

Report:

Manage network security in K8s cluster



**Supervised by :**

Pr. Amina Bengag

**Presented by :**

Fatima Zahra Lougmiri

Rajae El Khantach

**2025-2026**

**Outline:**

1. Introduction
2. Introduction to Kubernetes
3. Importance of Network Security in a Kubernetes Cluster
4. Solutions to Ensure Security
5. Network Policies
   1. Definition
   2. How They Work

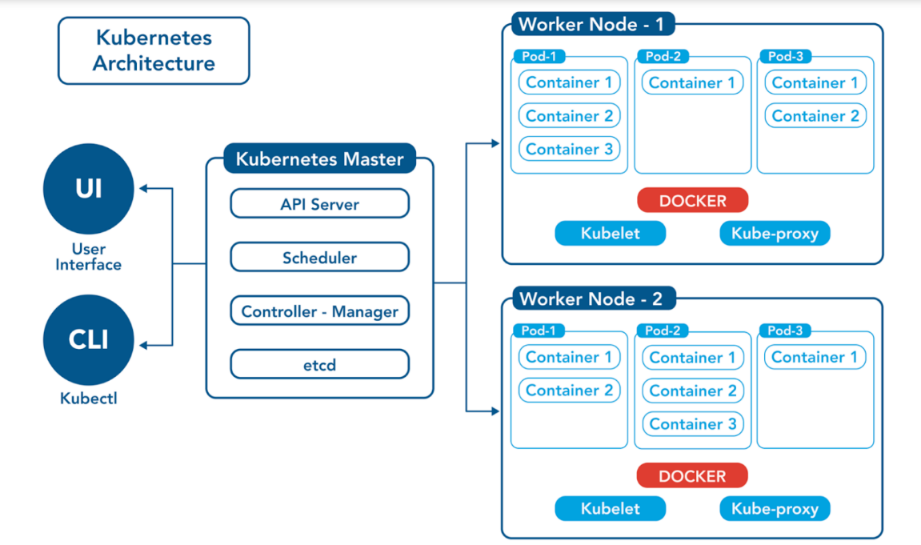
1.3 Demo

1. RBAC
   1. Definiton
   2. How It Works
   3. Demo
2. Conclusion
3. **Introduction :**
4. **Introduction to Kubernetes :**

Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation ([CNCF](https://www.cncf.io/about)).

Before diving into securing a Kubernetes cluster configuration, it is crucial to have a solid understanding of its architecture.

Kubernetes operates on a master-worker architecture. The **Kubernetes Master** manages the cluster, while **Worker Nodes** run the actual applications in containers. Containers are grouped into **Pods**, which are the smallest deployable units in Kubernetes.



**Components of the Architecture:**

User Interaction:

* **UI (User Interface)**: Users can interact with the Kubernetes cluster through a graphical interface, such as the Kubernetes Dashboard.
* **CLI (Kubectl)**: Users can also use the command-line tool kubectl to manage the cluster, issue commands, and deploy applications.

Kubernetes Master:

The Kubernetes Master is the control plane of the cluster, responsible for managing the overall state of the system. It consists of several key components:

* **API Server**: The central hub for all communication. It exposes the Kubernetes API, which the UI, CLI, and other components use to interact with the cluster. It handles requests, validates them, and updates the cluster state.
* **Scheduler**: Responsible for placing Pods onto Worker Nodes. It evaluates the resource requirements of Pods (e.g., CPU, memory) and the current state of the nodes to decide where to run each Pod.
* **Controller-Manager**: Ensures the desired state of the cluster matches the actual state. It runs various controllers, such as the ReplicaSet controller (to maintain the desired number of Pod replicas) and the Node controller (to monitor node health).
* **etcd**: A distributed key-value store that acts as the cluster’s database. It stores the configuration data and the current state of the cluster, ensuring consistency and reliability.

