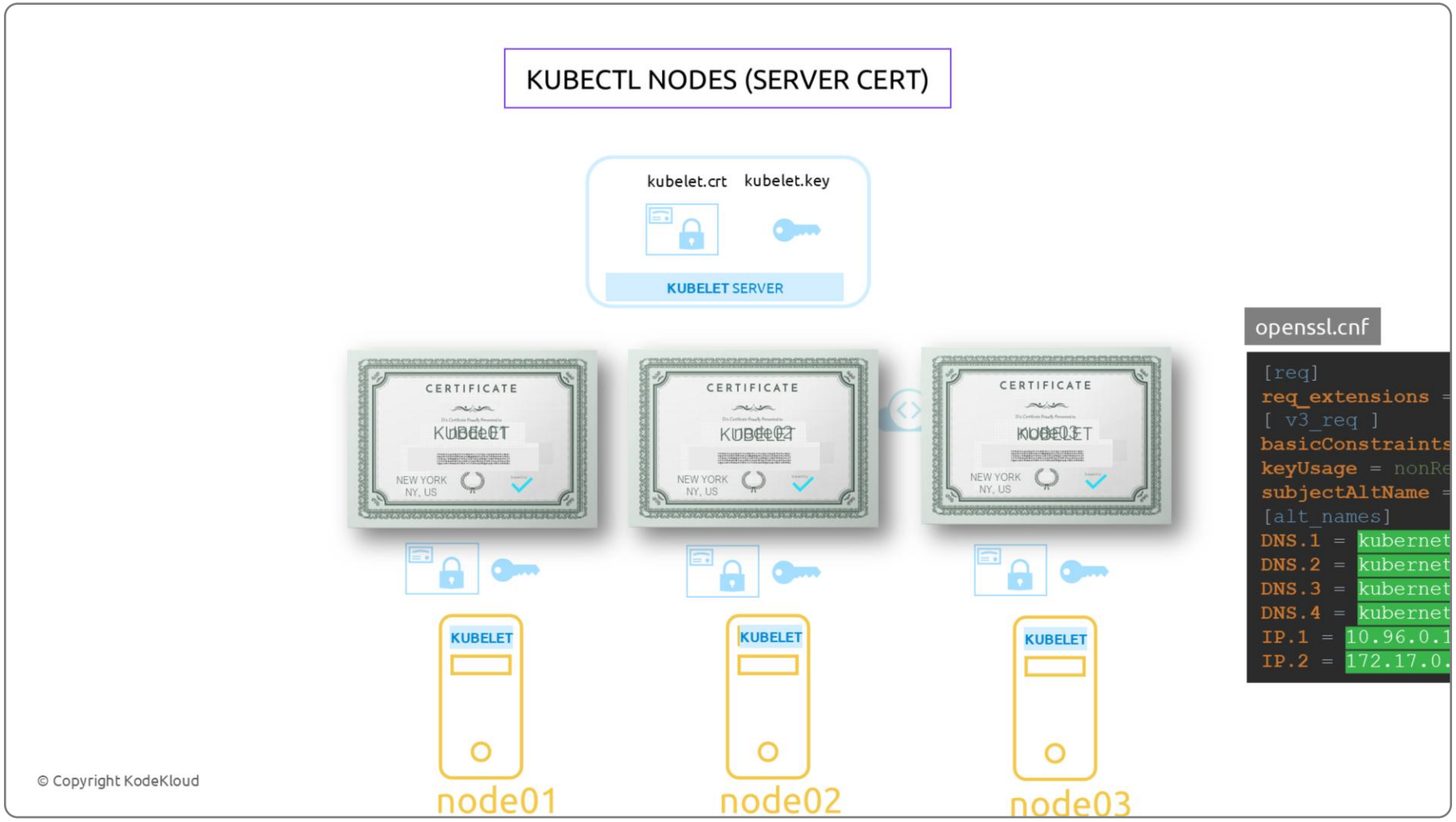
For that you must create an openssl config file. Create an openssl.cnf file and specify the alternate names in the alt_names section of the file. Include all the DNS names the api-server goes by as well as the IP address. Pass this config file as an option while generating the certificate signing request. Finally, sign the certificate using the CA certificate and key. You then have the kube api server certificate.

KUBE API SERVER



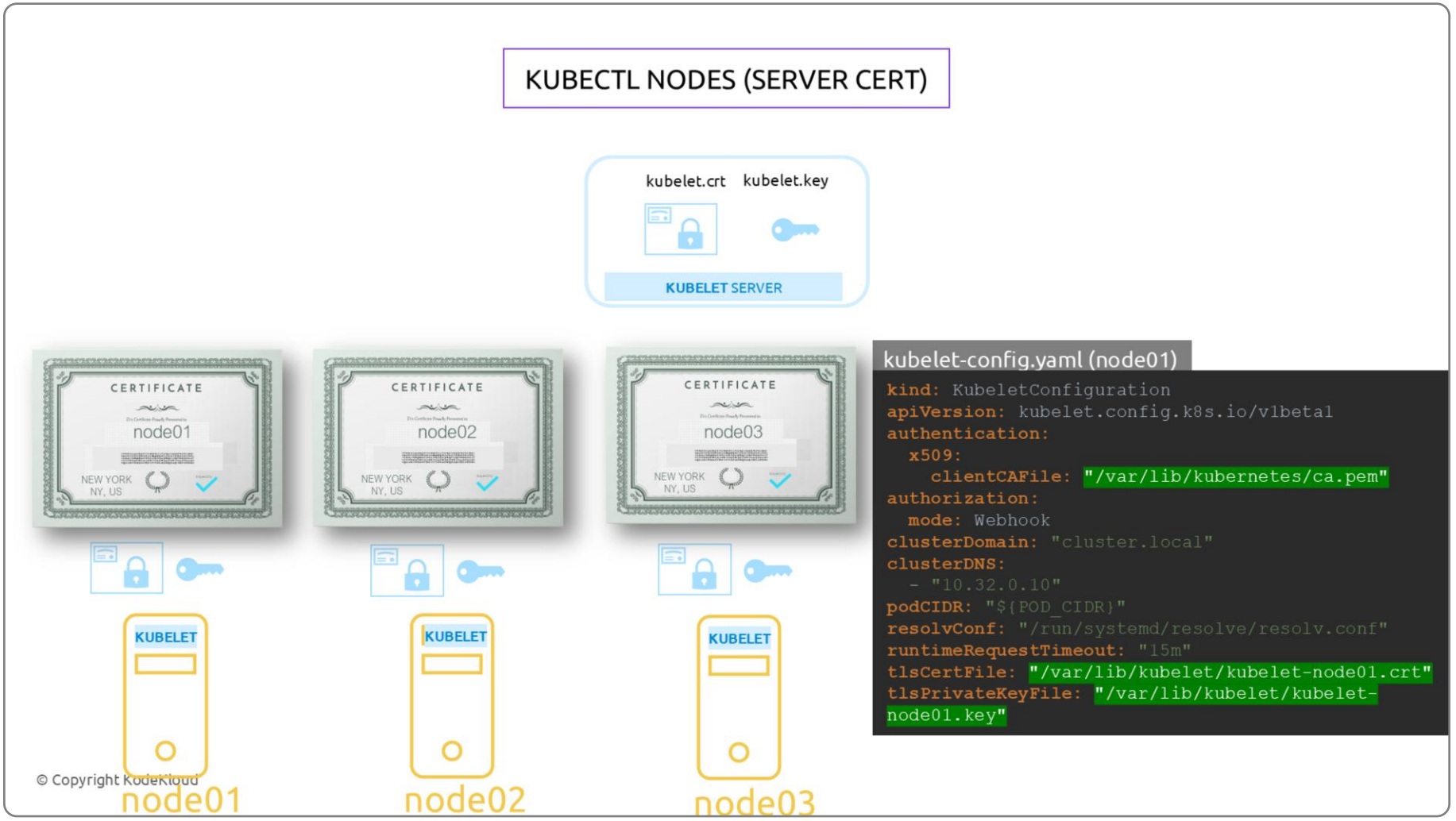

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/apiserver-client.crt \\
  --kubelet-client-key=/var/lib/kubernetes/apiserver-client.key \\
  --kubelet-https=true \\
  --runtime-config=api/all \\
  --service-account-key-file=/var/lib/kubernetes/service-account.pem \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --tls-cert-file=/var/lib/kubernetes/apiserver.crt \\
  --tls-private-key-file=/var/lib/kubernetes/apiserver.key \\
  --v=2
```


It is time to look at where we are going to specify these keys. Remember to consider the apiserver 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. Remember every component needs the CA certificate to verify its clients. Then we provide the apiserver certificates under the tls-cert options. We then specify the client certificates used by the kube-api server to connect to the etcd server, again with the CA file. And finally the kube-api server client certificate to connect to the kubelets.



Next comes the kubelet server. The kubelet 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 CA certificate. And then provide the kubelet node certificates. 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 controller-manger 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 kube-config files as we discussed earlier.

Well that's it for this lecture. In the next lecture we will see how you can view certificate information and how certificates are configured by the kubeadm tool.

www.kodekloud.com

In this lecture we look at how to view certificates in an existing cluster.

So you join a new team to help them manage their kubernetes environment. You are a new administrator to this team. You have been told that there are multiple issues related to certificates in the environment. So you are asked to perform a health check of all the certificates in the entire cluster. What do you?

First of all its important to know how the cluster was setup. There are different solutions available of 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, the kubeadm tool deploys

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.

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)						

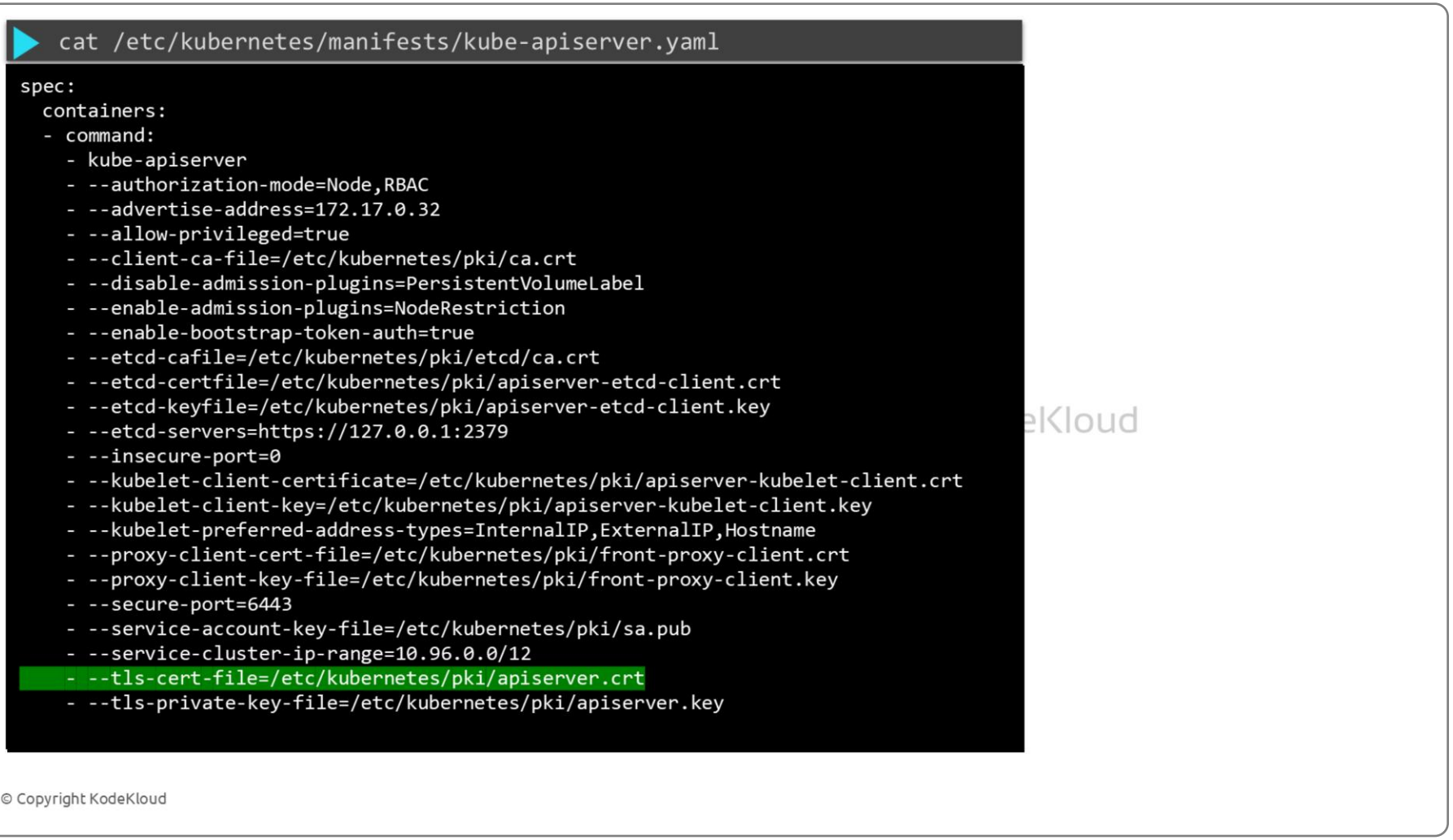
In this lecture we are going to look at a cluster provisioned by kubeadm as an example.

In order to perform a health check start by identifying all the certificates used in the system. I have created a sample Excel spreadsheet for you. Check out the resources link at the end of this lecture to access it.

So the idea is to create a list of certificate files used, their paths, the Names configured on them, the alternate names

configured if any, the organization the certificate account belongs to, the issuer of the certificate and the expiration date on the certificates.

So how do you get these? Start with the certificate files used.



For this, in an environment setup by kube-adm look for the kube-apiserver definition file under /etc/kubernetes/manifests folder. The command used to start the api server has information about all the certificates. Identify the certificate file used for each purpose and note it down. Next, take each certificate and look inside it to find more details. For example, we will start with the apiserver certificate file.

/etc/kubernetes/pki/apiserver.crt

```
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
```


Run the openssl x509 command and provide the certificate file as input to decode the certificate and view details.

Start with the name on the certificate under the Subject section. In this case its kube-apiserver. Then the alternative 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. Kubeadm names the kubernetes CA as kubernetes itself.

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

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

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

The certificate requirements are listed in detail in the Kubernetes Documentation page. Check the references section for the link.

Inspect Service Logs

```
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.
```


When you run into issues, you want to start looking at logs. If you setup 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 System's logging functionality.

View Logs

```
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.
```


In case you setup the cluster with kubeadm, then the various components are deployed as PODs. So you can look at the logs using kubectl logs command followed by the pod name.

View Logs

```
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
```

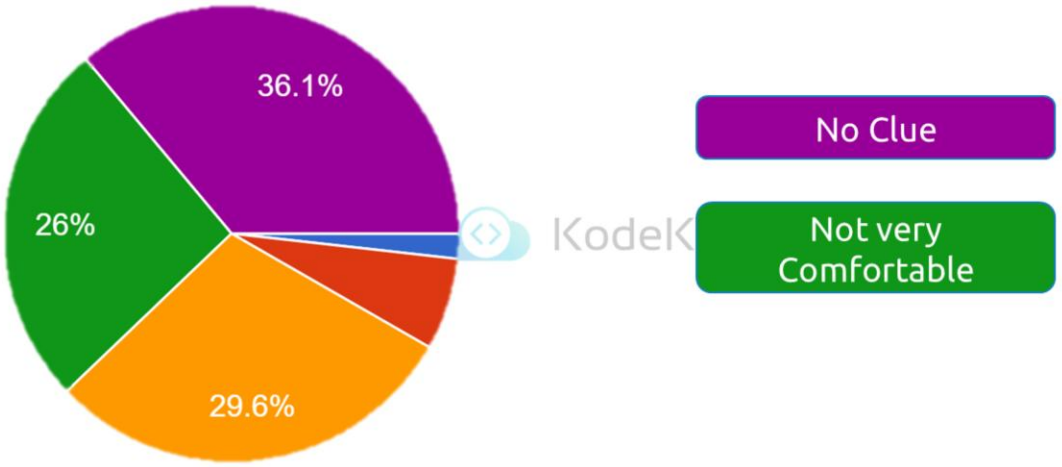

Sometimes if the core components such as the kubernetes api server or the etcd server are down, the kubectl commands wont function. In that case you have to go one level down to docker to fetch the logs. List all the containers using the docker ps –a command.

View Logs

