Kubernetes Security

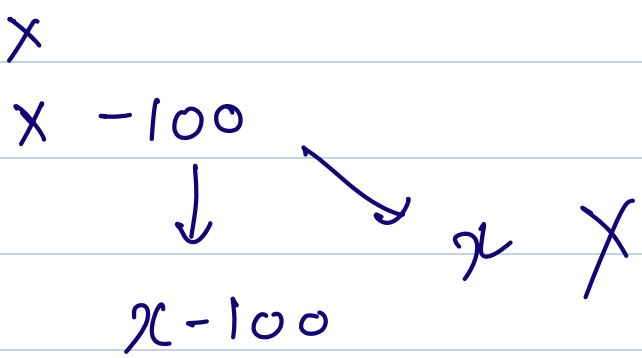
starts at 9:05pm

Agenda

1. Stateful sets
2. Authentication
    1. Types
3. Authorisation
    1. Types
4. Demo Authentication Authorisation
5. Security Contexts
6. Admission Controller

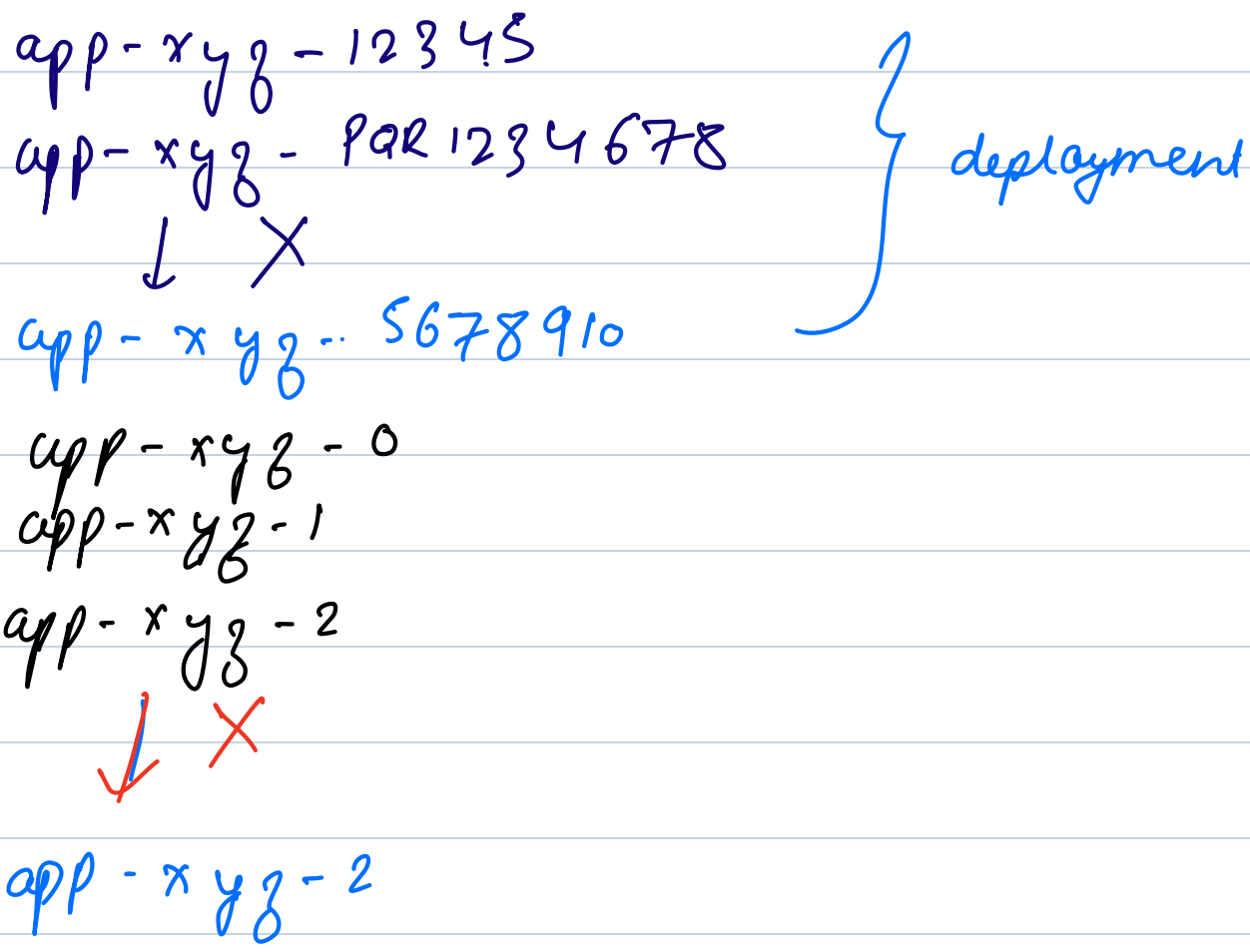
→ Stateful sets

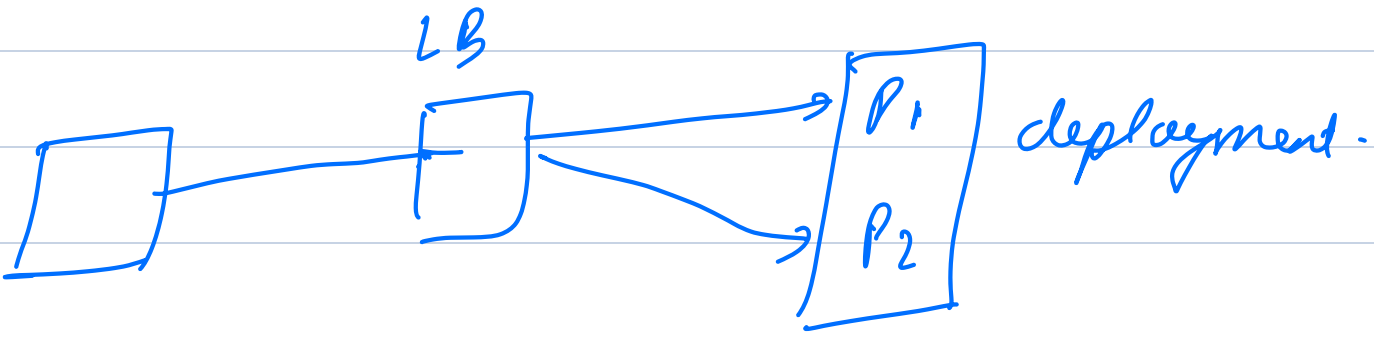
A **stateful application** is an application that **maintains its state** across different sessions, meaning it remembers data and context from one operation to the next.



A **StatefulSet** is a Kubernetes workload API object used to **manage stateful applications**.

Feature	Deployment	StatefulSet
Use Case	Stateless applications	Stateful applications (DBs, etc.)
Pod Identity	No stable identity, random names	Stable, unique identities
Storage	Ephemeral or shared storage	Persistent storage with PVCs
Scaling	Pods scaled randomly	Pods scaled sequentially
Service Type	Normal Service (for load balancing)	Headless Service (for unique DNS)






```
volumeClaimTemplates:
  - metadata:
      name: mysql-data
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 5Gi
      storageClassName: standard
```

Demo

(1) Headless Service.

my-db-0.db-service.my-namespace1.svc.cluster.local

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: db-service
spec:
  clusterIP: None  # This makes the service headless (no IP assignment)
  selector:
    app: my-db
  ports:
    - port: 5432
      targetPort: 5432
```

Stateful set (different pv, pvcs per each pod)

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-db
spec:
  serviceName: db-service  # Link to the headless service
  replicas: 3  # Number of replicas (PostgreSQL pods)
  selector:
    matchLabels:
      app: my-db
```

```
  template:
    metadata:
      labels:
        app: my-db
    spec:
      containers:
        - name: database
          image: postgres
          env:
            - name: POSTGRES_PASSWORD
              value: "mysecretpassword"  # Set the password for PostgreSQL
          ports:
            - containerPort: 5432
          volumeMounts:
            - name: db-storage
              mountPath: /var/lib/postgresql/data  # Path where the DB will store data
  volumeClaimTemplates:
    - metadata:
        name: db-storage
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
```

requests:

storage: 5Gi

Kubernetes Security.

