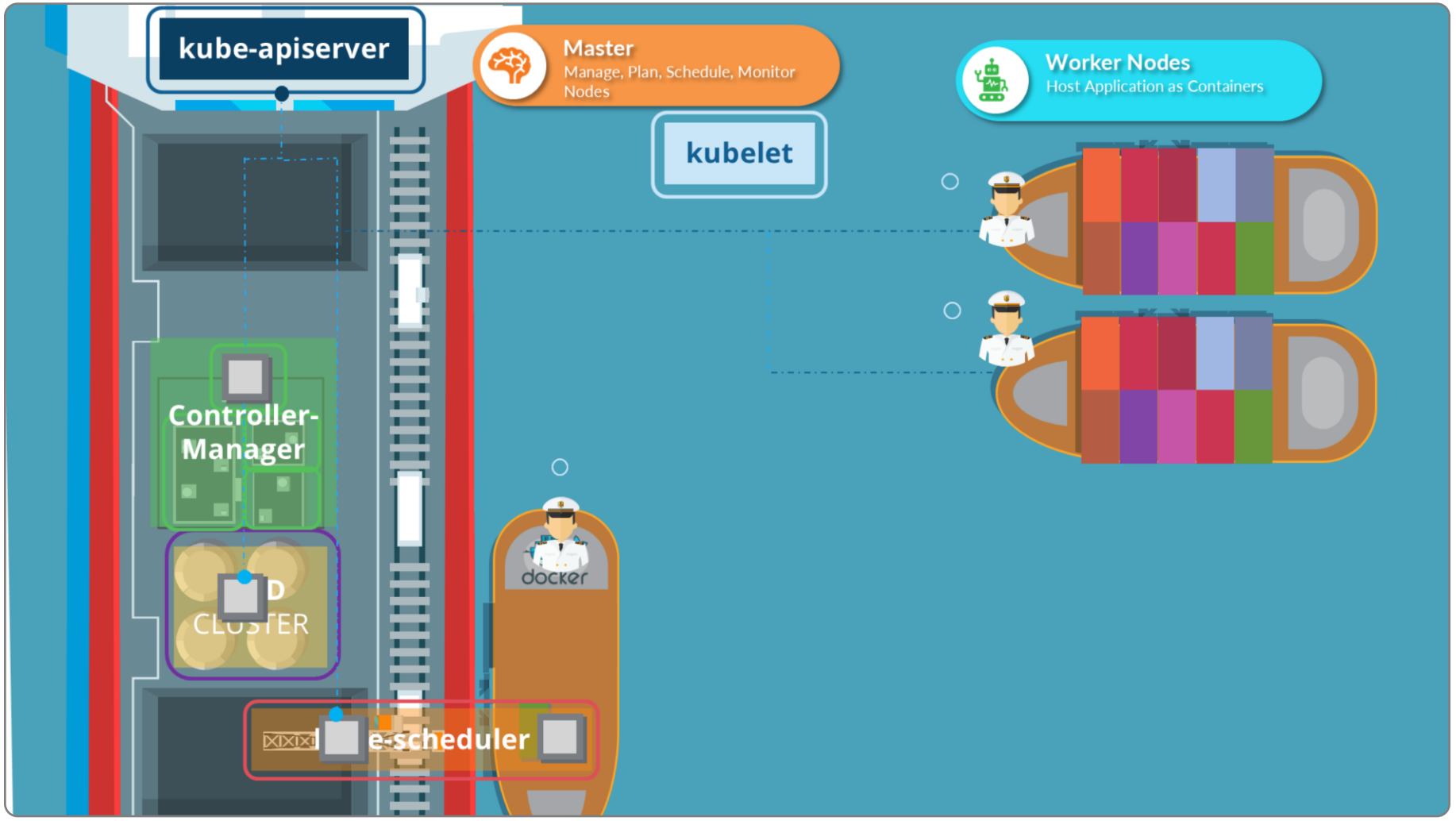
KodeKloud

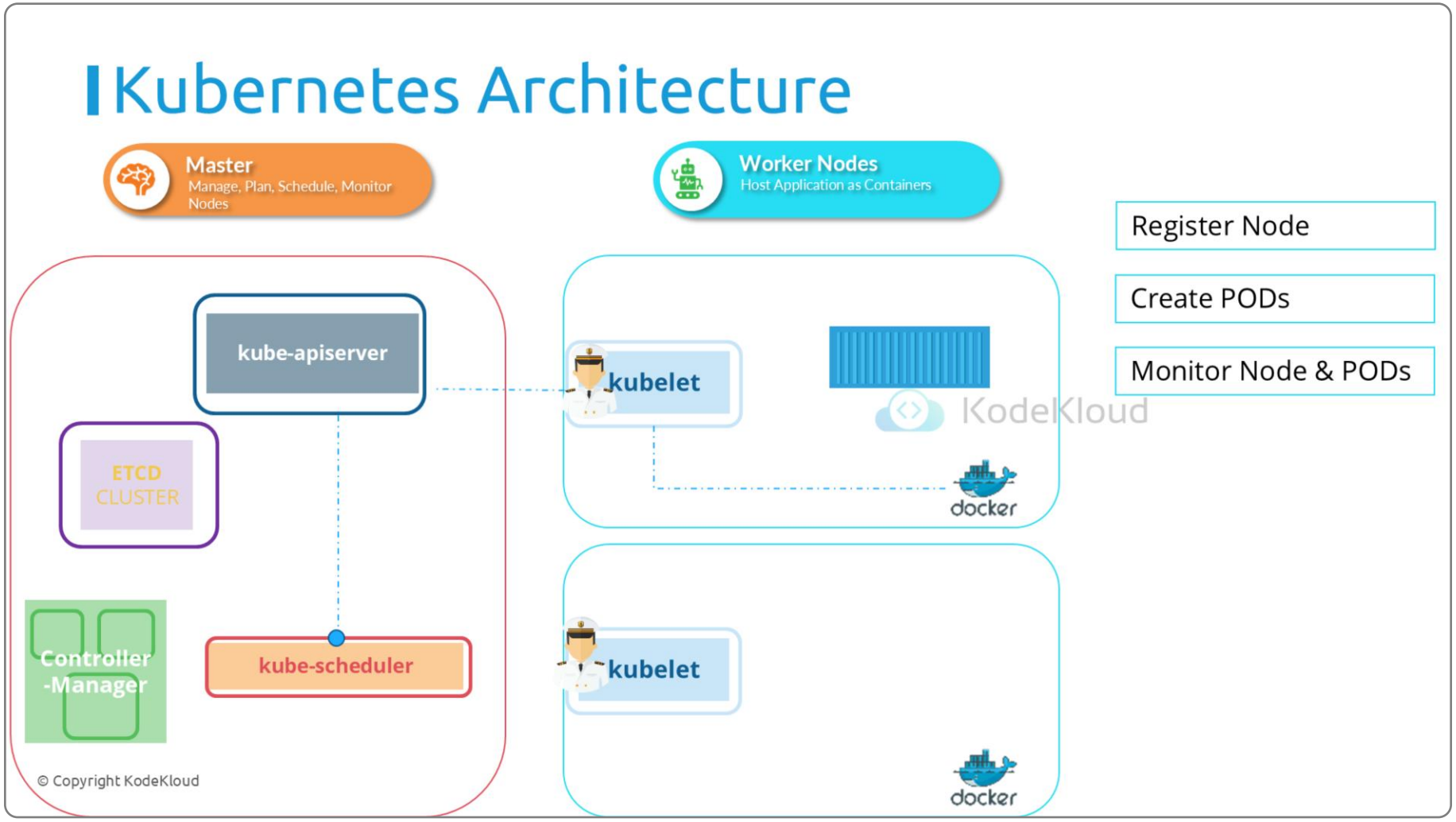
Visit www.kodekloud.com to learn more.

In this lecture we will revisit kubelet and the different approaches in configuring kubelets on nodes.



In the CKA course when we talked about the ship analogy, we discussed that the kubelet is like the captain on the ship. They lead all activities on a ship. They are the ones responsible for doing all the paperwork necessary to become part of the cluster, they are the sole point of contact from the master ship, they load or unload containers on the ship as instructed by the scheduler on the master, they also send back reports at regular intervals on the status of the ship and the containers on them.

What if the captain receives instructions from someone else pretending to be from the master ship? To whom should the captain reveal information regarding the cargo on his ship? How many are there, what are their contents, where are they going to etc.
It is important that all communications between the master ship or the kube-api server and the captain on the cargo ship or the kubelet is secure. And that is what we will see in this lecture.



The kubelet in the kubernetes worker node, registers the node with the kubernetes cluster. When it receives instructions to load a container or a POD on the node, it requests the container run time engine, which may be Docker, to pull the required image and run an instance. The kubelet then continues to monitor the state of the POD and the containers in it and reports to the kube-api server on a timely basis.

Installing kubelet

```
wget https://storage.googleapis.com/kubernetes-release/release/v1.20.0/bin/linux/amd64/kubelet
```

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  --container-runtime=docker \\
  --image-pull-progress-deadline=2m \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --v=2 \\
  --cluster-domain=cluster.local \\
  --file-check-frequency=0s \\
  --healthz-port=10248 \\
  --cluster-dns=10.96.0.10 \\
  --http-check-frequency=0s \\
  --sync-frequency=0s
```


Kubeadm does not deploy Kubelets

So we also discussed how to install the kubelet. Download the kubelet binary, and configure it to run as a service.

Just to refresh your memory, note that if you use the kubeadm tool to deploy your cluster we know that kubeadm automatically downloads the required binaries and bootstraps the cluster. However, it DOES NOT automatically deploy the kubelet. You must always manually install the kubelet on your worker nodes. So that's something to note.

Before we look into security, it is important to understand some basic details about the flags and configuration files for configuring the kubelet.

While configuring kubelet as a service we see that it has several options configured such as the container-runtime or the kubeconfig file which is used by the kubelet to authenticate to the kube-api server. The netowk plugin, the version, as well as some additional details such as cluster-domain, file check frequency, cluster dns and others.

Installing kubelet

```
wget https://storage.googleapis.com/kubernetes-release/release/v1.20.0/bin/linux/amd64/kubelet
```

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  --container-runtime=remote \\
  --image-pull-progress-deadline=2m \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --v=2 \\
  --config=/var/lib/kubelet/kubelet-config.yaml
  --file-check-frequency=0s \\
  --healthz-port=10248 \\
  --cluster-dns=10.96.0.10 \\
  --http-check-frequency=0s \\
  --sync-frequency=0s
```

kubelet-config.yaml

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
clusterDomain: cluster.local
fileCheckFrequency: 0s
healthzPort: 10248
clusterDNS:
- 10.96.0.10
httpCheckFrequency: 0s
syncFrequency: 0s
```


This is how it was originally. But with the release of version 1.10 most of these parameters were moved to another file called the kubelet-config file for ease of deployment and configuration management. The object created within the file is named KubeletConfiguration. And on the kubelet service we pass the path to this file as a command line argument named “config”. Note that within the file the parameters use a camel case. So all dashes that separate words are removed and words are written without spaces and the first letter of each word is capitalized except the first word. So http-check-frequency becomes httpCheckFrequency with c and f capitalized.

If you specify a flag both in the command line as well as in the file then the flag specified on the command line will override whatever is in the file.

We discussed that the kubeadm tool does not download or install the kubelet. However it can help in managing the kubelet-configuration. So if there are a large number of worker nodes, instead of manually creating these kubelet-config files, kubeadm tool can help in automatically configuring the kubelet-config files on those nodes when you run the kubeadmin join command.

View kubelet options

```
ps -aux | grep kubelet
```

```
root      2095  1.8  2.4 960676 98788 ?        Ssl  02:32   0:36 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --cgroup-driver=cgroupfs --cni-bin-dir=/opt/cni/bin --cni-conf-dir=/etc/cni/net.d --network-plugin=cni
```

