K8s Security

🔐 What is SSL/TLS?

🔸 SSL = Secure Sockets Layer

🔸 TLS = Transport Layer Security

- **SSL (Secure Sockets Layer)** and its modern version **TLS (Transport Layer Security)** are **security protocols.**
- They ensure that **data transferred between a client and a server is encrypted** and **safe from hackers.**
- They are the reason why websites use `https://` instead of `http://` .
- `https://` means the site is **secure and encrypted**, while `http://` is **not secure.**

TLS is the **newer and more secure version** of SSL.

They are **protocols used to encrypt data** sent between:

- Our browser (client)
- A website (server)

🌐 Why Do We Need It?

Imagine this:

👤 **User (Client)** → sends a request via HTTP → 🖥️ **Server**

Let's say we're on a public Wi-Fi (like in a coffee shop) and type our **bank password** on a website.

- If the site is **http://**, our data (password) goes in **plain text.** A hacker can **see it.**
- If the site is **https://**, our data is **encrypted** using TLS. A hacker **cannot read it.**

That's why most sites now use **https://**.

If it's just **HTTP**, the data (like passwords, bank info) travels **in plain text.**
💀 A **hacker** in between can intercept and read everything.

🔐 Symmetric Encryption (Why It's Not Safe Alone):

1️⃣ User Sends Request to Server:

- When a user wants to connect to a server (like logging into a website), the server may ask for identity verification.
- This is like showing our ID card at the entrance—it proves who we are before giving access.

2️⃣ User Sends Encrypted Data + Key:

- In symmetric encryption, both parties use the **same key** to encrypt and decrypt data.
- So the user must send their encrypted data **along with the secret key** to the server.

3️⃣ Server Uses the Key to Decrypt:

- The server uses this same key to decrypt the user's data.

4️⃣ ❌ Why It's Not Safe:

- While the data and key are being sent over the internet, a hacker can intercept both.
- This is like giving your locker key and our locker contents together—if someone catches it mid-way, they get full access.
- **Result:** Both our message and the secret key are exposed → the entire system is broken.

🛡️ TLS/SSL (Asymmetric Encryption)

TLS (Transport Layer Security) uses **asymmetric encryption**, which is more secure than symmetric. It works using **two keys**:

- 🔓 **Public Key**—Shared with anyone (user/client)
- 🔐 **Private Key**—Kept secret by the server

Here's how the secure connection is built:

1️⃣ User Sends Request to the Server:

- When we visit a secure website (like `https://example.com`), our browser sends a request to connect.

2️⃣ Server Creates a CSR (Certificate Signing Request):

- The server generates a **CSR**, which includes its **public key** and identity info (like domain name, organization, location).

3️⃣ CSR is Sent to CA (Certificate Authority):

- The server sends this CSR to a trusted **Certificate Authority (CA)**.

- The CA verifies the server's identity and then **digitally signs** the certificate.

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl certificate deny admin
certificatesigningrequest.certificates.k8s.io/admin denied
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get csr
NAME    AGE   SIGNERNAME                            REQUESTOR          REQUESTEDDURATION   CONDITION
admin   57s   kubernetes.io/kube-apiserver-client   kubernetes-admin   24h                 Denied
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl certificate approve admin
error: certificate signing request "admin" is already Denied
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get csr
NAME    AGE   SIGNERNAME                            REQUESTOR          REQUESTEDDURATION   CONDITION
admin   12s   kubernetes.io/kube-apiserver-client   kubernetes-admin   24h                 Pending
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl certificate approve admin
certificatesigningrequest.certificates.k8s.io/admin approved
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get csr
NAME    AGE   SIGNERNAME                            REQUESTOR          REQUESTEDDURATION   CONDITION
admin   33s   kubernetes.io/kube-apiserver-client   kubernetes-admin   24h                 Approved,Issued
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

4️⃣ Signed Certificate is Sent to the User:

- The server sends this **signed certificate** (includes the public key and CA's signature) to the user's browser.
- This tells the browser: "I'm a legit website, and here's proof from a trusted authority."

5️⃣ User Uses Public Key for Encryption:

- Now the user encrypts a **temporary symmetric key** using the **server's public key** and sends it back.

6️⃣ Only Server Can Decrypt It:

- Since only the server has the **private key**, only it can decrypt the symmetric key.
- From now on, both user and server use this symmetric key for faster secure communication.

✅ **Result:** Even if a hacker sees the public key, they **can't decrypt** anything without the private key.

🔁 What About User Authentication?

Until now, we talked about **server authentication** (browser verifies server).

But sometimes, the **server also needs to verify the user's identity**.

Two Types:

1. **Public Websites (like Amazon):**

- User authentication happens **later**, like when we **log in** with username and password.

🌐 Real-Life Example (Putting it All Together)

Let's say we open `https://www.amazon.com` :

1. Our browser sends a request to Amazon.
2. Amazon sends a certificate to prove it's really Amazon.
3. Our browser checks the certificate.
4. If valid, our browser generates a session key.
5. Encrypts it using Amazon's public key.
6. Amazon decrypts it using its private key.
7. Now both share a session key.
8. All further data (login, payment, etc.) is encrypted.

Done securely. No hacker can see our password or card details.

2. Private Systems (like internal company tools or APIs):

In internal/private environments (such as enterprise applications, DevOps pipelines, or internal APIs), companies often want **stronger user authentication**—not just passwords.

🔐 **Mutual TLS (mTLS)** is used for this.

🔄 What is Mutual TLS?

Normally, in regular TLS:

- The **server** proves its identity to the **client** (like our browser).
- The **client** is anonymous (it just trusts the server).

But in **Mutual TLS**:

- Both **server and client** must prove their identity to each other using **certificates**.

🛠️ How does it work in private systems?

1️⃣ Custom CA (Certificate Authority):

- Companies set up their **own internal CA** to issue client certificates
- This CA is **trusted only inside the organization**.

2️⃣ User/Client gets a certificate:

- Each user or system is given a **client certificate** signed by the internal CA.
- This certificate acts like a **digital ID**.

3️⃣ **During TLS handshake:**

- The **server requests** a client certificate.
- The **client sends** its certificate.
- The server checks the certificate using the **custom CA.**

4️⃣ If it's valid, access is granted.

- This proves the user/system is trusted (not just using a password).

📌 **So yes:**

> "User authentication" happens via certificates in private systems, and a **custom CA** is used to issue and verify those certs.

This makes the system **highly secure**—even if someone knows our password, they **can't connect without the valid certificate.**

👥 Scenario

We are already an **admin** of a Kubernetes cluster.

Now, a **new admin** (say some x person) joins the team.

We want to give them access **using certificate-based authentication.**

🔐 Step-by-Step: Client Certificate Authentication for New Kubernetes User

✅ Step 1: New User Generates Private Key and CSR

This step is done **on the new user's local machine** (or securely managed environment).

```
# 1. Generate a private key (3072-bit RSA)
openssl genrsa -out admin.key 3072

# 2. Create a CSR (Certificate Signing Request)
openssl req -new -key admin.key -out admin.csr -
```

- `CN=admin` → Common Name (the username)
- `O=system:masters` → Group (must be `system:masters` for full admin privileges)

✅ Step 2: Admin (We) Approve the Request in the Cluster

We now take the `.csr` file from the user and **generate a certificate signed by the Kubernetes cluster CA.**

🛠️ Method 1: Using Kubernetes CA Directly

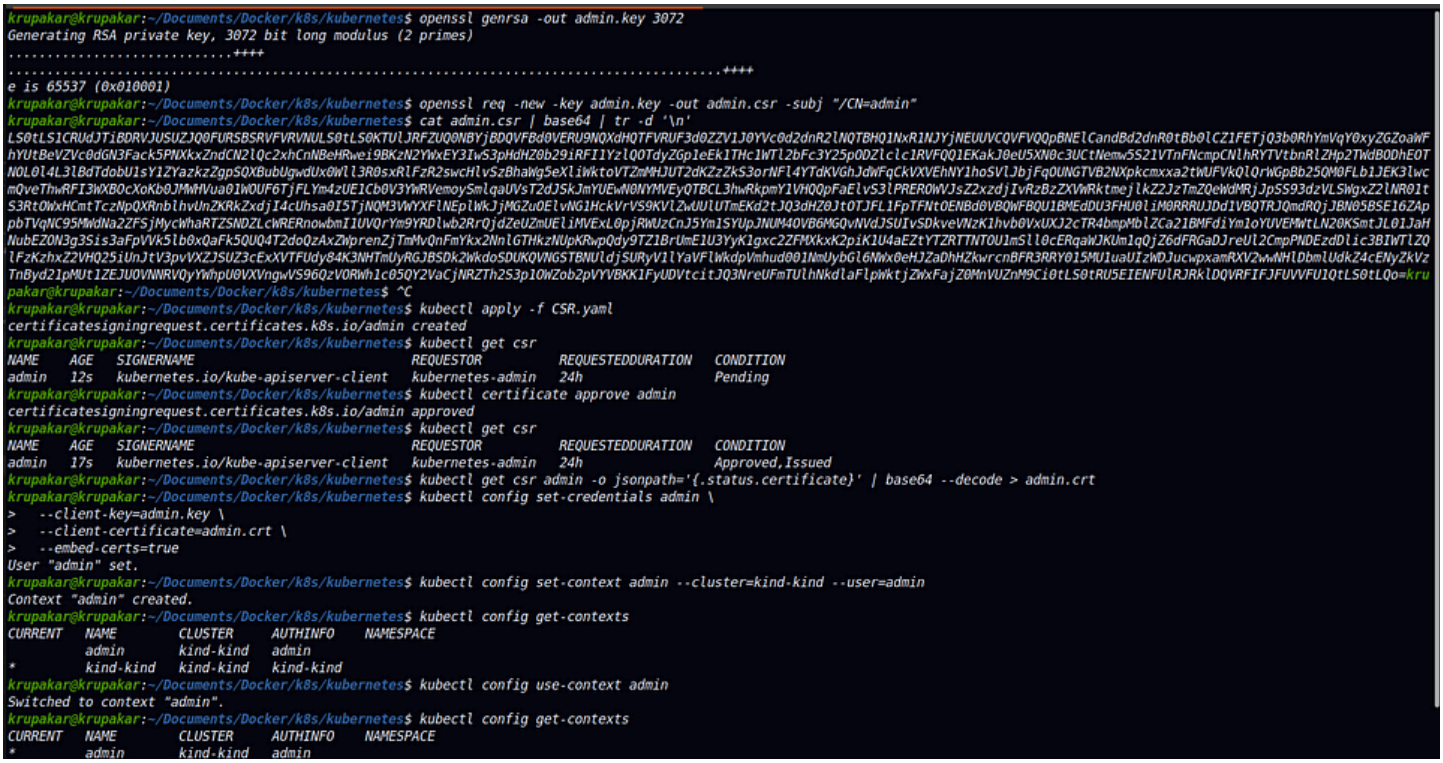
This works **only if we have access to the Kubernetes CA key and cert** (usually available on control plane node under `/etc/kubernetes/pki`):

```
root@kind-control-plane:/# cd /etc/kubernetes/pki/
root@kind-control-plane:/etc/kubernetes/pki# ls -lrt
total 60
-rw------- 1 root root 1675 Jul 17 13:03 ca.key
-rw-r--r-- 1 root root 1107 Jul 17 13:03 ca.crt
-rw------- 1 root root 1679 Jul 17 13:03 apiserver-kubelet-client.key
-rw-r--r-- 1 root root 1176 Jul 17 13:03 apiserver-kubelet-client.crt
-rw------- 1 root root 1675 Jul 17 13:03 front-proxy-ca.key
-rw-r--r-- 1 root root 1123 Jul 17 13:03 front-proxy-ca.crt
-rw------- 1 root root 1675 Jul 17 13:03 front-proxy-client.key
-rw-r--r-- 1 root root 1119 Jul 17 13:03 front-proxy-client.crt
drwxr-xr-x 2 root root 4096 Jul 17 13:03 etcd
-rw------- 1 root root 1679 Jul 17 13:03 apiserver-etcd-client.key
-rw-r--r-- 1 root root 1123 Jul 17 13:03 apiserver-etcd-client.crt
-rw------- 1 root root  451 Jul 17 13:03 sa.pub
-rw------- 1 root root 1675 Jul 17 13:03 sa.key
-rw------- 1 root root 1675 Jul 18 11:33 apiserver.key
-rw-r--r-- 1 root root 1326 Jul 18 11:33 apiserver.crt
root@kind-control-plane:/etc/kubernetes/pki#
```

```
# Sign the CSR using the Kubernetes CA
openssl x509 -req -in admin.csr -CA /etc/kuberne
  -CAkey /etc/kubernetes/pki/ca.key -CAcreateser
```