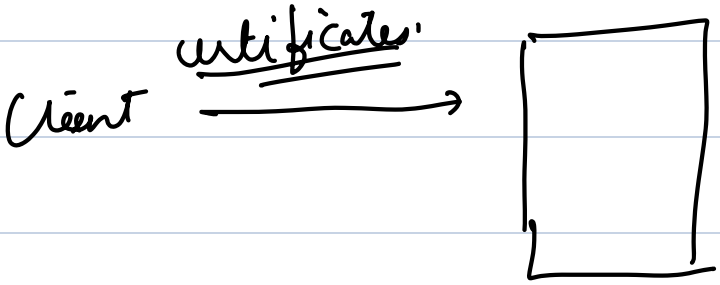
Authentication
Authorisation
Admission Controller.

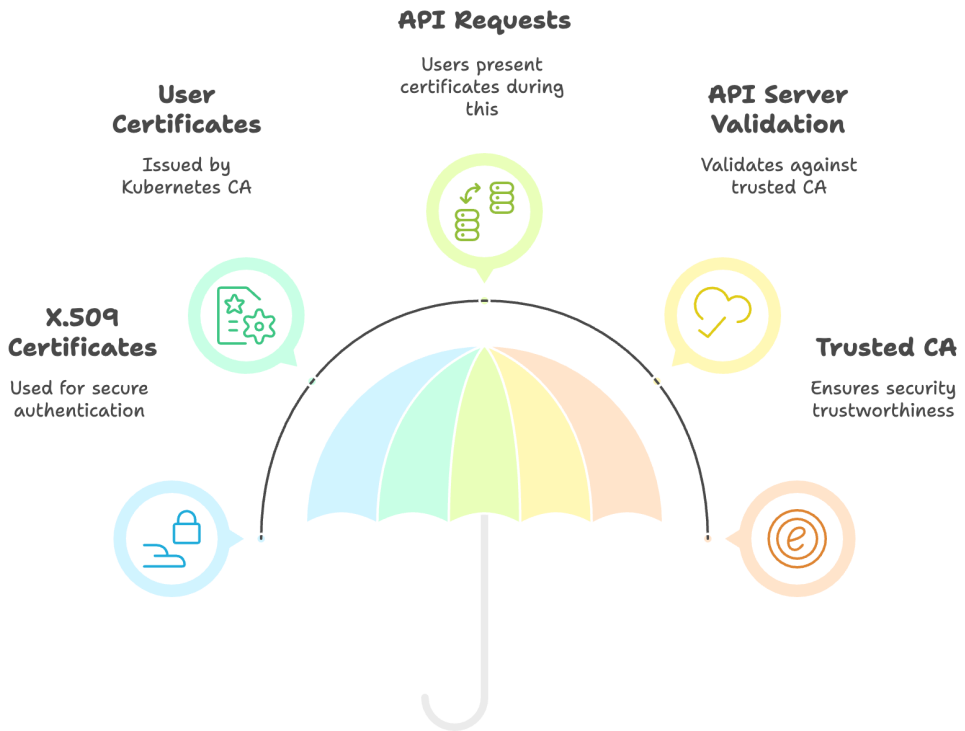
Authentication.

Authentication is the process of verifying the identity of users and systems that interact with the Kubernetes cluster.

Different Authentication Methods.

1) Client certificates.

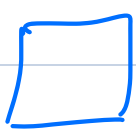




(2) Bearer Token Authentication.

(1) Service Account Token.

→ Pod.



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

This token allows the Pod to authenticate with the API Server.

② static Token file.

Users → SA → grant API access.

/etc/kubernetes/static_tokens.csv

Token never expires
!
Security Risk.