```
cat /var/lib/kubelet/config.yaml
```

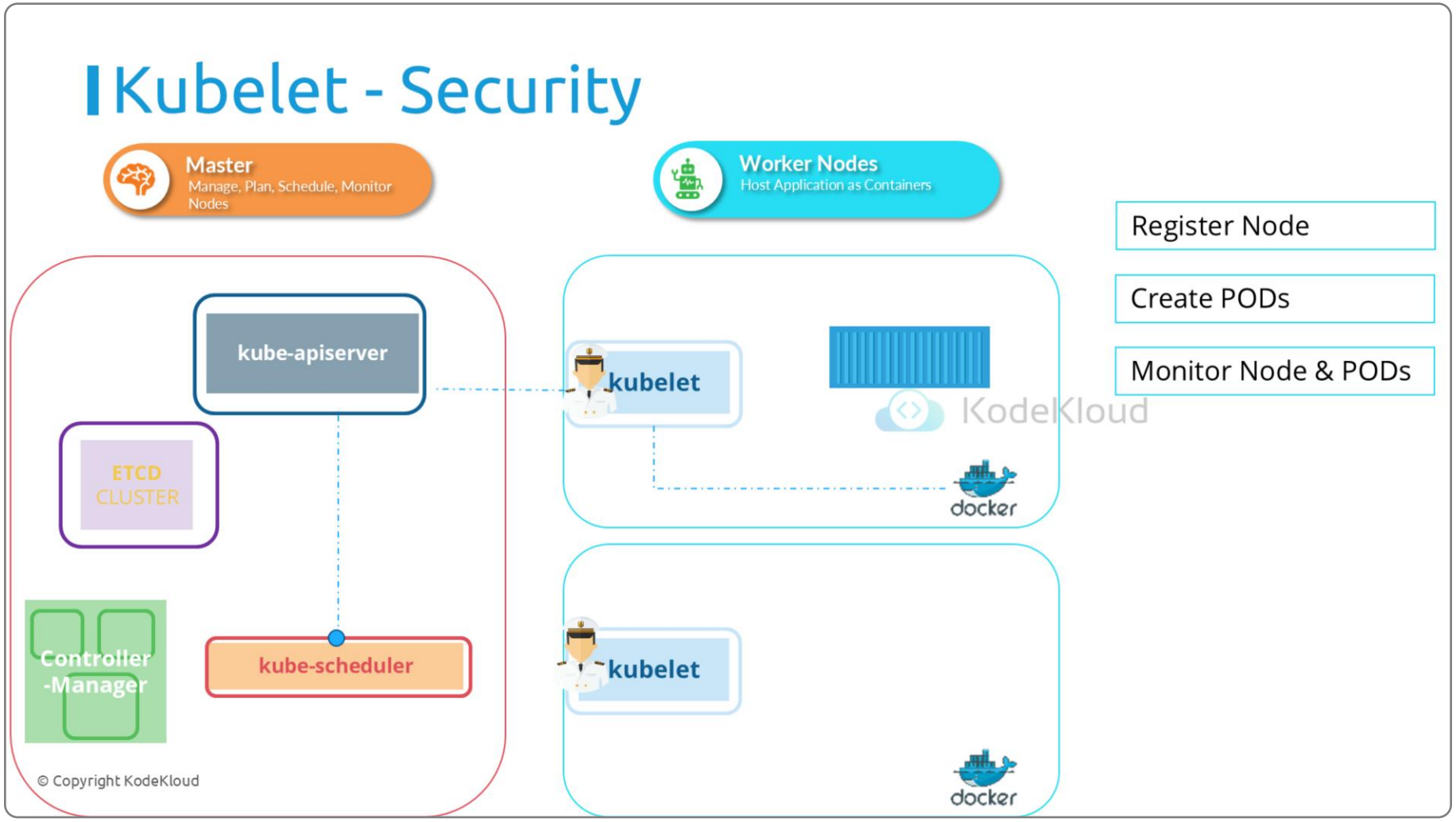
```
apiVersion: kubelet.config.k8s.io/v1beta1
clusterDNS:
- 10.96.0.10
clusterDomain: cluster.local
cpuManagerReconcilePeriod: 0s
evictionPressureTransitionPeriod: 0s
fileCheckFrequency: 0s
healthzBindAddress: 127.0.0.1
healthzPort: 10248
httpCheckFrequency: 0s
imageMinimumGCAge: 0s
kind: KubeletConfiguration
nodeStatusReportFrequency: 0s
nodeStatusUpdateFrequency: 0s
rotateCertificates: true
runtimeRequestTimeout: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
```


Once the kubelet is configured inspecting the kubelet process on a node shows the different options configured, including the path to the kubeconfig file. In this case it happens to be at /var/lib/kubelet/config.yaml

Inspecting that file gives us a list of parameters configured for the kubelet.

So the reason we discussed about all of these is to get to this point. To know how to inspect the current configurations of

a kubelet. There are some parameters passed in with the kubelet service and many of the flags are within the kubelet configuration file.



Let us now go over some of the security aspects of kubelet. How do we make sure that the kubelet only responds to requests from the kube-api server and not anyone else.

Kubelet

Port	Description
10250	Serves API that allows full access
10255	Serves API that allows unauthentcated read-only access

The kubelet serves on 2 ports. Port 10250 which is where the kubelet serves its API server that allows full access. And port 10255 which servers an API that allows unauthenticated/unauthorized read-only access.

Kubelet

Port	Description
10250	Serves API that allows full acces
10255	Serves API that allows unauthe access

```
curl -sk https://localhost:10250/pods/
```

```
{"kind":"PodList","apiVersion":"v1","metadata":{},"items":[{"meta
nngzb","generateName":"kube-proxy-","namespace":"kube-system","se
proxy-nngzb","uid":"3c5e5745-f67e-4f91-beb9-f11d4da7a0d0","resour
```

```
curl -sk https://localhost:10250/logs/syslog
```

```
Nov 10 11:26:59 host01 kernel: [    0.000000] Linux version 4.15.
7.3.0 (Ubuntu 7.3.0-16ubuntu3))
#31-Ubuntu SMP Tue Jul 17 15:39:52 UTC 2018 (Ubuntu 4.15.0-29.31-
Nov 10 11:26:59 host01 kernel: [    0.000000] Command line: BOOT_
root=/dev/mapper/host01--vg-root ro quiet spl
ash vt.handoff=1
Nov 10 11:26:59 host01 kernel: [    0.000000] KERNEL supported cp
Nov 10 11:26:59 host01 kernel: [    0.000000]   Intel GenuineInte
Nov 10 11:26:59 host01 kernel: [    0.000000]   AMD AuthenticAMD
Nov 10 11:26:59 host01 kernel: [    0.000000]   Centaur CentaurHa
Nov 10 11:26:59 host01 kernel: [    0.000000] x86/fpu: x87 FPU wi
Nov 10 11:26:59 host01 kernel: [    0.000000] e820: BIOS-provided
```



```
"/attach/{podNamespace}/{podID}/{containerName}":       "p
"/attach/{podNamespace}/{podID}/{uid}/{containerName}": "p
"/configz": "proxy",
"/containerLogs/{podNamespace}/{podID}/{containerName}": "
"/cri/":                    "proxy",
"/cri/foo":                 "proxy",
"/debug/flags/v":           "proxy",
"/debug/pprof/{subpath:*}": "proxy",
"/exec/{podNamespace}/{podID}/{containerName}":       "pro
"/exec/{podNamespace}/{podID}/{uid}/{containerName}": "pro
"/healthz":                             "proxy",
"/healthz/log":                         "proxy",
"/healthz/ping":                        "proxy",
"/healthz/syncloop":                    "proxy",
"/logs/":                               "log",
"/logs/{logpath:*}":                    "log",
"/metrics":                             "metrics",
"/metrics/cadvisor":                    "metrics",
"/metrics/probes":                      "metrics",
"/metrics/resource/v1alpha1":           "metrics",
"/pods/":                               "proxy",
"/portForward/{podNamespace}/{podID}": "proxy",
"/portForward/{podNamespace}/{podID}/{uid}":         "prox
"/run/{podNamespace}/{podID}/{containerName}":       "prox
"/run/{podNamespace}/{podID}/{uid}/{containerName}": "prox
"/runningpods/":    "proxy",
"/spec/":           "spec",
"/stats/":          "stats",
"/stats/container": "stats",
"/stats/summary":   "stats",
```

By default the kubelet allows anonymous access to its API. So if you do a curl to port 10250 at the API - pods, you'll be able to see the list of pods running on the node. Similarly you can access other APIs such as viewing the system logs of the node on which the kubelet is running by going to the API /logs/syslog.

Kubelet

Port	Description
10250	Serves API that allows full access
10255	Serves API that allows unauthentcated read-only access

```
curl -sk http://localhost:10255/metrics
# HELP apiserver_audit_event_total [ALPHA] Counter of audit events generated and sent to the audit backend.
# TYPE apiserver_audit_event_total counter
apiserver_audit_event_total 0
# HELP apiserver_audit_requests_rejected_total [ALPHA] Counter of apiserver requests rejected due to an error in audit lo
# TYPE apiserver_audit_requests_rejected_total counter
apiserver_audit_requests_rejected_total 0
# HELP apiserver_client_certificate_expiration_seconds [ALPHA] Distribution of the remaining lifetime on the certificate
requ
est.
# TYPE apiserver_client_certificate_expiration_seconds histogram
apiserver_client_certificate_expiration_seconds_bucket{le="0"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="1800"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="3600"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="7200"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="21600"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="43200"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="86400"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="172800"} 0
```

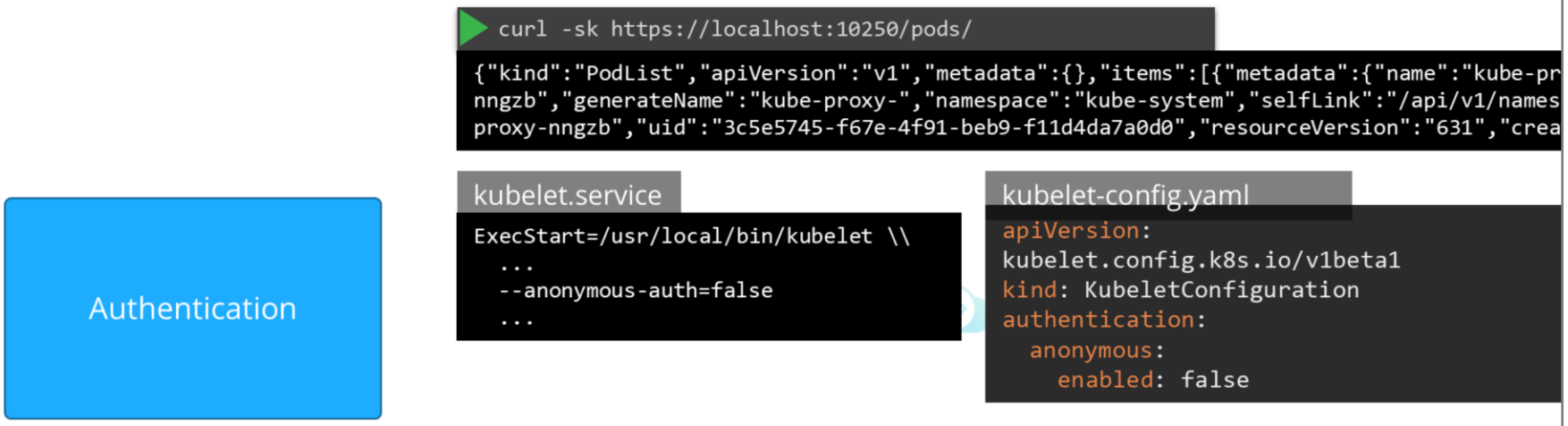

Similary the service running at port 10255 provides read-only access to metrics to any unauthenticated unauthorized clients.

As you can imagine, This is of-course a big security risk. Anyone that knows the IP address of these hosts can access these APIs to perform anything that the API server can do – such as view existing pods, create new pods, exec into existing pods, view usage metrics and stats, logs and more.

Kubelet Security

So any request that comes to the kubelet is first authenticated and then authorized. The authentication process decides if the user or the requesting entity has access to the API and the authorization process decides what areas of the API can the user access and what operations can they perform.

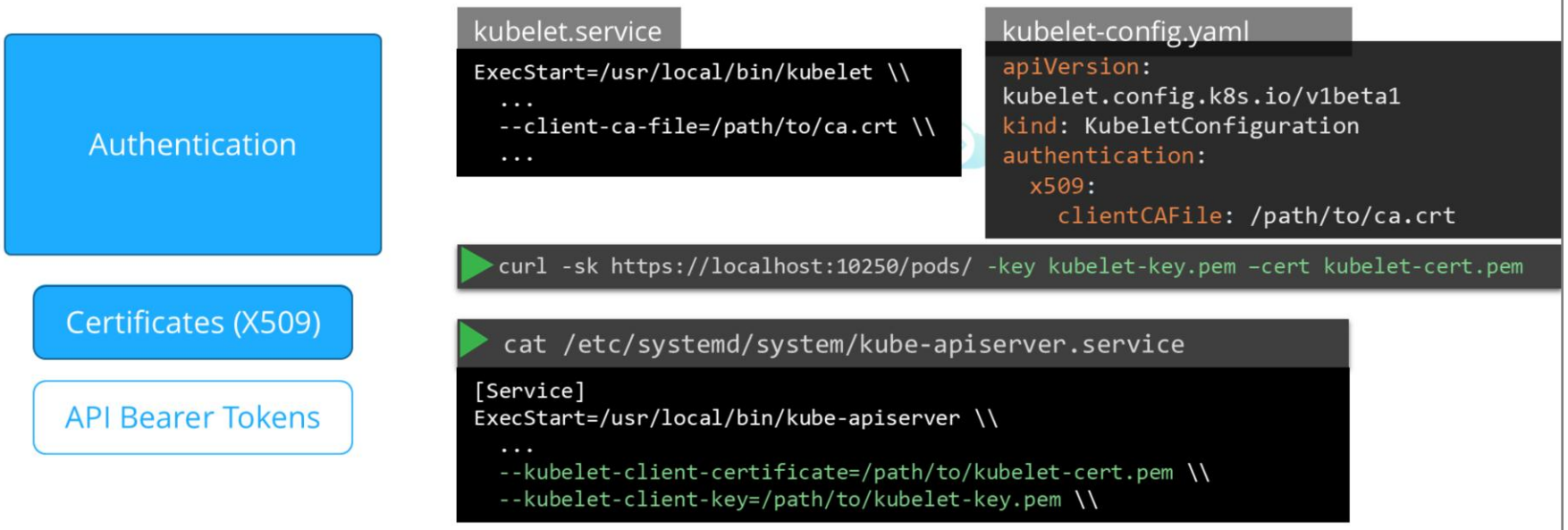
Kubelet Security



Let's look at authentication first. We saw that if you did a curl to the API server at port 10250 it returns the list of pods. That's because by default the kubelet permits all requests to go through without any kind of authentication. These requests are marked as anonymous users part of an unauthenticated group. This behaviour can be changed by setting the anonymous-auth flag to false in the kubelet service configuration file. Of course this parameter can also be set in the external KubeletConfiguration file as we saw before. Only that it is now under the authentication section like this.

So the best practice is to disable anonymous authentication and enable supported authentication mechanisms.

Kubelet Security



There are 2 authentication mechanisms. <c> Certificate based or API bearer token based authentication. <c> Of these we will look at the certificates based authentication approach. Provide the ca file using the client-ca-file command line argument on the kubelet service with the path to the CA bundle. As we discussed before this can also be set using the kubelet-config file with the authentication section and the clientCAFile parameter in it. And that is the recommended approach. Now that the certificate is configured here, to access the API you must provide the client certificates to authorize. So if you were doing a curl you now have to pass in the certificate and the associated key to authenticate into

the kubelet.

As far as the kubelet is concerned the kube-api server is a client. So the kube-api server also has to authenticate to the kubelet to interact with it. For this the api server must have the kubelet-client certificate and key configured. And that's the flags you see in the kube api server service configuration file. So remember that the kube api server is itself a server. And it has its own set of certificates and authentication configurations. And all the other components of the Kubernetes architecture that requires to communicate with the kube-api server must have certificates configured. However here we are referring to the kubelet alone. And in this particular instance when the kube api server tries to reach the kubelet it is a client of kubelet and it must have the kubelet client certificate and key configured correctly for it to function.

And this is also the approach used by the kubeadm tool to secure the kubelet.

Now if neither of these 2 authentication mechanisms explicitly reject a request, the default behaviour of the kubelet is to allow the request as "anonymouse requests" with a username of system:anonymous and a group of "system:unauthenticated".

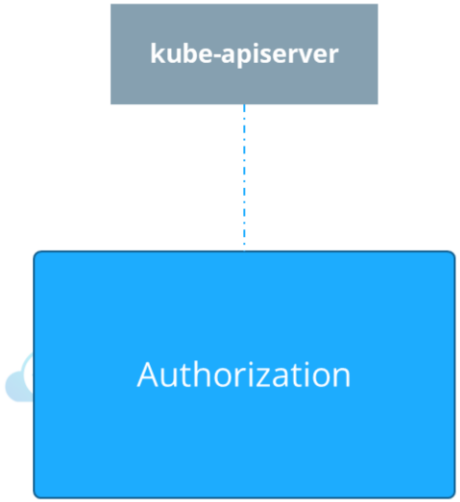
Kubelet Security

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  ...
  --authorization-mode=Webhook
  ...
```

kubelet-config.yaml

```
apiVersion:
kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authori
  mode: Webhook
```



Next let's talk about Authorization. Once the user gains access to the system what resources of the kubelet API can they interact with and what actions can they perform.