Worker Nodes:

Worker Nodes are the machines (physical or virtual) where the actual workloads (applications) run. Each Worker Node contains the following components:

* **Pods**: The smallest deployable unit in Kubernetes. A Pod can contain one or more containers that share the same network namespace (e.g., IP address) and storage volumes. In the diagram:
  + Worker Node 1 has three Pods (Pod-1, Pod-2, Pod-3), each with multiple containers (e.g., Container 1, Container 2, etc.).
  + Worker Node 2 also has three Pods with similar container setups.
* **Docker**: The container runtime that runs the containers within Pods. While Docker is shown here, Kubernetes supports other runtimes like containerd or CRI-O as well.
* **Kubelet**: An agent that runs on each Worker Node. It communicates with the Kubernetes Master (via the API Server) to ensure that the containers in the Pods are running as expected. It also reports the node’s status back to the Master.
* **Kube-proxy**: Manages networking on the Worker Node. It maintains network rules to enable communication between Pods, Services, and external traffic. It often uses mechanisms like iptables or IPVS to handle traffic routing.
* **How it all works together:**

In Kubernetes, a user interacts with the cluster through the UI or kubectl, sending commands like "deploy an application with 3 replicas" to the API Server. The API Server validates the request and stores the desired state in etcd. The Scheduler then assigns Pods to Worker Nodes based on resource availability and policies. The Controller-Manager ensures the desired state is maintained, such as replacing failed Pods. On each Worker Node, the Kubelet follows the Master’s instructions to manage Pods, Docker runs the containers, and Kube-proxy handles networking to enable communication between Pods and external services.

1. **Importance of Network Security in a Kubernetes Cluster :**

Network security is a critical aspect of managing a Kubernetes cluster, as it ensures the protection of sensitive data, prevents unauthorized access, and maintains the integrity of applications running in the cluster. Kubernetes environments are highly dynamic, with containers constantly being created, scaled, and destroyed, which increases the attack surface if not properly secured. Implementing robust network security measures—such as Network Policies, encryption (TLS/SSL), and role-based access control (RBAC)—helps restrict communication between pods, namespaces, and external systems, reducing the risk of lateral movement by attackers. Additionally, securing the cluster’s ingress and egress traffic prevents data breaches and denial-of-service (DoS) attacks. Without proper network security, Kubernetes clusters are vulnerable to exploits like man-in-the-middle attacks, unauthorized API access, and malicious lateral movement, which can compromise the entire infrastructure. Therefore, integrating strong network security practices is essential for maintaining compliance, ensuring business continuity, and safeguarding cloud-native applications.

1. **Solutions to Ensure Security :**

To strengthen Kubernetes cluster security, a multi-layered approach combining network segmentation and granular access controls is essential. At the namespace level, we will work on Network Policies (NP) and at the cluster level, we will work on Role-Based Access Control (RBAC).

1. **Network Policies :**
   1. **Definition :**

**Network Policies** in Kubernetes are **a set of rules** that define how **pods** can **communicate** with each other and other network endpoints. They act as a **firewall** for cluster traffic, allowing administrators to **control and restrict communication** between pods based on labels, namespaces, or IP blocks. By default, Kubernetes allows all pods to communicate freely; Network Policies enforce segmentation, ensuring only authorized connections **are permitted**. This enhances security by preventing unauthorized lateral movement, reducing the attack surface, and enforcing zero-trust principles within the cluster.

* 1. **How They Work :**

Supposing we have two namespaces, and we want to control the communication between the Pods in each namespace. To achieve this, we must apply a NetworkPolicy to the namespace we want to manage, regulating both the incoming and outgoing traffic. In the figure below, we aim to control both the ingress and egress traffic for the specified namespace.

So let’s understand the structure of the NP:

As initial context, the NetworkPolicy (NP) is defined in a YAML file, which we apply to our namespace. Its structure is as follows:

* Metadata :

 **apiVersion: networking.k8s.io/v1**: Specifies the Kubernetes API version for NetworkPolicy resources.

 **kind:** Indicates that this is a NetworkPolicy resource.

 **metadata**: Contains metadata about the resource.

**-name: test-net-policy**: The name of the NetworkPolicy, used to identify it in the cluster.

**-namespace: default**: The namespace where this policy is applied. Here, it’s the default namespace.

* Spec :

The spec section defines the actual rules of the NetworkPolicy.

 **podSelector**: Identifies the Pods to which this policy applies.

**matchLabels: role: db**: This policy targets Pods with the label role: db. In the Kubernetes architecture, the API Server uses this selector to determine which Pods in the Worker Nodes are affected.