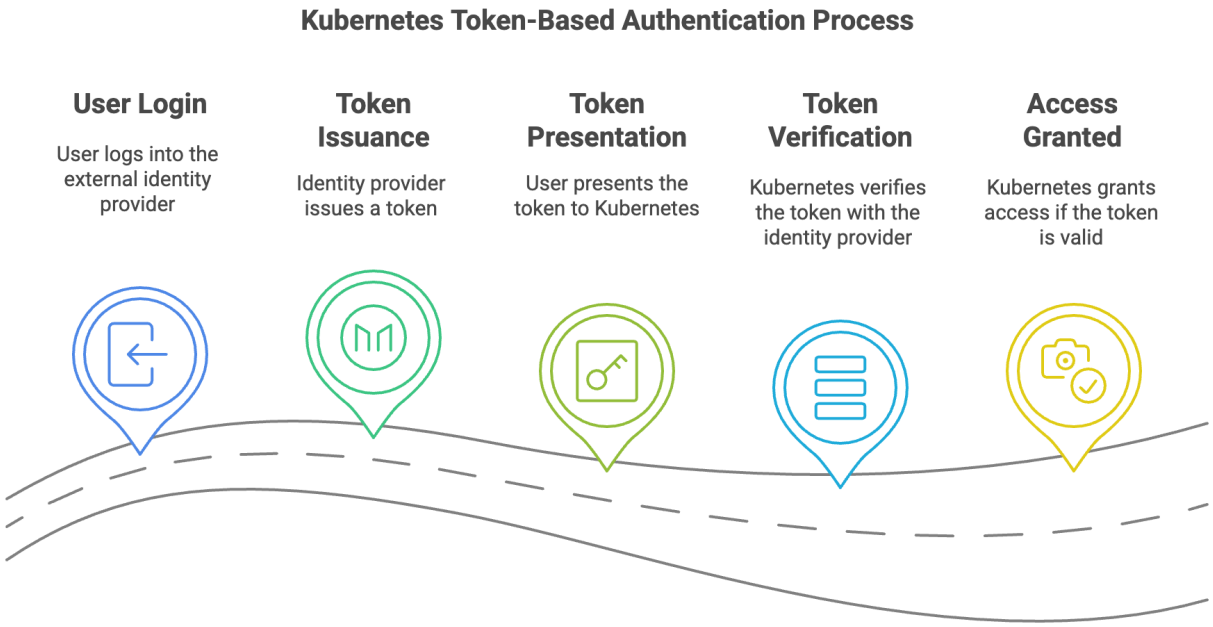
echo "my-static-token,user1,user1,system:masters" > /etc/kubernetes/static_tokens.csv

③ OpenID Connect Authentication.

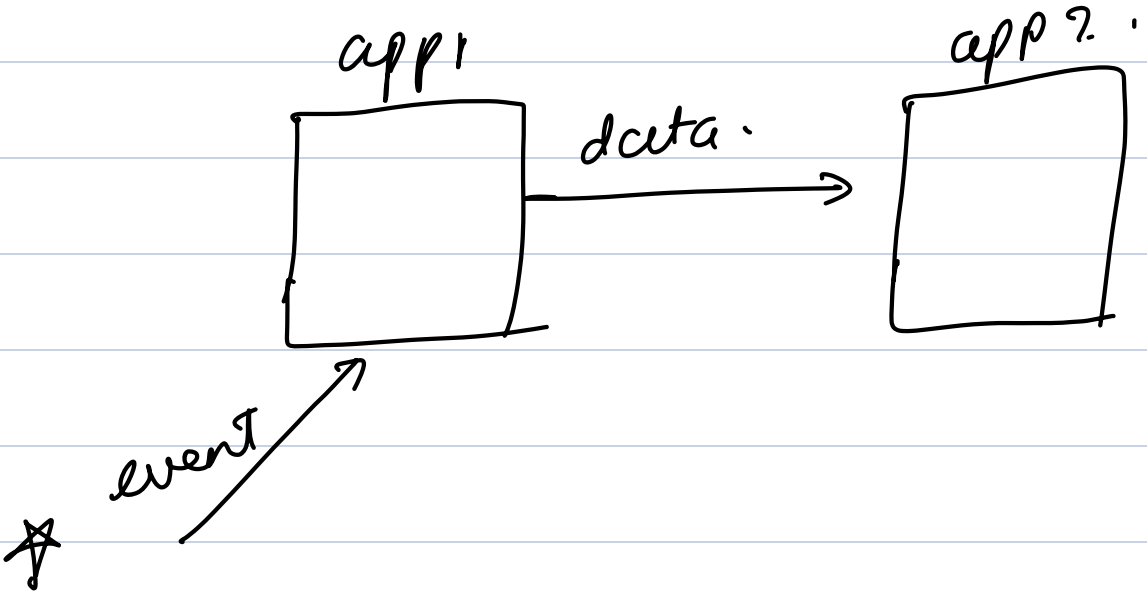
It enables Kubernetes to authenticate users using external identity providers.