```
docker logs 87fc
```

```
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.
```


And then view the logs using docker logs command followed by the container ID.



We sent out a Poll during the creation of this course about your knowledge on TLS Certificates. Most of you got back to us saying you had no clue about it or were not very comfortable. So we decided to put together a set of lectures that will help you gain enough knowledge to work with certificates in kubernetes. And of course these lectures will help you otherwise as well. If you are comfortable with this concept, feel free to skip the pre-requisite lectures and head straight over to the ones relevant to kubernetes.

Goals!

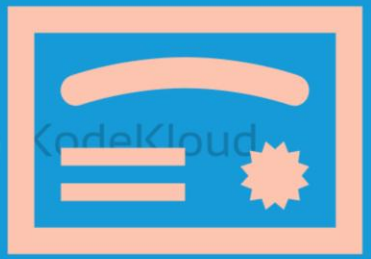
- ❑ What are TLS Certificates?
- ❑ How does Kubernetes use Certificates?
- ❑ How to generate them?
- ❑ How to configure them?
- ❑ How to view them?
- ❑ How to troubleshoot issues related to Certificates

KodeKloud

By the end of this section you should be an expert when it comes to working with certificates in general as well as with kubernetes. You should be able to easily configure and troubleshoot issues related to certificates. And the only way to do it, as I understand, is to learn and understand how it works in and out. And that's what we are going to do in this section.

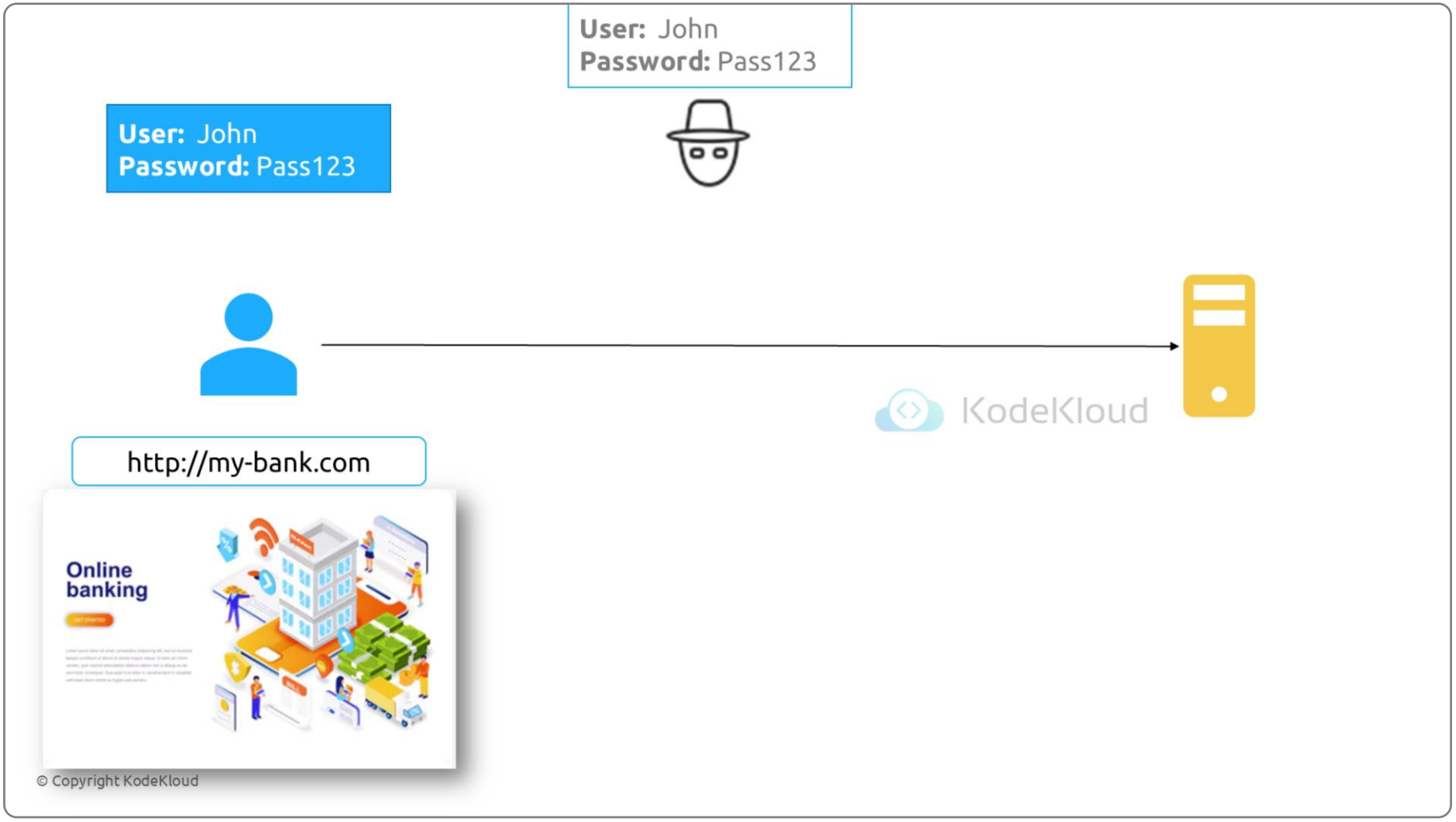
www.kodekloud.com

TLS CERTIFICATES (PRE-REQ)

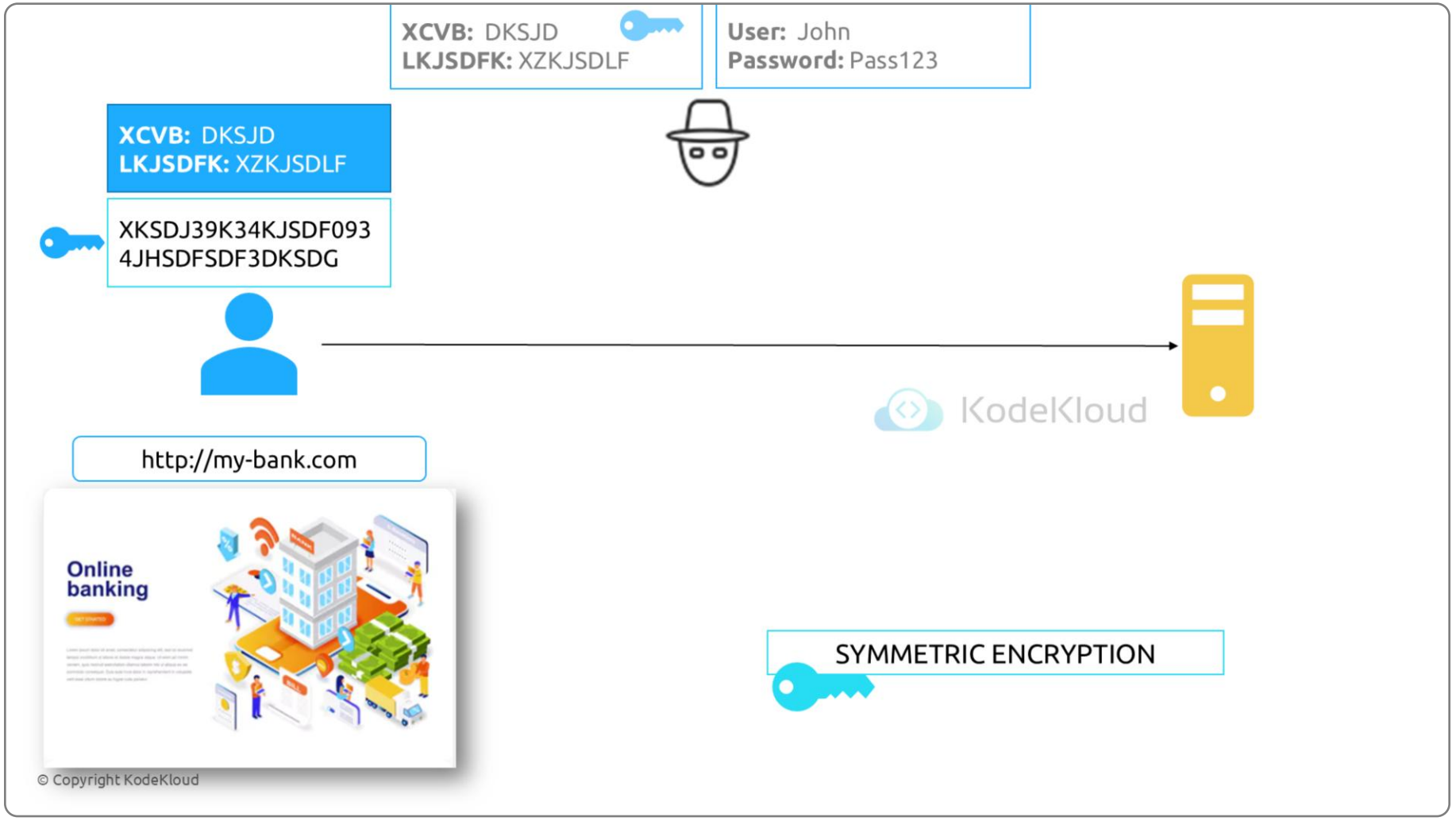


We will start with the basics of Certificates in general. If you are already comfortable with certificates, certificate authorities and how they work feel free to skip this lecture and go over to the next where we discuss this in the kubernetes terms.

A Certificate is used to guarantee trust between two parties during a transaction. For example, when a user tries to access a web server, TLS certificates ensure that the communication between the user and the server is encrypted and the server is who it says it is.



Let's take a look at a scenario. Without secure connectivity, if a user where to access his online banking application, the credentials he types in would be sent in a plain text format. A hacker sniffing network traffic could easily retrieve the credentials and use it to hack into the users bank account. Well that's obviously not safe. So you must encrypt the data being transferred using encryption keys.



The data is encrypted using a key, which is basically a set of random numbers and alphabets.

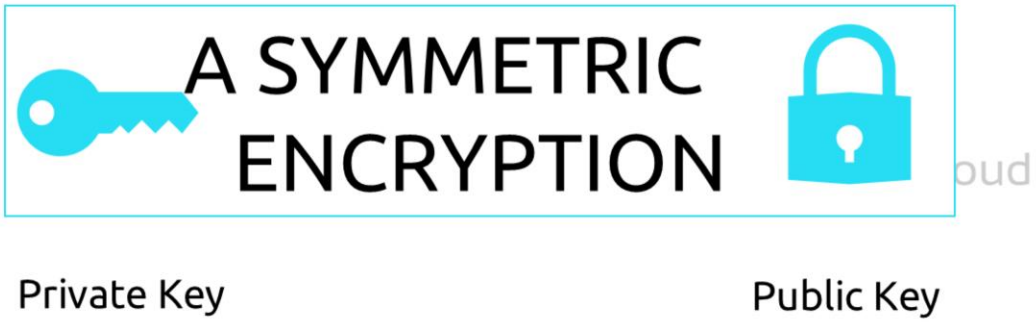
You add the random number to your data and you encrypt it into a format that cannot be recognized.
The data is then sent to the server. The hacker sniffing the network gets the data, but can't do anything with it.

However, the same is the case with the server receiving the data. It cannot decrypt the data without the key.

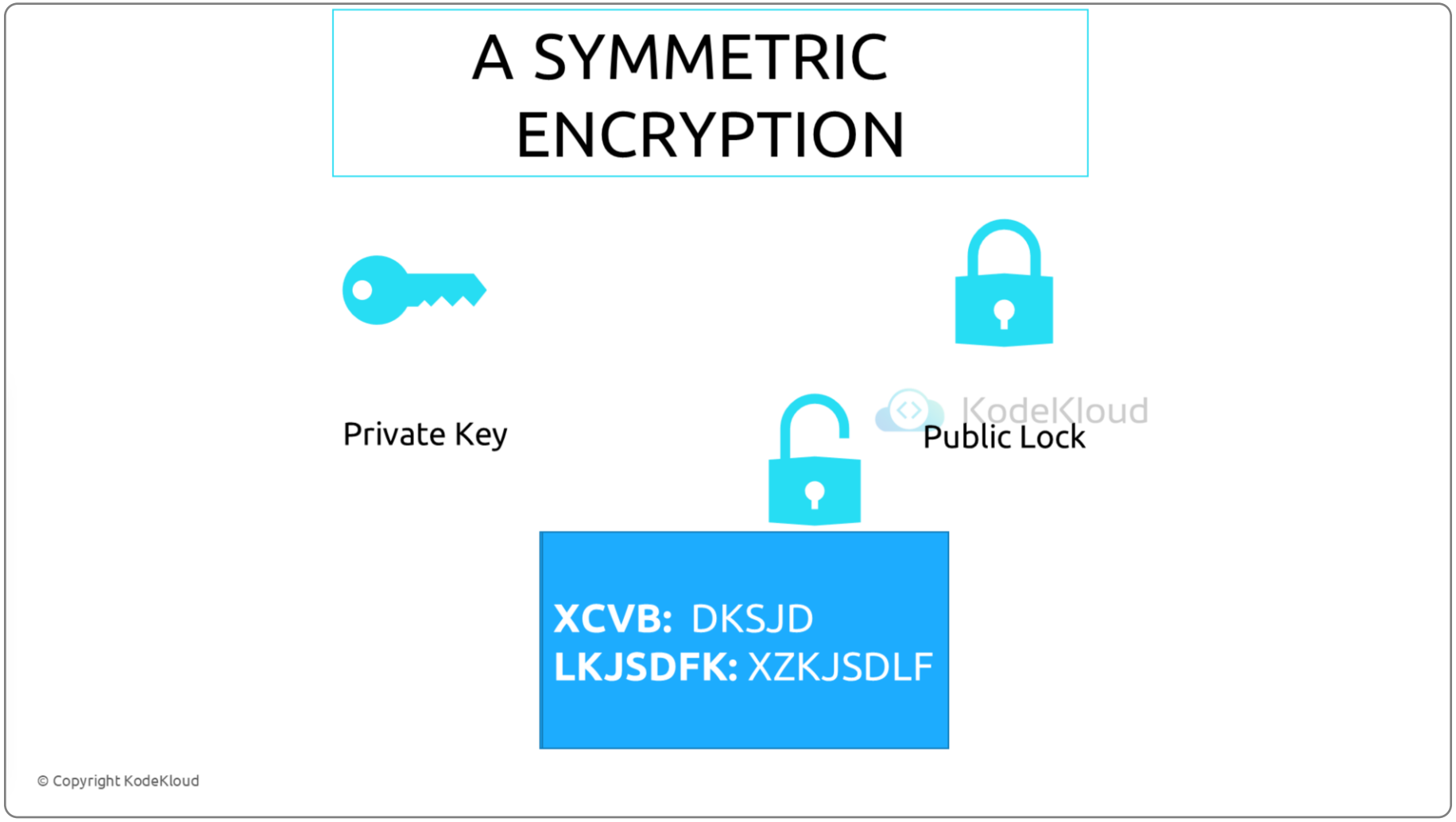
So a copy of the key must also be sent to the server so that the server can decrypt and read the message. Since the key is also sent over the same network, the attacker can sniff that as well and decrypt the data with it.

This is known as SYMMETRIC ENCRYPTION.

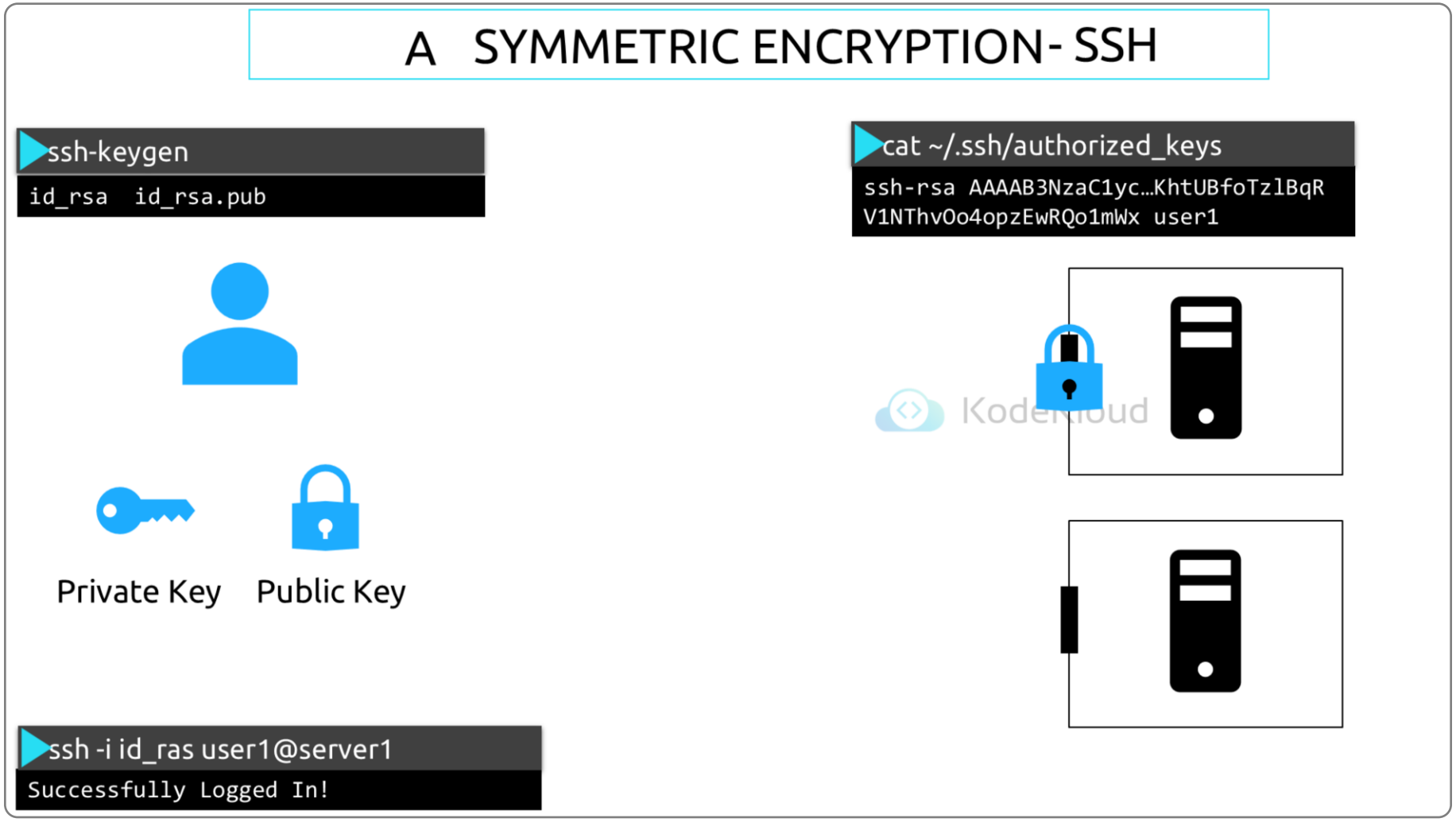
It is a secure way of encryption, but since it uses the same key to encrypt and decrypt the data and since the key has to be exchanged between the sender and the receiver, there is a risk of a hacker gaining access to the key and decrypting the data.



And that's where Asymmetric encryption comes in. Instead of using a single key to encrypt and decrypt data, asymmetric encryption uses a pair of keys. A private key and a public key. Well they are private and public KEYS, but for the sake of this example, we will call it a private key and a public lock. We will get back to that at the end, but for now think of it as a key and lock pair. A key which is only with me, so its private. A lock that anyone can access, so its public.



The trick here is if you encrypt or lock data with your lock, you can only open it with the associated key. So your key must always be secure with you and not be shared with anyone else. It's private. But the lock is public and may be shared with others. But they can only lock something with it. No matter what is locked using the public lock, it can only be unlocked by your private key.



Before we go back to our web server example, let's look at an even simpler use case of securing SSH access to servers through key pairs.

You have a server in your environment that you need access to. You don't want to use passwords as they are too risky. So you decide to use key pairs. You generate a public and private key pair. You can do this by running

the ssh_keygen command. It creates two files. Id_rsa is the private key and id_rsa.pub is the public key. Well, not a public key, a public lock.

You then secure your server by locking down all access to it, except through a door that is locked using your public lock. It's usually done by adding an entry with

your public key into the servers .ssh authorized_keys file.

So you see, the lock is public and anyone can attempt to break through, but as long as no one get's their hands on your private key which is safe with you on your laptop, no one can gain access to the server.

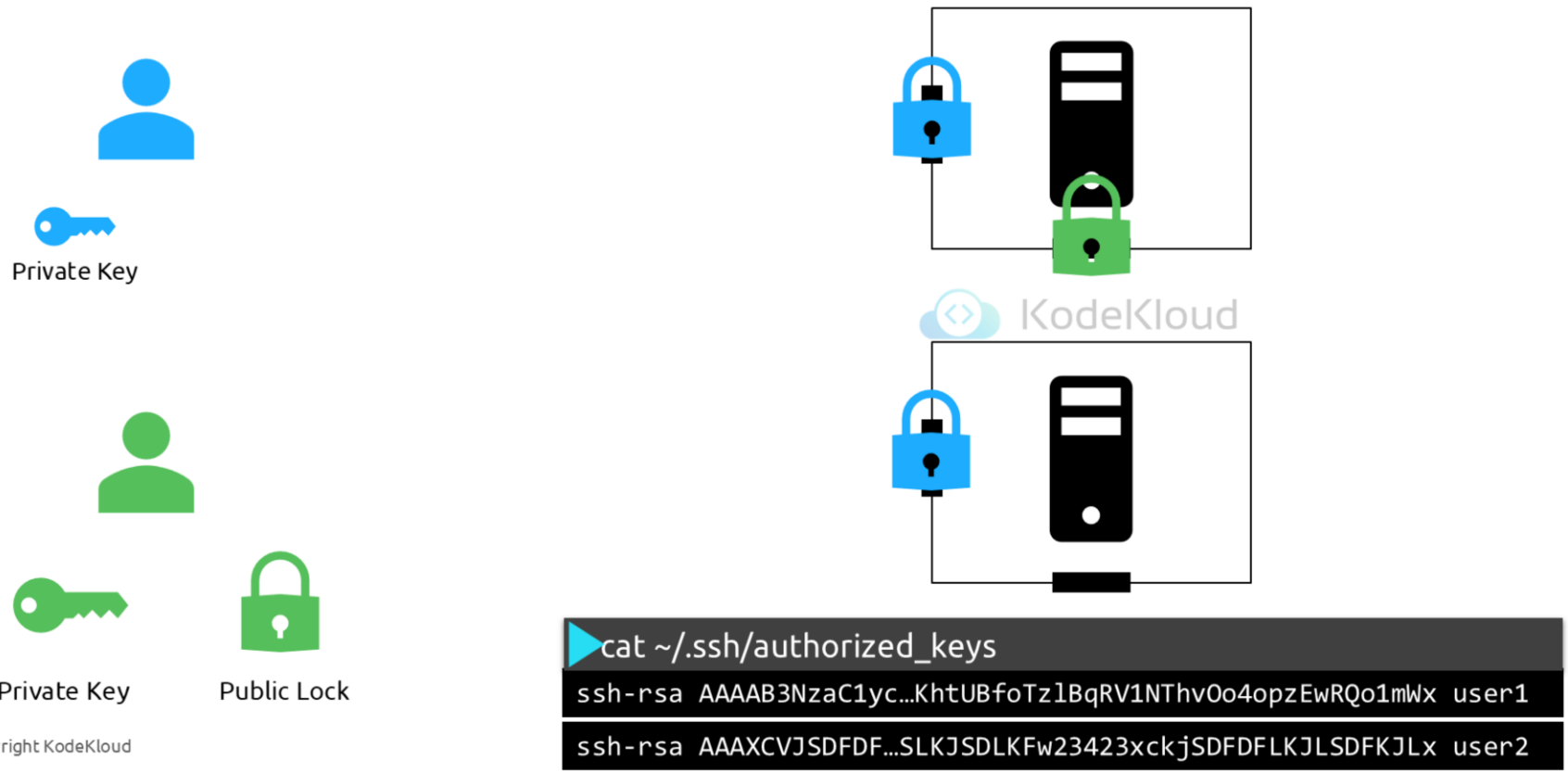
When you try to SSH you specify the location of your private key in your SSH command.

What if you have other servers in your environment?

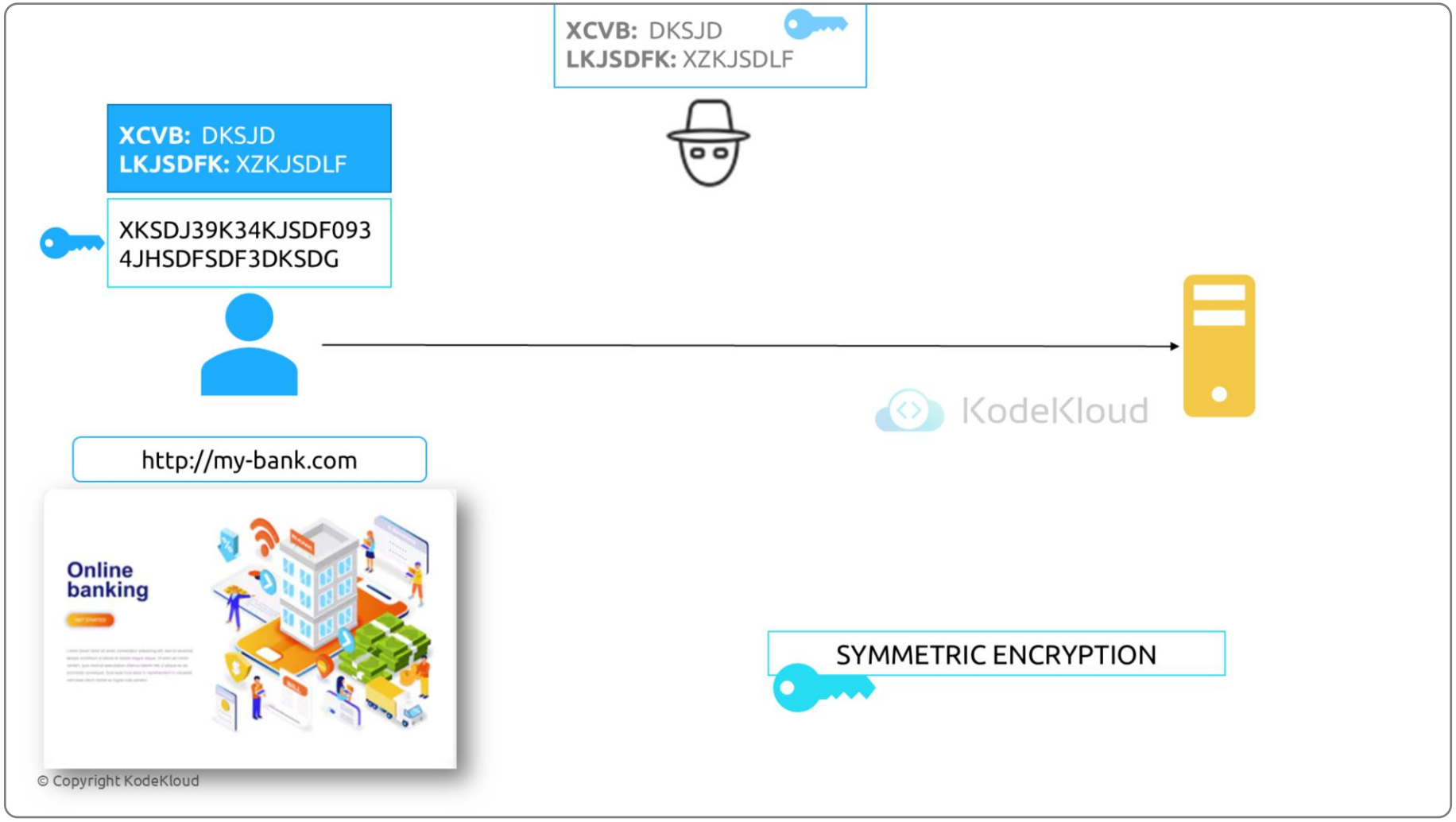
How do you secure more than one server with your key pair?

Well, you can create copies of your public lock and place them on as many servers as you want. You can use the same private key to SSH into all of your servers securely.

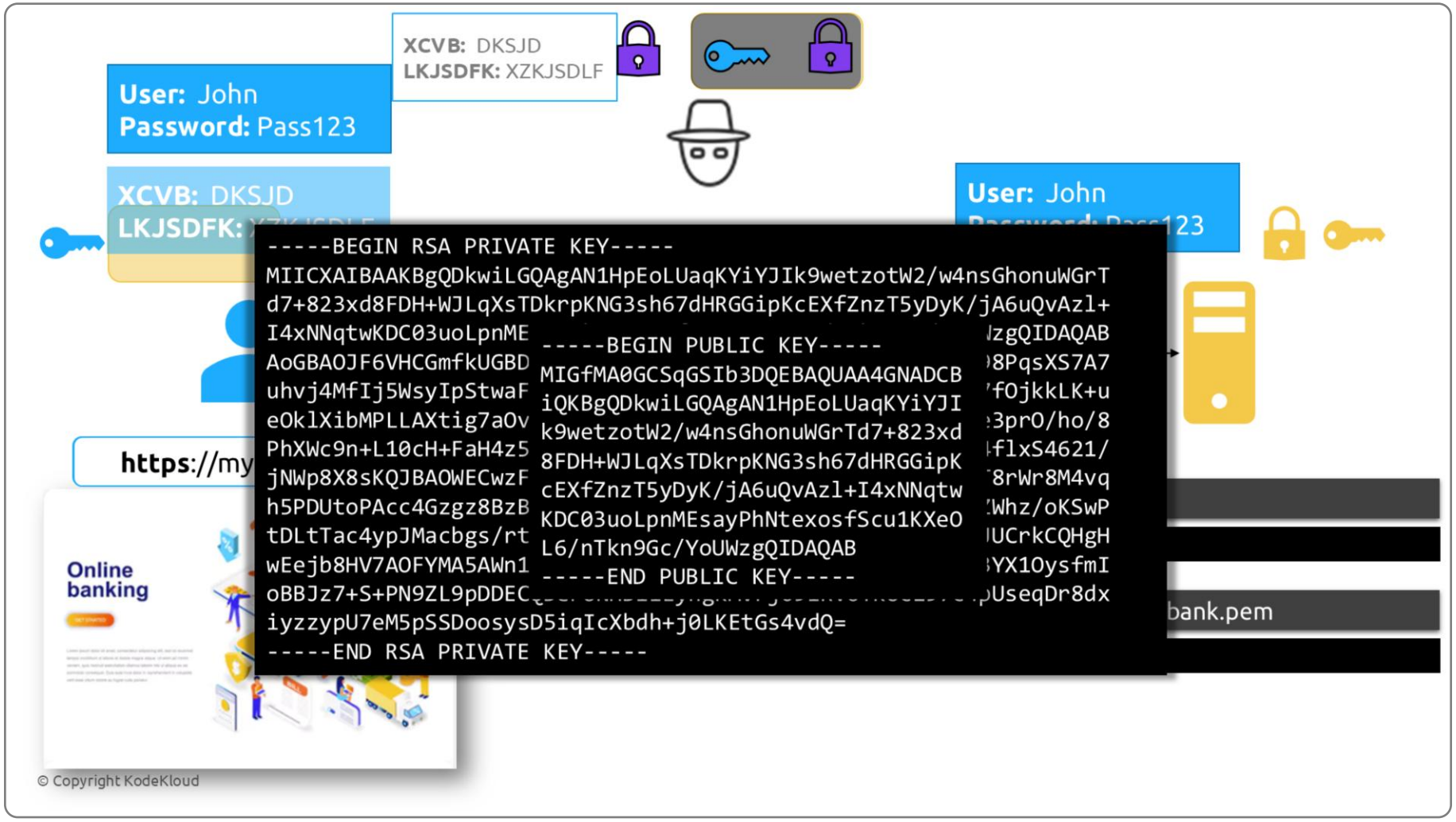
A SYMMETRIC ENCRYPTION- SSH