This creates:

- ✅ `admin.crt` (user's client certificate)
- 🔑 `admin.key` (user's private key)



📥 Extracted the Signed Certificate:

```
kubectl get csr admin -o jsonpath='{.status.cert
```

This command decodes the signed certificate (`admin.crt`) for the user `admin` after the CSR was approved.

✅ Step 3: Set Up kubeconfig File for the New User

Set User Credentials in Kubeconfig:

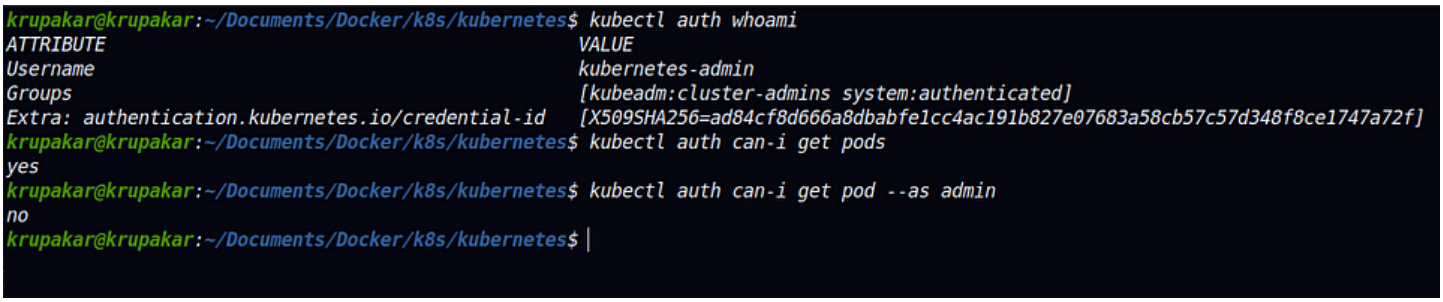
Now we configure a **kubeconfig file** for `admin` with their credentials

```
kubectl config set-credentials admin\
  --client-key=admin.key \
  --client-certificate=admin.crt \
  --embed-certs=true
```

- We configured kubeconfig with the user's credentials (private key and signed certificate).
- `--embed-certs=true` embeds the actual certificate and key data inside the kubeconfig file—good for portability.

✅ What's Next? (Step 4: Authorization)

Now that the `admin` user is **authenticated**, Kubernetes recognizes them—but they **still can't access cluster resources** until they are **authorized**.



🔐 Authentication vs Authorization in Kubernetes

Authentication = Identity Check

- **In Kubernetes:** The user presents credentials (like a client certificate) to prove **who they are.**

- **Analogy:** We show our **ID card** at the office entrance. Security confirms who we are .

Authorization = Access Control

- **In Kubernetes:** Once our identity is confirmed, **RBAC** rules determine **what we're allowed to do.**
- **Analogy:** After entering the building, we're told **which room/hall** to go to (e.g., training room) and **which areas we're not allowed to enter** (e.g., other departments).

In our example:

- The **new employee is authenticated** when their identity is verified at the entrance.
- The **new employee is authorized** when the company decides **what rooms or resources they can access.**

```
krupakar@krupakar:~/.kube$ cd ..
krupakar@krupakar:~$ cd $HOME/.kube
krupakar@krupakar:~/.kube$ ls -lrt
total 12
drwxr-x--- 4 krupakar krupakar 4096 Apr 30 22:30 cache
-rw------- 1 krupakar krupakar 5602 Jul 17 18:33 config
krupakar@krupakar:~/.kube$
```

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl config use-context admin
Switched to context "admin".
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://127.0.0.1:34477
  name: kind-kind
contexts:
- context:
    cluster: kind-kind
    user: admin
  name: admin
- context:
    cluster: kind-kind
    user: kind-kind
  name: kind-kind
current-context: admin
kind: Config
preferences: {}
users:
- name: admin
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
- name: kind-kind
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

The user entry gets recorded in the config file

> After the user is authenticated (their identity is verified), Kubernetes needs to **authorize** what they can do—like viewing pods, deleting deployments, etc.

There are 4 main **authorization modes** in Kubernetes:

We should now assign RBAC permissions:

🔹 1. RBAC (Role-Based Access Control)

- **Most commonly used mode.**
- Access is given based on **roles** assigned to **users or groups.**
- Defined using:
- `Role`, `ClusterRole`

- `RoleBinding`, `ClusterRoleBinding`

📌 **Use case:** "Give read-only access to a user in a namespace."

🔹 2. ABAC (Attribute-Based Access Control)

- Access is granted based on a **policy file** that contains rules with user attributes.
- **Not dynamic** like RBAC; needs changes in static files.
- Rarely used in production now.

📌 **Use case:** Older systems with complex conditions or where RBAC is not sufficient.

🔹 3. Node Authorization

- Used **only by kubelets (nodes).**
- Grants permissions to nodes (e.g., reading secrets or pod info **of its own pods**).
- Prevents nodes from accessing other node's data.

📌 **Use case:** Allow a node to read its own pod info but **not other nodes' data.**

🔹 4. Webhook Authorization

- Kubernetes sends a request to an **external service** (like a custom API).
- The external service **decides** whether to allow the request.
- Used in **custom or enterprise** setups.

📌 **Use case:** “Allow access only during office hours”—logic handled outside Kubernetes.

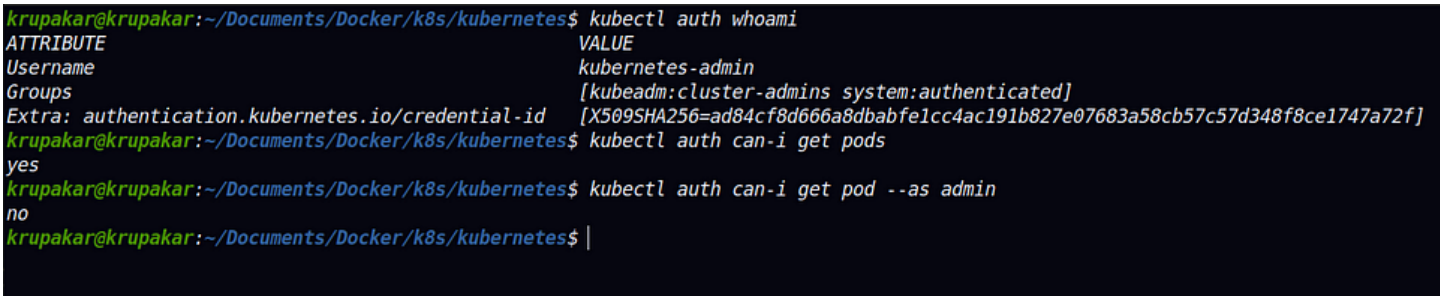
Role, RoleBinding, and ClusterRoleBinding

🔸 What is a Role?

A `Role` defines **permissions** (what actions are allowed) **within a specific namespace.**

- 🔧 It lists what verbs (get, list, create, delete, etc.) a user can perform on which Kubernetes resources (pods, secrets, configmaps, etc.).
- 📦 **Namespace-scoped**—it only applies to one namespace.