eg:- Okta?
google
Azure.

IPd.

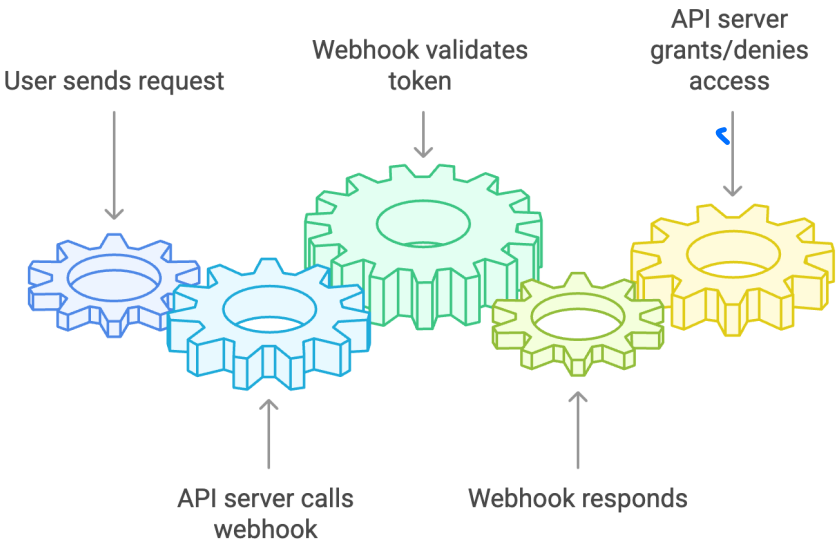


(4) Webhook Token Authentication.



Webhook authentication allows Kubernetes to **delegate authentication** to an external service.

Kubernetes API Authentication Process



Authentication Method	How It Works	Use Case	Pros	Cons
Client Certificates	Uses X.509 certificates to authenticate users and systems.	Secure authentication for admins and system processes.	Strong security, cryptographic validation.	Requires **certificate management**.
Service Account Tokens	Kubernetes generates tokens for Pods to access the API.	Used by applications running inside the cluster.	Automatic, easy to use.	Token rotation and security risks if leaked.
Static Token File	Stores predefined tokens in a file on the API server.	Used for **testing or simple setups**.	Easy setup.	**Insecure**, manual management.
OIDC Authentication	Uses an external identity provider (Google, Azure, Okta).	Enterprise authentication with **corporate credentials**.	Centralized identity management, easy integration.	Requires an external provider and setup.
Webhook Token Authentication	API server calls an external webhook to validate tokens.	Custom authentication using a separate service.	**Flexible**, integrates with external systems.	Requires **maintaining a separate authentication service**.

Break → 10:20pm.

Authorisation.

→ authenticated → what all can you do inside the cluster?

Authorization is used for controlling access to resources

Different Authorisation Modes.

1. Node.

   used by Kubelet.

This mode authorizes kubelets to read and write to their node objects.