What if other users need access to your servers? Well they can do the same thing, They can generate their own public and private key pairs. As the only person who has access to those servers, you can create an additional door for them and lock it with their public locks. Copy their public keys to all the servers and now other users can access the servers using their private keys.



Let’s go back to our web server example. You see, the problem we had earlier with symmetric encryption was that the key used to encrypt data had to be sent to the server over the network along with the encrypted data. And so there is a risk of the hacker getting the key to decrypt data. What if we could somehow get the key to the server safely? Once the key is safely made available to the server, the server and client can safely continue to communicate with each other using symmetric encryption.



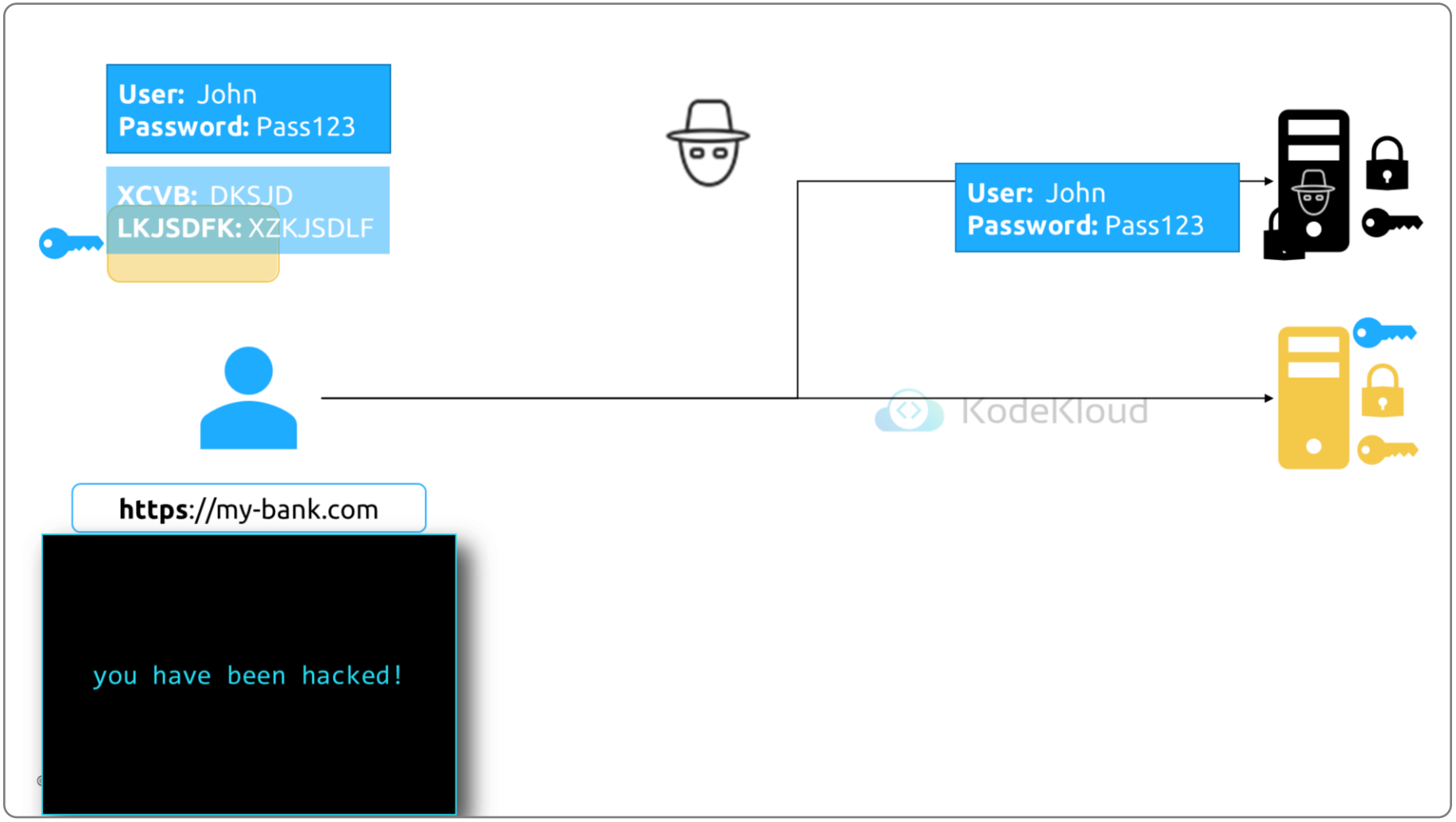
To securely transfer the Symmetric key from the client to the server, we use Asymmetric Encryption. So, we generate a public and private key pair on the server. We are going to refer to the public lock as public key, now that you have got the idea. The ssh-keygen command we used earlier creates a pair of keys for SSH purposes. So the format is a bit different. Here we use the openssl command to generate a private and public key pair. And that's how they look.

When the user first accesses the web server using https, he gets the public key from the server. Since the hacker is sniffing

all traffic, let us assume he too gets a copy of the public key. We'll see what he can do with it.

The user (in fact the user's browser) then encrypts the symmetric key using the public key provided by the server. The symmetric key is now secure. The user then sends this to the server. The hacker also gets a copy. The server uses the private key to decrypt the message and retrieve the symmetric key. However, the hacker does not have the private key to decrypt and retrieve the symmetric key from the message it received. The hacker only has the public key, with which he can only lock or encrypt a message and not decrypt the message.

The symmetric key is now safely available only to the user and the server. They can now use the symmetric key to encrypt data and send to each other. The receiver can use the same symmetric key to decrypt data and retrieve information. The hacker is left with encrypted messages and public keys with which he can't decrypt any data. With asymmetric encryption we have successfully transferred the symmetric keys from the user to the server and with symmetric encryption we have secured all future communication between them. Perfect!



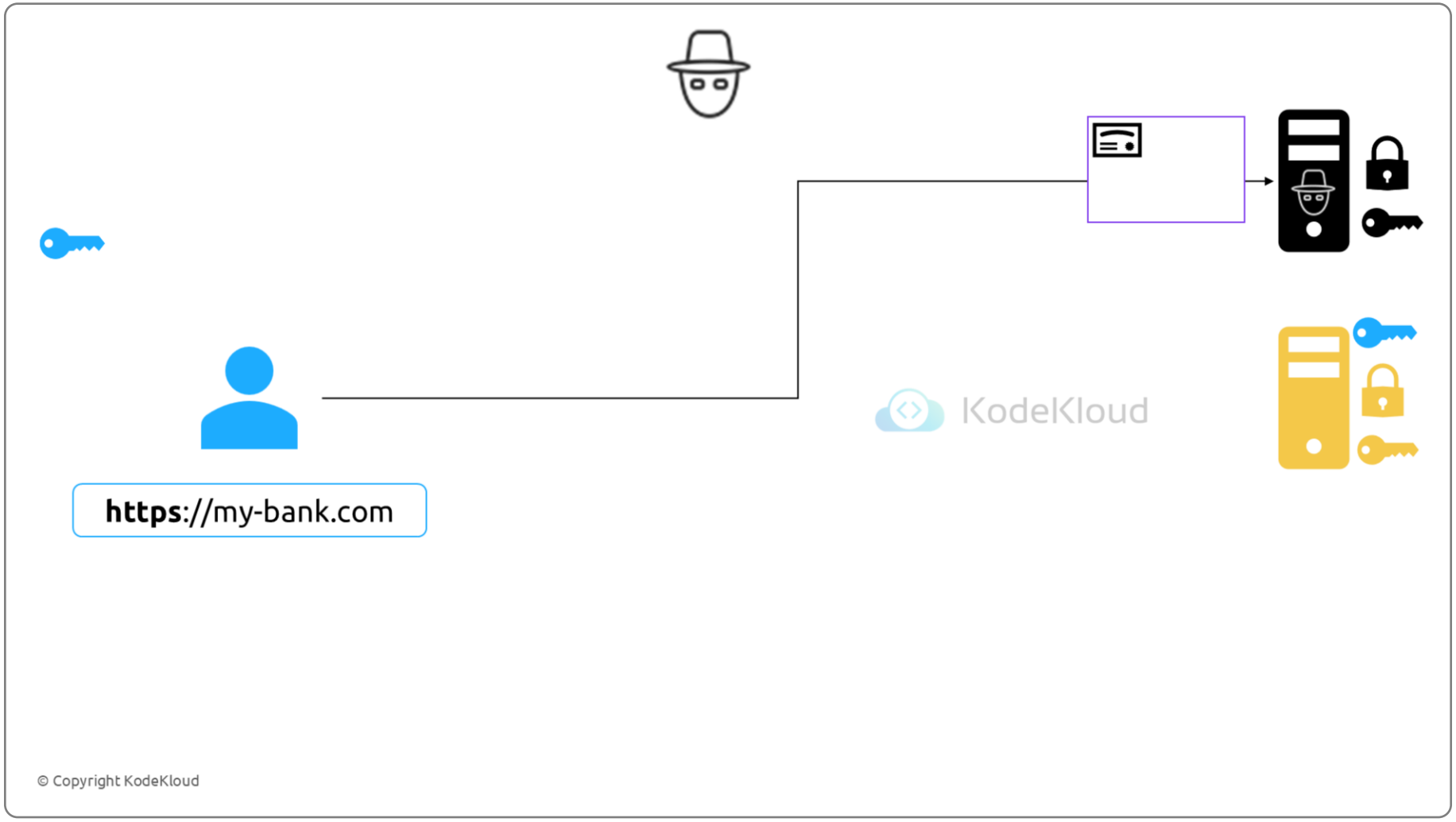
The hacker now looks for new ways to hack your account. And so he realizes that the only way he can get your credentials is by getting you to type it into a form he presents. So he creates a website that looks exactly like your banks website. The design is the same, the graphics are the same, the website is a replica of the actual banks website.

He hosts the website on his own server. He wants you to think, its secure too, so he generates his own set of public and private keys and configures them on his web server. And finally he manages to tweak your environment or your network

to route your requests to his server, when you try to reach your banks website.

When you open your browser and type the website address in, you see a very familiar page. The same login page of your bank that you are used to seeing. So you go ahead and type in the username and password. You made sure you typed in HTTPS in the URL to make sure the communication is secure and encrypted.

Your browser receives a key, you send the encrypted symmetric key, and then you send your credentials encrypted with the key and receiver decrypts the credentials with the same symmetric key. You have been communicating securely in an encrypted manner but with the hackers server. As soon as you send in your credentials, you see a dashboard that doesn't look very much like your bank's.



What if you could look at the key you received from the server and say if it is a legitimate key from the real bank's server? When the server sends the key, it does not send the key alone. It sends a certificate that has the key in it.

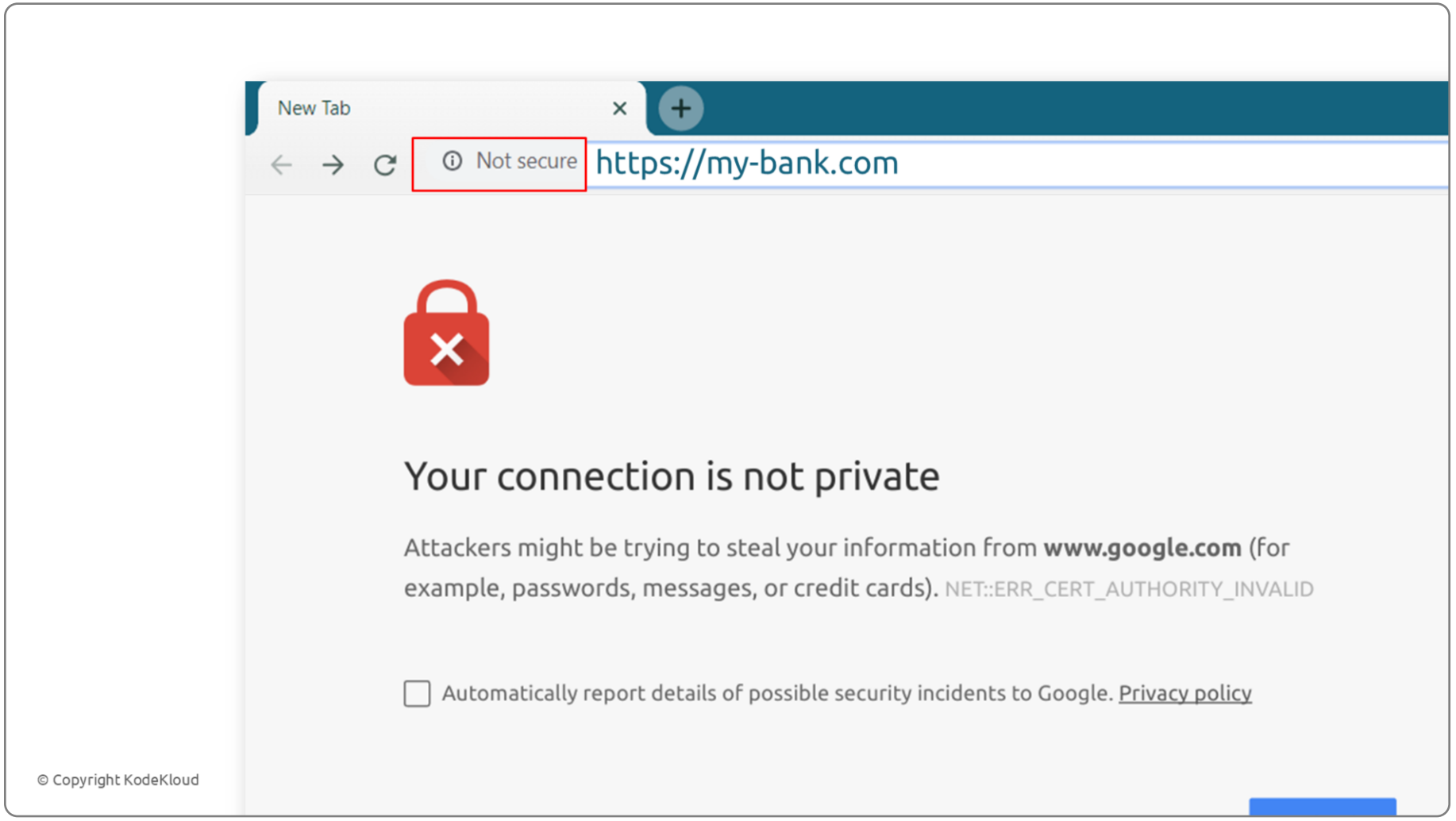
If you take a closer look at the certificate, you will see that it is like an actual certificate but in a digital format. It has information about who the certificate is issued to, the public key of that server, the location of that server etc.

On the right you see output of an actual certificate. Every certificate has a name on it – the person or subject to whom the certificate is issued to. That is very important as that is the field that helps you validate their identity. If this is for a web server, this must match what the user types in, in the url on his browser.

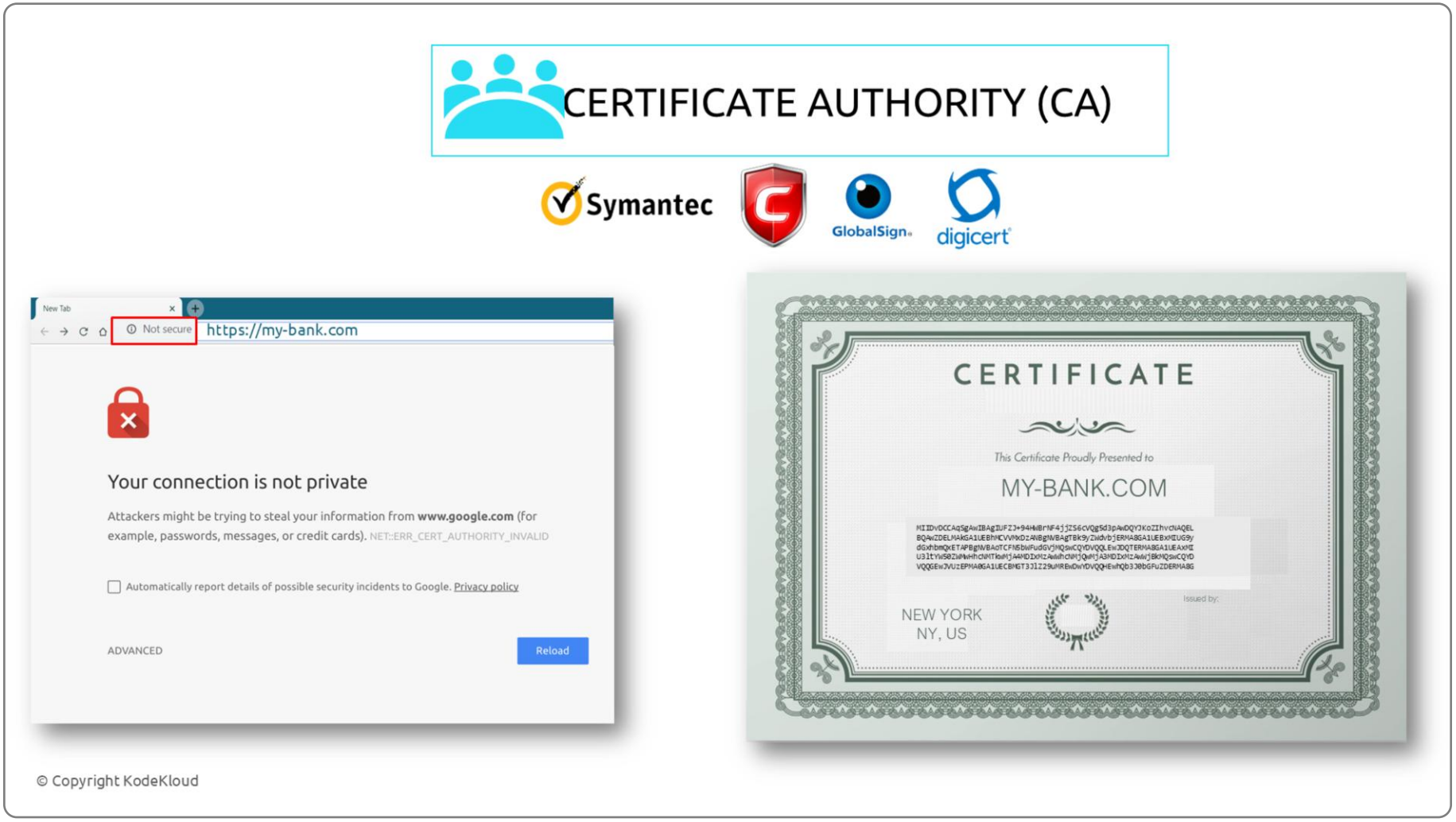
If the bank is known by any other names, and if they'd like their users to access their application with the other names as well, then all those names should be specified in this certificate, under the subject alternative names section.

But you see anyone can generate a certificate like this. You could generate one for yourself saying you are google. And that's what the hacker did in this case. He generated a certificate saying he is your banks website. So how do you look at a certificate and verify if it is legit? That is where the most important part of the certificate comes in to play – Who signed and issued this certificate? If you generated a certificate then you will have to sign it by yourself. That is known as a self signed certificate. Anyone looking at the certificate you generated will immediately know that it is not a safe certificate.

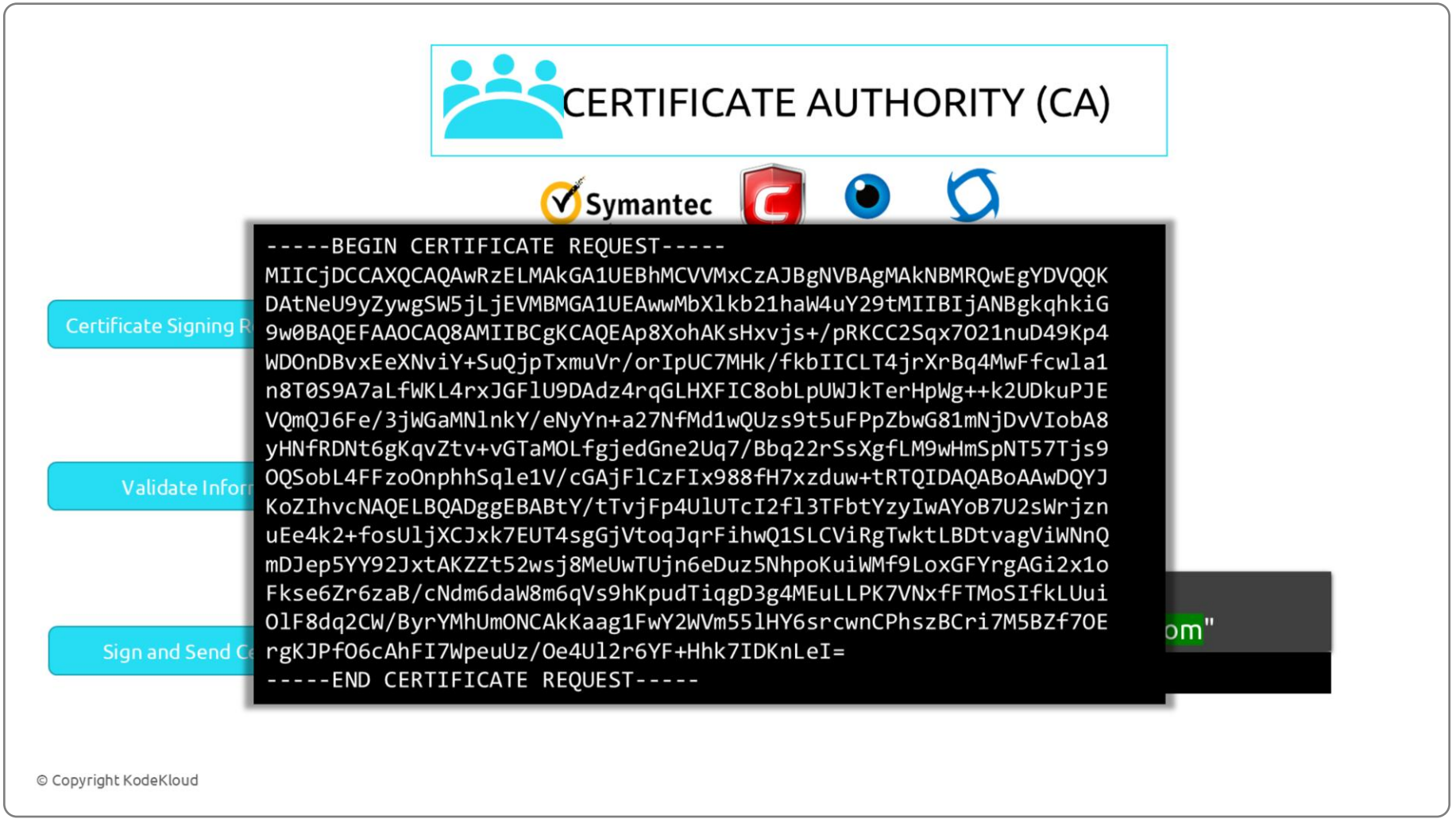
If you looked at the Certificate you received from the hacker closely, you would have noticed that it was a fake certificate that was signed by the hacker himself.