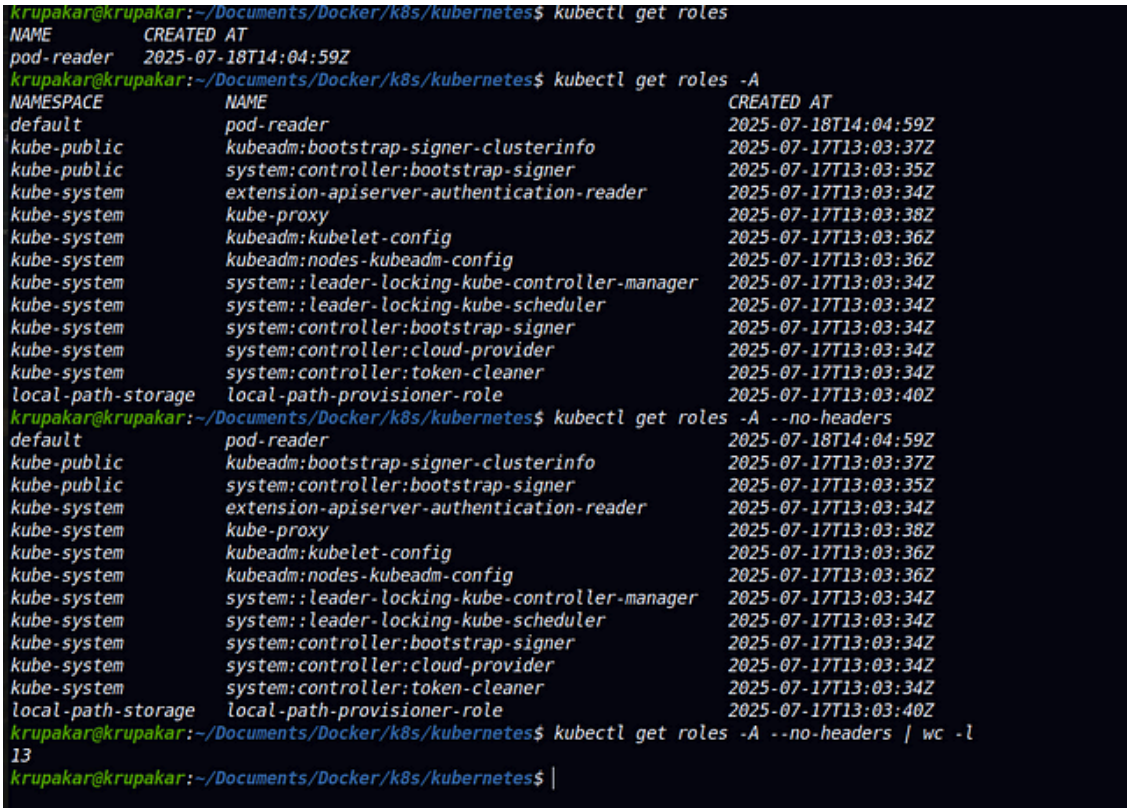
> Think of this like assigning permissions **within one department** (e.g., Sales team folder or Development namespace).



```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl api-resources --namespaced=true
NAME                        SHORTNAMES   APIVERSION                     NAMESPACED   KIND
bindings                                 v1                             true         Binding
configmaps                  cm           v1                             true         ConfigMap
endpoints                   ep           v1                             true         Endpoints
events                      ev           v1                             true         Event
limitranges                 limits       v1                             true         LimitRange
persistentvolumeclaims      pvc          v1                             true         PersistentVolumeClaim
pods                        po           v1                             true         Pod
podtemplates                             v1                             true         PodTemplate
replicationcontrollers      rc           v1                             true         ReplicationController
resourcequotas              quota        v1                             true         ResourceQuota
secrets                                  v1                             true         Secret
serviceaccounts             sa           v1                             true         ServiceAccount
services                    svc          v1                             true         Service
controllerrevisions                      apps/v1                        true         ControllerRevision
daemonsets                  ds           apps/v1                        true         DaemonSet
deployments                 deploy       apps/v1                        true         Deployment
replicasets                 rs           apps/v1                        true         ReplicaSet
statefulsets                sts          apps/v1                        true         StatefulSet
localsubjectaccessreviews                authorization.k8s.io/v1        true         LocalSubjectAccessReview
horizontalpodautoscalers    hpa          autoscaling/v2                 true         HorizontalPodAutoscaler
cronjobs                    cj           batch/v1                       true         CronJob
jobs                                     batch/v1                       true         Job
leases                                   coordination.k8s.io/v1         true         Lease
endpointslices                           discovery.k8s.io/v1            true         EndpointSlice
events                      ev           events.k8s.io/v1               true         Event
ingresses                   ing          networking.k8s.io/v1           true         Ingress
networkpolicies             netpol       networking.k8s.io/v1           true         NetworkPolicy
poddisruptionbudgets        pdb          policy/v1                      true         PodDisruptionBudget
rolebindings                             rbac.authorization.k8s.io/v1   true         RoleBinding
roles                                    rbac.authorization.k8s.io/v1   true         Role
csistoragecapacities                     storage.k8s.io/v1              true         CSIStorageCapacity
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

Namespace-scoped

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ code .
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ cat role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl apply -f role.yaml
role.rbac.authorization.k8s.io/pod-reader created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get role
NAME         CREATED AT
pod-reader   2025-07-18T14:04:59Z
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl describe role pod-reader
Name:         pod-reader
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  pods       []                 []              [get watch list]
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

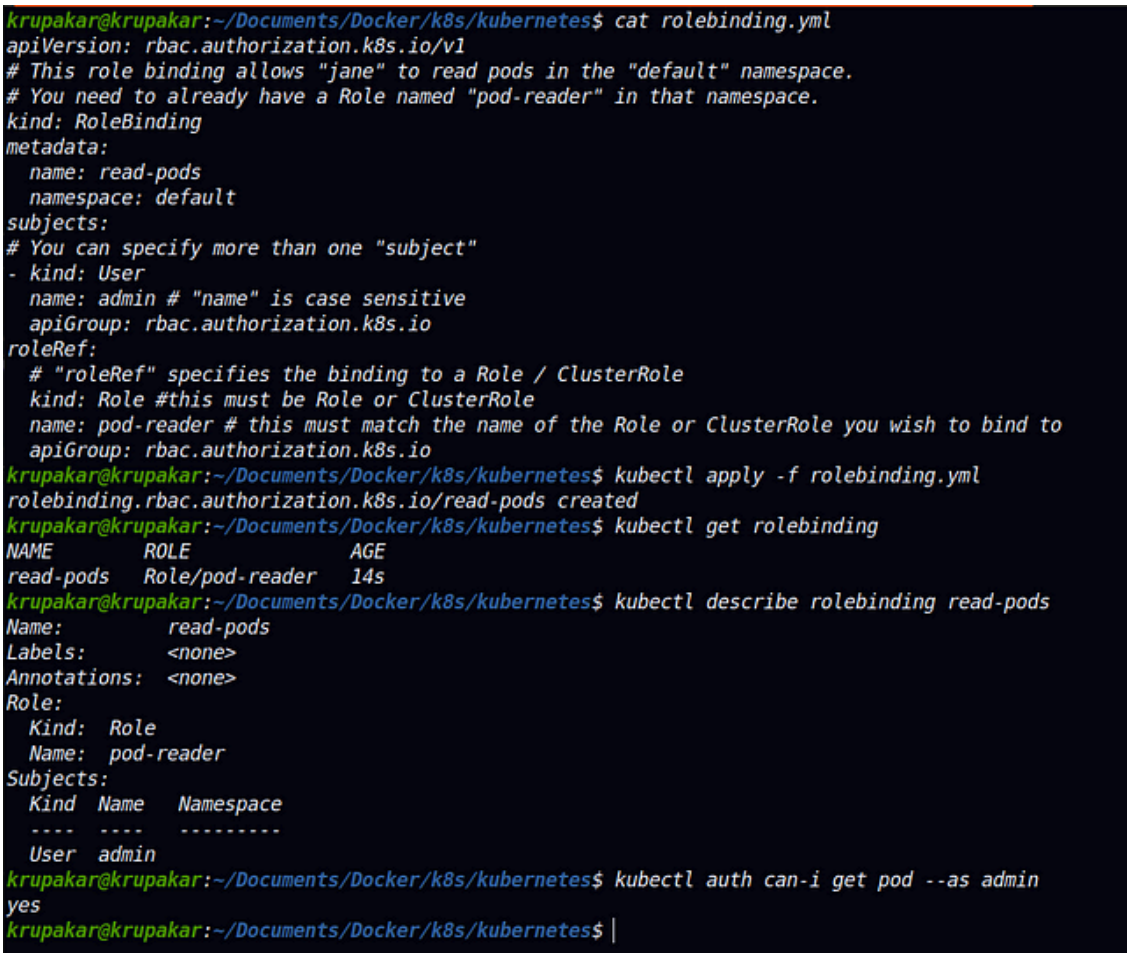


🔸 What is a RoleBinding?

A `RoleBinding` connects a **user/group/serviceaccount** to a **Role**—essentially saying "this user can have these permissions" in that namespace.

- 🤝 Binds a Role to a user/group **within a namespace.**

> This is like saying: “Assign this employee (user) to the **Sales department** with these specific duties.”



🔸 What is a ClusterRole?

A `ClusterRole` defines permissions like a Role, **but it can be applied cluster-wide** (across all namespaces).

- It can also define **non-namespaced resources** (like nodes, persistent volumes).
- 💡 It can be used in two ways:

1. Grant access to resources across all namespaces.
2. Grant access to cluster-scoped resources.

> Think of this like assigning access to **entire company-wide systems** (like HR software, internal email systems, etc.)

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl auth can-i get nodes
Warning: resource 'nodes' is not namespace scoped

yes
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl auth can-i get nodes --as admin
Warning: resource 'nodes' is not namespace scoped

no
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl create clusterrole node-reader --verb=get,list,watch --resource=node
clusterrole.rbac.authorization.k8s.io/node-reader created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get clusterrole
NAME                                                                   CREATED AT
admin                                                                  2025-07-17T13:03:28Z
cluster-admin                                                          2025-07-17T13:03:28Z
edit                                                                   2025-07-17T13:03:28Z
kindnet                                                                2025-07-17T13:03:39Z
kubeadm:get-nodes                                                      2025-07-17T13:03:36Z
local-path-provisioner-role                                            2025-07-17T13:03:40Z
node-reader                                                            2025-07-18T17:14:30Z
system:aggregate-to-admin                                              2025-07-17T13:03:28Z
system:aggregate-to-edit                                               2025-07-17T13:03:28Z
system:aggregate-to-view                                               2025-07-17T13:03:28Z
system:auth-delegator                                                  2025-07-17T13:03:28Z
system:basic-user                                                      2025-07-17T13:03:28Z
system:certificates.k8s.io:certificatesigningrequests:nodeclient       2025-07-17T13:03:29Z
system:certificates.k8s.io:certificatesigningrequests:selfnodeclient   2025-07-17T13:03:29Z
```

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get nodes
NAME                 STATUS     ROLES           AGE   VERSION
kind-control-plane   NotReady   control-plane   28h   v1.33.1
kind-worker          Ready      <none>          28h   v1.33.1
kind-worker2         Ready      <none>          28h   v1.33.1
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl delete node kind-worker
Error from server (Forbidden): nodes "kind-worker" is forbidden: User "admin" cannot delete resource "nodes" in API group "" at the cluster scope
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

🔸 What is a ClusterRoleBinding?

A `ClusterRoleBinding` connects a user/group/serviceaccount to a **ClusterRole**—giving them access to **cluster-level or all namespaces.**

🔓 Grants cluster-wide access.

> This is like giving a new employee **company-wide admin access**—they can walk into any department, use any system.