The default authorization mode is "AlwaysAllow" and this allows all access to the API. To prevent this we set the authorization mode to webhook. When set to webhook, the kubelet makes a call to the API server to determine whether each request is authorized.

Kubelet Security

```
curl -sk https://localhost:10255/metrics
```

```
# HELP apiserver_audit_event_total [ALPHA] Counter of audit events generated and sent to the audit backend.
# TYPE apiserver_audit_event_total counter
apiserver_audit_event_total 0
# TYPE apiserver_client_certificate_expiration_seconds histogram
apiserver_client_certificate_expiration_seconds_bucket{le="0"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="1800"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="86400"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="172800"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="345600"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="604800"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="2.592e+06"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="7.776e+06"} 0
apiserver_client_certificate_expiration_seconds_bucket{le="1.5552e+07"} 0
```

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  ...
  --read-only-port=10255
  ...
```

kubelet-config.yaml

```
apiVersion:
kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
readOnlyPort: 0
```


We also talked about the metrics service running on port 10255. The one with the read-only access that does not require authentication or authorization mechanisms. This service is enabled when the read only port flag on the kubelet service is not set as it defaults to port 10255 or it is set to some other port. If this value is set to 0, then this service is disabled.

Note that if the kubelet config file is in use, then this flag has a default value of 0. So its disabled by default. So you'll probably see this disabled on most systems.

Kubelet Security

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  ...
  --anonymous-auth=false \\
  --client-ca-file=/path/to/ca.crt \\
  --authorization-mode=Webhook
  --read-only-port=0
  ...
```

kubelet-config.yaml

```
apiVersion:
kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
authentication:
  anonymous:
    enabled: false
  x509:
    clientCAFile: /path/to/ca.crt
authorization:
  mode: Webhook
readOnlyPort:
0
```


So let us summarize what we have learned. The kubelet by default allows anonymous authentication, and so to prevent it we set the anonymous flag to false. All of these settings can be done either in the command line or the recommended approach is to set them in the kubelet configuration file.

Kubelet supports 2 authentication mechanisms. The certificate based authentication and bearer token based authentication. For certificate based authentication set the client-ca-file path to the kubelet.

By default the kubelet allows all requests without authorization. To configure authorization, set the authorization mode to webhook. When this is set the kubelet authorizes requests through the kube api server.

By default the read only port is set to 10255 which enables the metrics server to expose metrics and other vulnerable information. This can be disabled by setting it to 0.

Well that’s it for now and head over to the labs and practice working with kubelet security.

Labs – Kubelet Security

- Explore kubelet security
- Enable Authentication/Authorization on Kubelet

In the upcoming labs you will explore securing the kubelet by first exploring current settings on a kubelet and then enabling authentication and authorization on the kubelet. Head over to the labs and I will see you in the next lecture.

References

https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/
https://kubernetes.io/docs/tasks/administer-cluster/kubelet-config-file/
https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/kubelet-integration/#configure-kubelets-using-kubeadm
https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-authentication-authorization/
https://gist.github.com/lizrice/c32740fac51db2a5518f06c3dae4944f

In this lecture we will take a look at Pod security polocies.

Let's take this pod definition file for example. It is used to create a pod with an ubuntu image. It has a number of configurations that we might not want to allow users to perform on our cluster. For example under the security context section we have the privileged flag set to True. This will allow the processes running inside the pod to have root privileges on the host. The container is configured to run as user 0. That is the root user. We have capabilities added to the cluster. We have volume mounts configured to use hostpath. And all of these can make your container vulnerable to attack. So we may want to create policies that prevent users from creating containers with certain configurations. And that's where Pod

Security Policies come in. Pod security policies help in defining policies to restrict pods from being created with specific capabilities or privileges.

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, …. PodSecurityPolicy, TaintNodesByCondition,
ValidatingAdmissionWebhook. The order of plugins in this flag does not matter.
```


So we have been talking about admission controllers so far. One of the admission controller that comes built-in with Kubernetes is the Pod security policy admission controller. And this is not enabled by default. You may enable it by updating the enable admission plugins flag on the api server.

Pod Security Policy


```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
   - name: ubuntu
     image: ubuntu
     command: ["sleep", "3600"]
     securityContext:
       privileged: True
       runAsUser: 0
       capabilities:
          add: ["CAP_SYS_BOOT"]
   volumes:
   - name: data-volume
     hostPath:
       path: /data
```

KodeKloud

When enabled the pod security policy admission controller observes all pod creation requests and validates the configuration against a set of pre-configured rules. If it detects a match that we have configured the request is rejected.

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=PodSecurityPolicy
```

/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=PodSecurityPolicy
    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
```


As we discussed before to enable podsecurity policy add it to the enable admission plugins flag on the api server.

Define Pod Security Policy

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
   - name: ubuntu
     image: ubuntu
     command: ["sleep", "3600"]
     securityContext:
       privileged: True
       runAsUser: 0
       capabilities:
          add: [”CAP_SYS_BOOT"]
  volumes:
  - name: data-volume
    hostPath:
       path: /data
       type: Directory
```

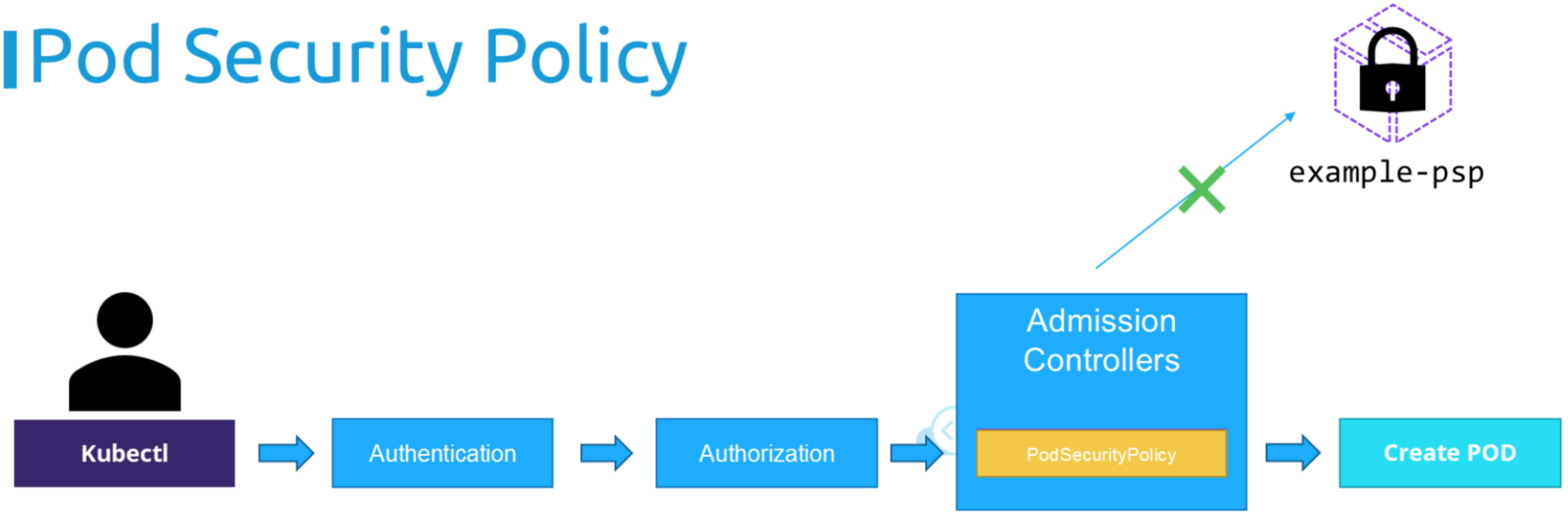
psp.yaml

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
   name: example-psp
spec:
    privileged: false
    seLinux:
      rule: RunAsAny
    supplementalGroups:
      rule: RunAsAny
    runAsUser:
      rule: RunAsAny
    fsGroup:
      rule: RunAsAny
```


Next we create a POD Security policy object defining our policy requirements. In this case we don’t want to allow creating containers with privileged flag set to true. So we create a pod security policy with the apiversion ,kind metadata and spec. Apiversion is policy/v1beta1 as of this recording. kind is pod security policy . Metadata has the name example-psp. And then under spec we have the privileged flag set to false. This will reject all pod creation requests with the privileged flag set to true. And there are some required fields such as seLinux, runAsUser etc. For now these are all set to RunAsAny, which essentially allows pods to be created with any of these flags. We then create this object to create the pod security

policy object.

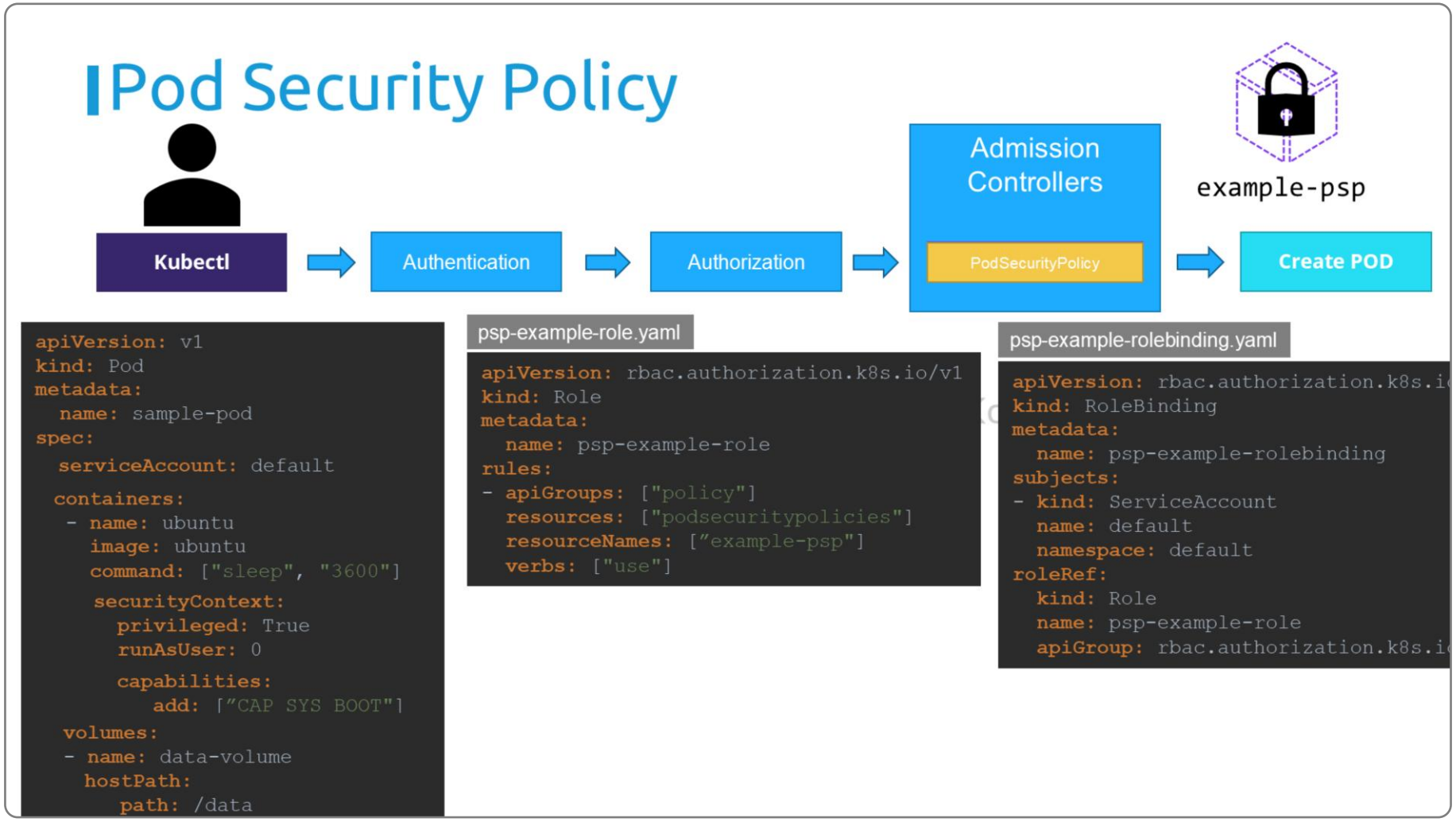
Pod Security Policy