2. ABAC.

   Attribute Based Access Control.

```
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
  "kind": "Policy",
```

```
  "spec": {
    "user": "alice",
    "namespace": "default",
    "resource": "pods",
    "readonly": true
  }
}
```

```
- --authorization-mode=Node,RBAC,ABAC
- --authorization-policy-file=/etc/kubernetes/abac-policy.json
```

After adding abac-policy.json
↓
Restart the API server pod.

Why is ABAC Less Preferred?

- Policies are stored in a **static file**, requiring API server restarts for updates.
- **Not scalable** for large clusters.
- **RBAC is preferred** because it is **dynamic and easier to manage**.

3 Webhook Authorization.

A **custom external service** makes authorization decisions. When a request is received:

1. Kubernetes sends it to an external **webhook server**.
2. The server **verifies the request** based on custom logic.
3. The server **returns allow/deny decisions**.

```
- --authentication-token-webhook-config-file=/etc/kubernetes/webhook-config.yaml
- --authentication-token-webhook-cache-ttl=5m
```

```
apiVersion: v1
kind: Config
clusters:
 - name: webhook-auth
   cluster:
     server: https://auth.example.com/validate
users:
 - name: webhook-auth
contexts:
 - context:
     cluster: webhook-auth
     user: webhook-auth
   name: webhook-auth
```

current-context: webhook-auth

→ RBAC

Roles → Permissions

Rolebindings → Assigned to users that make use of this job.

Roles. → within a namespace

ClusterRoles. → across cluster.

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "watch", "list"]

API Group	Example Resources
"" (empty, core API)	pods, services, configmaps, secrets, nodes, namespaces
`apps`	deployments, daemonsets, statefulsets, replicasets
`batch`	jobs, cronjobs
`rbac.authorization.k8s.io`	roles, rolebindings, clusterroles, clusterrolebindings
`networking.k8s.io`	networkpolicies, ingresses
`storage.k8s.io`	storageclasses, volumeattachments

Verb	Description
`get`	Read a specific resource (e.g., `kubectl get pod my-pod`)
`list`	List all resources of a type (e.g., `kubectl get pods`)
`watch`	Monitor changes to resources in real-time
`create`	Create a new resource
`update`	Modify an existing resource
`patch`	Partially update an existing resource
`delete`	Remove a resource
`deletecollection`	Delete multiple resources at once

RoleBindings
ClusterRoleBindings.

Associates role with a user or group.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
```

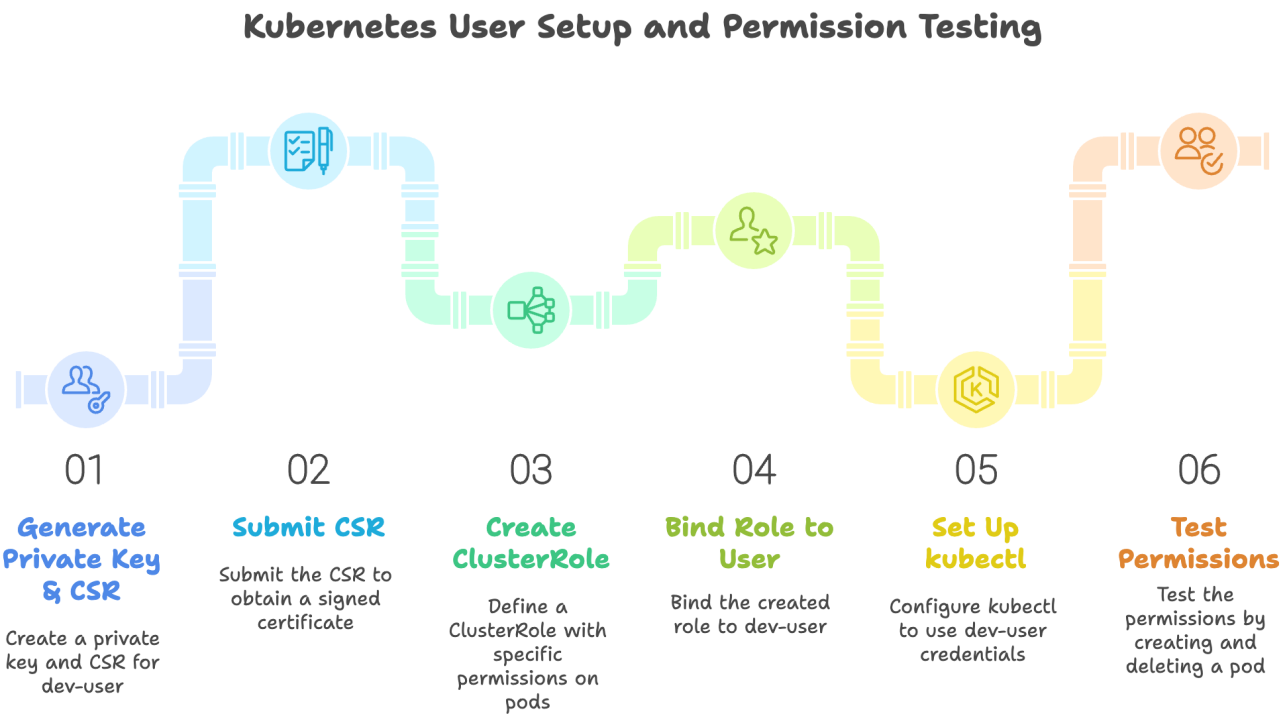
```
  namespace: default
subjects:
- kind: User
  name: "jane"
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io