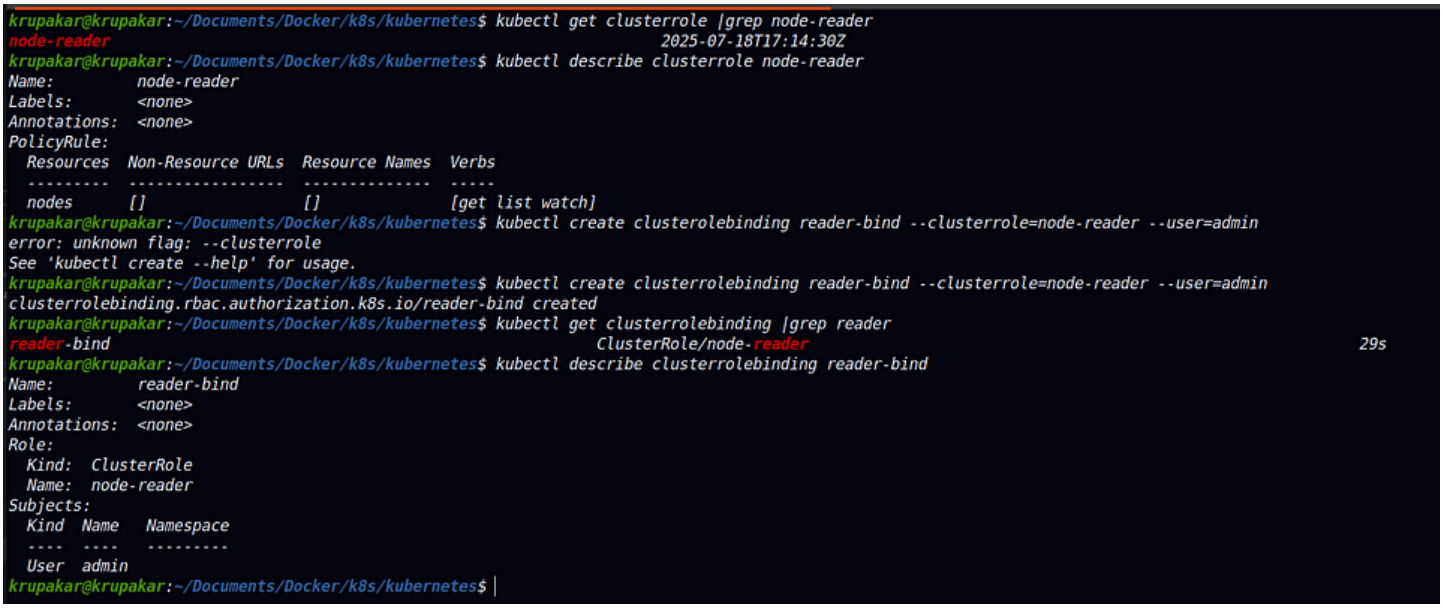
Final Flow (from joining to access)

1. 👤 **Employee joins company** → Gets ID (Create certificate and user in Kubernetes).
2. 🔐 **Authentication** → Security checks ID (Kubernetes verifies client cert).

3. 🎯 **Authorization** → Management tells which dept/resource he can access.
4. 📂 **Assign Role** → In dev team, can only read pods.
5. 🔗 **RoleBinding** → Actually binds employee to that dev role.
6. 🏢 **If Admin** → Gets full ClusterRole + ClusterRoleBinding.

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl api-resources --namespaced=false
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED   KIND
componentstatuses                 cs           v1                                     false        ComponentStatus
namespaces                        ns           v1                                     false        Namespace
nodes                             no           v1                                     false        Node
persistentvolumes                 pv           v1                                     false        PersistentVolume
mutatingwebhookconfigurations                  admissionregistration.k8s.io/v1        false        MutatingWebhookConfiguration
validatingadmissionpolicies                    admissionregistration.k8s.io/v1        false        ValidatingAdmissionPolicy
validatingadmissionpolicybindings              admissionregistration.k8s.io/v1        false        ValidatingAdmissionPolicyBinding
validatingwebhookconfigurations                admissionregistration.k8s.io/v1        false        ValidatingWebhookConfiguration
customresourcedefinitions         crd,crds     apiextensions.k8s.io/v1                false        CustomResourceDefinition
apiservices                                    apiregistration.k8s.io/v1              false        APIService
selfsubjectreviews                             authentication.k8s.io/v1               false        SelfSubjectReview
tokenreviews                                   authentication.k8s.io/v1               false        TokenReview
selfsubjectaccessreviews                       authorization.k8s.io/v1                false        SelfSubjectAccessReview
selfsubjectrulesreviews                        authorization.k8s.io/v1                false        SelfSubjectRulesReview
subjectaccessreviews                           authorization.k8s.io/v1                false        SubjectAccessReview
certificatesigningrequests        csr          certificates.k8s.io/v1                 false        CertificateSigningRequest
flowschemas                                    flowcontrol.apiserver.k8s.io/v1        false        FlowSchema
prioritylevelconfigurations                    flowcontrol.apiserver.k8s.io/v1        false        PriorityLevelConfiguration
ingressclasses                                 networking.k8s.io/v1                   false        IngressClass
ipaddresses                       ip           networking.k8s.io/v1                   false        IPAddress
servicecidrs                                   networking.k8s.io/v1                   false        ServiceCIDR
runtimeclasses                                 node.k8s.io/v1                         false        RuntimeClass
clusterrolebindings                            rbac.authorization.k8s.io/v1           false        ClusterRoleBinding
clusterroles                                   rbac.authorization.k8s.io/v1           false        ClusterRole
priorityclasses                   pc           scheduling.k8s.io/v1                   false        PriorityClass
csidrivers                                     storage.k8s.io/v1                      false        CSIDriver
csinodes                                       storage.k8s.io/v1                      false        CSINode
storageclasses                    sc           storage.k8s.io/v1                      false        StorageClass
volumeattachments                              storage.k8s.io/v1                      false        VolumeAttachment
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

cluster-level resources



🔑 Kubernetes Service Account?

In Kubernetes, **Service Accounts (SAs)** are **non-human user identities** used by **applications, bots, or controllers** to interact with the cluster.

Unlike regular **user accounts** (used by admins or developers), service accounts are for automation or internal cluster tasks.

📍 Analogy

Imagine a company:

- **Human users** are employees: Admins, Developers, QA, etc.—they log in with their own ID to do things manually.
- **Service accounts** are like automated machines: A CCTV monitoring system, or a daily auto-email bot—they perform tasks on behalf of the company without human input.

🔧 When and Why Are Service Accounts Used?

- When **applications running inside the cluster** (like Jenkins, Prometheus, ArgoCD, etc.) need to talk to the Kubernetes API.

- For **automation** tasks like:
- A Jenkins pipeline deploying apps.
- A monitoring tool scraping metrics.
- A cronjob interacting with the cluster.

Human users **don't trigger these manually**, so we don't want to assign our own credentials. We want a **dedicated identity**: that's the service account.

🔁 How Are Service Accounts Managed?

1. **Automatically Created:**
   Every namespace has a default service account.
   Run this command:

```
kubectl get sa -A | grep default
```

1. If there are 5 namespaces, we'll see 5 default service accounts (1 per namespace).
2. **Usage by Pods:**
   Every pod automatically uses the **default service account of its namespace**—unless we tell it to use a specific one.

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get sa
NAME      SECRETS   AGE
default   0         2d2h
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get sa -A
NAMESPACE           NAME                                 SECRETS   AGE
default             default                              0         2d2h
kube-node-lease     default                              0         2d2h
kube-public         default                              0         2d2h
kube-system         attachdetach-controller              0         2d2h
kube-system         bootstrap-signer                     0         2d2h
kube-system         certificate-controller               0         2d2h
kube-system         clusterrole-aggregation-controller   0         2d2h
kube-system         coredns                              0         2d2h
kube-system         cronjob-controller                   0         2d2h
kube-system         daemon-set-controller                0         2d2h
kube-system         default                              0         2d2h
kube-system         deployment-controller                0         2d2h
kube-system         disruption-controller                0         2d2h
kube-system         endpoint-controller                  0         2d2h
kube-system         endpointslice-controller             0         2d2h
kube-system         endpointslicemirroring-controller    0         2d2h
kube-system         ephemeral-volume-controller          0         2d2h
kube-system         expand-controller                    0         2d2h
kube-system         generic-garbage-collector            0         2d2h
kube-system         horizontal-pod-autoscaler            0         2d2h
kube-system         job-controller                       0         2d2h
kube-system         kindnet                              0         2d2h
kube-system         kube-proxy                           0         2d2h
kube-system         legacy-service-account-token-cleaner 0         2d2h
kube-system         namespace-controller                 0         2d2h
kube-system         node-controller                      0         2d2h
```

Default

🗒️ Why Does Every Namespace Have a Default Service Account?

When Kubernetes creates a **new namespace**, it **automatically creates a `default` service account** inside it.

🧠 The Reason: So Pods Can Talk to the Kubernetes API (If Needed)

Here's why:

- When we deploy a Pod, that Pod **might need to communicate with the Kubernetes API server**—for example:
- To check its own status.
- To watch for ConfigMap changes.

- To interact with other services.
- Kubernetes needs a way to **authenticate that Pod** to the API server.
- That's where the **default service account** comes in—it's like giving every Pod a badge saying:
  "Hi, I'm from namespace X. Here's my identity."
- If we don't explicitly assign a service account in our pod spec, Kubernetes **assigns the default one in that namespace**.

3. Assigning Permissions:
Just like human users, we can assign roles to service accounts using:

- `RoleBinding` (within a namespace)
- `ClusterRoleBinding` (for cluster-wide access)

1. **Example:**
   If Jenkins needs to deploy apps, we create a `jenkins-service-account`, and bind it to a role that gives it the required access (like create pods, manage deployments, etc.).

📍 Analogy

Imagine we join a new department in a company (a namespace in Kubernetes).

- The company automatically gives every department a **generic access badge** (default service account).
- If a machine (pod) is installed in that department, it **gets that default badge** unless we request a specific one.

This ensures the machine has **some identity** if it ever needs to ask for internal resources or perform tasks.

🧠 Why Not Use Human Users?

Let's say there are 100 people in the Operations team. If we use their personal accounts to run automation like Jenkins pipelines:

- Logs will show *who* ran it, even if it was triggered by code.
- If a person leaves the company, automation breaks.
- It's insecure, unscalable, and hard to manage.

Service accounts solve all this: consistent identity, proper RBAC, easily manageable.

Use Case:

When a **pod** runs, it pulls the **container image** from a registry (like Docker Hub or ECR).
If the image is in a **public** registry, no login is needed.
But if the image is in a **private registry**, Kubernetes needs **credentials** to pull that image.

Solution: Image Pull Secret

Kubernetes uses an `imagePullSecret` to **store Docker registry credentials** securely.
Then, it can use that secret to log in and **pull the private image.**

🔐 How to create an image pull secret?

Let's say our Docker registry username is `myuser` and password is `mypassword` .

```
kubectl create secret docker-registry my-registr
  --docker-username=myuser \
  --docker-password=mypassword \
  --docker-email=myemail@example.com \
  --docker-server=https://index.docker.io/v1/
```

This creates a secret of type `kubernetes.io/dockerconfigjson`.

🚀 How to use it in a Pod?

In the pod YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: private-image-pod
spec:
  containers:
  - name: myapp
    image: myuser/myprivateimage:latest
  imagePullSecrets:
  - name: my-registry-secret
```

This tells the pod to use `my-registry-secret` to **log in and pull** the image.