```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
   - name: ubuntu
     image: ubuntu
     command: ["sleep", "3600"]
     securityContext:
       privileged: True
       runAsUser: 0
       capabilities:
```


So now we have the pod security policy admission controller enabled. And the pod security policy object created with our rules in it. The next time a request comes in the pod security policy admission controller will try to query the pod security policies objects to looks for rules to deny or approve the request. However if you have only enabled the admission controller and not authorized security policies api, then the admission controller will not be able to communicate with the pod security policies api and thus reject all requests. Even if the requests are valid. So remember that as soon as the pod security policy admission controller is enabled it will start denying all requests to create a pod.

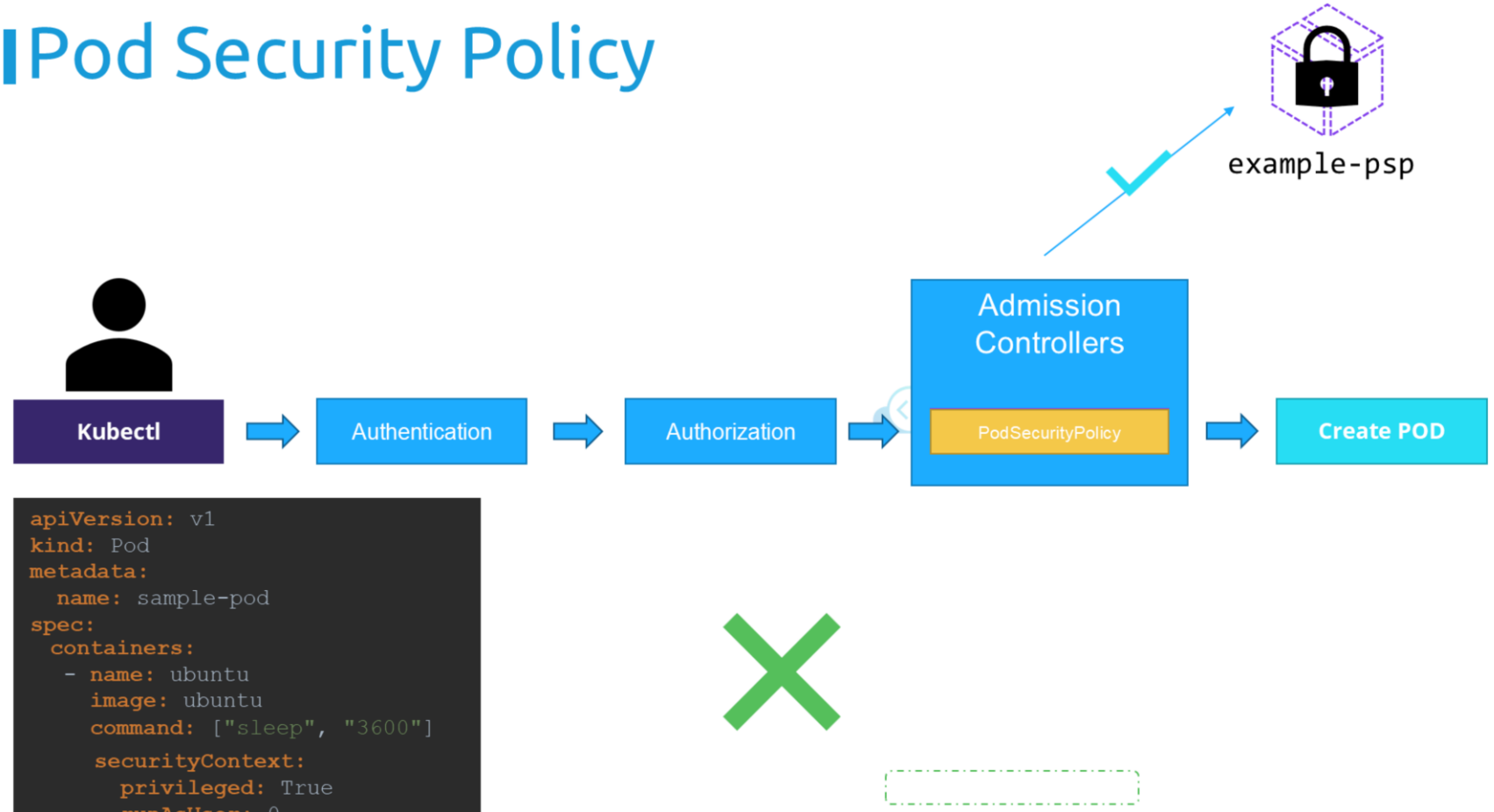
The entity that is trying to create a pod in this case a user or the Pod itself must have access to the pod security policy api. So how do we do that? How do we give one object access to an API in Kubernetes? We've learned that this can be done using role based access controls.



Every pod when created has a service account associated with it. Even if you don't specify one explicitly the service account named default in that namespace that that pod belongs to is assigned to the pod. To authorize the pod with the pod security policy create a role and bind it to the default service account in that namespace. The role isnamed psp-example-role. And is configured with access to use the API pod security policies under the policy apigroup with the resourcename example-psp. That's the name assigned to the pod security policy we created. Next we create a rolebinding to bind the role to the default service account. In our case since we are in the default namespace the namespace would

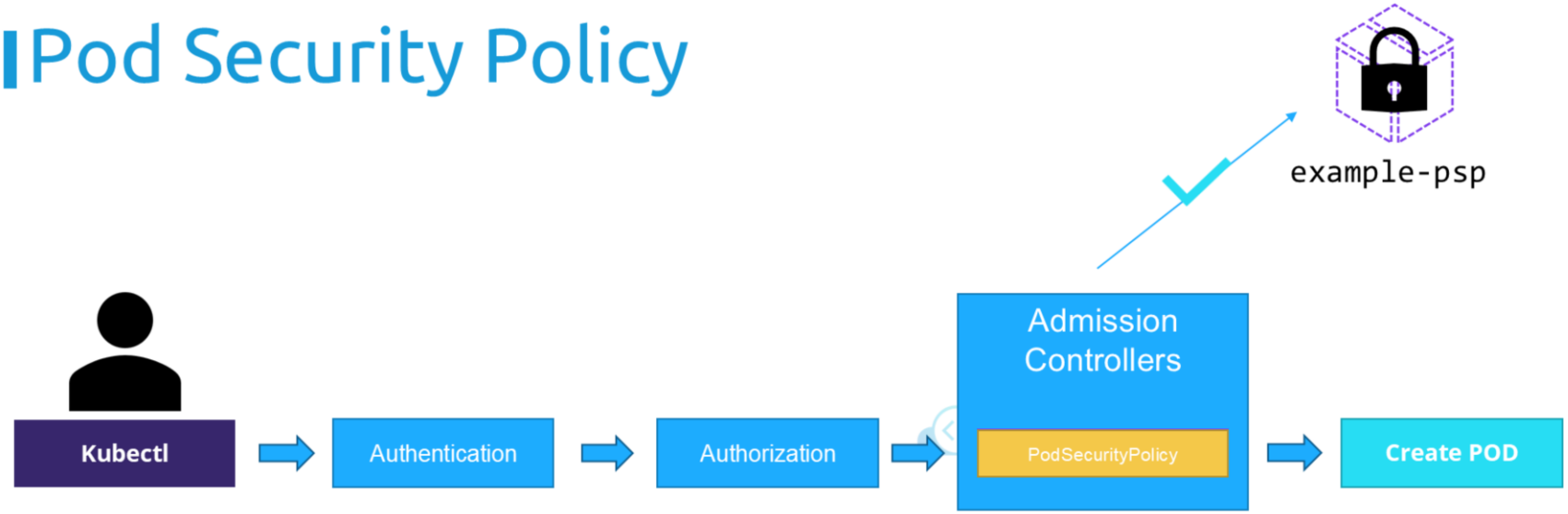
also be default. These role and role bindings will ensure access to pod security policies are authorized.

Pod Security Policy



So now with the appropriate role binding set, when the pod creation request comes in, the admission controller can now authorize through to the pod security policies that we created, and since we have disabled use of privileged flag, the request is denied.

Pod Security Policy




```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
   - name: ubuntu
     image: ubuntu
     command: ["sleep", "3600"]
     securityContext:
       privileged: False
       runAsUser: 0
       capabilities:
```


However if we were to send a request with the privileged flag set to false, then the request is approved.

Define Pod Security Policy

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
   - name: ubuntu
     image: ubuntu
     command: ["sleep", "3600"]
     securityContext:
       privileged: True
       runAsUser: 0
       capabilities:
          add: ["CAP_SYS_BOOT"]
  volumes:
  - name: data-volume
    hostPath:
       path: /data
       type: Directory
```