 **policyTypes**: Specifies the types of traffic this policy controls.

* **Ingress**: Controls incoming traffic to the selected Pods.
* **Egress**: Controls outgoing traffic from the selected Pods.

**ingress**: Defines rules for incoming traffic to the Pods selected by podSelector (i.e., Pods with role: db).

 **from**: Specifies the sources allowed to send traffic to these Pods. It has two conditions:

* **ipBlock**: Allows traffic from a specific IP range.
  + **cidr: 172.17.0.0/16**: Permits traffic from IPs in the range 172.17.0.0 to 172.17.255.255.
  + **except: 172.17.1.0/24**: Excludes the subnet 172.17.1.0 to 172.17.1.255 from the allowed range. So, traffic from 172.17.1.0/24 is blocked, but the rest of 172.17.0.0/16 is allowed.
* **namespaceSelector**: Allows traffic from Pods in namespaces that match certain labels.
  + **matchLabels: project: myproject**: Permits traffic from Pods in namespaces labeled project: myproject. In the architecture, this would involve Kube-proxy on the Worker Nodes enforcing these rules to route traffic between namespaces.

 **ports**: Specifies which ports and protocols are allowed for incoming traffic.

* **protocol: TCP**: Allows TCP traffic.
* **port: 6379**: Allows traffic on port 6379 (commonly used by Redis)

**egress**: Defines rules for outgoing traffic from the Pods selected by podSelector (i.e., Pods with role: db).

 **to**: Specifies the destinations to which these Pods can send traffic.

-**ipBlock: cidr: 10.0.0.0/24**: Allows traffic to IPs in the range 10.0.0.0 to 10.0.0.255.

 **ports**: Specifies which ports and protocols are allowed for outgoing traffic.

* **protocol: TCP**: Allows TCP traffic.
* **port: 5978**: Allows traffic on port 5978.
* Pods with the label role: db can receive TCP traffic on port 6379 from either:

-IPs in the range 172.17.0.0/16 (except the subnet 172.17.1.0/24), OR

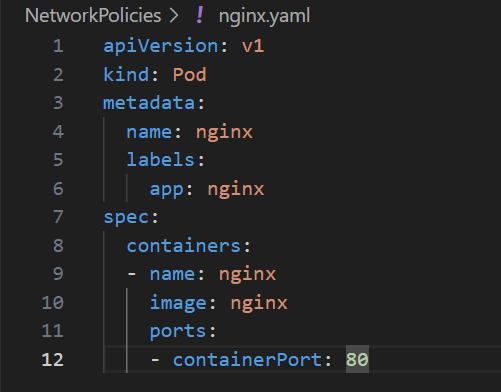
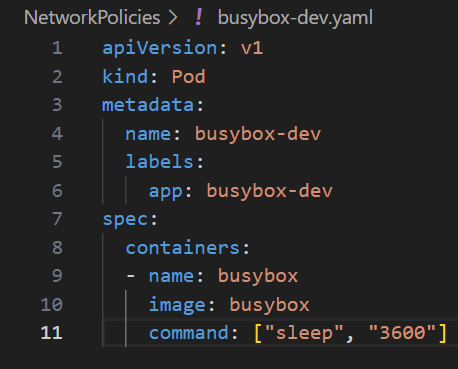
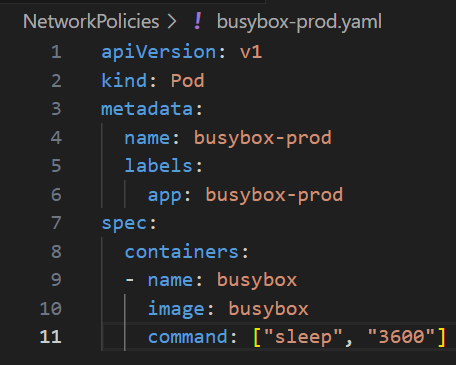
-Pods in namespaces labeled project: myproject.

This OR logic applies because the from section in the NetworkPolicy YAML lists these conditions as separate entries under ingress, each prefixed with a -, which Kubernetes interprets as allowing traffic if any of the conditions are met.

Additionally, Pods with role: db can send TCP traffic on port 5978 to IPs in the range 10.0.0.0/24.