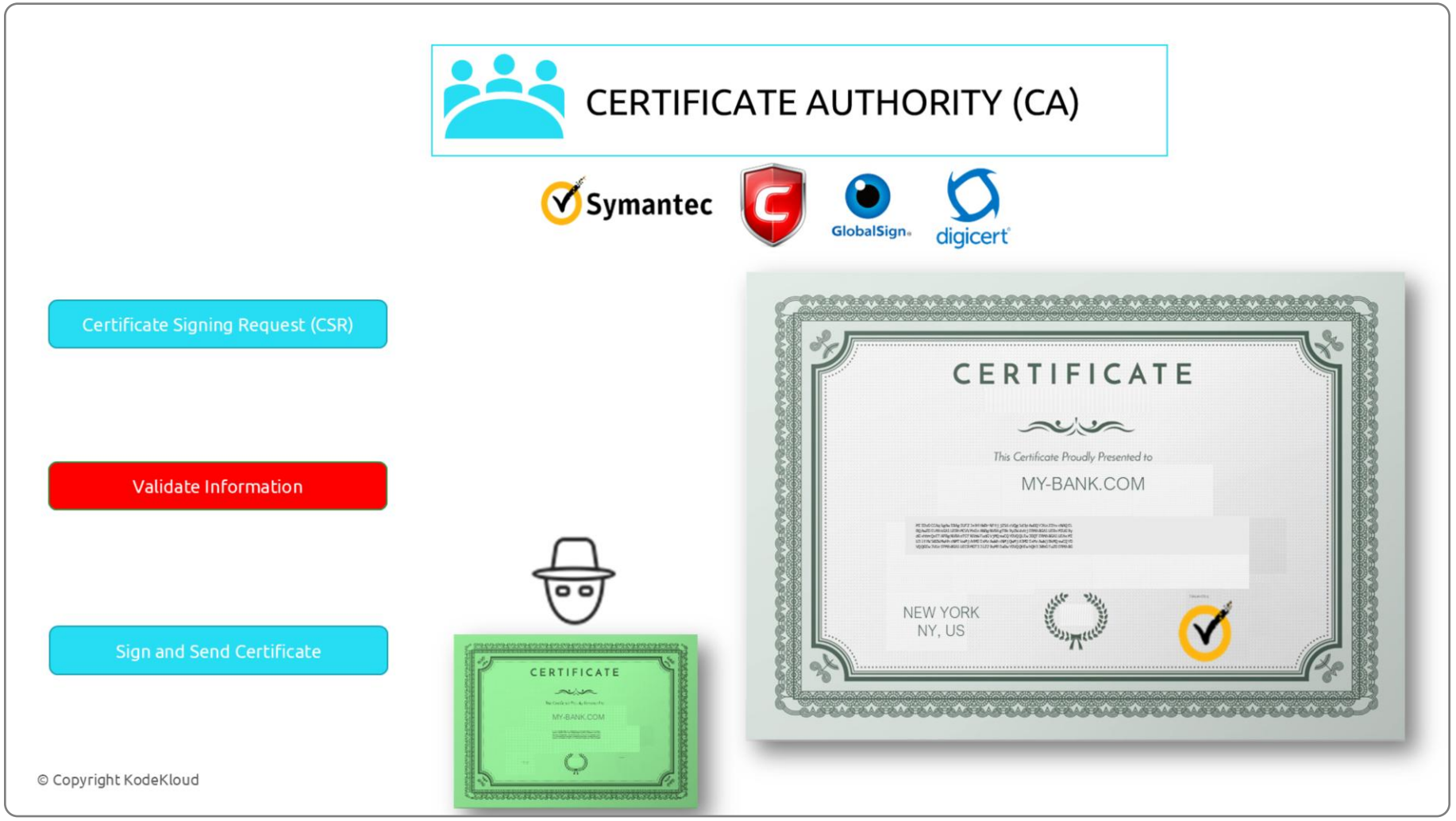
As a matter of fact, your browser does that for you. All of the web browsers are built in with a Certificate validation mechanism, wherein the browser checks the certificate received from the server and validates it to make sure it is legitimate. If it identifies it to be a fake certificate, then it actually warns you.



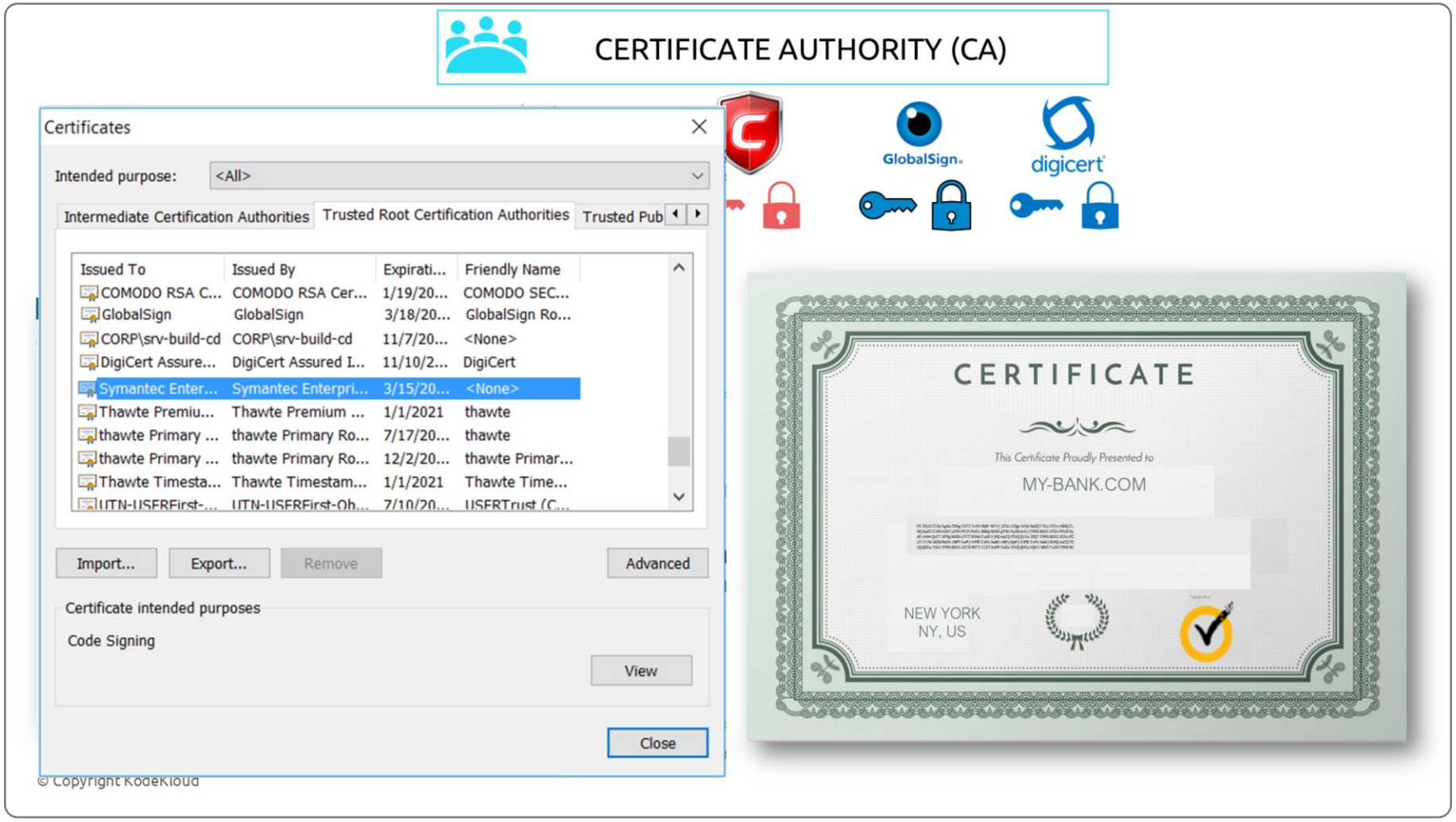
So then, how do you create a legitimate certificate for your web servers that the web browsers will trust? How do you get your certificates signed by someone with authority. That's where Certificate Authorities or CAs comes in. They are well known organizations that can sign and validate your certificates for you. Some of the popular ones are Symantec, Digicert, Comodo, GlobalSign etc.



The way this works is, you generate a Certificate Signing Request or CSR, using the key you generated earlier and the domain name for your website. You can do this, again, using the openssl command. This generates a my-bank.csr file which is the certificate signing request that should be sent to the CA for signing. It looks like this. The certificate authorities verify your details and once it checks out, signs the certificate and sends it back to you.



You now have a certificate signed by a CA that the browsers trust. If hacker tried to get his certificate signed the same way, he would fail during the validation phase and his certificate would be rejected by the CA. So the website that he is hosting won't have a valid certificate. The CAs use different techniques to make sure that you are the actual owner of that domain.

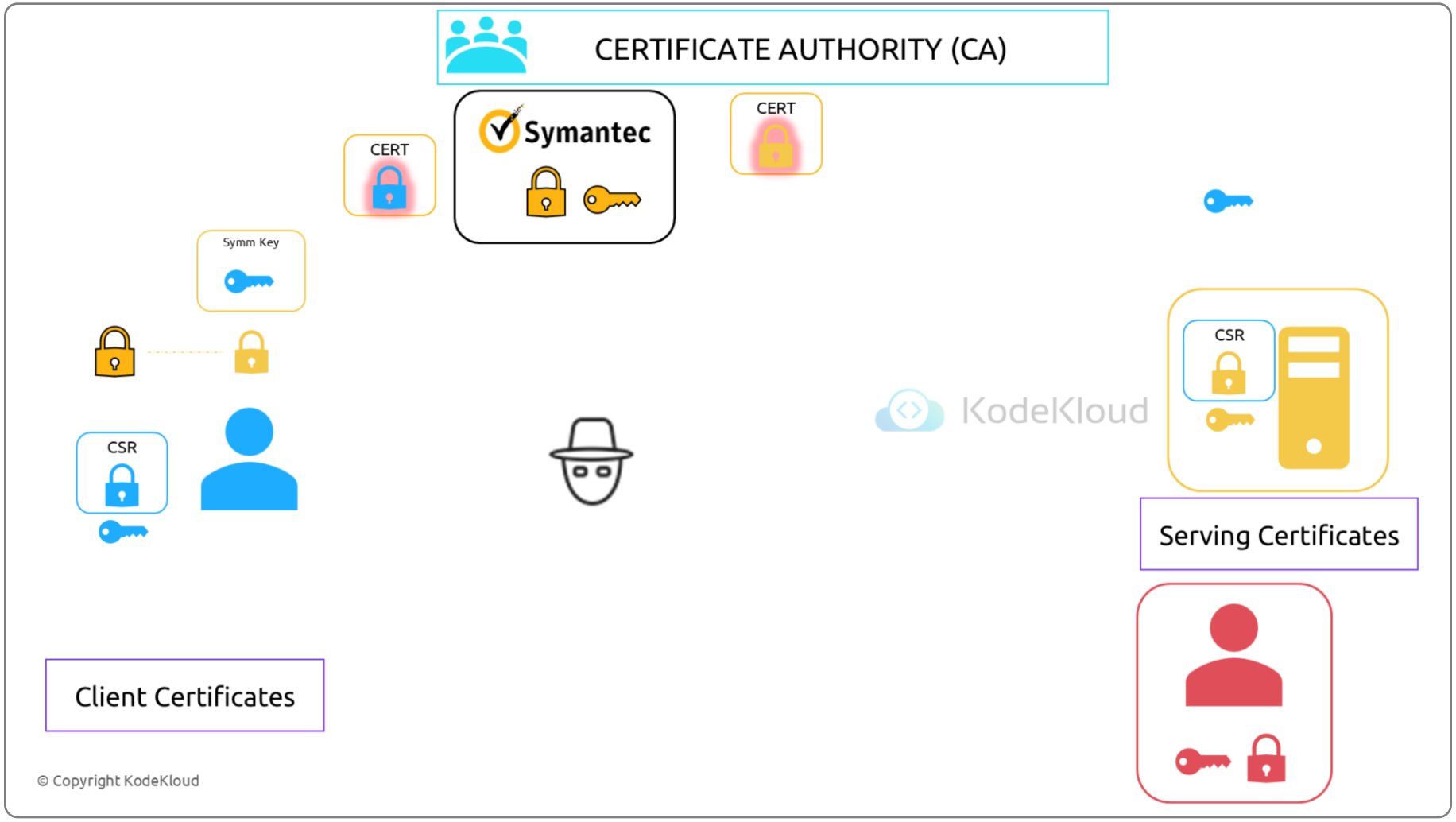


You now have a certificated signed by a CA that the browsers trust. But how does the browsers know that the CA itself was legitimate? For example, what if the certificate was signed by a fake CA? In this case our certificate was signed by Symantec? How would the browser know Symantec is a valid CA and that the certificate was in fact signed by Symantec and not someone who says they are Symantec?

The CAs themselves have a set of public and private key pairs. The CA uses their private key pairs to sign the certificates.

The public keys of all the CA's are built-in to the browsers. The browser uses the public key of the CA to validate that the certificate was actually signed by the CA themselves. You can actually see them in the settings of your web browser, under certificates. They are under trusted CAs.

Now these are public CAs that help us ensure the public websites we visit, like our banks, email etc are legitimate. However, they don't help you validate sites hosted privately, say within your organization. For example, for accessing your payroll or internal email applications. For that you can host your own private CAs. Most of these companies listed here have a private offering of their services. A CA server that you can deploy internally within your company. You can then have the public key of your internal CA server installed on all your employees browsers and establish secure connectivity within your organization.



So let's summarize real quick. We have seen why you may want to encrypt messages being sent over a network. To encrypt messages, we use Asymmetric Encryption with a pair of public and private keys. An admin uses a pair of keys to secure SSH connectivity to the servers.

The server uses a pair of keys to secure HTTPS traffic. But for this, the server first sends a Certificate Signing Request to a CA. The CA uses its private key to sign the CSR. Remember all users have a copy of the CA's public key. The signed

certificate is then sent back to the server. The server configures the web application with the Signed certificate.

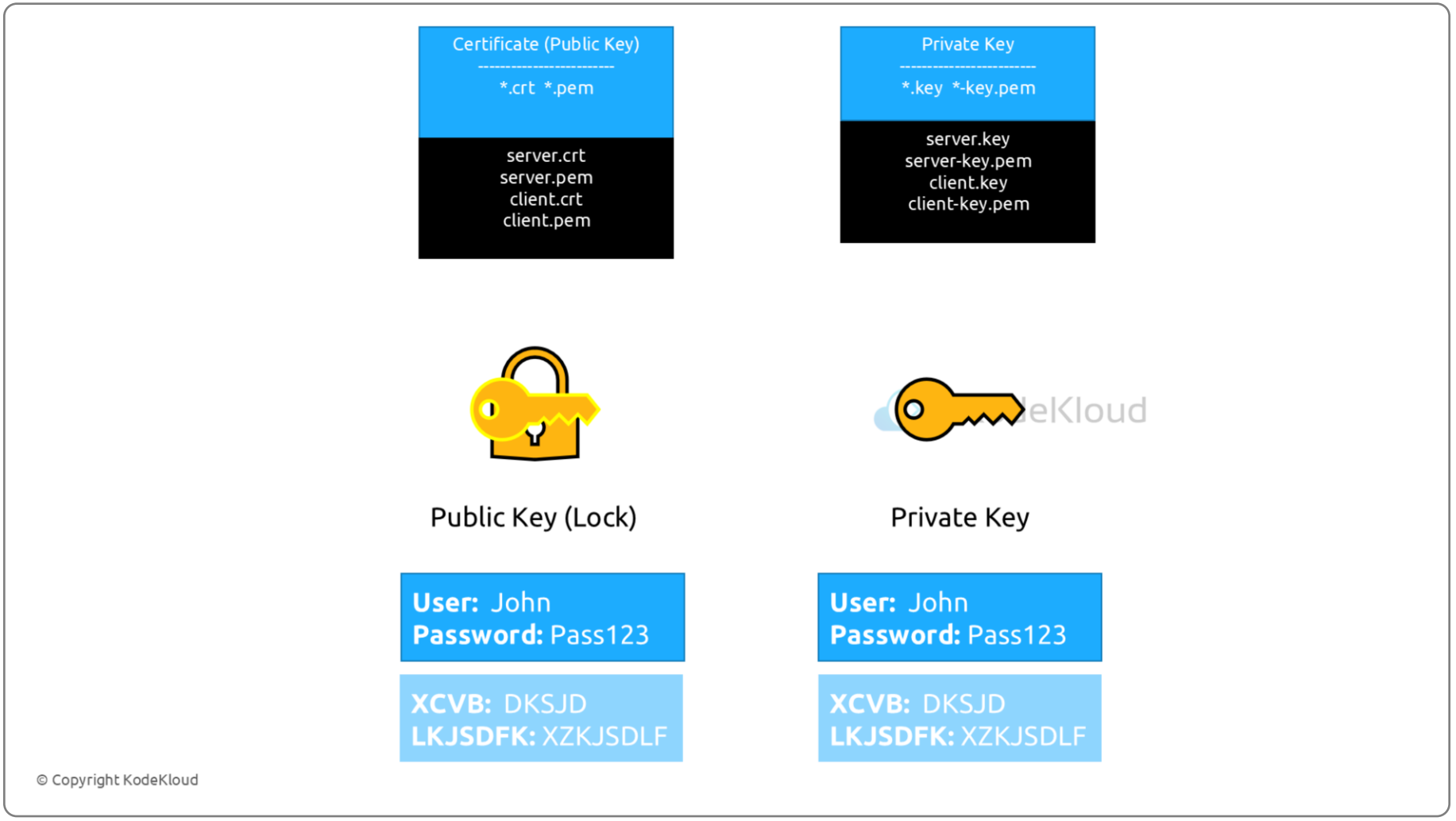
Whenever a user accesses the web application, the server first sends the certificate with its public key. The user (or rather the users browser) reads the certificate and uses the CA's public key to validate and retrieve the servers public key. It then generates a symmetric key that it wishes to use going forward for all communication. The symmetric key is encrypted using the servers public key and sent back to the server. The server uses its private key to decrypt the message, and retrieve the symmetric key. The symmetric key is used for all communication going forward.

So the administrator generates a key pair for securing SSH, the web server generates a key pair for securing the web site with HTTPS, the Certificate Authority generates its own set of key pair to sign certificates. The end user though only generates a single symmetric key. Once he establishes trust with the website, he uses his username and password to authenticate to the web server.

With the servers key pairs the client was able to validate that the server is who they say they are. But the server does not for sure know if the client is who they say they are. It could be a hacker impersonating a user, by somehow gaining access to his credentials, not over the network for sure as we have secured it already with TLS. May be some other means. Anyway, so what can the server do to validate that the client is who they say they are.

For this as part of the initial trust building exercise, the server can request a certificate from the client. And so the client must generate a pair of keys and a signed certificate from a valid CA. The client then sends this certificate to the server for it to verify that the client is who they say they are. Now you must be thinking, you haven't generated a client certificate to access any website. That's because TLS client certificates are not generally implemented on web servers. Even if they are, its all implemented under the hoods, so a normal user don't have to generate and manage the certificates manually. So that was the final piece about Client Certificates.