psp.yaml

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
   name: example-psp
spec:
   privileged: false
    seLinux:
      rule: RunAsAny
    supplementalGroups:
      rule: RunAsAny
    runAsUser:
      rule: 'MustRunAsNonRoot'
    requiredDropCapabilities:
    - 'CAP_SYS_BOOT'
    defaultAddCapabilities:
    - 'CAP_SYS_TIME'
    volumes:
    - 'persistentVolumeClaim'
```

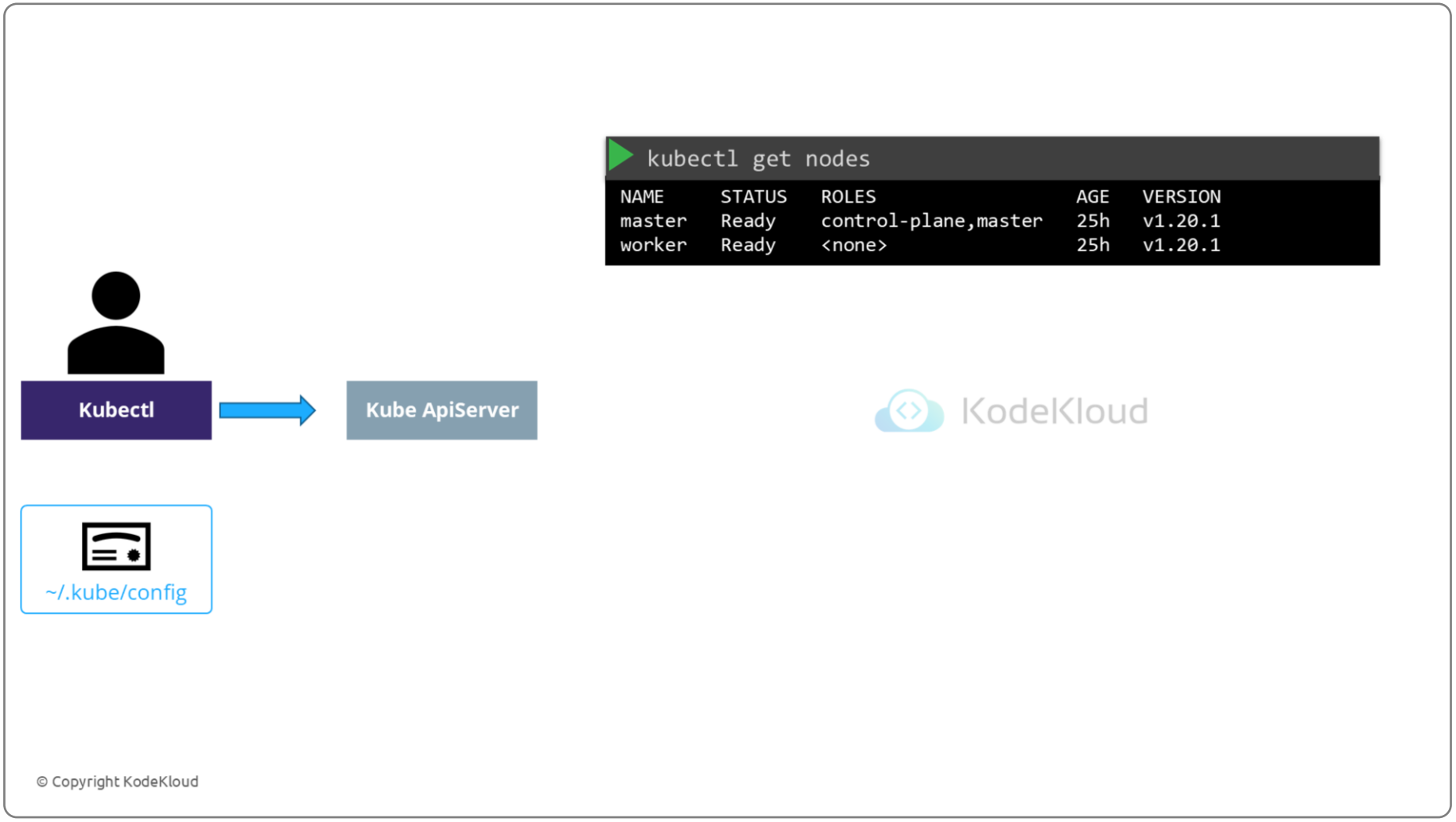
https://kubernetes.io/docs/concepts/policy/pod-security-policy/

Let's take a closer look at the pod security policy object itself. To disallow privileged containers, set the privileged option to false. To disallow runas root user set the runasuser rule to MustRunAsNonRoot. To disallow certain capabilities, add the requiredDropCapabilities and provide a list of capabilities that must be dropped. You may use the defaultAddCapabilities to provide a list of capabilities that must be added by default. Those listed here will be added to the pod definition automatically. This indicates that pod security policies are not only used to verify and approve or reject pod creation requests but they can also change or mutate the pod definition by adding default values.

To ensure hostPath volumes are not set, define a set of allowed volume types under volumes section within the pod security policy. For more examples, refer to the documentation on pod security policies.

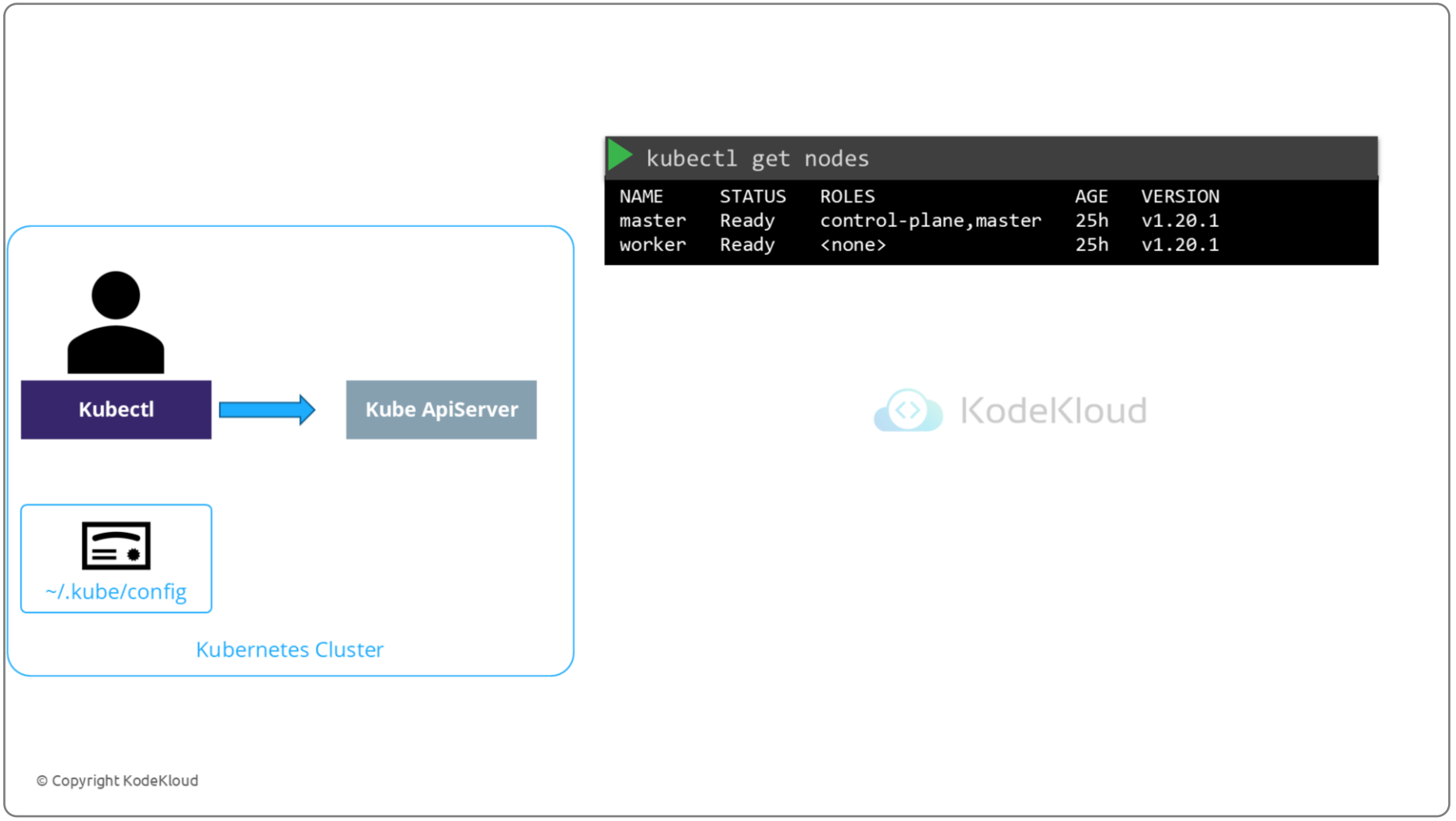
Well that’s it for now. Head over to the labs and practice working with pod security policies.

In this lecture we will take a look at the kubectl proxy in a bit more detail.

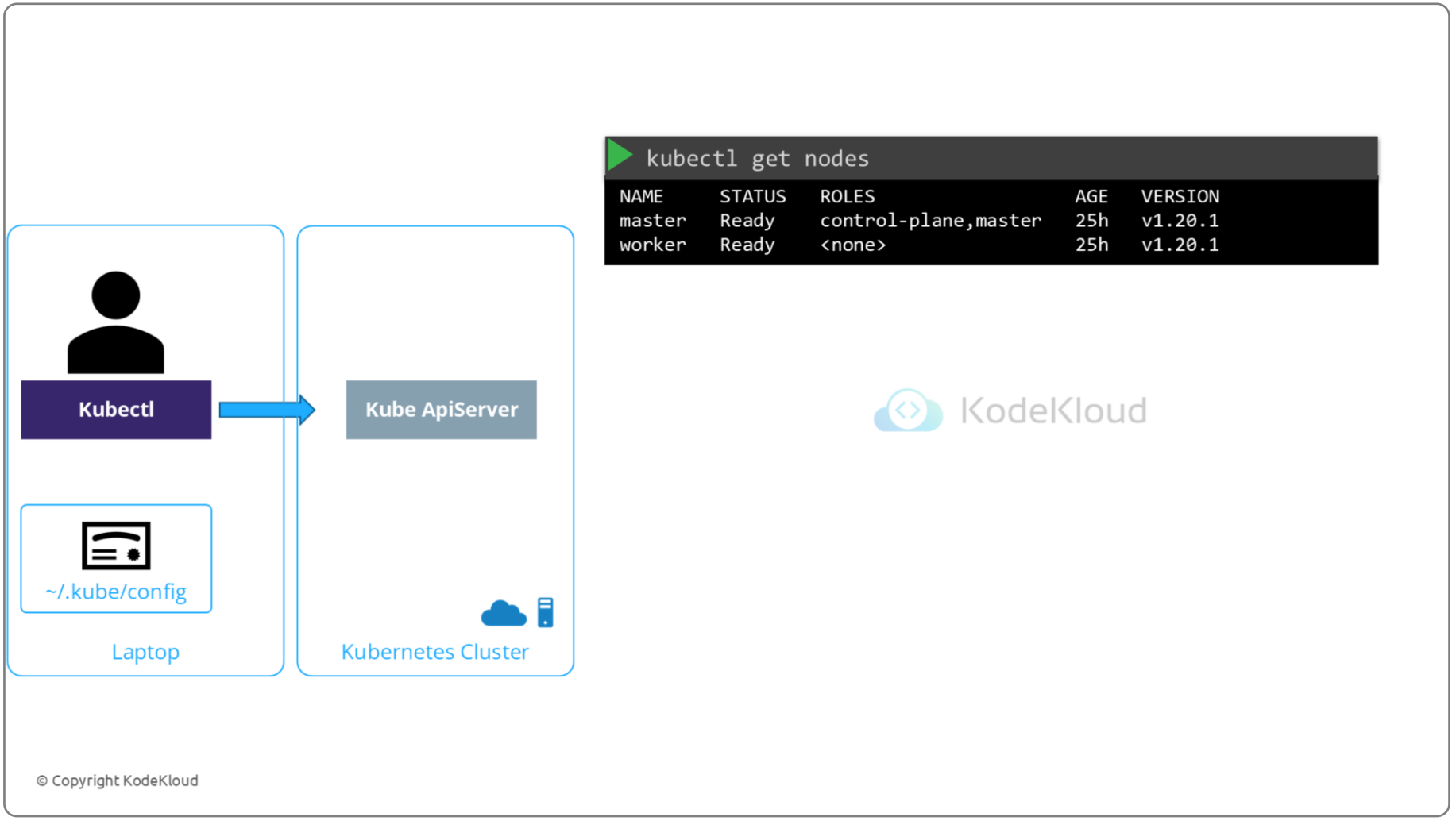


We learned that we can use the kubectl utility to interact with the Kubernetes api server. And when doing so we don't need to provide authentication mechanism in the command line because we configure it in the kube config file on our system. We learned that the kube config file has the necessary user details and credentials in it to access and interact with the Kubernetes cluster through the api server.

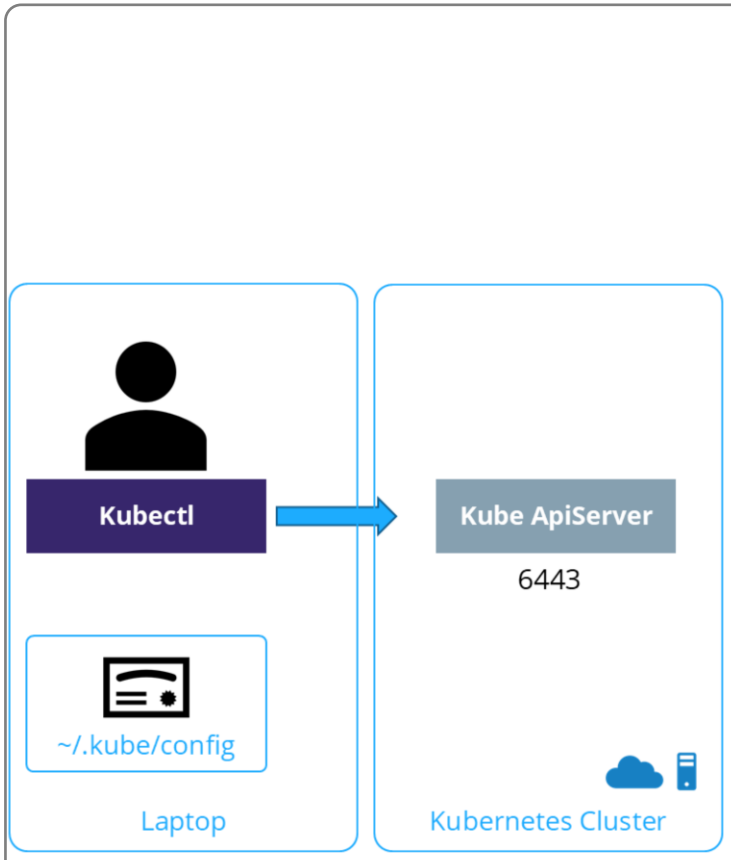
Now the kubectl utility could be anywhere.



It may be on the same host as the master node in the Kubernetes cluster as it is in our labs



or it could be elsewhere say on our laptop. The cluster could be within a VM on my laptop or on a private server in the environment or a server on the public cloud environment or it could be from <c> some managed Kubernetes service provider. No matter where the cluster is hosted you can manage it locally using the kubectl utility from your laptop as long as you have the kubeconfig file with the necessary security credentials.



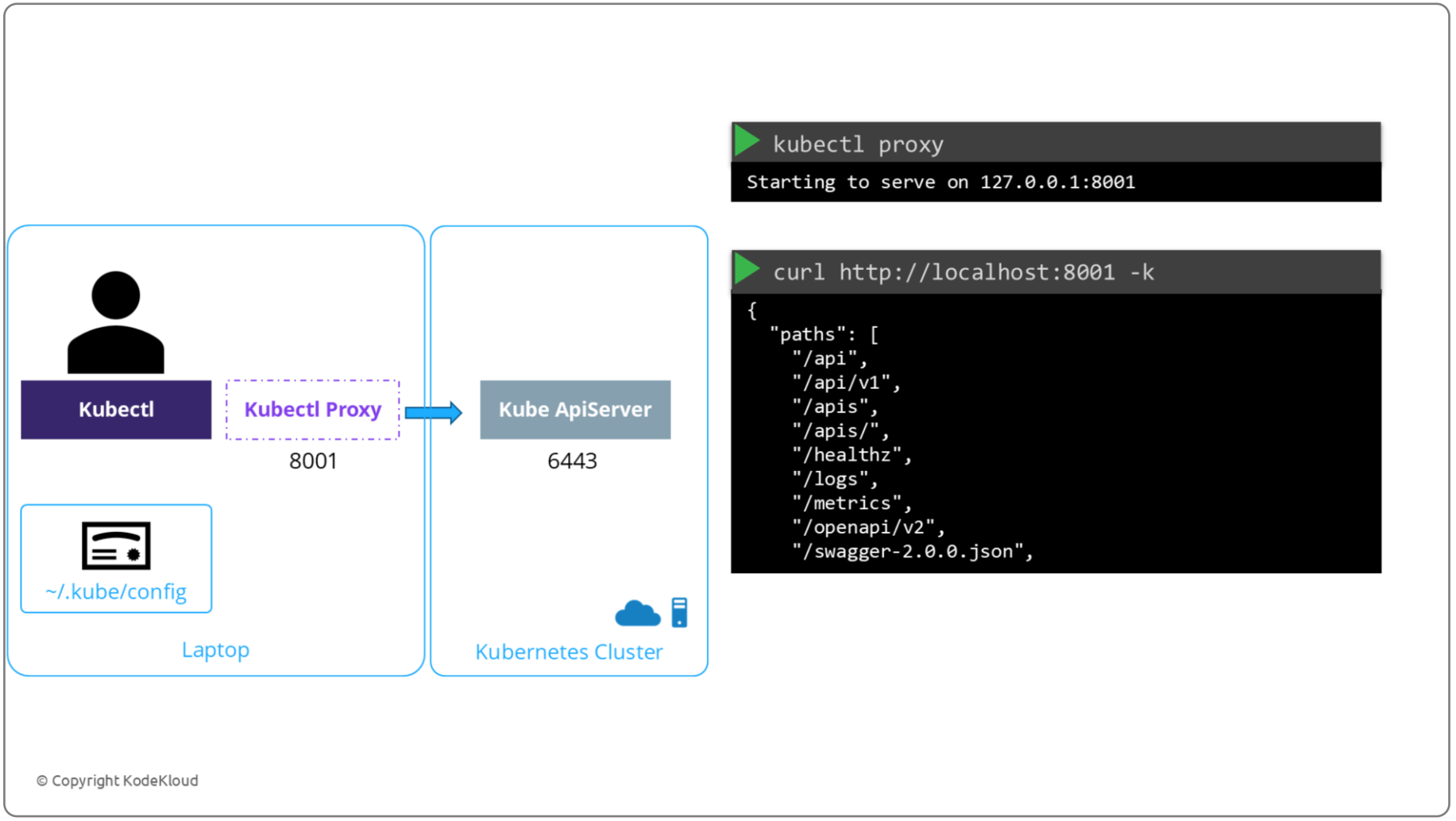