* 1. **Demo :**

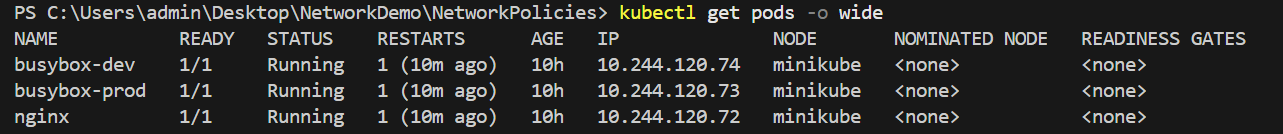
To begin, I will create resources to test connectivity. I will set up a Pod for an Nginx server and two Busybox Pods—one for the development environment and another for production—using YAML configuration files.

To deploy these Pods, I will apply the YAML files to the Kubernetes cluster using the following command:

**“kubectl apply -f nginx.yaml -f busybox-dev.yaml -f busybox-prod.yaml ”**

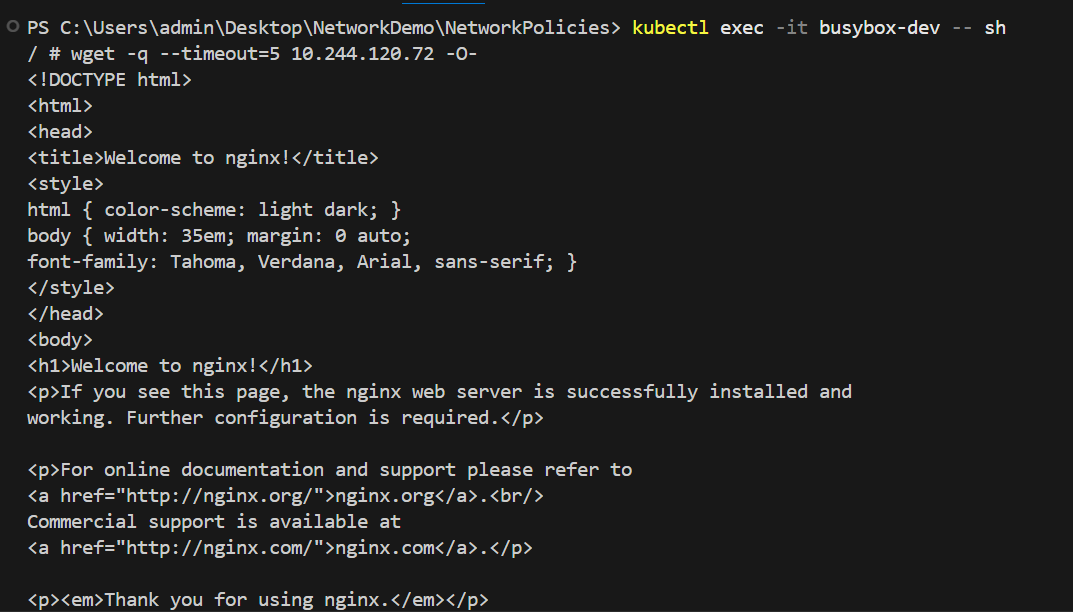
Now we will verify that they are running correctly: (STATUS: RUNNING)