This whole infrastructure including the CA, the servers, the people and the process of generating, distributing and maintaining digital certificates is known as Public Key Infrastructure or PKI.



Finally, let me clear up something before you leave. I have been using the analogy of a key and lock for private and public keys. If I gave you the impression that only the Lock or the public key can encrypt data, then please forgive me as its not true. These are in fact two related or paired keys.

You can encrypt data with any one of them and only decrypt with the other.

You cannot encrypt with one and decrypt with the same.

So you must be careful what you encrypt your data with. If you encrypt your data with your private key, then remember anyone with your public key (which could really be everyone out there) will be able to decrypt and read your message.

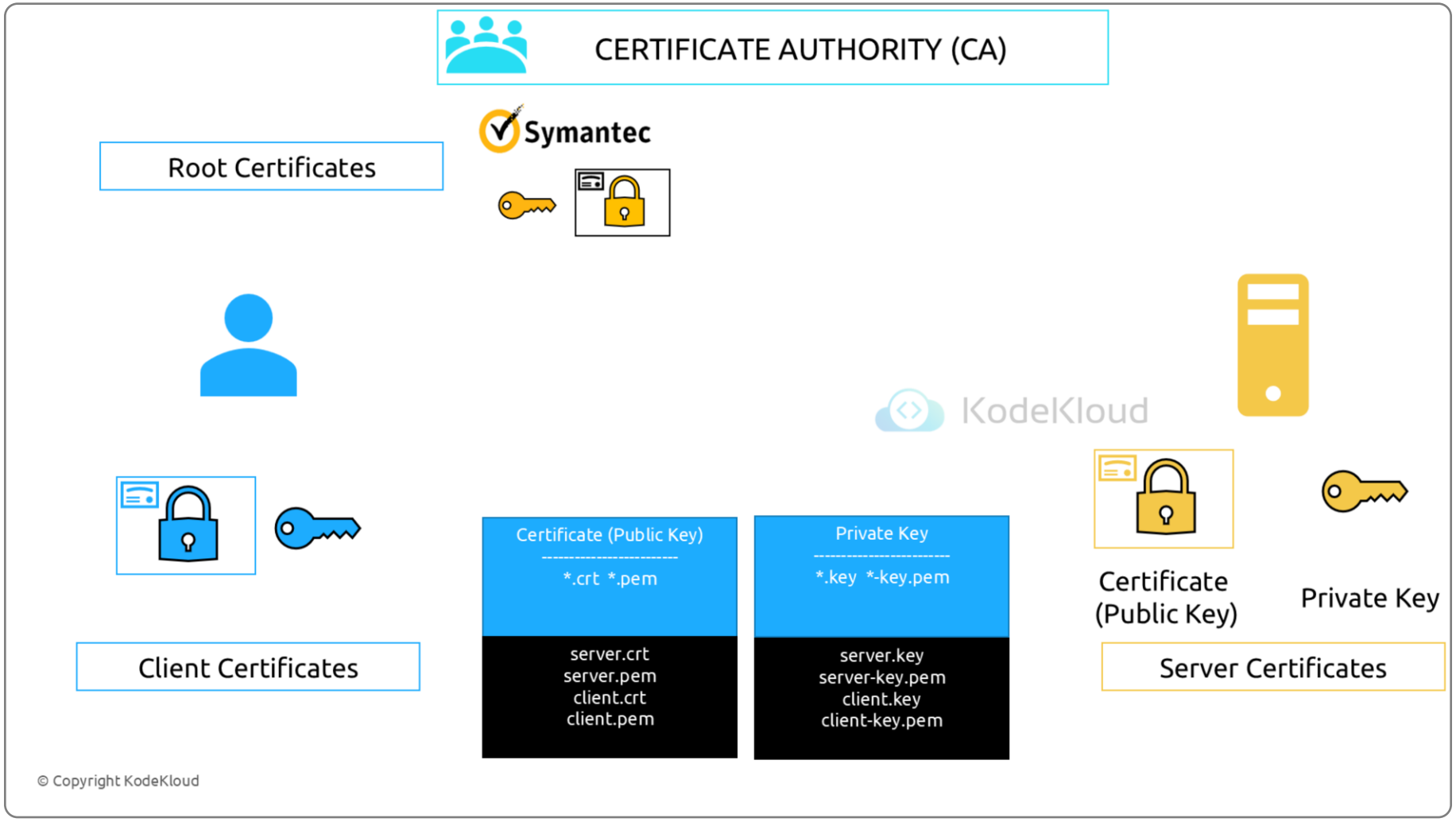
Finally, a quick note on naming convention. Usually certificates with Public key are named crt or pem extension. So that's server.crt, server.pem for server certificates or client.crt or client.pem for client certificates.

And private keys are usually with extension .key, or –key.pem. For example server.key or server-key.pem. So just remember private keys have the workd 'key' in them. Either as an extension or in the name of the certificate. And one that doesn't have the word key in them is usually a public key or certificate.

Well, that's it for this lecture. Thank you for your watching and I will see you in the next lecture.

www.kodekloud.com

Hello and welcome to this lecture. In this lecture we look at securing your Kubernetes cluster with TLS Certificates.

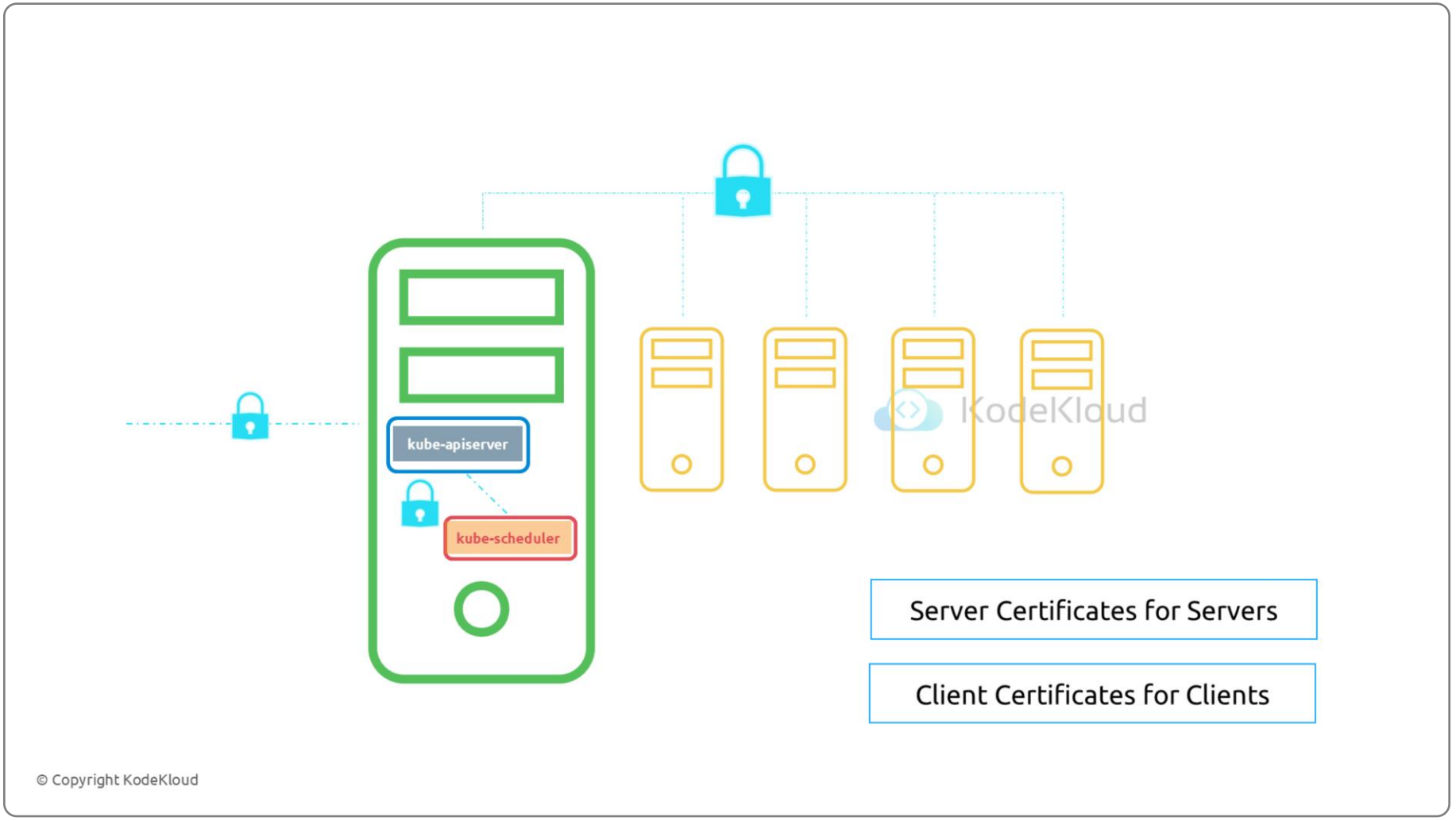


In the previous lecture we saw what a public and private key are. How a server uses public and private keys to secure connectivity. We will call them serving certificates. We saw what a certificate authority is. We learned that the CA has its own set of public and private key pairs that it uses to sign server certificates. We will call them Root Certificates. We also saw how a server can request a client to verify themselves using Client Certificates.

And a quick note on naming convention before we go forward. You are going to see a lot of certificate files in this lecture

and it could be confusing. So use this technique to know which one is which. Usually certificates with Public key are named crt or pem extension. So that's server.crt, server.pem for server certificates or client.crt or client.pem for client certificates. And private keys are usually with extension .key, or with a –key in the filename. For example server.key or server-key.pem. So just remember private keys have the workd 'key' in them. Either as an extension or in the name of the certificate. And one that doesn't have the word key in them is usually a public key or certificate. That's how I remember it.

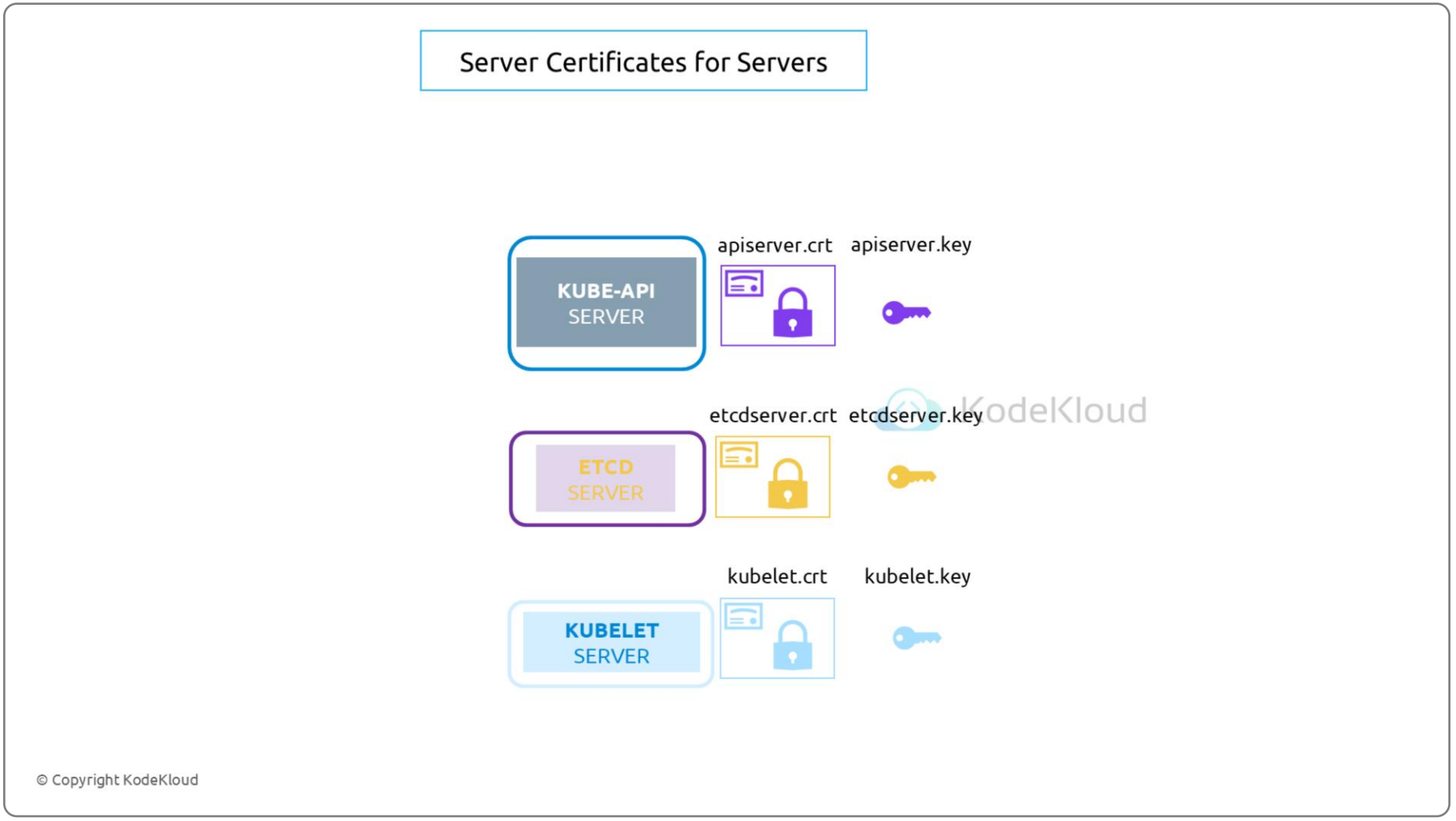
We will now see how these concepts relate to a kubernetes cluster.



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 secured. For example an administrator interacting with the kubernetes cluster through the kubectl utility or via 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.

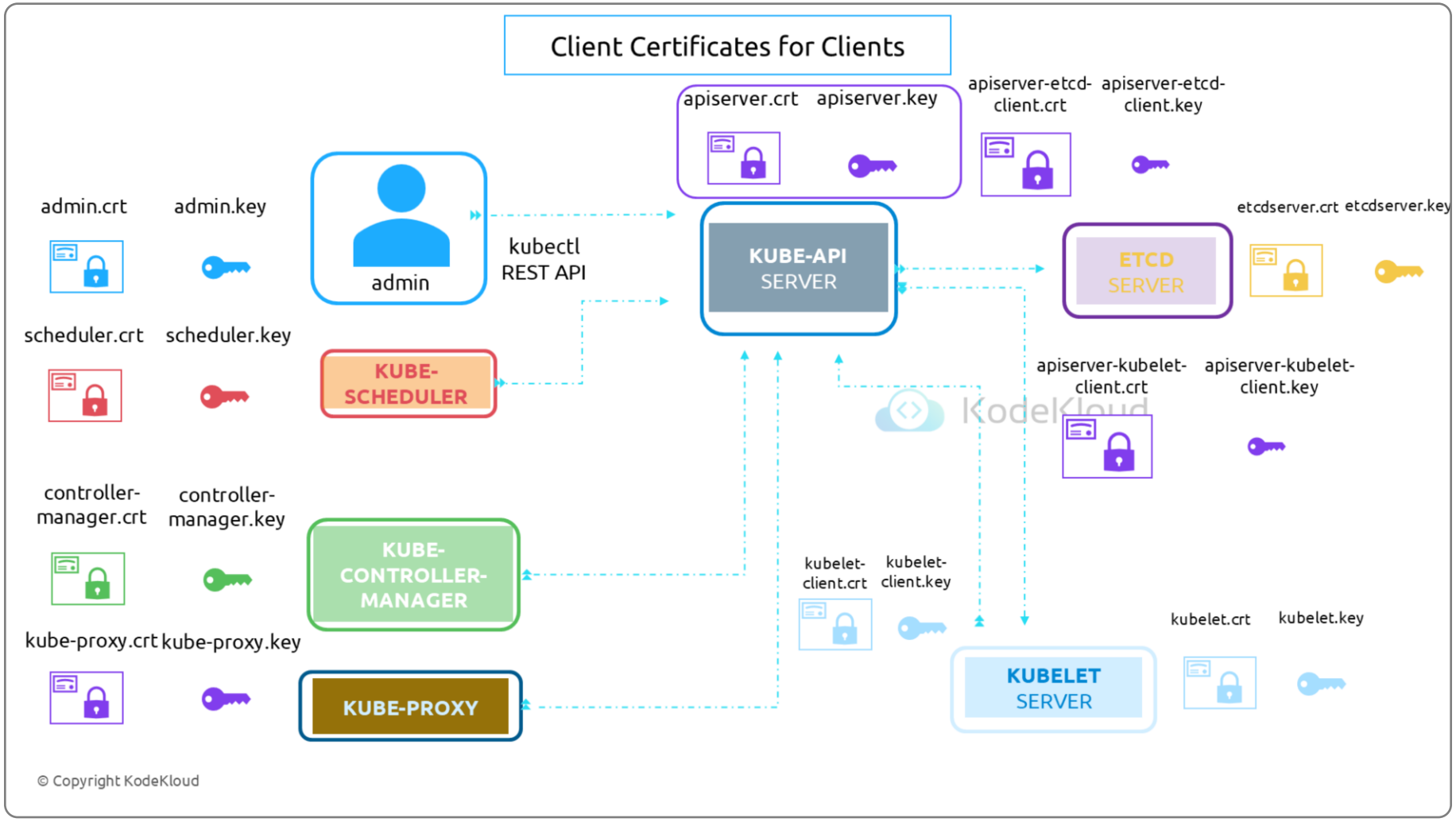


Let's look at the different components within the kubernetes cluster and identify the various servers and clients and who talks to who. Let's start with the kube-apiserver. As we know already, the API server exposes an HTTPS service that other components as well as external users use to manage the kubernetes 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.crt 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. Depends on who and how the cluster was setup. So these names may be different in yours. In this lecture, we will try to use names that help us easily identify the certificate files.

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

The other server component in the cluster is on the worker nodes. They are the kubelet services. They also expose an HTTPs API endpoint that the kube-api server talks to, to interact with the work nodes. Again that requires a certificate and key pair. We will call it kubelet.crt and kubelet.key.



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-api server are us, the administrators. Through kubectl or REST 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. The scheduler talks to the kube-api server 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-api server. As far as the kube-api server is concerned, the scheduler is just another client like the admin user. So the scheduler needs

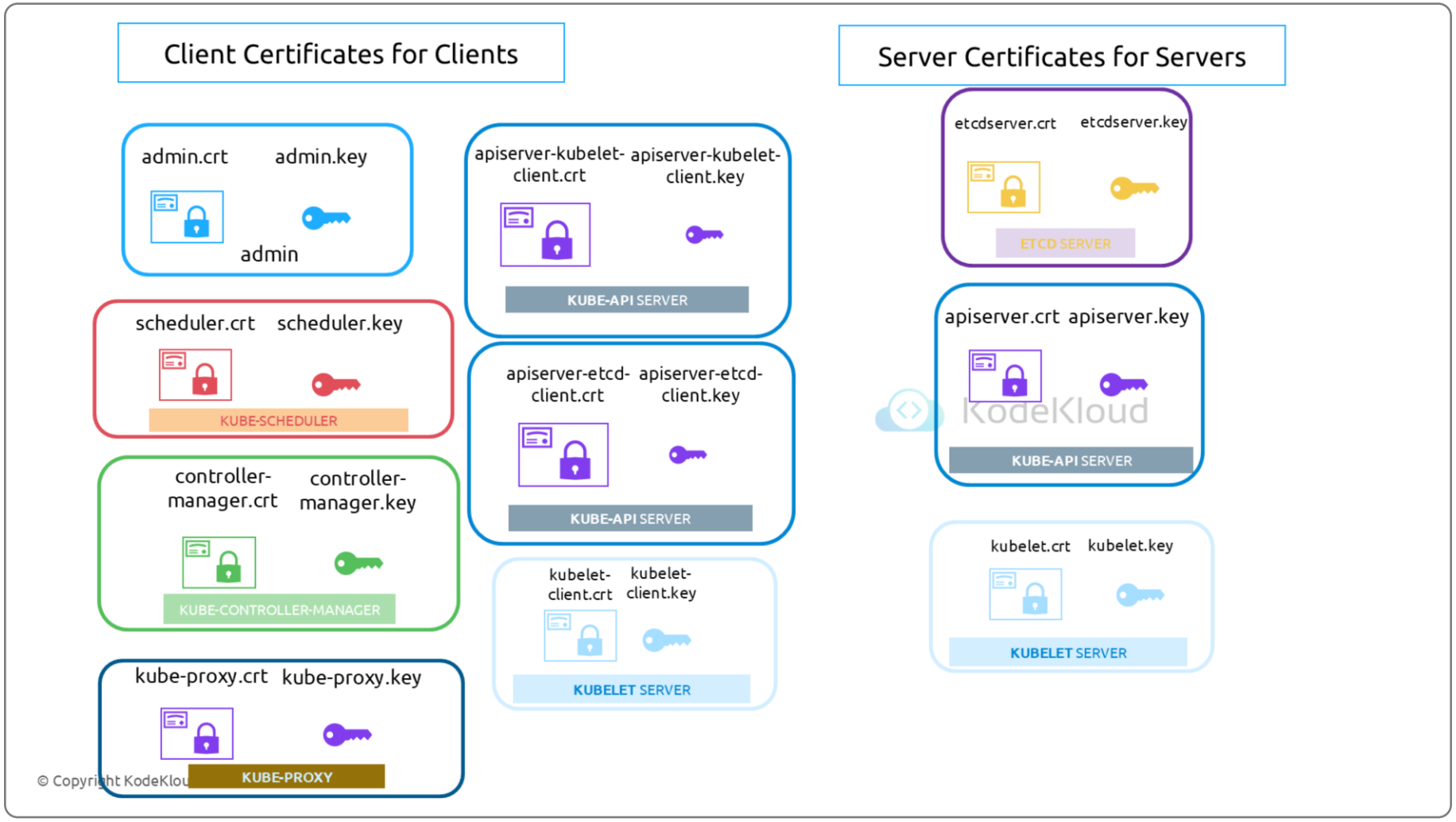
to validate its identity using a client TLS certificate. So it needs its own pair of certificate and keys. We will call it scheduler.crt and scheduler.key.

The kube-controller manager is another client that accesses the kube-api server. So it also requires a certificate for authentication to the kube-api server. So we create a certificate pair for it.

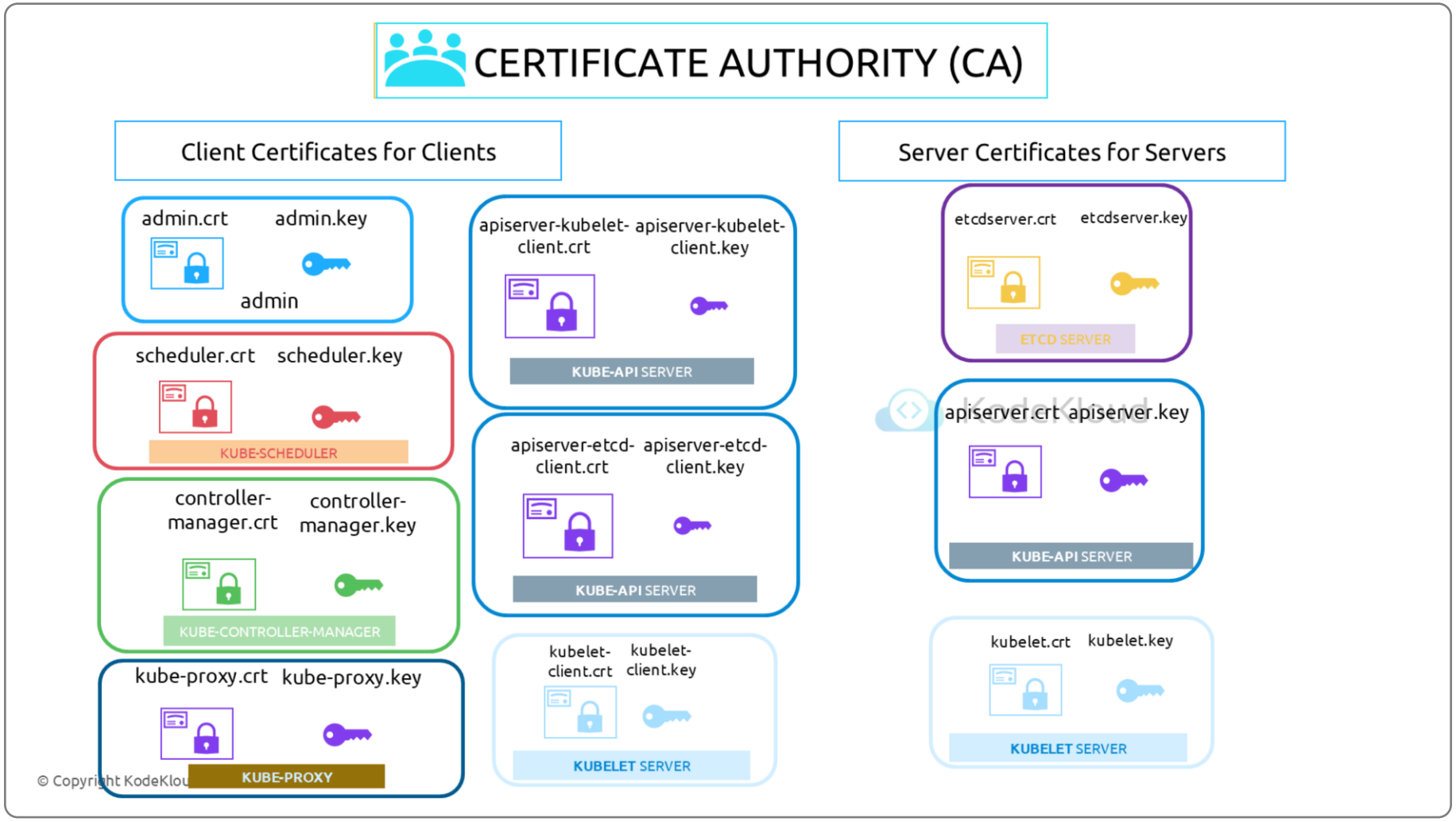
The last client component is the kube-proxy. The kube-proxy requires a client certificate to authenticate to the kube-api server. And so it requires its own pair of certificate and keys. We will call them kube-proxy.crt and kube-proxy.key.

The servers communicate amongst them as well. For example, the kube-api server communicates with the ETCD server. In fact of all the components the kube-api server is the only server that talks to the ETCD server. So as far as the ETCD server is concerned the kube-api server is a client. So it needs to authenticate. The kubeapi server 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-api server to authenticate to the etcd server.

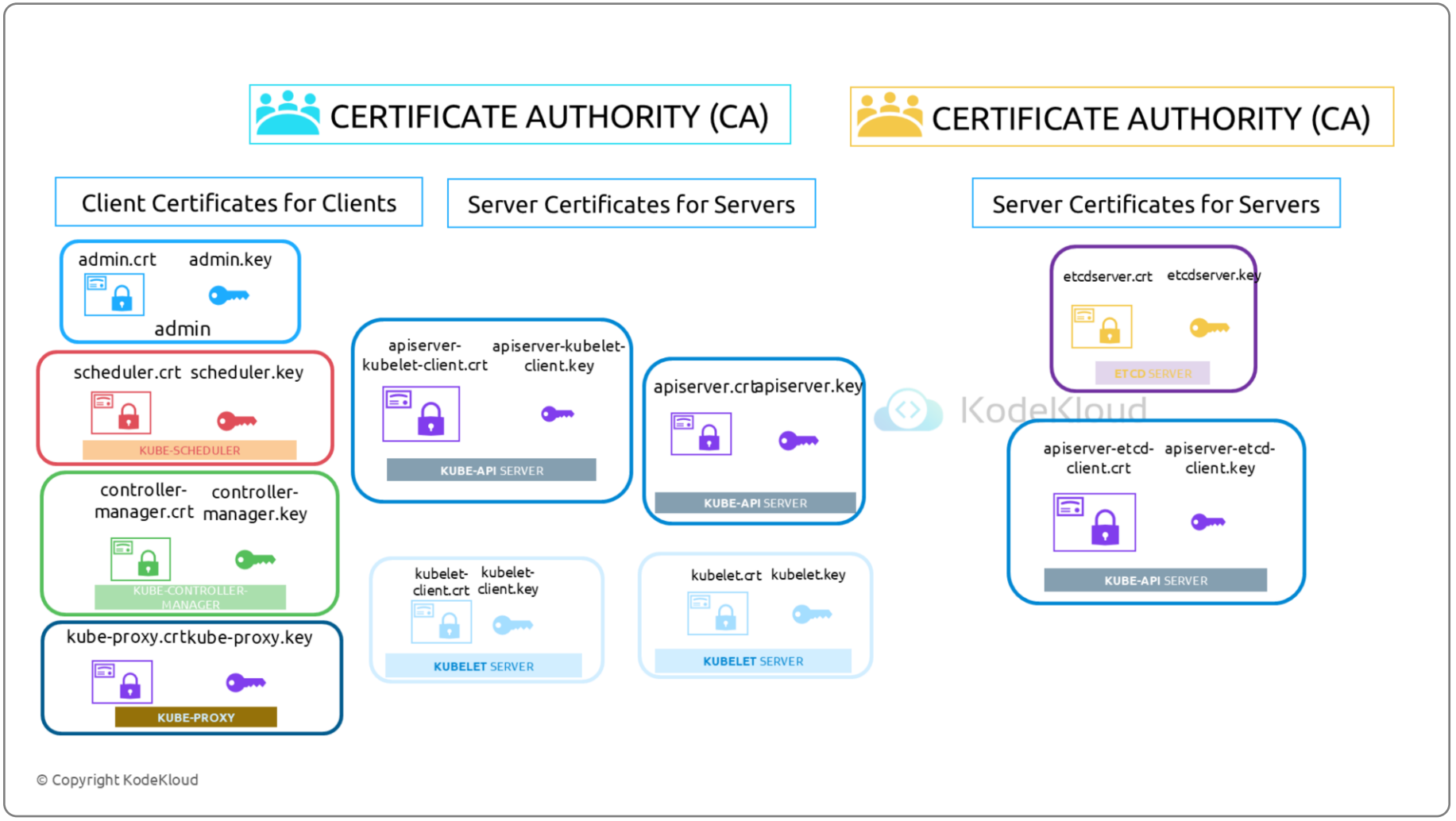