```
curl http://<kube-api-server-ip>:6443 -k
```


```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
}
```


```
curl http://<kube-api-server-ip>:6443 -k
     --key admin.key
     --cert admin.crt
     --cacert ca.crt
```


```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
```


We also discussed that another way to interact with the Api server at port 6443 is using curl. If you were to access the API directly through curl as shown here, then you will not be allowed access, as you have not specified any authentication mechanisms. So you have to authenticate to the API using your certificate files by passing them in the command line like this.

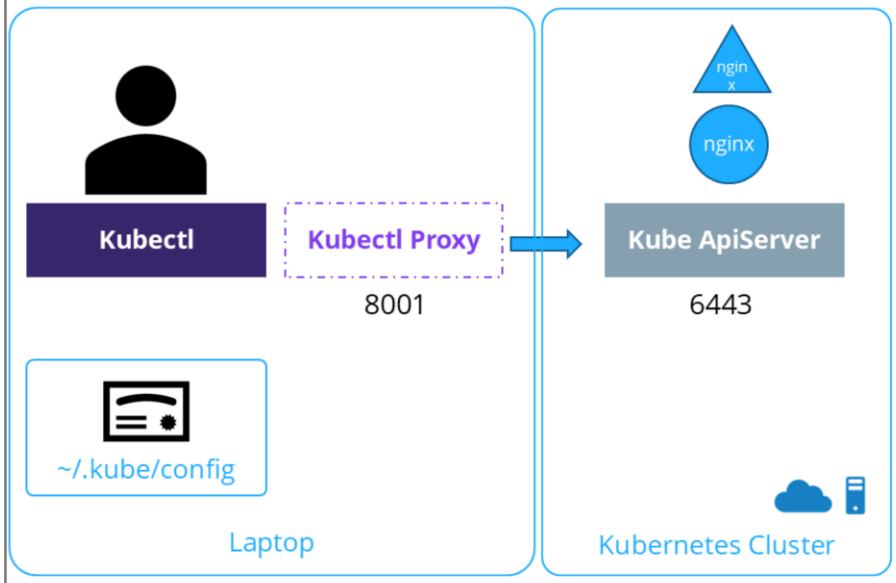


An alternate option is to start a kubectl proxy client. <c> The kubectl proxy command, launches a proxy service locally on port 8001 and uses credentials and certificates from your kubeconfig file to access the cluster. That way you don't have to specify those in the curl command. <c> Now you can access the kubectl proxy service on localhost at port 8001 and the proxy will use the credentials from kube-config file to forward your request to the kube api server. This will list all available APIs at root.

So remember that the proxy runs on your laptop and is only accessible within your laptop. By default it accepts traffic from the loopback address at 127.0.0.1 only. So its not accessible from outside your laptop.

Kubectl Proxy

```
curl http://localhost:8001/api/v1/namespaces/default/services/nginx/proxy/
```



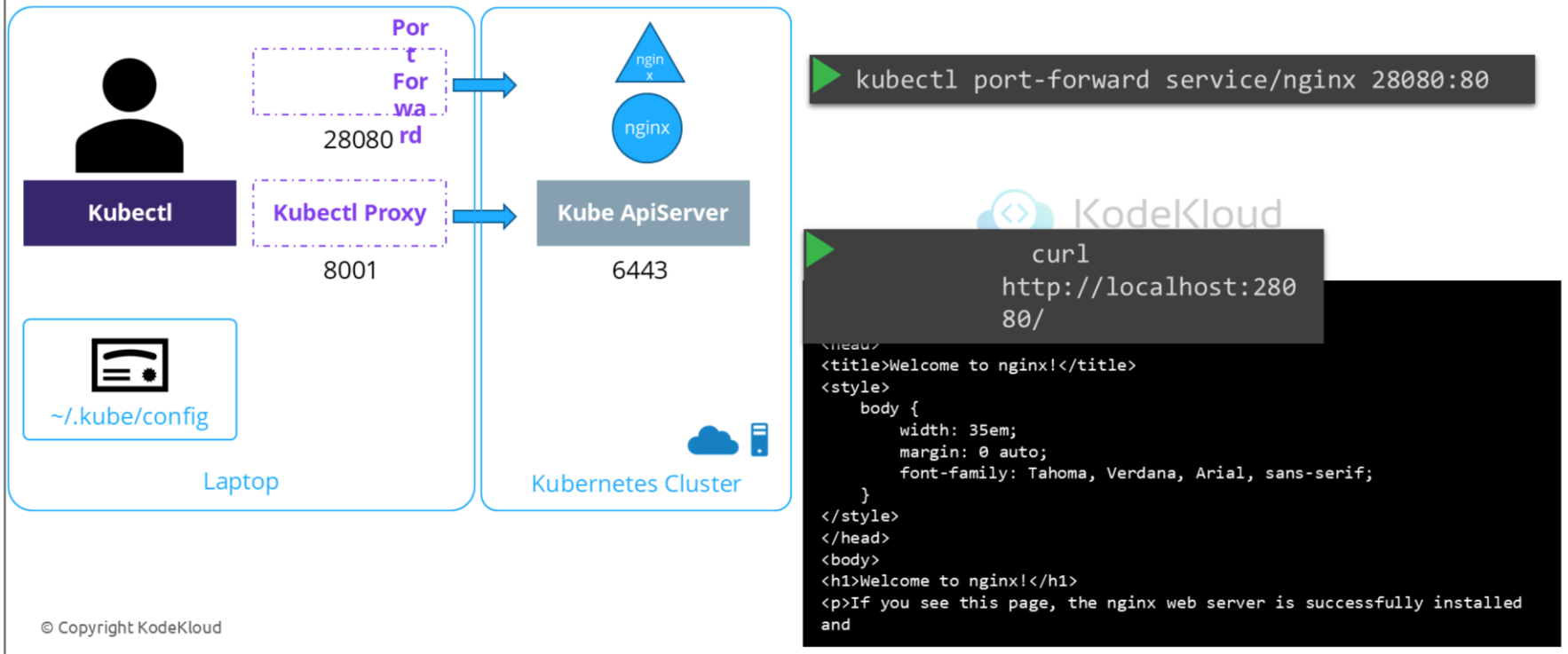

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed
and
working. Further configuration is required.</p>


<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>


<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```


So what all can you do with the kubectl proxy? You can make any kind of API request to the API server with this. And not just that you can proxy requests to any service within the cluster. For example, say we have an nginx pod exposed with an nginx service. But say for some security reason we have not exposed it to the outside world through nodeport or a load balancer. The nginx service is a clsuterip type service that is only accessible within the cluster.

You could form a url through your kubectl proxy at port 8001, that is to the api/v1/namespaces/ and then provide the

namespace the application is in, in this case its in the default namespace, followed by services and the name of the service that you would like to access followed by proxy. This way you will be able to access services hosted within your cluster through the kubectl proxy that’s running on your laptop as if the cluster is running locally on your laptop.

Kubectl Port Forward



Another option to access the service is to configure a port forward. With kubectl you could forward a port from your laptop to a port on a service within the kubernetes cluster. The kubectl port-forward command takes a pod, deployment, replicaset or service as an argument in this case the service nginx is what we want to forward traffic to. And then we specify a port on our host which is 28080 and that is forward to port 80 on the service. Now to access the service running on the remote cluster, we can just go to port 28080 on our localhost.

So any service can be accessed like this.

So these are different ways that we can run a local proxy using kubectl utility to access a service running on a remote kubernetes cluster.

We discuss this approach as it will come in handy in understanding the upcoming lectures.

Security
KUBECONFIG

In this lecture we look at how to manage certificates and what the Certificate API is.

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

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

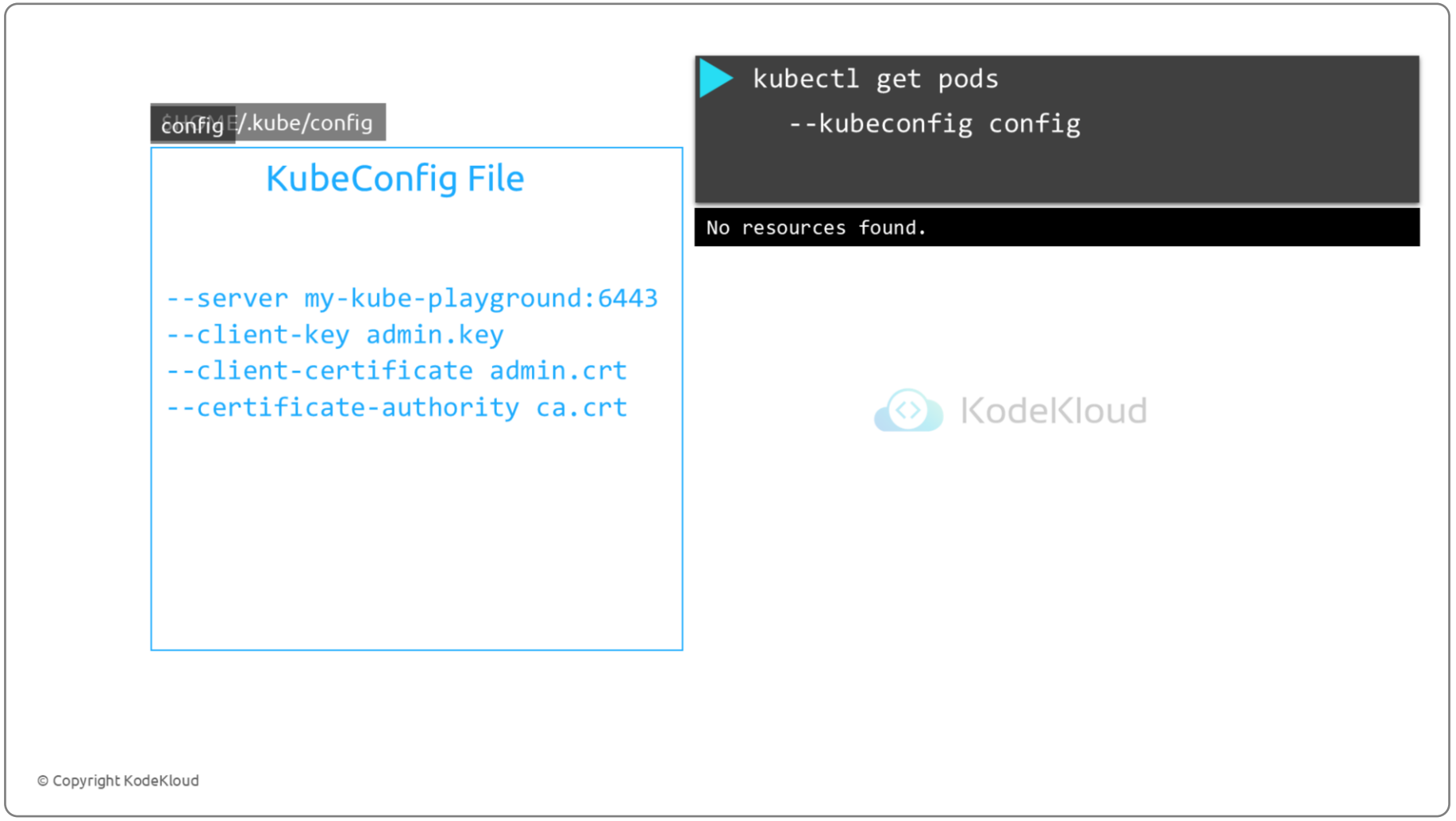
```
kubectl get pods
     --server my-kube-playground:6443
     --client-key admin.key
     --client-certificate admin.crt
     --certificate-authority ca.crt
```