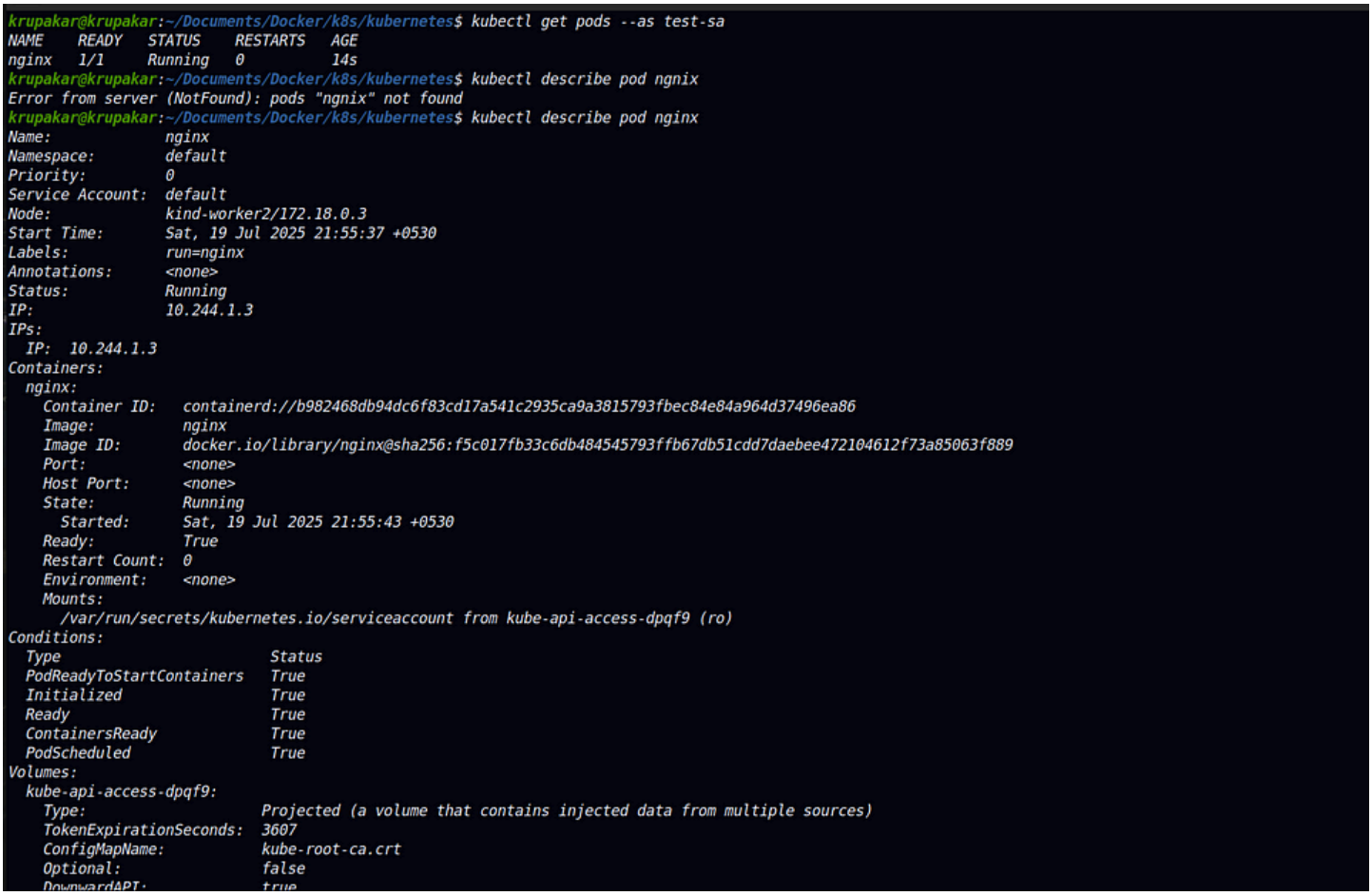
```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl create sa test-sa
serviceaccount/test-sa created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl describe sa test-sa
Name:                test-sa
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   <none>
Tokens:              <none>
Events:              <none>
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ code .

krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ code .
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl apply -f SAsecret.yaml
secret/build-robot-secret created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get secret
NAME                  TYPE                                  DATA   AGE
build-robot-secret    kubernetes.io/service-account-token   3      14s
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl describe secret build-robot-secret
Name:         build-robot-secret
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: test-sa
              kubernetes.io/service-account.uid: a481bfcd-dbaa-47c9-bcf1-811af2295728

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1107 bytes
namespace:  7 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6ImozaU83TjFPelNYcV9kYmF5eUxnS2VQLXRkQ09JbWN6Nll5T2oxYnVFTlEifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNj
b3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6ImJ1aWxkLXJvYm90LXNlY3JldCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5
hbWUiOiJ0ZXN0LXNhIiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQudWlkIjoiYTQ4MWJmY2QtZGJhYS00N2M5LWJjZjEtODExYWYyMjk1NzI4Iiwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50OmRlZm
F1bHQ6dGVzdC1zYSJ9.07hDnjvamncczpBSKo3i0x_W3_KfO2zry1Hh9zYoPVo5F2RQcJQglEwuRYzCzuNYAkOqPVES9MU_9m2boeZHOsJKpVfq86-w9DzNm4kNQIiB4JJ1CF8JLYGBPZeCXHwRZEjIa7hm7ABPx7Gn7Q1CNUackGH2j1FE344P
OmsEpU4289-xDTcJ463jjc3B21YlTl9qTuFGTUNRTmmWXKZc1DKYEvUr5OzGFAmlAfw4xpwEKtlMWA4p8QAU3-Lf4_hWlsKWsPw8dmv2HR8BalTdIXQTjxtaOxfxJBr4YLb0AiAn4RD9i_6chGi0JXMUSZsXOimYtZRewFAKBokUAJbezg
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl auth can-i get pods --as test-sa
no
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl create role test-role \
> --verb=list,get,watch \
> resource=pod
error: exactly one NAME is required, got 2
See 'kubectl create role -h' for help and examples
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl create role test-role --verb=list,get,watch --resource=pod
role.rbac.authorization.k8s.io/test-role created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl create rolebinding test-rb --role=test-role --user=test-sa
rolebinding.rbac.authorization.k8s.io/test-rb created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get pods --as test-sa
No resources found in default namespace.
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl auth can-i get pods --as test-sa
yes
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

🔁 Do We need to add it for every pod?

Yes—**by default**, we have to specify `imagePullSecrets` for each pod using a private image.

But we can attach this **once to a ServiceAccount**, and all pods using that SA will use the secret.



```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get pods --as test-sa
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          14s
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl describe pod ngnix
Error from server (NotFound): pods "ngnix" not found
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl describe pod nginx
Name:             nginx
Namespace:        default
Priority:         0
Service Account:  default
Node:             kind-worker2/172.18.0.3
Start Time:       Sat, 19 Jul 2025 21:55:37 +0530
Labels:           run=nginx
Annotations:      <none>
Status:           Running
IP:               10.244.1.3
IPs:
  IP:  10.244.1.3
Containers:
  nginx:
    Container ID:   containerd://b982468db94dc6f83cd17a541c2935ca9a3815793fbec84e84a964d37496ea86
    Image:          nginx
    Image ID:       docker.io/library/nginx@sha256:f5c017fb33c6db484545793ffb67db51cdd7daebee472104612f73a85063f889
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Sat, 19 Jul 2025 21:55:43 +0530
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-dpqf9 (ro)
Conditions:
  Type                        Status
  PodReadyToStartContainers   True
  Initialized                 True
  Ready                       True
  ContainersReady             True
  PodScheduled                True
Volumes:
  kube-api-access-dpqf9:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    Optional:                false
    DownwardAPI:             true
```

🔁 Attach Image Pull Secret to Service Account

```
kubectl patch serviceaccount default \
  -p '{"imagePullSecrets": [{"name": "my-registr
```

This adds the image pull secret to the **default service account** in the namespace.

So now, **any pod** using the default SA will use the secret automatically.

📍 Where is the token stored?

When a pod is created using a service account, the **API token** (used to talk to Kubernetes API) is auto-mounted at:

```
/var/run/secrets/kubernetes.io/serviceaccount/to
```

But this is **different from image pull secret.**
This is for **API access,** while imagePullSecret is for **pulling container images.**

🎯 In Simple Steps

1. We have a **private image** in Docker Hub (or another registry).
2. We create a **Docker registry secret** in Kubernetes.
3. We either.

- Add the secret in our pod YAML using `imagePullSecrets` , or
- Attach it once to the service account (like `default`) so all pods can use it.

4. Kubernetes uses that secret to **login to the registry** and **pull the image.**

Network Policy (Kubernetes)

A **NetworkPolicy** in Kubernetes is a resource that controls how **pods communicate** with each other and with the **outside world.**

> It helps **allow or block traffic** between pods based on **rules** like labels, namespaces, ports, and IP blocks.

🌐 1. Application Architecture Example

Imagine a typical 3-tier app deployed in Kubernetes:

- **Frontend Pod (Web)** → Port **80**
- **Backend Pod (App logic)** → Port **443**
- **Database Pod (DB)** → Port **3306**

The flow of a user request looks like this:

User → Frontend → Backend → Database

This is the **normal communication flow**—it happens in **two directions**:

📥 2. Types of Network Flows

1. **Ingress**
   Traffic *coming into a pod* from outside (e.g., User to Frontend)
2. **Egress**
   Traffic *going out of a pod* (e.g., Backend calling the Database)

⚠️ 3. Default Kubernetes Behavior

By default:

> 🚨 **All pods can talk to all other pods in the cluster!**

- **Frontend can talk to Backend** ✅
- **Backend can talk to Database** ✅

- But **Frontend can also talk directly to Database**
  Traffic going out of a pod (e.g., Backend calling the Database)
  ⚠️ 3. Default Kubernetes Behavior
  By default:

- 🚨 All pods can talk to all other pods in the cluster!
  Frontend can talk to Backend ✅
  Backend can talk to Database ✅
  But Frontend can also talk direc

- **Database can also reply directly to Frontend**

This is **not secure**—it breaks the proper layered access.

🔐 4. Why We Need Network Policies

We want to **control** which pods can talk to which—for security and proper design.

So we write **Network Policies** like:

- Frontend **cannot** access Database
- Frontend can only talk to Backend
- Backend can talk to DB
- DB cannot talk to Frontend directly

🔌 5. CNI Plugins and Policy Enforcement

- Kubernetes needs a **CNI plugin** that supports Network Policies.
- **Kind** uses `kindnet` by default—it does **not support** enforcing network policies.

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get po -n=kube-system
NAME                                          READY   STATUS        RESTARTS      AGE
coredns-674b8bbfcf-sk2t7                      1/1     Running       2 (57m ago)   45h
coredns-674b8bbfcf-sv9xh                      1/1     Terminating   0             2d19h
coredns-674b8bbfcf-tx457                      1/1     Terminating   0             2d19h
coredns-674b8bbfcf-v7vxq                      1/1     Running       2 (57m ago)   45h
etcd-kind-control-plane                       1/1     Running       0             2d19h
kindnet-bnx66                                 1/1     Running       3 (57m ago)   2d19h
kindnet-dcd6n                                 1/1     Running       3 (57m ago)   2d19h
kindnet-flzm2                                 1/1     Running       0             2d19h
kube-apiserver-kind-control-plane             1/1     Running       0             2d19h
kube-controller-manager-kind-control-plane    1/1     Running       0             2d19h
kube-proxy-g298v                              1/1     Running       3 (57m ago)   2d19h
kube-proxy-j78rm                              1/1     Running       0             2d19h
kube-proxy-vm7wt                              1/1     Running       3 (57m ago)   2d19h
kube-scheduler-kind-control-plane             1/1     Running       0             2d19h
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get ds -A
NAMESPACE     NAME         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
kube-system   kindnet      3         3         2       3            2           kubernetes.io/os=linux   2d19h
kube-system   kube-proxy   3         3         2       3            2           kubernetes.io/os=linux   2d19h
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

- So we install **Calico** (or Cilium) to support and enforce network policies in Kind.

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ cat kindnetwork.yml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 30001
    hostPort: 30001
- role: worker
- role: worker
networking:
  disableDefaultCNI: true
  podSubnet: 192.168.0.0/16krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kind create cluster --name kind-kind --config kindnetwork.yml
Creating cluster "kind-kind" ...
 ✓ Ensuring node image (kindest/node:v1.33.1) 🖼
 ✓ Preparing nodes 📦 📦 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing StorageClass 💾
 ✓ Joining worker nodes 🚜
Set kubectl context to "kind-kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind-kind

Thanks for using kind! 😊
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get nodes
NAME                  STATUS     ROLES           AGE    VERSION
kind-kind-control-plane   NotReady   control-plane   4m5s   v1.33.1
kind-kind-worker          NotReady   <none>          86s    v1.33.1
kind-kind-worker2         NotReady   <none>          86s    v1.33.1
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.30.2/manifests/calico.yaml
poddisruptionbudget.policy/calico-kube-controllers created
serviceaccount/calico-kube-controllers created
serviceaccount/calico-node created
serviceaccount/calico-cni-plugin created
```

```
Every 2.0s: kubectl get pods -l k8s-app=calico-node -A

NAMESPACE     NAME                READY   STATUS     RESTARTS   AGE
kube-system   calico-node-5rzp7   0/1     Init:0/3   0          92s
kube-system   calico-node-ls842   0/1     Init:0/3   0          93s
kube-system   calico-node-t55fx   0/1     Init:0/3   0          92s
```

🛠️ Scenario: NetworkPolicy to Control Access Between Frontend → Backend → DB

We'll do this in a structured way:

1️⃣ Create 3 Deployments + 3 Services

- **frontend**
- **backend**
- **database**

Each Deployment will:

- Have labels like `role: front`, `role: back`, `role: db`
- Have its own **ClusterIP service**

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl apply -f NWfe.yml
pod/frontend created
service/frontend created
pod/backend created
service/backend created
service/db created
pod/mysql created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get po -o wide
NAME       READY   STATUS              RESTARTS   AGE   IP       NODE                 NOMINATED NODE   READINESS GATES
backend    0/1     ContainerCreating   0          32s   <none>   kind-kind-worker2    <none>           <none>
frontend   0/1     ContainerCreating   0          41s   <none>   kind-kind-worker2    <none>           <none>
mysql      0/1     ContainerCreating   0          16s   <none>   kind-kind-worker2    <none>           <none>
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get svc -o wide
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE     SELECTOR
backend      ClusterIP   10.96.120.81    <none>        80/TCP     38s     role=backend
db           ClusterIP   10.96.211.213   <none>        3306/TCP   30s     name=mysql
frontend     ClusterIP   10.96.114.128   <none>        80/TCP     46s     role=frontend
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP    9m13s   <none>
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get po -o wide --watch
NAME       READY   STATUS              RESTARTS   AGE   IP       NODE                 NOMINATED NODE   READINESS GATES
backend    0/1     ContainerCreating   0          54s   <none>   kind-kind-worker2    <none>           <none>
frontend   0/1     ContainerCreating   0          63s   <none>   kind-kind-worker2    <none>           <none>
mysql      0/1     ContainerCreating   0          38s   <none>   kind-kind-worker2    <none>           <none>
^Ckrupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get po -o wide
NAME       READY   STATUS              RESTARTS   AGE     IP       NODE                 NOMINATED NODE   READINESS GATES
backend    0/1     ContainerCreating   0          2m50s   <none>   kind-kind-worker2    <none>           <none>
frontend   0/1     ContainerCreating   0          2m59s   <none>   kind-kind-worker2    <none>           <none>
mysql      0/1     ContainerCreating   0          2m34s   <none>   kind-kind-worker2    <none>           <none>
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get po -o wide --watch
NAME       READY   STATUS              RESTARTS   AGE     IP       NODE                 NOMINATED NODE   READINESS GATES
backend    0/1     ContainerCreating   0          3m      <none>   kind-kind-worker2    <none>           <none>
frontend   0/1     ContainerCreating   0          3m9s    <none>   kind-kind-worker2    <none>           <none>
mysql      0/1     ContainerCreating   0          2m44s   <none>   kind-kind-worker2    <none>           <none>
mysql      0/1     ContainerCreating   0          3m10s   <none>   kind-kind-worker2    <none>           <none>
backend    0/1     ContainerCreating   0          3m26s   <none>   kind-kind-worker2    <none>           <none>
frontend   0/1     ContainerCreating   0          3m38s   <none>   kind-kind-worker2    <none>           <none>
mysql      1/1     Running             0          6m5s    192.168.197.2   kind-kind-worker2   <none>           <none>
backend    1/1     Running             0          8m11s   192.168.197.1   kind-kind-worker2   <none>           <none>
frontend   1/1     Running             0          8m34s   192.168.197.3   kind-kind-worker2   <none>           <none>
```

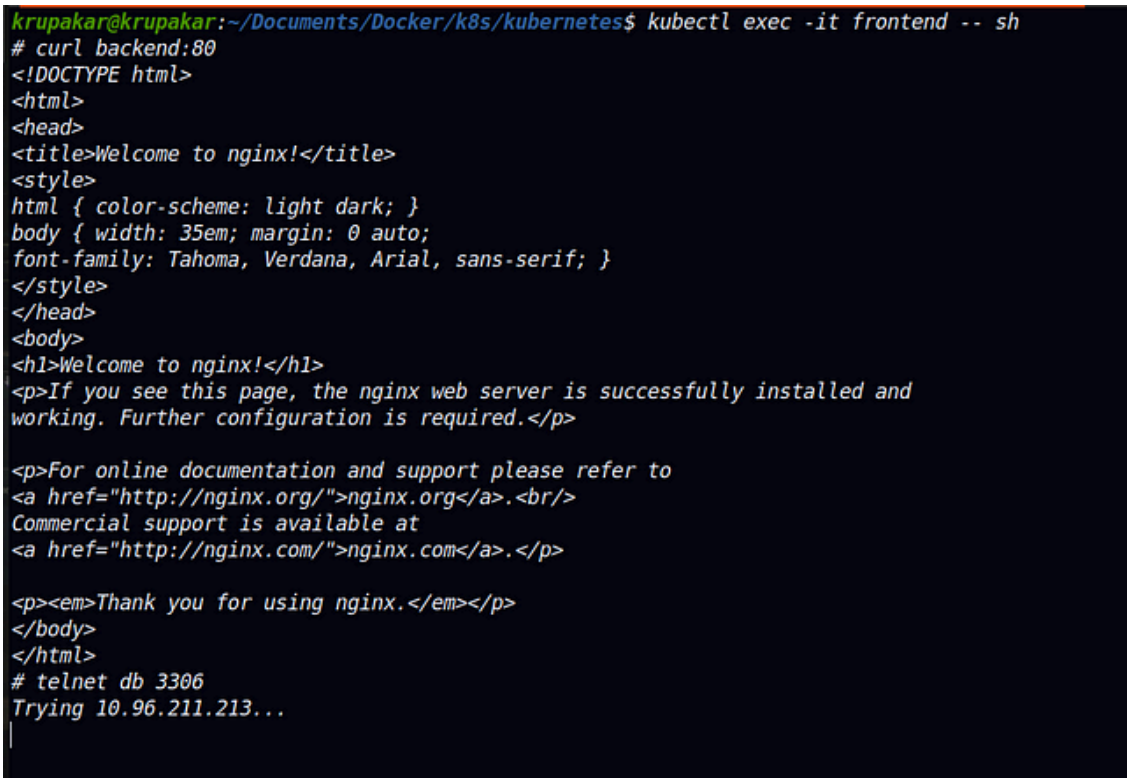
2️⃣ Default Behavior (No Network Policy)

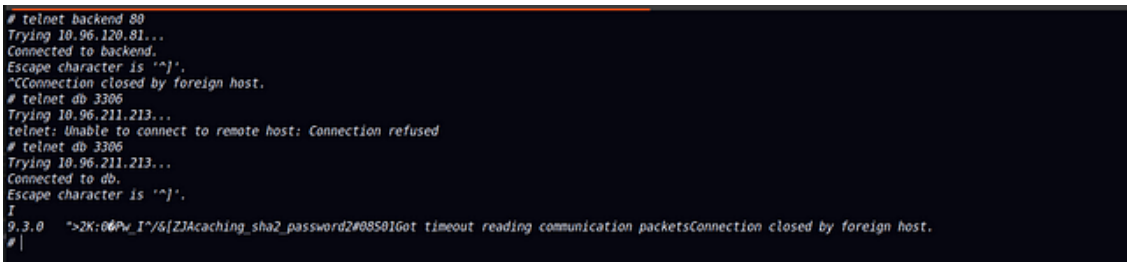
- We `kubectl exec` into any pod (e.g., frontend pod)
- We can `curl` all services (backend, db)

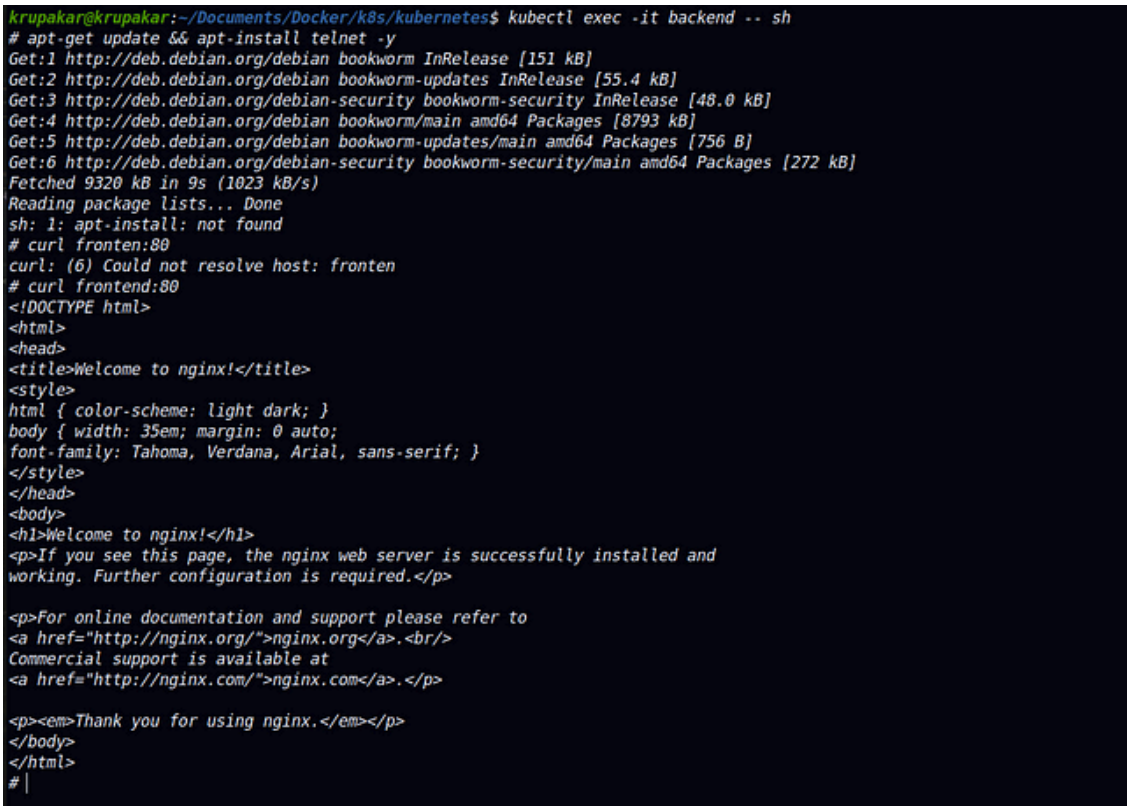
- That means: **all pods can access all pods**

`curl backend` → OK
`curl db` → OK
`curl frontend` from db → OK

⚠️ This is **insecure**







3️⃣ Use telnet Instead of curl for DB Testing

Since DB doesn't give response on `curl`, we'll use `telnet`:

```
apt update && apt install -y telnet
telnet db 3306
```

4️⃣ Create Initial Network Policy

Now we **want to restrict** the access:

- Frontend can talk to Backend
- Backend can talk to Database
- Frontend should NOT talk to Database

- Database should NOT talk to Frontend

5️⃣ How to Write the Network Policy

We'll write a **NetworkPolicy** that allows **Ingress traffic** only from the pod that has `role=front`.

Example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-front-to-back
spec:
  podSelector:
    matchLabels:
      role: back
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: front
```

- This means: **Only frontend pods can talk to backend**
- All others (e.g., DB or other) → ❌ denied

Repeat similar for backend → db

6️⃣ Add Egress Control (Optional)

We can also control who a pod **can talk to (egress)**

Example: Backend should **only** be allowed to talk to DB:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: back-egress-to-db
spec:
  podSelector:
    matchLabels:
      role: back
  policyTypes:
  - Egress
  egress:
  - to:
    - podSelector:
        matchLabels:
          role: db
```

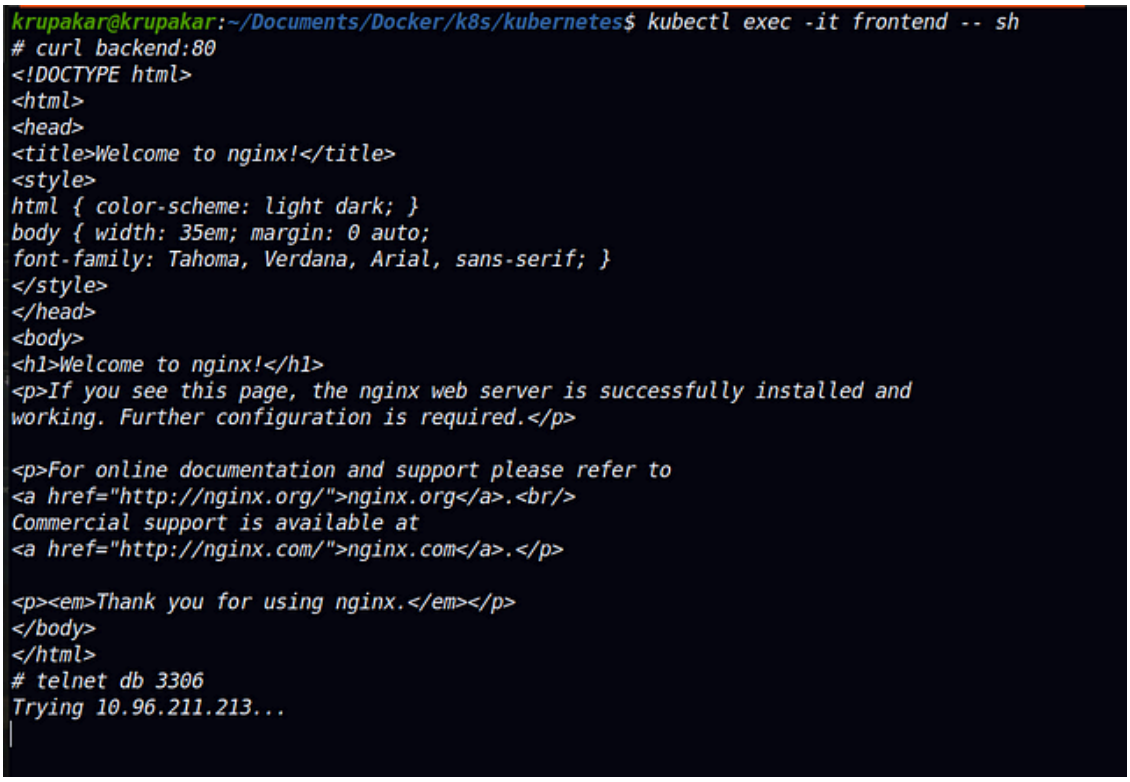
```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ cat networkpolicy.yml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-test
spec:
  podSelector:
    matchLabels:
      name: mysql
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: backend
    ports:
    - port: 3306krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl apply -f networkpolicy.yml
networkpolicy.networking.k8s.io/db-test created
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get netpole
error: the server doesn't have a resource type "netpole"
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl get netpol
NAME      POD-SELECTOR   AGE
db-test   name=mysql     35s
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl describe netpol db-test
Name:         db-test
Namespace:    default
Created on:   2025-07-20 16:35:06 +0530 IST
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     name=mysql
  Allowing ingress traffic:
    To Port: 3306/TCP
    From:
      PodSelector: role=backend
  Not affecting egress traffic
  Policy Types: Ingress
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$
```

7️⃣ Test the Policy

Now test again:

- ✅ `curl backend` from frontend → OK
- `curl db` from frontend → Denied
- `telnet db 3306` from backend → OK
- `curl frontend` from db → Denied