The kube-api server also talks to the kubelet server on each of the individual nodes. That's how it monitors the worker nodes and instructs them to load pods etc. For this, again, it can use the original certificates or generate new ones specifically for this purpose.



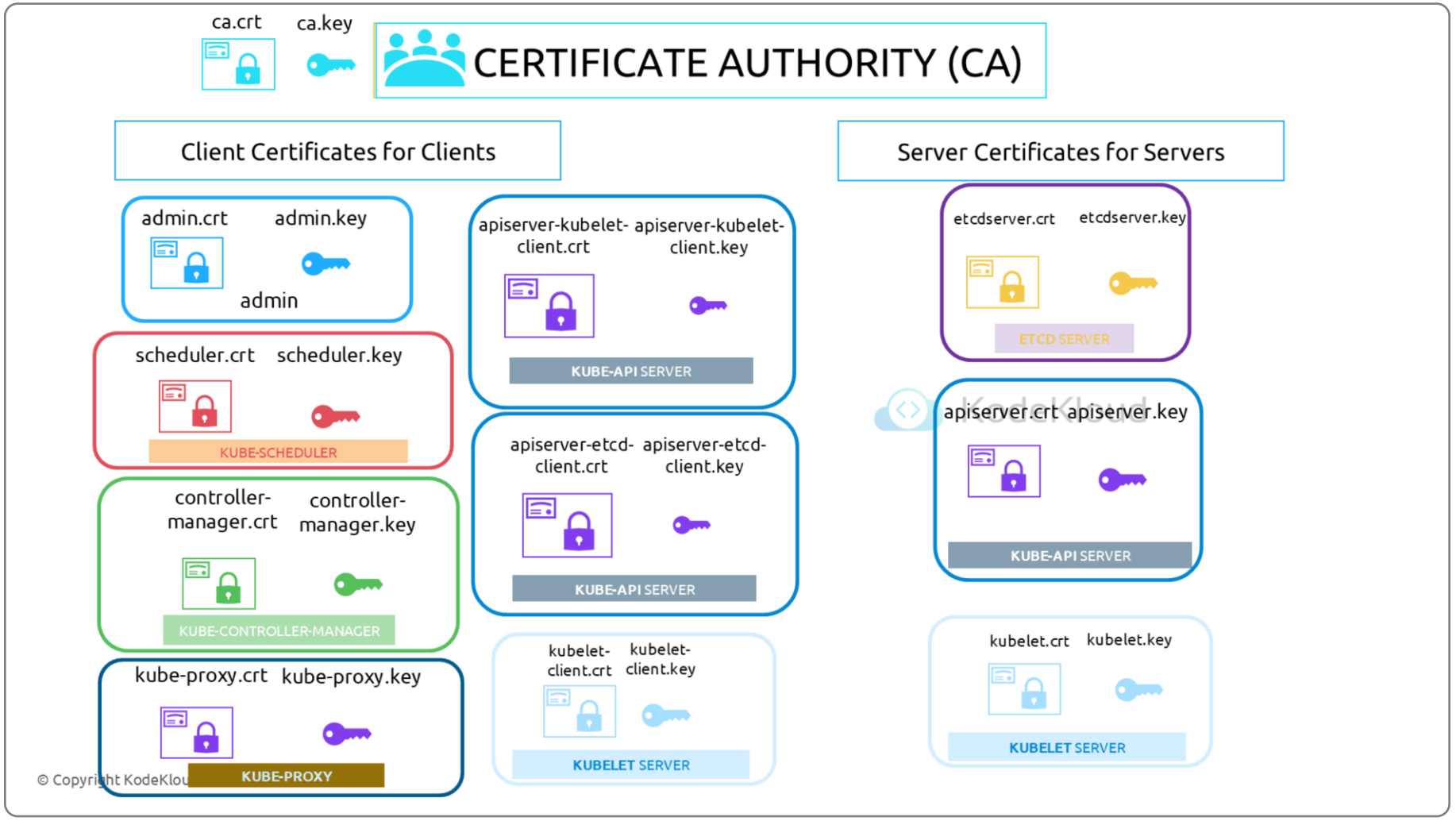
So that's too many certificates. Let's try and group them. There are a set of client certificates mostly used by clients to connect to the kube-api server. And there are a set of server side certificates used by the kube-api server, etcd-server and kubelet to authenticate their clients.



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 one certificate authority for your cluster.



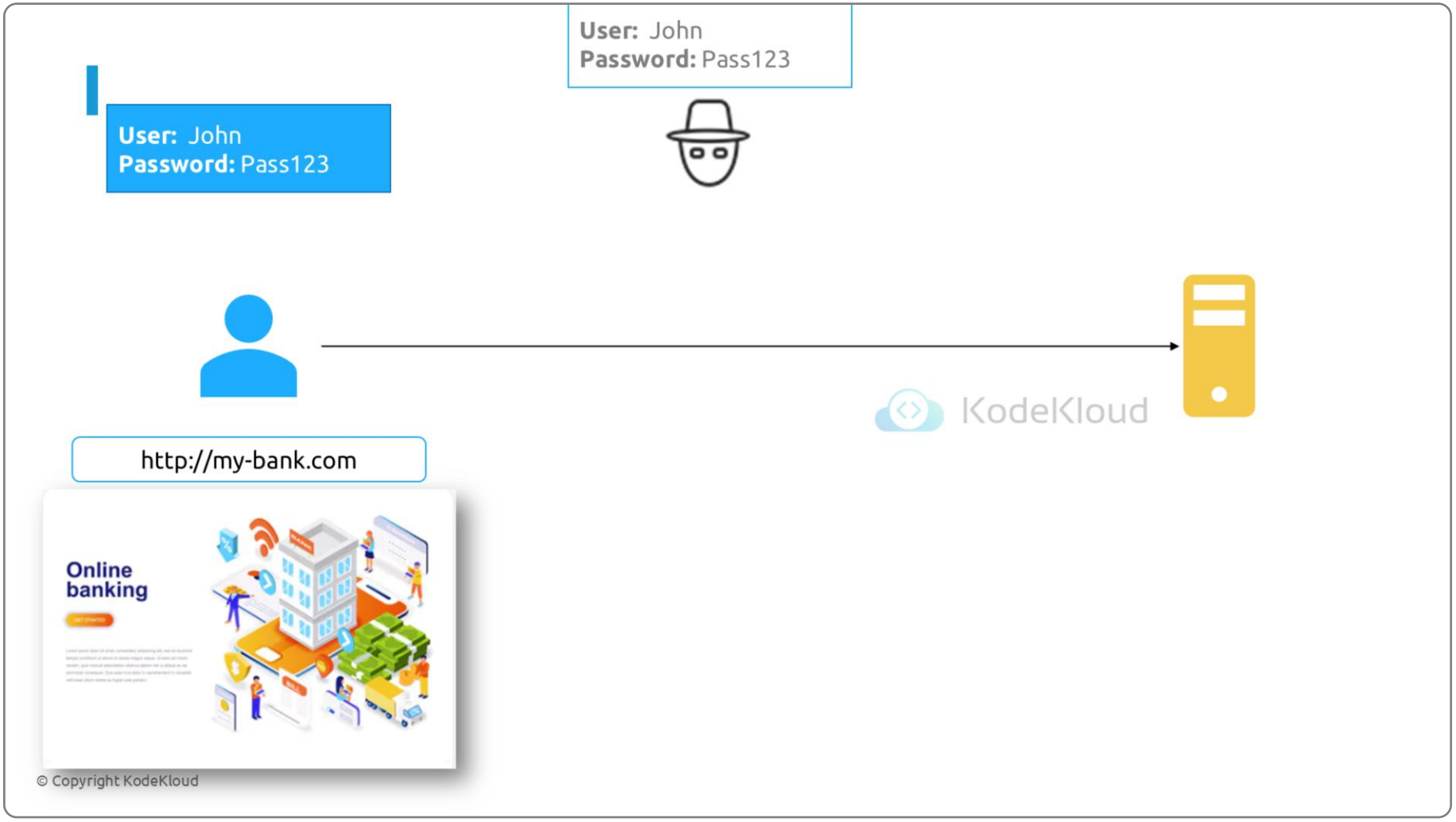
In fact you can have more than one. One for all the components in the cluster and another one specifically for ETCD. In that case the ETCD servers certificates and the ETCD servers client certificates, which in this case is the api-server client certificate will be all signed by the ETCD server CA.



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. Let us now see how to generate them.

One way SSL vs Mutual SSL

In this lecture, we will understand the basics of mTLS or mutual TLS and see how we can use it to encrypt the traffic between two systems.



In one of the earlier lectures, we got an introduction to TLS. We learnt how to create SSL certificates, and how to use them to secure SSH and web servers.

<c>We then used the example of a customer accessing their online banking to demonstrate how, without encryption, an attacker can easily retrieve the users credentials by sniffing the network and hack in to the users bank account.

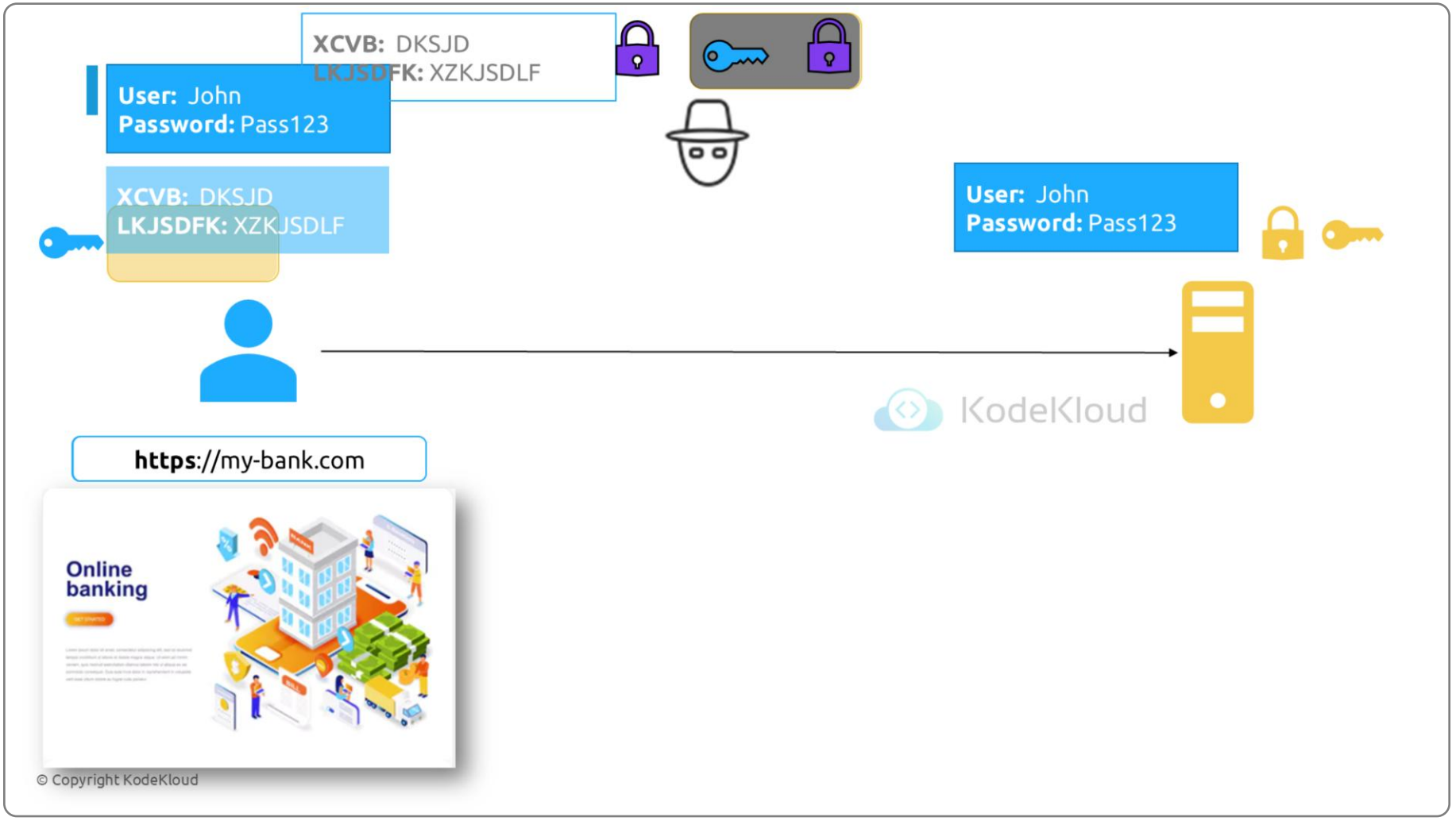
We then learnt how to make use of TLS certificates to encrypt the data between the user and the bank, enabling a secure connection via HTTPS.<c>

When the user first establishes connection with the bank, he receives its public certificate.

The Web browser on the client side will verify if the certificate from the bank is legitimate.<c>

To do this the browser checks if the certificate authority used to sign the certificate is valid.
<c> The public keys of all the CA’s are built-in to the browsers trust store.

The browser uses the public key of the CA to validate that the certificate was actually signed by the CA themselves.

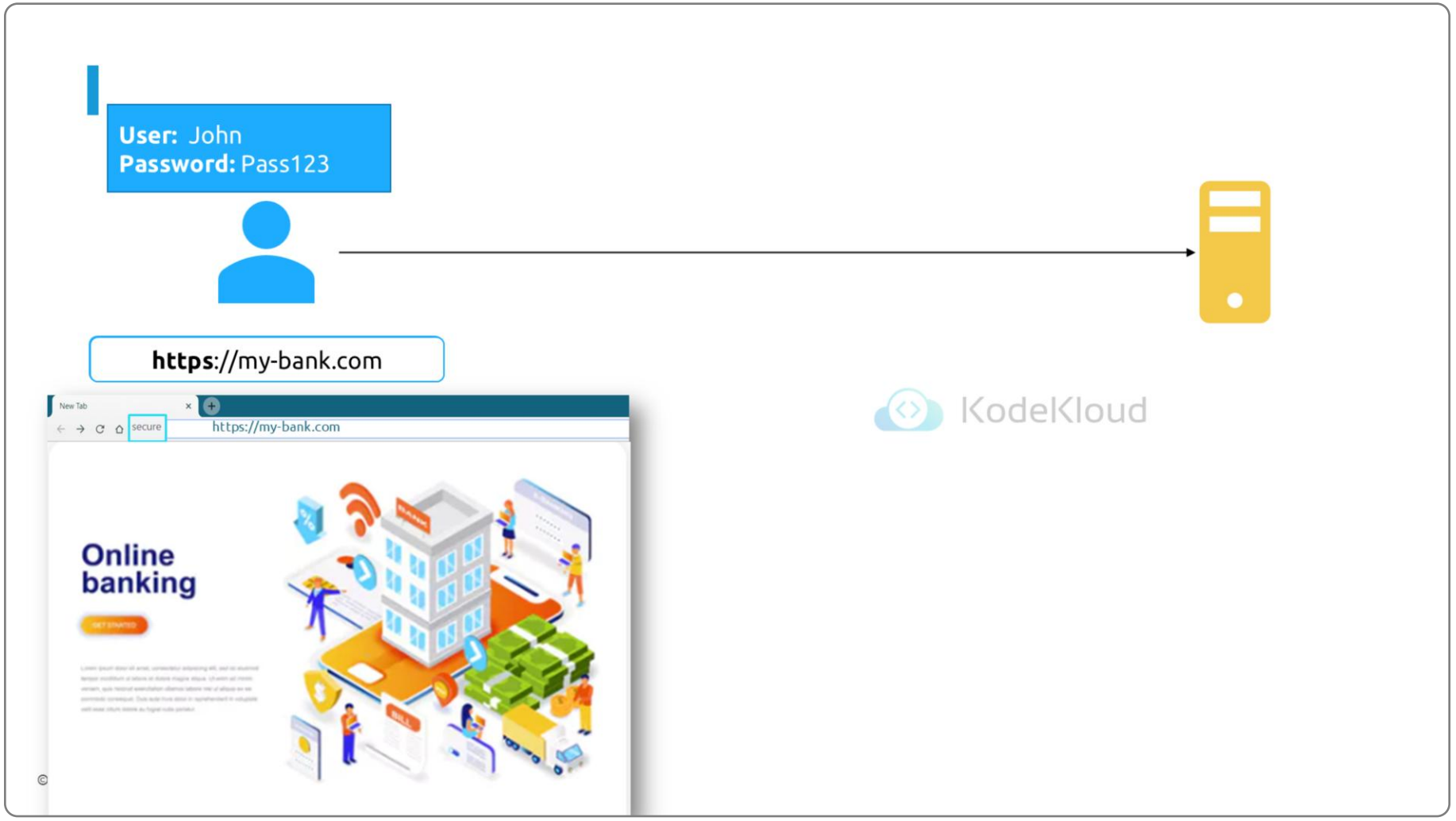


Once it is validated, the client which is the web browser uses the public certificate from the bank to encrypt the symmetric key so that it can be sent over to the bank securely.

<c>The bank then uses its private key to decrypt this symmetric key. This is called as asymmetric encryption.

Once this is done, the client uses the symmetric key to encrypt all its data and transfer it to the bank. The bank then uses the same key to decrypt the data.

By using asymmetric encryption, we were able to securely transfer our symmetric key from the client machine to the bank. And then using symmetric encryption, we were then able to securely exchange information with the bank. A hacker snooping on the network cannot get hold of the data sent to the bank as he cannot decrypt the symmetric key



Here the client verifies the server i.e the bank's certificates. The Bank however, does not verify the clients certificates. <c>The only way it knows who you are is based on the actual data you send to it i.e the username, password/client number e.t.c.

This is an example of a one way SSL and is most commonly used in scenarios where a user wants to connect to web service

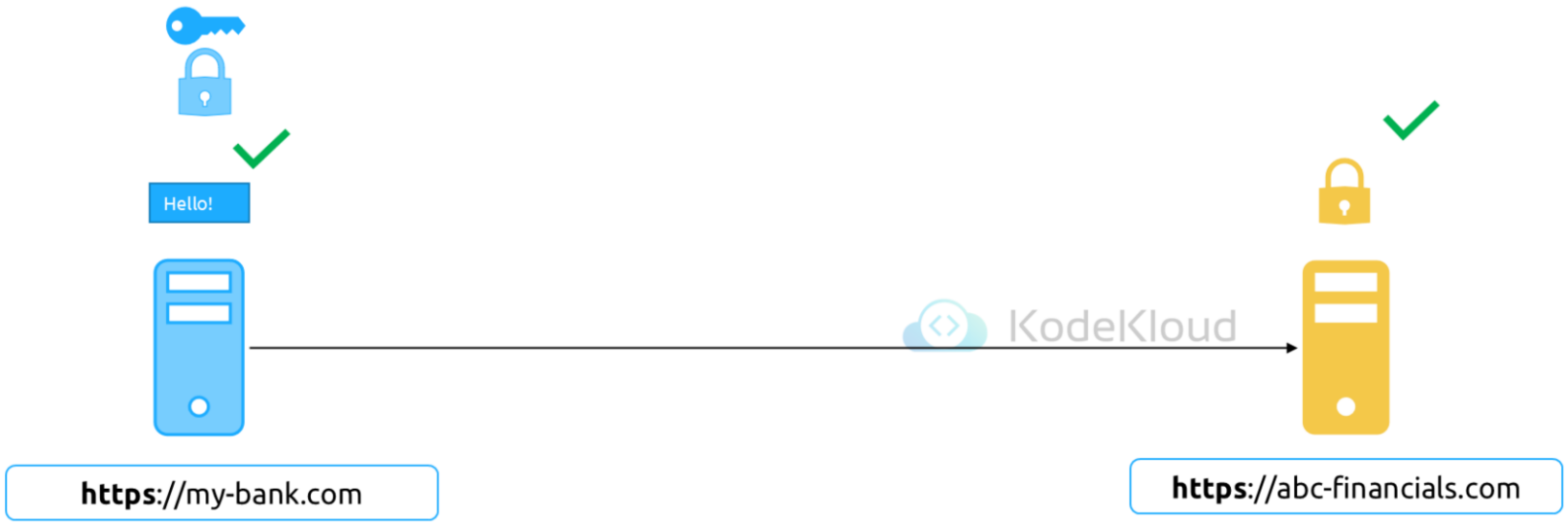
over the internet – for example, accessing your email account, social media sites, financial institutions e.t.c. Just like in the example of my-bank.com, all these web services establish the authenticity of the user based on some form confidential information – such as username and passwords.

However, consider a scenario where there is no end user to key in this information.

Let us assume that two organizations want to securely exchange confidential information between themselves. <c>

Now, say my-bank.com as the client wants to get some data to the server abc-financials.<c> How can the server validate that the data is actually being sent by my-bank.com?

This is where two-way TLS or mutual TLS is used. With mutual TLS or mTLS, the client and also the server will now verify the authenticity of each other.



To make this easier, let us call my-bank as the client and abc-financials as the server.

When requesting data from the server,<c> the Client first requests the servers public certificate.
The server replies back with its public certificate<c> and then requests the certificate of the client.
Meanwhile, the Client checks with the <c>CA if the certificate belongs to the Server.

Once this is verified, the client sends its certificate<c> to server and also shares a symmetric key which is encrypted with the now public key of the server.

The server now validates<c> with the CA if the public certificate that was sent by the client indeed belongs to my-bank.com

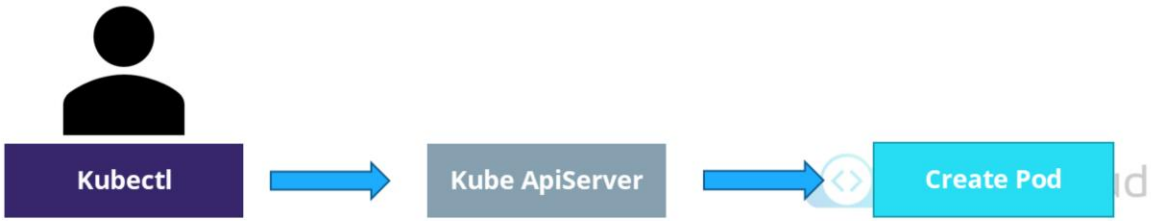
Now that both the server and the client have mutually authenticated each other all further communication from the client to server can be encrypted using the symmetric key.

Well, that’s it for this lecture. In the upcoming lecture, we will learn how to make use of mTLS to secure pod to pod communication in a kubernetes cluster.

Admission Controllers

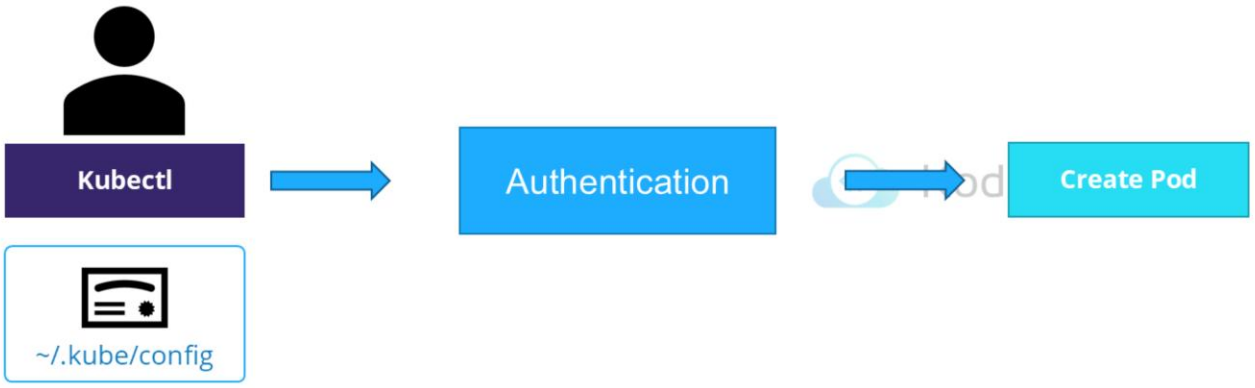
In this lecture we look at how to verify platform binaries before deploying a cluster.

Securing Kubernetes



So we've been running commands from our command line using the kubectl utility to perform various kinds of operations on our Kubernetes cluster. And we know everytime we send a request say to create a pod, the request goes to the API server and then the pod is created and the information is finally persisted in the ETCD database.

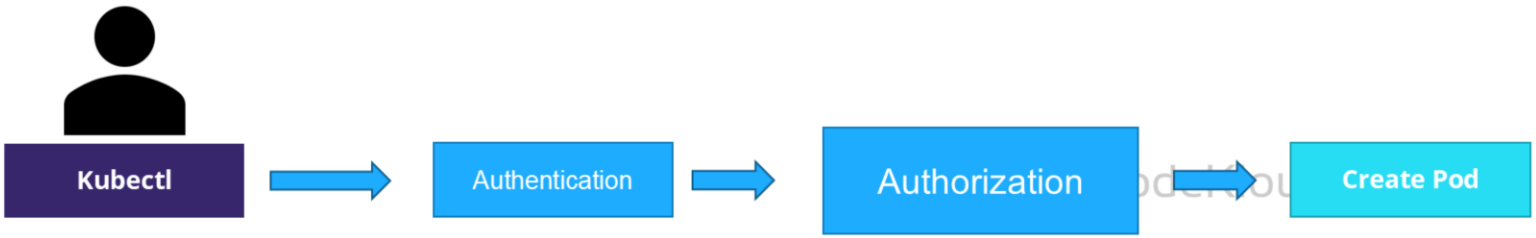
Authentication