```
No resources found.
```


So far we have seen how to generate a certificate for a user. We have seen how a client uses the certificate file and key to query the kubernetes Rest API for a list of PODs using Curl. In this case my cluster is called my-kube-playground, so send a CURL request to the address of the kube-api server while passing in the pair of files along with the ca certificate as options. This is then validated by the API server to authenticate the user.

Now how do you do that while using the kubectl command? You can specify the same information using the options server,

client-key, client-certificate and certificate-authority with the kubectl utililty.

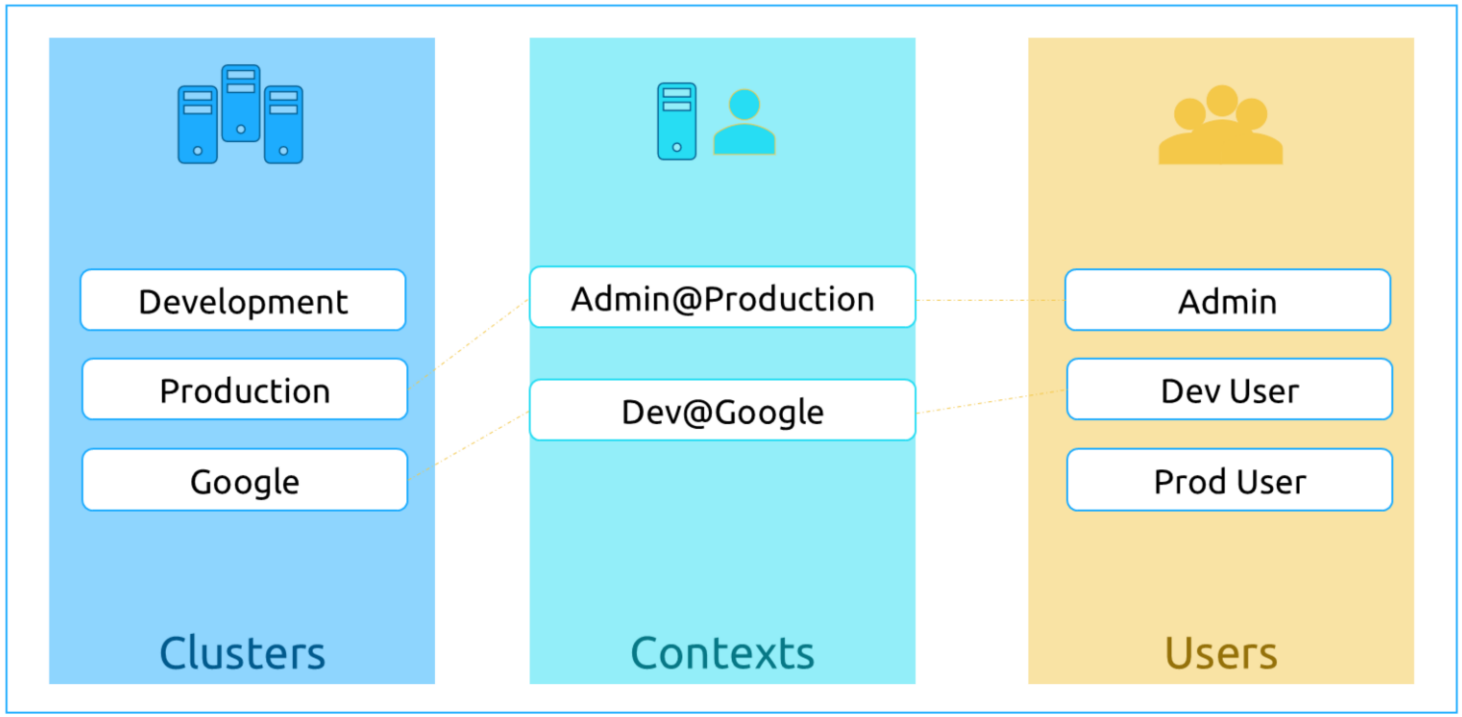


Obviously typing those in every time is a tedious task. So you move these information to a configuration file called as KubeConfig. And then specify this file as the kubeconfig option. By default the kubectl tool looks for a file named config under a directory .kube under the users home directory. So if you create the KubeConfig file there, you don't have to specify the path to the file explicitly in the kubectl command. That's the reason you haven't been specifying any options for your kubectl commands so far.

The kubeconfig file is in a specific format. Let’s take a look at that.

KubeConfig File

$HOME/.kube/config



The config file has 3 sections. Clusters, Users and Contexts.

Clusters are the various kubernetes clusters that you need access to. Say you have a multiple clusters for a Development environment , or testing environment or prod, or for different organizations or on different cloud providers etc. All those go here.

Users are the user accounts with which you have access to these clusters. For example the admin user. A Dev User, a prod user etc. These users may have different privileges on different clusters.

Finally, contexts marry these together. Contexts define which user account will be used to access which cluster. For example you could create a context named Admin@Production that will use the admin account to access a production cluster.

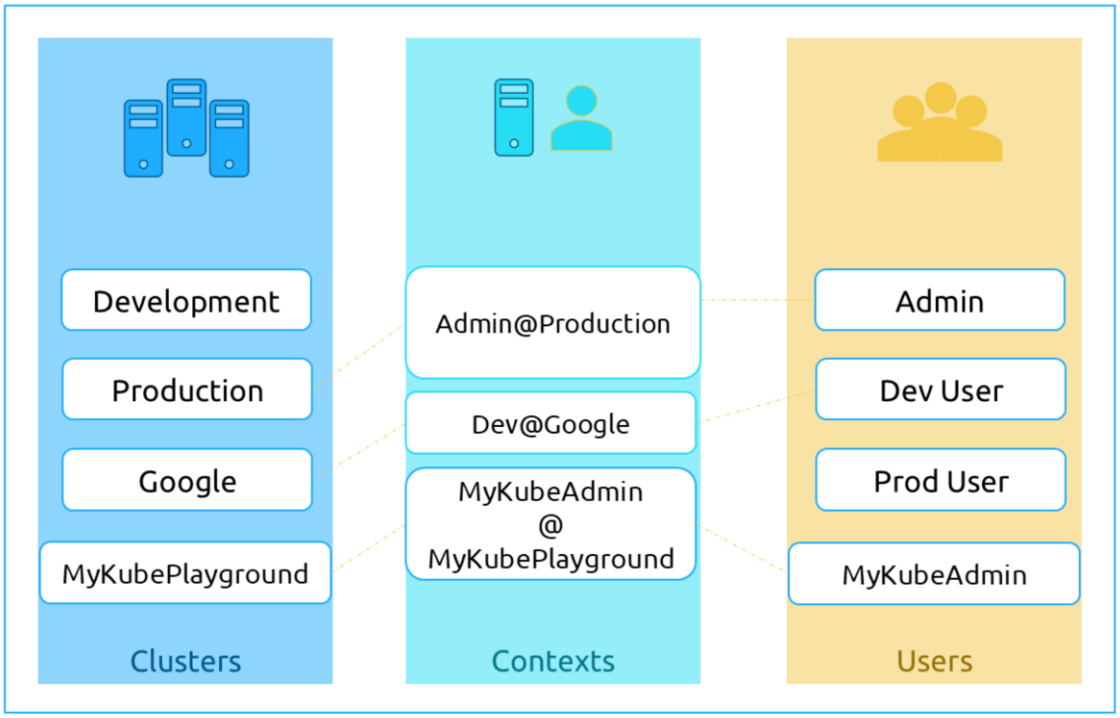
Or I may want to access the cluster I have setup on Google with the devusers credentials to test deploying the application I built.

Remember, you are not creating any new users or configuring any kind of user access or authorization in the cluster. You are using existing users with their existing privileges and defining what user you are going to use to access what cluster. That way you don’t have to specify the user certificates and server address in each and every kubectl command you run.

KubeConfig File

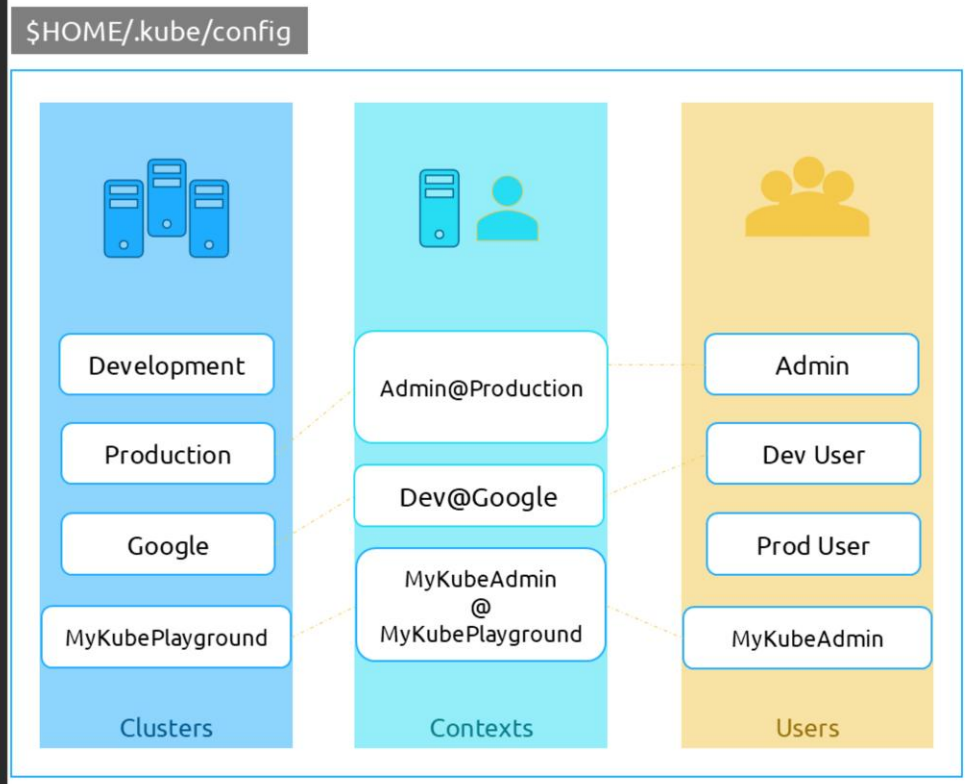
$HOME/.kube/config

```
--server my-kube-playground:6443
--client-key admin.key
--client-certificate admin.crt
--certificate-authority ca.crt
```



So how does it fit into our example? The server specification in our command goes into the clusters section. The admin users keys and certificates goes into the users section. You then create a context that specifies to use the MyKubeAdmin user to access the MyKubePlayground cluster.

```yaml
apiVersion: v1
kind: Config

clusters:
- name: my-kube-playground
  cluster:
    certificate-authority: ca.crt
    server: https://my-kube-playground:6443

contexts:
- name: my-kube-admin@my-kube-playground
  context:
    cluster:
    user:

users:
- name: my-kube-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```



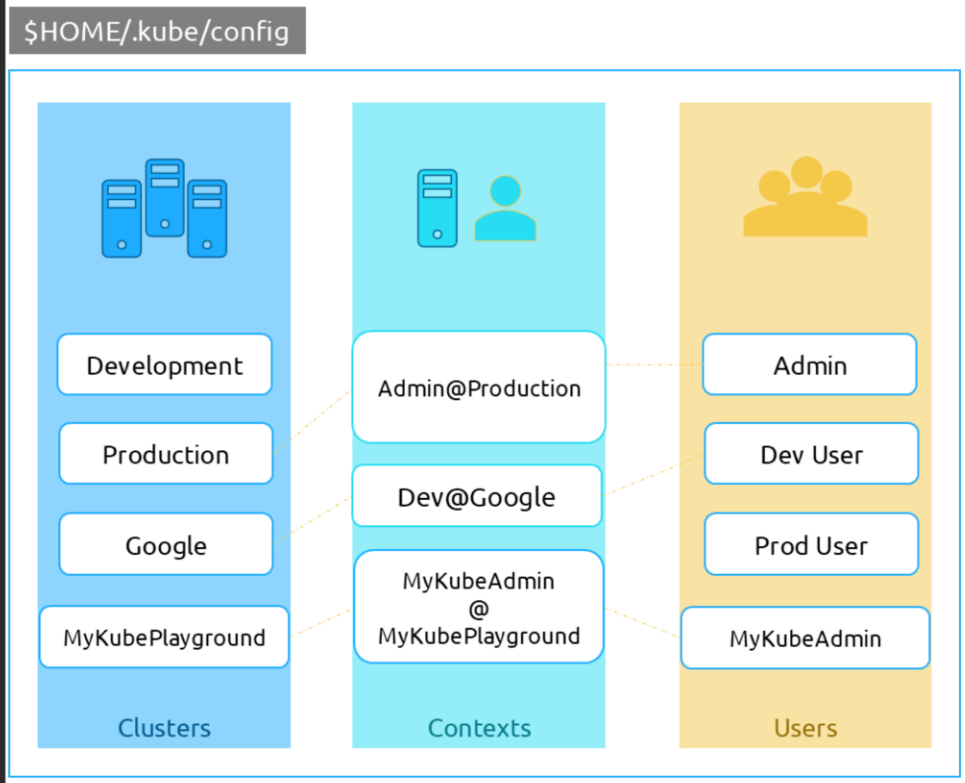
Let's look at a real KubeConfig file now. The kubeConfig is in a YAML format. It has apiVersion set to v1. The kind is config. And then it has 3 sections as we discussed. One for clusters, one for contexts and one for users. Each of these is in an array format. That way you can specify multiple clusters, users or contexts within the same file.

Under clusters we add a new item for our kube-playground cluster. We name it mukubeplayground and specify the server address under the server field. It also requires the certificate of the certificate authority.

We then add an entry into the users section to specify details of my kube admin user. Provide the location of the client's certificate and key pair. So we have now defined the cluster and the user to access the cluster.

Next create an entry under context to link the two together. We will name the context my kube admin @ my kube playground. We will then specify the same name we used for cluster and user.

```
apiVersion: v1
kind: Config
current-context: dev-user@google

clusters:
- name: my-kube-playground   (values hidden...)
- name: development
- name: production
- name: google

contexts:
- name: my-kube-admin@my-kube-playground
- name: dev-user@google
- name: prod-user@production

users:
- name: my-kube-admin
- name: admin
- name: dev-user
- name: prod-user
```



Follow the same procedure to add all the clusters you daily access, the user credentials you use to access them as well as the contexts.

Once the file is ready, remember that you don't have to create any object like you usually do for other kubernetes objects. The file is left as is and is read by the kubectl command and the required values are used.

Now, how does kubectl know which context to chose from? We have 3 defined here. Which one should it start with?

You can specify the default context to use by adding a field current-context to the kubeconfig file. Specify the name of the context to use. In this case kubectl will always use the context dev-user@google to access the google clusters using the dev-user's credentials.