subjects:
- kind: User
  name: "jane"
  apiGroup: rbac.authorization.k8s.io
- kind: Group
  name: "dev-team"
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount
  name: "app-sa"
  namespace: default
roleRef:
  kind: Role
```

Jane → Pod-reader.

name: pod-reader

apiGroup: rbac.authorization.k8s.io

→ Authentication and Authorisation Demo.



dev user → Authentication.

→ Key

↓

→ CSR → Submit to CA → Certificate.

kube config.

↓

certificate & key (dev-user)

↓ cluster ?

Context.

cluster + user.

→ Security Contexts

Security contexts are settings applied to Pods and containers to define privilege and access control settings.

→ Pod level

→ Container level -

Pod level

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:  # Security settings apply to all containers in the Pod
    runAsUser: 1000
```

```
    runAsGroup: 1000
    fsGroup: 1000
  containers:
    - name: app-container
      image: nginx
```

Container level

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-container
spec:
  containers:
    - name: app-container
      image: nginx
      securityContext:  # Only applies to this container
        runAsUser: 2000
        readOnlyRootFilesystem: true
```

Key Security Context Fields

① runAsUser & runAsGroup.

securityContext:

runAsUser: 1000 # Runs as non-root user with UID 1000

runAsGroup: 3000 # Uses GID 3000

(2) Privileged.

securityContext:

privileged: true # Grants full access to the host

(3) Allow Privilege Escalation.

Blocking Privilege Escalation

securityContext:

allowPrivilegeEscalation: false

(4) Read Only Filesystem.

ecurityContext:

readOnlyRootFilesystem: true

(5) Linux capabilities

```
securityContext:
  capabilities:
    add: ["NET_ADMIN"]  # Allows changing network settings
    drop: ["ALL"]  # Drops all unnecessary capabilities


apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:  # Applies to all containers in the pod
    runAsUser: 1000  # Runs as non-root user
    runAsGroup: 3000 # Group ID for security
    fsGroup: 2000    # Ensures correct file permissions
  containers:
    - name: secure-container
      image: busybox
      securityContext:
        allowPrivilegeEscalation: false  # Prevent privilege escalation
        readOnlyRootFilesystem: true     # Prevents modifying root FS
      command: ["sleep", "3600"]
```

kubectl exec -it secure-pod -- id

kubectl exec -it secure-pod -- sh -c "su root"

kubectl exec -it secure-pod -- touch /testfile