```
krupakar@krupakar:~/Documents/Docker/k8s/kubernetes$ kubectl exec -it backend -- sh
# telnet db 3306
Trying 10.96.211.213...
Connected to db.
Escape character is '^]'.
I
9.3.0
     gRL``*C0<RP\W!lpbcaching_sha2_password
```



✅ Final Setup Flow:

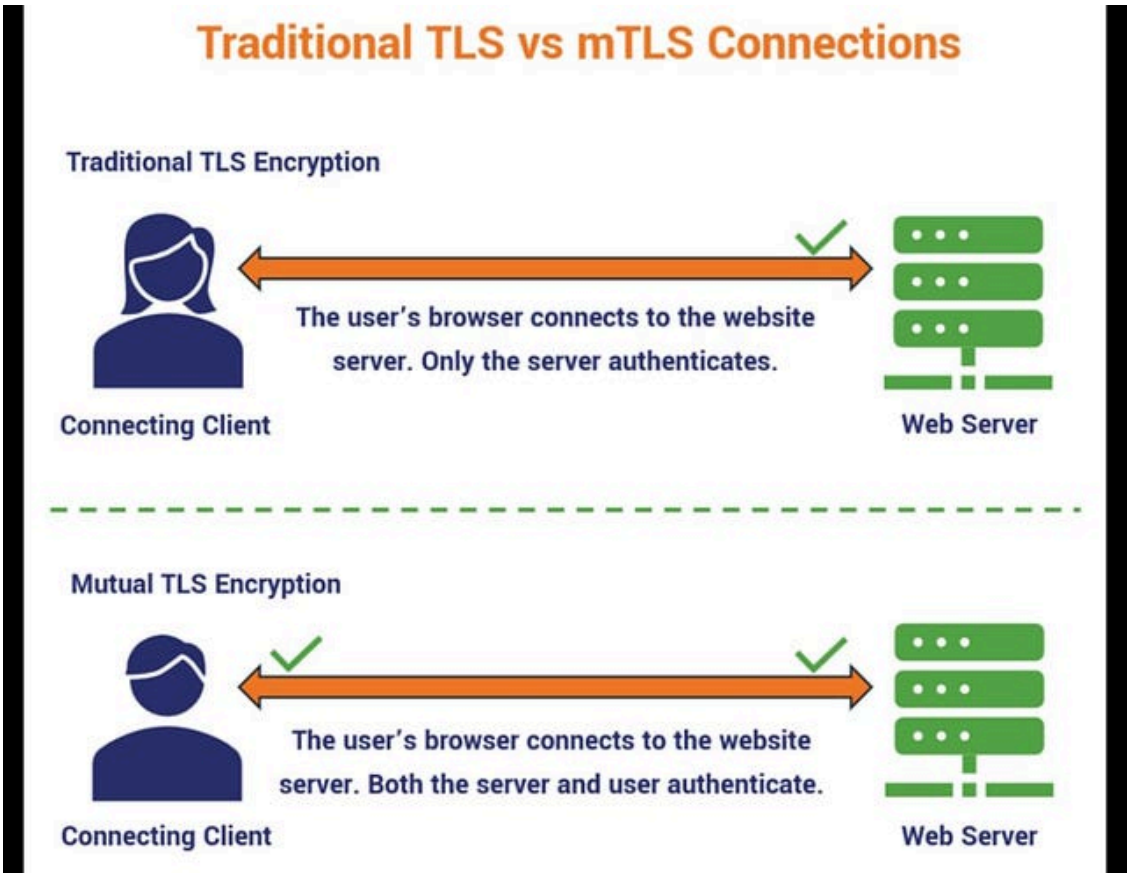
Frontend → Backend → DB

| ^

❌————————————|

Visual Guide: How TLS, mTLS, and Certificates Work in Kubernetes

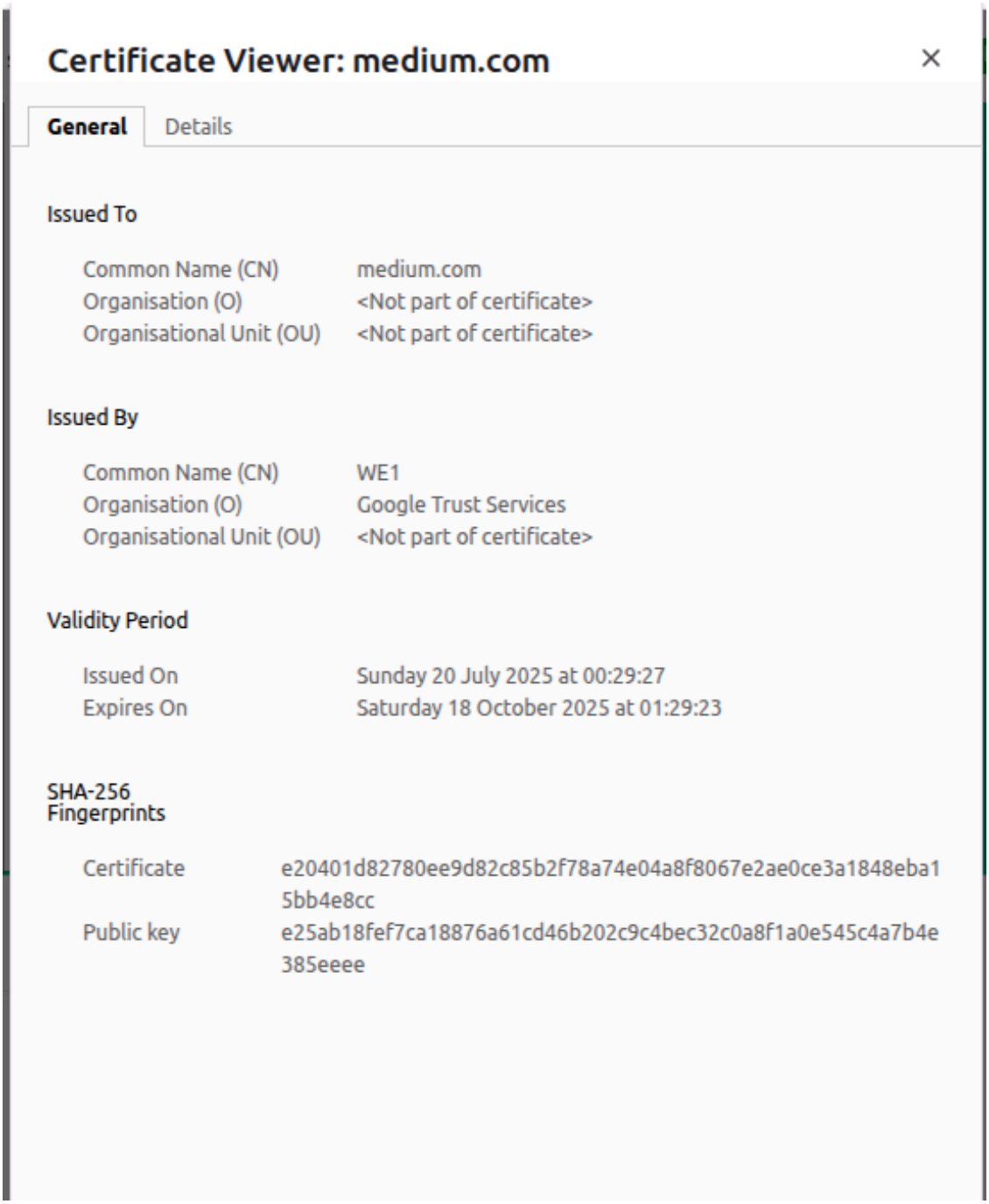
> A simplified look at how Kubernetes uses certificates for secure communication and authentication between components.



Traditional TLS vs Mutual TLS (mTLS)

The diagram shows the key difference between traditional TLS and mutual TLS:

- In **Traditional TLS**, only the **server presents a certificate.** It's like when we visit a website—our browser checks if the site is secure, but the site doesn't check who we are.



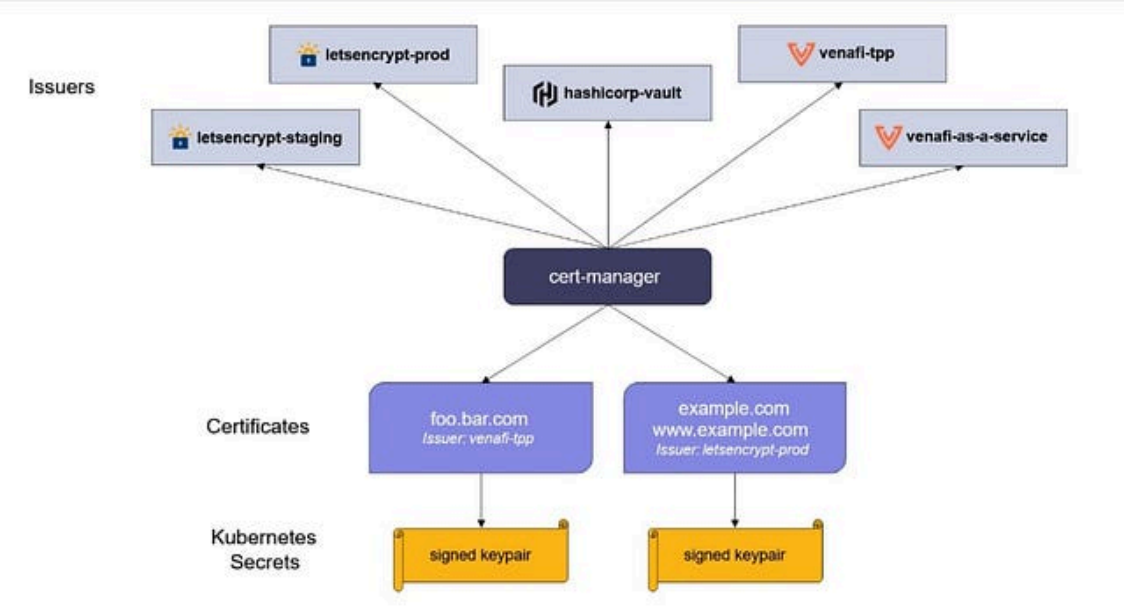
Secure Connection Proof:
Our browser shows this like to confirm the site is real and the connection is safe.

- In **Mutual TLS (mTLS), both the client and the server present certificates.** It's like a secure handshake where **both parties verify each other's identity.**

💡 This extra layer of trust is especially important in Kubernetes environments—like when two internal services (e.g., frontend and

backend) communicate. mTLS ensures that **only trusted components** talk to each other.

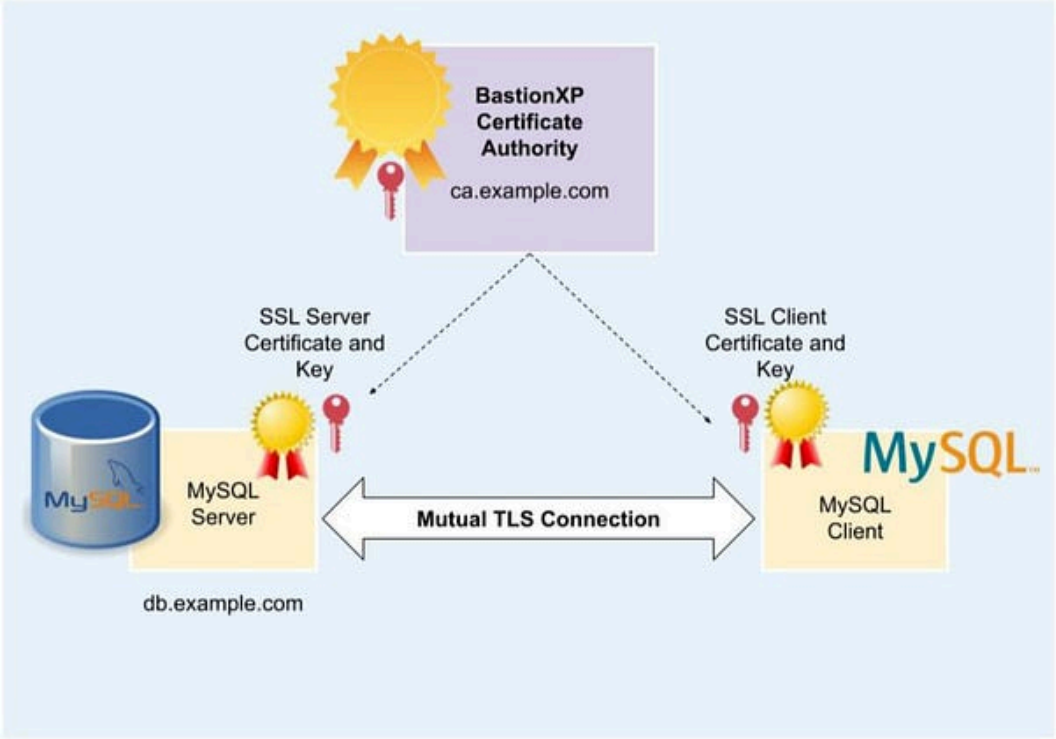
. . .



The diagram shows how **cert-manager** helps handle certificates automatically inside Kubernetes.

It works with tools like **Let's Encrypt**, **Vault**, or **Venafi** (called *Issuers*) to request and renew certificates for us.

These certificates are saved as Kubernetes **secrets** and used to secure communication using **TLS** or **mTLS**.



Mutual TLS Authentication—MySQL Example

The diagram shows how **mutual TLS (mTLS)** works between a MySQL client and server.

Both the client and server get certificates from the same **Certificate Authority (CA).**

Before connecting, they **check each other's certificates** to make sure they're trusted.

This way, **both sides are verified**, making the connection secure and trustworthy.

https://medium.com/@shirishareddy.yasa