Kubectl config

```
kubectl config view
```

```
apiVersion: v1

kind: Config
current-context: kubernetes-admin@kubernetes

clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://172.17.0.5:6443
  name: kubernetes

contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes


users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

```
kubectl config view –kubeconfig=my-custom-config
```

```
apiVersion: v1

kind: Config
current-context: my-kube-admin@my-kube-playground


clusters:
- name: my-kube-playground
- name: development
- name: production


contexts:
-  name: my-kube-admin@my-kube-playground
-  Name: prod-user@production


users:
- name: my-kube-admin
- name: prod-user
```


There are command line options available within kubectl to view and modify the kubeconfig files. To view the current file being used run the kubectl config view command. It lists the clusters, contexts and users as well as the current-context set.

As we discussed earlier, if you do not specify which kubeconfig file to use, it ends up using the default file located in the folder .kube in users home directory. Alternatively you can specify a kubeconfig file by passing the kubeconfig option in the command line like this.

Kubectl config

```
kubectl config view
```

```
apiVersion: v1

kind: Config
current-context: my-kube-admin@my-kube-playground


clusters:
- name: my-kube-playground
- name: development
- name: production


contexts:
-  name: my-kube-admin@my-kube-playground
-  Name: prod-user@production


users:
- name: my-kube-admin
- name: prod-user
```

```
kubectl config use-context prod-user@production
```

```
apiVersion: v1

kind: Config
current-context: prod-user@production


clusters:
- name: my-kube-playground
- name: development
- name: production


contexts:
-  name: my-kube-admin@my-kube-playground
-  Name: prod-user@production


users:
- name: my-kube-admin
- name: prod-user
```


We will move our custom config to the home directory so this becomes our default config file. So how do you update your current-context? Say you have been using my-kube-admin user to access my-kube-playground. How do you change the context to use prod-user to access the production cluster?

Run the kubectl config use-context command to change the current-context to the prod-user@production context. This change can be seen in the current-context field in the file. So yes the changes made by kubectl config command actually

reflects in the file.

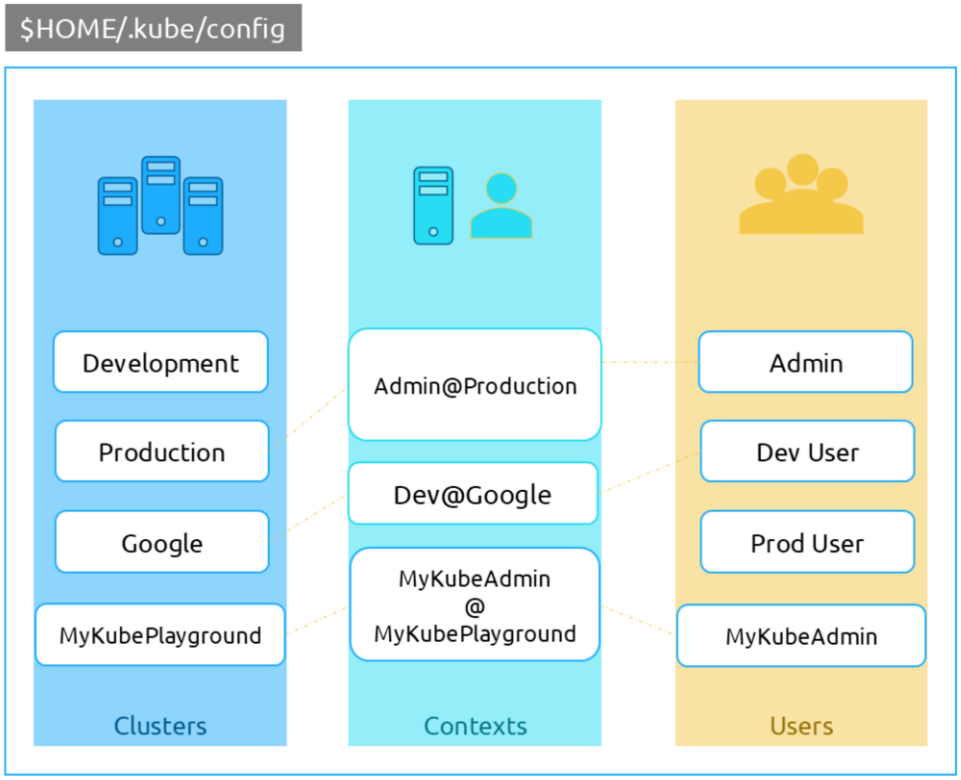
Kubectl config

```
kubectl config -h
```

```
Available Commands:
  current-context Displays the current-context
  delete-cluster  Delete the specified cluster from the kubeconfig
  delete-context  Delete the specified context from the kubeconfig
  get-clusters    Display clusters defined in the kubeconfig
  get-contexts    Describe one or many contexts
  rename-context  Renames a context from the kubeconfig file.
  set             Sets an individual value in a kubeconfig file
  set-cluster     Sets a cluster entry in kubeconfig
  set-context     Sets a context entry in kubeconfig
  set-credentials Sets a user entry in kubeconfig
  unset           Unsets an individual value in a kubeconfig file
  use-context     Sets the current-context in a kubeconfig file
  view            Display merged kubeconfig settings or a specified kubeconfig file
```


You can make other changes in the file, update or delete items in it, using other variations of the kubectl config command. Check them out when you get time.

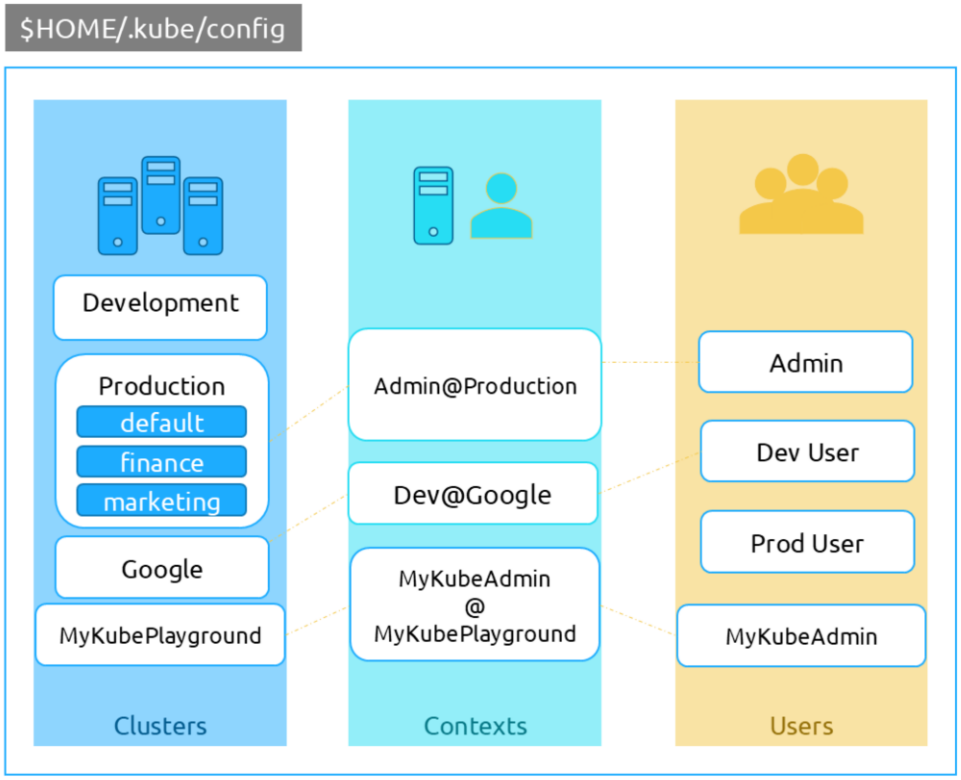
Namespaces



What about namespaces? For example, each cluster may be configured with multiple namespaces within it.

Namespaces

```yaml
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: ca.crt
    server: https://172.17.0.51:6443

contexts:
- name: admin@production
  context:
    cluster: production
    user: admin
    namespace: finance

users:
- name: admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```



Can you configure a context to switch to a particular namespace? Yes! The context section in the kubeconfig file can take additional field called namespace where you can specify a particular namespace. This way, when you switch to that context, you will automatically be in a specific namespace.

Certificates in KubeConfig

```
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    server: https://172.17.0.51:6443

contexts:
- name: admin@production
  context:
    cluster: production
    user: admin
    namespace: finance

users:
- name: admin
  user:
    client-certificate: /etc/kubernetes/pki/users/admin.crt
    client-key: /etc/kubernetes/pki/users/admin.key
```


Finally, a word on certificates. You have seen path to certificate files mentioned in kubeconfig like this. Well, its better to use the full path like this.

Certificates in KubeConfig

```
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt

    certificate-authority-data: LS0tLS1CRUdJTiBDRVJU
                      SUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ
                      1dEQ0NBVUFDQVFBd0V6RVJNQThHQTFVRU
                      F3d0libVYzTFhWelpYSXdnZ0VpTUEwR0N
                      TcUdTSWIzRFFFQgpBUVVBQTRJQkR3QXdn
                      Z0VLQW9JQkFRRE8wV0pXK0RYc0FKU0lya
                      nBObzV2UklCcGxuemcrNnhjOStVVndrS2
                      kwCkxmQzI3dCsxZUVuT041TXVxOTlOZXZ
                      tTUVPbnJ
```

```
-----BEGIN CERTIFICATE -----
MIICWDCCAUACAQAwEzERMA8GA1UEAwwIbmV3LXVzZXIwggEiMA0GC
AQUAA4IBDwAwggEKAoIBAQDO0WJW+DXsAJSIrjpNo5vRIBplnzg+6
LfC27t+1eEnON5Muq99NevmMEOnrDUO/thyVqP2w2XNIDRXjYyF40
y3BihhB93MJ7Oql3UTvZ8TELqyaDknRl/jv/SxgXkok0ABUTpWMx4
IF5nxAttMVkDPQ7NbeZRG43b+QWlVGR/z6DWOfJnbfezOtaAydGLT
EcCXAwqChjBLkz2BHPR4J89D6Xb8k39pu6jpyngV6uP0tIbOzpqNv
j2qEL+hZEWkkFz80lNNtyT5LxMqENDCnIgwC4GZiRGbrAgMBAAGgA
9w0BAQsFAAOCAQEAS9iS6C1uxTuf5BBYSU7QFQHUzalNxAdYsaORR
hOK4a2zyNyi44OOijyaD6tUW8DSxkr8BLK8Kg3srREtJql5rLZy9L
P9NL+aDRSxROVSqBaB2nWeYpM5cJ5TF53lesNSNMLQ2++RMnjDQJ7
Wr2EUM6UawzykrdHImwTv2mlMY0R+DNtV1Yie+0H9/YElt+FSGjh5
4l3E/y3qL71WfAcuH3OsVpUUnQISMdQs0qWCsbE56CC5DhPGZIpUb
vwQ07jG+hpknxmuFAeXxgUwodALaJ7ju/TDIcw==
-----END CERTIFICATE -----
```

```
cat ca.crt | base64
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNU
tLS0KTUlJQ1dEQ0NBVUFDQVFBd0V6RVJNQThHQTF
F3d0libVYzTFhWelpYSXdnZ0VpTUEwR0NTcUdTSW
FFFQgpBUVVBQTRJQkR3QXdnZ0VLQW9JQkFRRE8wV
K0RYc0FKU0lyanBObzV2UklCcGxuemcrNnhjOStV
rS2kwCkxmQzI3dCsxZUVuT041TXVxOTlOZXZtTUV
J
```

But remember there is also another way to specify the certificate credentials. Let's look at the first one for instance where we configure the path to the certificate authority. We have the contents of the ca.crt file on the right. Instead of using the certificate-authority field and the path to the file, you may optionally use the certificate-authority-data field and provide the contents of the certificate itself. But not the file as is, convert the contents to a base64 encoded format and then pass that in. Similarly if you see a file with the certificates data in the encoded format, use the base64 --decode option to decode the certificate.

Certificates in KubeConfig

```
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt

    certificate-authority-data: LS0tLS1CRUdJTiBDRVJU
                      SUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ
                      1dEQ0NBVUFDQVFBd0V6RVJNQThHQTFVRU
                      F3d0libVYzTFhWelpYSXdnZ0VpTUEwR0N
                      TcUdTSWIzRFFFQgpBUVVBQTRJQkR3QXdn
                      Z0VLQW9JQkFRRE8wV0pXK0RYc0FKU0lya
                      nBObzV2UklCcGxuemcrNnhjOStVVndrS2
                      kwCkxmQzI3dCsxZUVuT041TXVxOTlOZXZ
                      tTUVPbnJ
```

```
echo "LS0…bnJ" | base64 --decode
```

```
-----BEGIN CERTIFICATE -----
MIICWDCCAUACAQAwEzERMA8GA1UEAwwIbmV3LXVzZXIwggEiMA0GCSq
AQUAA4IBDwAwggEKAoIBAQDO0WJW+DXsAJSIrjpNo5vRIBplnzg+6xc
LfC27t+1eEnON5Muq99NevmMEOnrDUO/thyVqP2w2XNIDRXjYyF40Fb
y3BihhB93MJ7Oql3UTvZ8TELqyaDknRl/jv/SxgXkok0ABUTpWMx4Bp
IF5nxAttMVkDPQ7NbeZRG43b+QWlVGR/z6DWOfJnbfezOtaAydGLTZF
EcCXAwqChjBLkz2BHPR4J89D6Xb8k39pu6jpyngV6uP0tIbOzpqNv0Y
j2qEL+hZEWkkFz80lNNtyT5LxMqENDCnIgwC4GZiRGbrAgMBAAGgADA
9w0BAQsFAAOCAQEAS9iS6C1uxTuf5BBYSU7QFQHUzalNxAdYsaORRQN
hOK4a2zyNyi44OOijyaD6tUW8DSxkr8BLK8Kg3srREtJql5rLZy9LRV
P9NL+aDRSxROVSqBaB2nWeYpM5cJ5TF53lesNSNMLQ2++RMnjDQJ7ju
Wr2EUM6UawzykrdHImwTv2mlMY0R+DNtV1Yie+0H9/YElt+FSGjh5L5
4l3E/y3qL71WfAcuH3OsVpUUnQISMdQs0qWCsbE56CC5DhPGZIpUbnK
vwQ07jG+hpknxmuFAeXxgUwodALaJ7ju/TDIcw==
-----END CERTIFICATE -----
```

Similarly if you see a file with the certificates data in the encoded format, use the base64 --decode option to decode the certificate.

Well that's it for this lecture. Head over to the practice exercises section and practice working with kubeconfig files and troubleshooting issues.

Reference

https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/

KodeKloud

Visit www.kodekloud.com to learn more.