```
❯ cat .kube/config
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM1ekNDQWMrZ
EFWTVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRFRJeE1ETXdOVEExTkRFeE1Gb1hEVE14TURNd0
NWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBRENDQVFvQ2dnRUJBTWZOCnFRc2V
6NmpRQVYwNmNDa2gzbUUyNlEwRWxMT3czWU1aMGwKdVgxblhIcjk0bHd1RitNdXU3SWI3cnFOZkk0dXlMSG50
RgpJdGx6R0lQS01iUlh5MmIrZFFpbWNQYjJvSUgvQXJZR0UvS0VML1BQU1pWVzBGemNGQzAzcEpkRkZJdldlc
3BtZWVwWEJVVHF1M1VXckJRWG92WkpPc3EzM2tLK205NHNFdGoKU1VQVnlrdnF6MEVlVnhGQ0lHYVNZaWpjWU
ZNbHZYZDhWeQprY3pHY0hZb1Qwd3M4V0pPVUxjQ0F3RUFBYU5DTUVBd0RnWURWUjBQQVFIL0JBUURBZ0trTUE
PQkJZRUZJQUwzNHpGYmdETDQrZFpENUtHT0ROSFgzZ3ZNQTBHQ1NxR1NJYjMKRFFFQkN3VUFBNElCQVFCWWNB
U2VPSkFrQnZrMEZUTjNyMQp4VkZLTklnOWZlZHBFU1ZYUWp2OXU1cUhRRUhUV1gxclRGdjBNbUNVNnlFTlhpZ
HVHeGd4dUNZV3grYmZ5UGtzZGdMcnYxb1B0SFZ5Ym9wQzBnVXNkSUZ0TGpOYWNyRkR1RGsKNEYrZmNnVXpUVT
JVUElDYllljYnFieG14SHJ2MkJRYzA3Mwp4aUd1MWo3SklMQ1daejV0alQ0VG5oYW9nd25QOHRIWnBIY1VDYXB
0djI2YUFqUWJFeGtRZy9aUkViNXM2MnNMWjI1YWg1YwotLS0tLUVORCBDRVJUSUZJQ0FURS0tLS0tCg==
    server: https://10.10.10.15:6443
```

When the requests hits the API server, we've learned that it goes through an authentication process. And this is usually done through certificates. If the request was sent through kubectl we know the kubeconfig file has these certificates configured. And the authentication process is responsible for identifying the user who sent the request and making sure the user is valid.

Authorization



developer-role.yaml

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


And then the request goes through an authorization process. And this is when we check if the user has permission to perform that operation. And we have learned that this is achieved through Role based access controls. So in this case if the user was assigned this particular role of a developer, the user is allowed to list, get create, update or delete pods. And so if the request that came in matched any of these conditions, in this case it does as the request is to create a pod, it is allowed to go through, other wise its rejected. So that's authorization with Role based access control.

Authorization - RBAC

- ✓ Can list PODs/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

developer-role.yaml

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


With role based access control you could place in different kinds of restrictions such as to allow or deny those with a particular role to say create, list or delete different kinds of objects like pods, deployments or services. We could even restrict access to specific resource names such as say a developer can only work on pods named blue or orange. Or restrict access within a namespace alone. As you can see most of these rules that you can create with role based access control is at the Kubernetes API level. What user is allowed access to what kind of API operations. And it does not go beyond that.

But what if you want to do more that just define what kind of access a user has to an object?

Authorization - RBAC

- ✓ Can list PODs/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

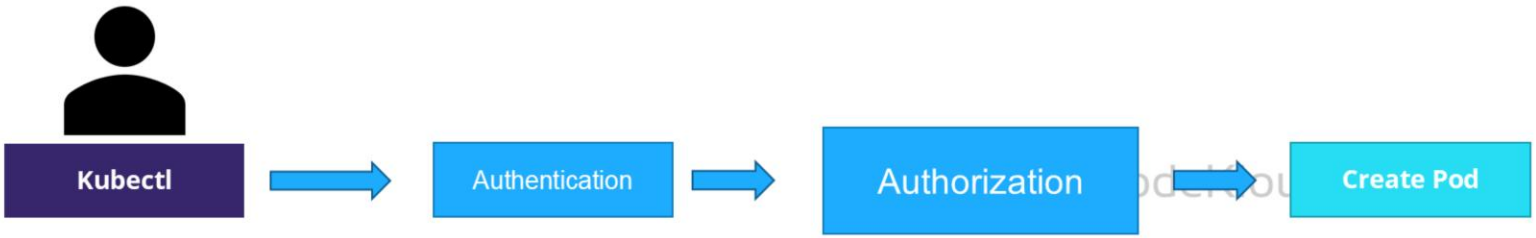
- ❖ Only permit images from certain registry
- ❖ Do not permit runAs root user
- ❖ Only permit certain capabilities
- ❖ Pod always has labels

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"]
```