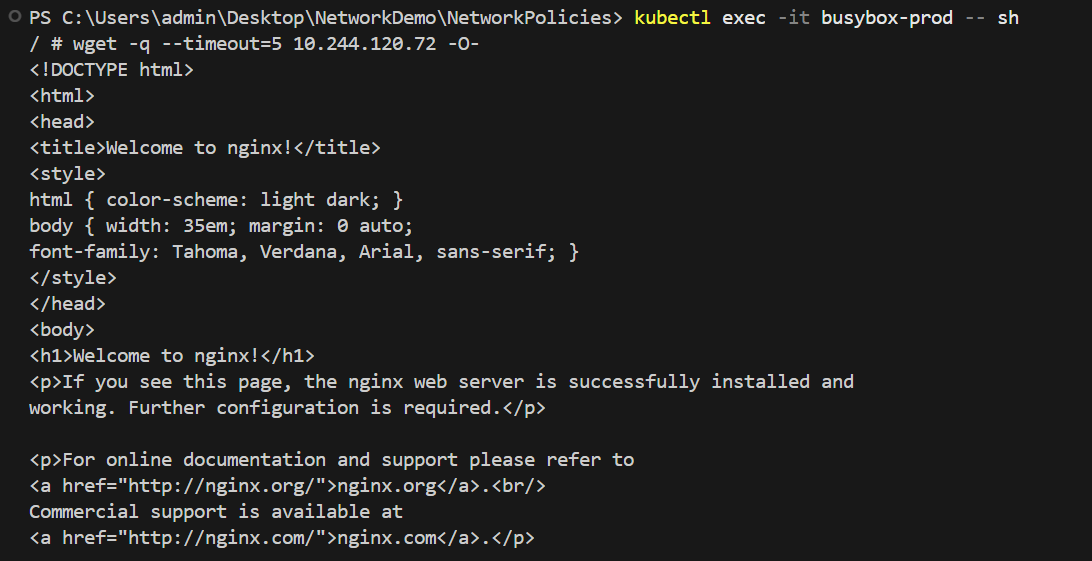
And before implementing network policies, I will test connectivity from the two Busybox Pods (development and production) to the Nginx server Pod.

***Step1***: I will access to busybox-dev pod by the commend “kubectl exec –it busybox-dev – sh”

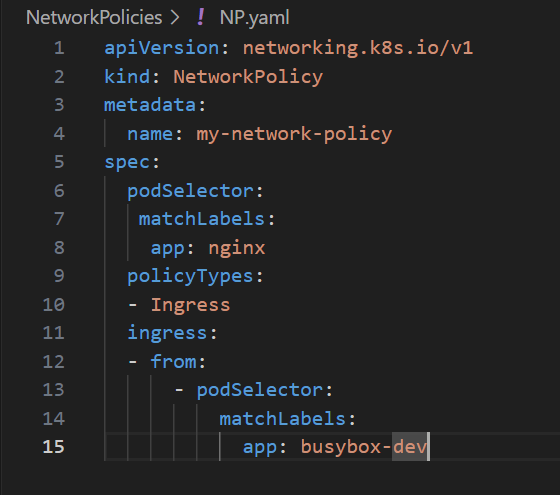
***Step2***: From within the Busybox Pod, test connectivity to the Nginx server using: “*wget –q –timeout=5 <IP-of-Nginxserver>*”

****

***Result***: The connectivity test was successful ✅

And the same result for busybox-prod:

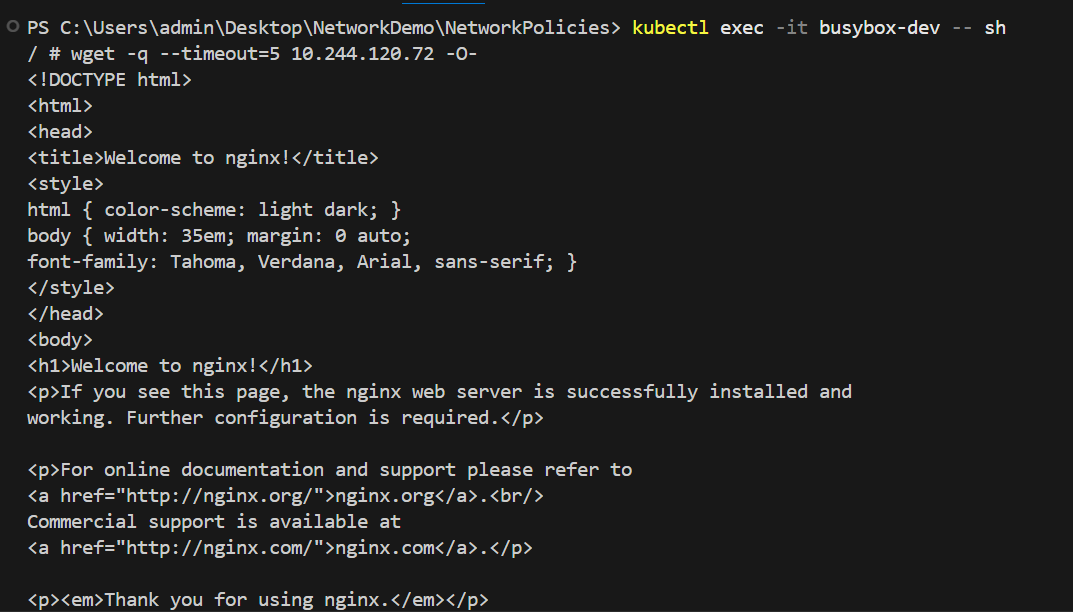
To control network traffic, We will create a Network Policy that permits only the busybox-dev Pod to connect and send traffic to the Nginx server Pod, denying access from other sources, including the busybox-prod Pod. This policy will be defined in a YAML file named **NP.yaml**