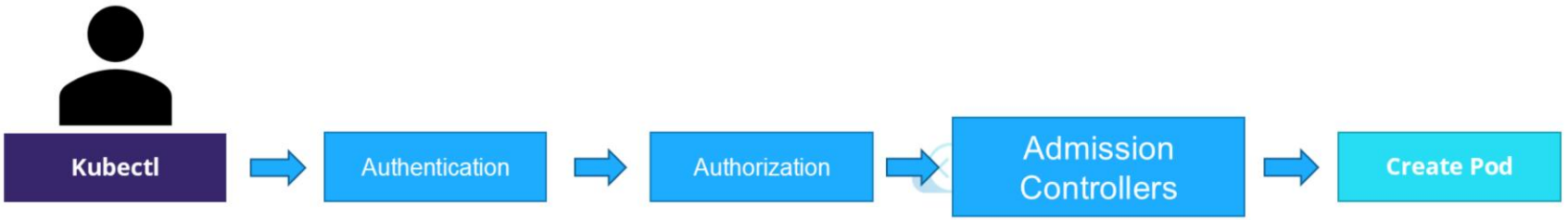

For example when a pod creation request comes in you'd like to review the configuration file and look at the image name and say do not allow images from public docker hub registry. Only allow images from a specific internal registry only. Or to enforce that we must never use the latest tag for any images. Or say for example you'd like to say if the container is running as the root user, do not allow that request. Or allow certain capabilities only. Or enforce that metadata always contains labels.

Authorization



So these can’t be achieved with Role based access controls.

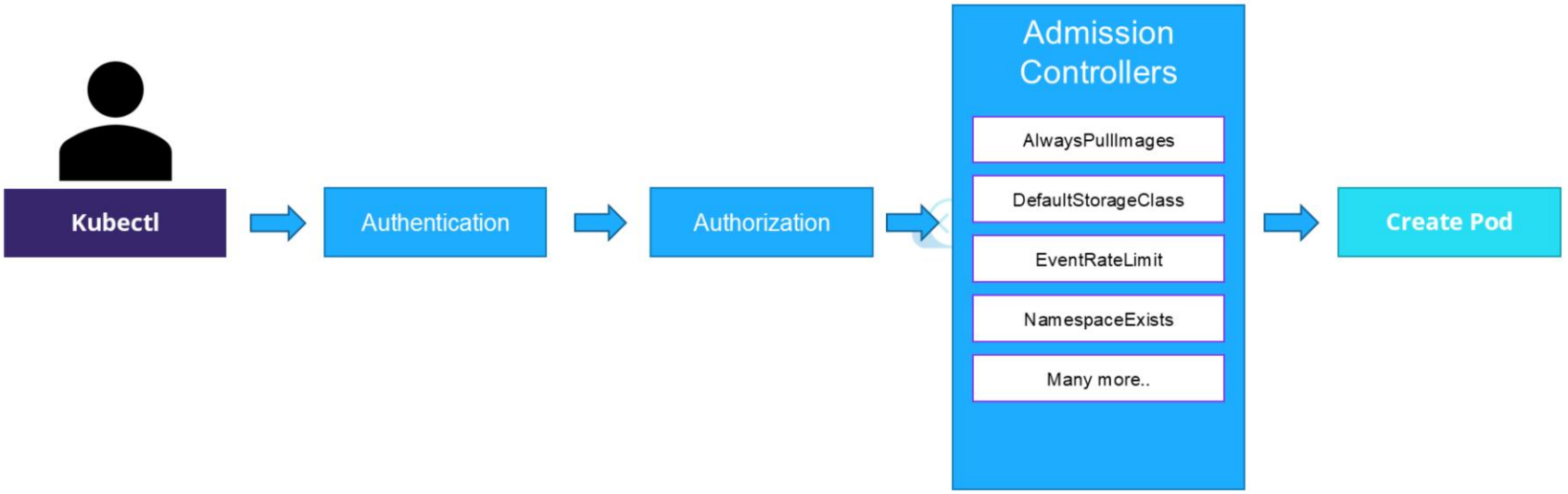
Admission Controllers



And that's where Admission Controllers comes into play. 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 such as change the request itself or perform additional operations before the pod gets created. We will go over some examples in the upcoming slides.

Admission Controllers

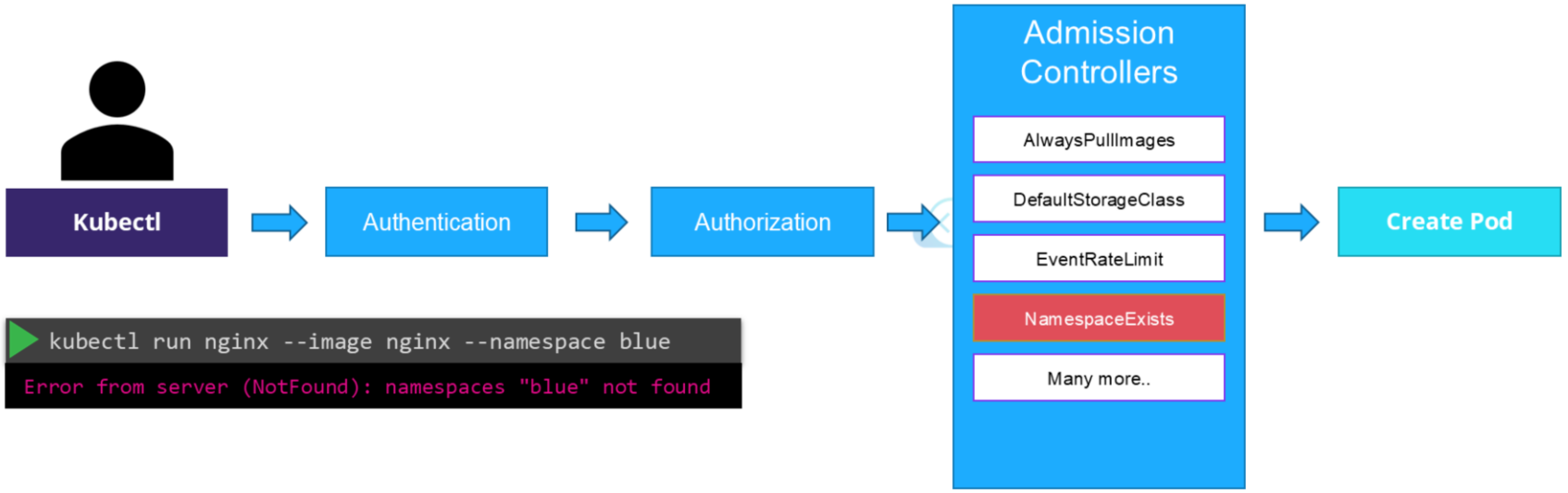


There are a number of admission controllers that come pre-built with Kubernetes such as AlwaysPullImages that ensures that everytime a pod is created the images are always pulled. The DefaultStorageClass admission controller observes the creation of PVCs and automatically adds a default storage class to them if one is not specified.

The event rate limit admission controller can help set a limit on the requests the API server can handle at a time to prevent the API server from flooding with requests.

The namespaceexists admission controller rejects requests to namespaces that do not exist. So let’s take that as an example and look at it in more detail.

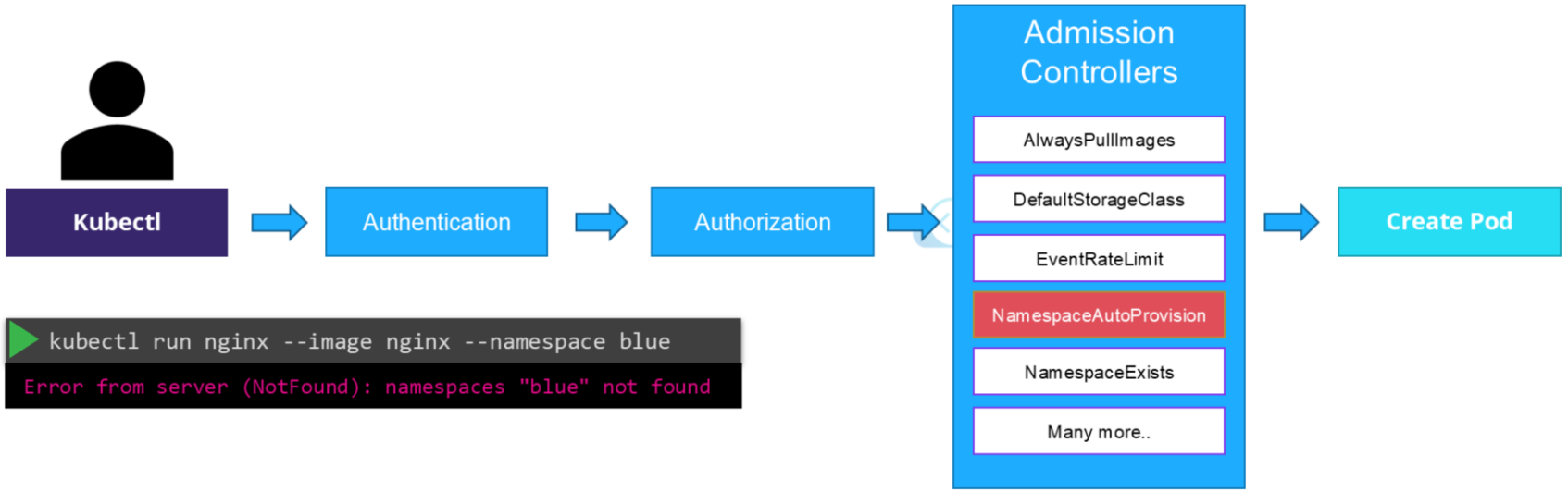
Admission Controllers



Say we want to create a POD in a namespace called blue that doesn't exist. If I run this command it would throw an error that says the namespace blue is not found.

What's happening here is that my request gets authenticated, then authorized, and then it goes through the Admission controllers. The namespaceexists admission controller handles the request and 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.

Admission Controllers



There is another admission controller that is not enabled by default. And that is called as the NamespaceAutoProvision admission controller. This will automatically create the namespace if it does not exist. This can be enabled by configuring the –enable-admission-plugins flag on the kube-api server.

View Enabled Admission Controllers

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

```
--enable-admission-plugins strings       admission plugins that should be enabled in addition to default enabled ones
(NamespaceLifecycle, LimitRanger, ServiceAccount, TaintNodesByCondition, Priority, DefaultTolerationSeconds,
DefaultStorageClass, StorageObjectInUseProtection, PersistentVolumeClaimResize, RuntimeClass, CertificateApproval,
CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass, MutatingAdmissionWebhook,
ValidatingAdmissionWebhook, ResourceQuota). Comma-delimited list of admission plugins: AlwaysAdmit, AlwaysDeny,
AlwaysPullImages, CertificateApproval, CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass,
DefaultStorageClass, DefaultTolerationSeconds, DenyEscalatingExec, DenyExecOnPrivileged, EventRateLimit,
ExtendedResourceToleration, ImagePolicyWebhook, LimitPodHardAntiAffinityTopology, LimitRanger, MutatingAdmissionWebhook,
NamespaceAutoProvision, NamespaceExists, NamespaceLifecycle, NodeRestriction, …. TaintNodesByCondition,
ValidatingAdmissionWebhook. The order of plugins in this flag does not matter.
```

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


First, to see a list of admission controllers enabled by default run the kube-apiserver –h command and grep for enable-admission-plugins. Here you'll see a list of admission controllers that are enabled by default. Note that if you are running this in a kubeadm based setup then you must run this command within the kubeapi-server-controplane pod using kubectl exec like this.

Enable Admission Controllers

kube-apiserver.service

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --runtime-config=api/all \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --v=2
  --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision
  --disable-admission-plugins=DefaultStorageClass
```

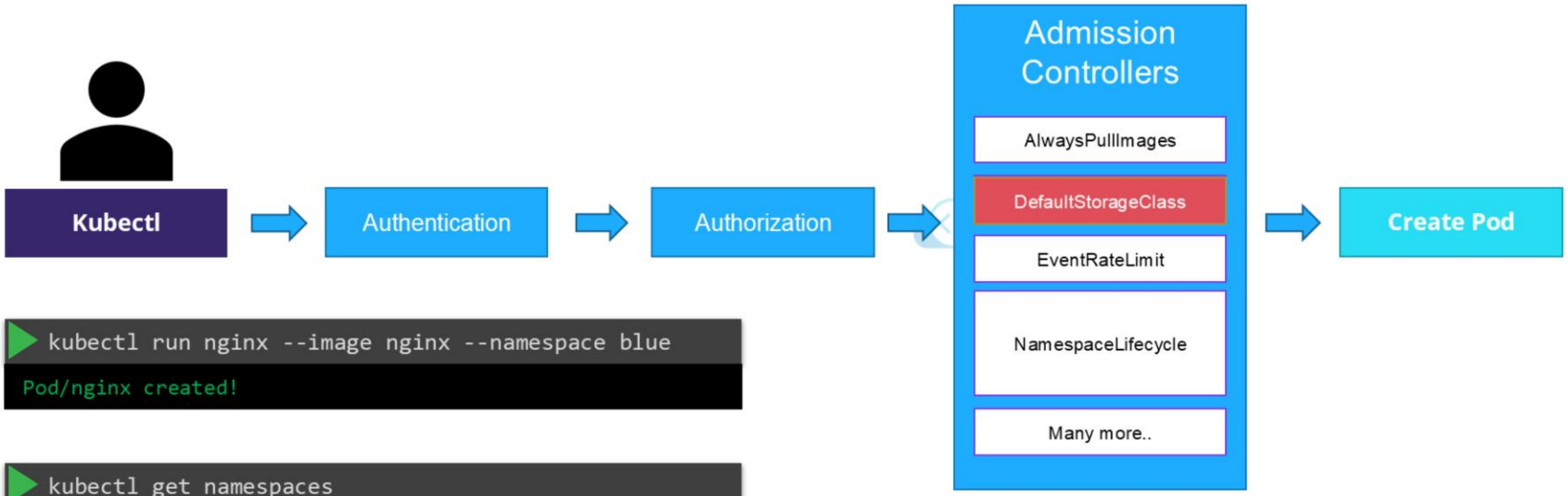
/etc/kubernetes/manifests/kube-apiserver.yaml

```
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
    - --enable-admission-plugins=NodeRestriction,NamespaceAutoPro
    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. If in a kube-adm based setup then update the flag withn the kube-apiserver manifest file as shown here on the right.

Similarly, to disable admission controller plugins you could use the disable admission plugins flag.

Admission Controllers




```
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
```


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, and then namespace autoprovisin controller, at which point it realizes that the namespace doesn't exist. So it creates the namespace automatically and the request goes through successfully to create the pod.

If you list the namespaces now you'll see that the blue namespace is automatically created.

So that’s one example of how an admission controller works. It can not only validate and reject requests from users, it can also perform operations in the backend, or change the request itself.

Note that the namespaceautoprovision and the namespace exists admission controllers are deprecated and is now replaced by namespacelifecycle admission controller. The namespace lifecycle 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.

KodeKloud

Visit www.kodekloud.com to learn more.