Then apply this Network Policy to our cluster:

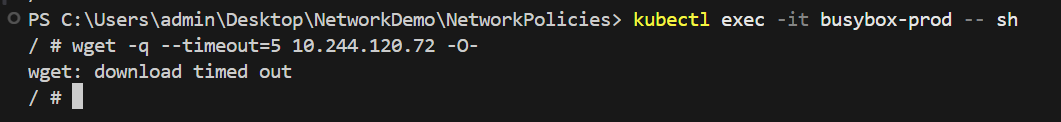


Try to connect to nginx server again from busybox-dev Pod :



***Result***: The connectivity test was successful ✅

I will attempt to connect from the busybox-prod Pod:



***Result***: The connectivity test was unsuccessful ❌, confirming the Network Policy restricts access as intended ✅

1. **RBAC :**
   1. **Definition :**

RBAC, or **Role-Based Access Control**, in the context of Kubernetes, is a method for regulating access to resources within a cluster based on the roles assigned to users, groups, or service accounts. It allows administrators to define permissions through roles and bind those roles to specific entities, ensuring that only authorized users or processes can perform actions like creating, modifying, or deleting resources (e.g., Pods, Services, or Deployments).

In Kubernetes, Role-Based Access Control (RBAC) assigns permissions to entities like **users**, **groups**, and **service accounts** to control access to cluster resources.

* **A user** in Kubernetes represents a human or external system interacting with the cluster, typically via tools like kubectl or the Kubernetes Dashboard .

Kubernetes does not manage users directly; it relies on external systems for authentication (e.g., X.509 certificates, bearer tokens..). Users are identified by their credentials, which are presented to the **API Server** when making requests.

* **A group** is a collection of users, used to simplify permission management by assigning roles to multiple users at once.

Like users, groups are not managed by Kubernetes directly; they are defined by the external authentication system. Groups are referenced in RBAC RoleBindings or ClusterRoleBindings (explained in the next axis) to grant permissions to all users within the group.

* **A service account** is a special type of account in Kubernetes used by processes (e.g., Pods or applications) running inside the cluster to authenticate to the API Server and interact with cluster resources. It’s managed directly by Kubernetes and is namespaced resources.

Service accounts are created automatically when a namespace is created (e.g., the default service account in each namespace) or can be created manually.

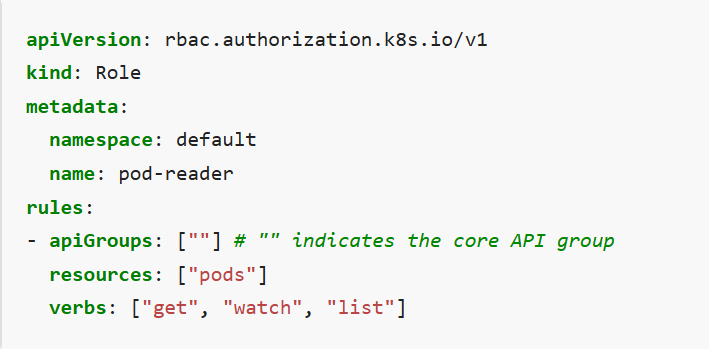
* 1. **How It Works :**

The Key Components of RBAC in Kubernetes are:

1. **Role**: Defines a set of permissions (rules) for specific actions (e.g., "get," "list," "create") on resources within a namespace.
2. **RoleBinding**: Assigns a Role to a user, group, or service account within a namespace.
3. **ClusterRole**: Similar to a Role but applies cluster-wide, not limited to a specific namespace.
4. **ClusterRoleBinding**: Assigns a ClusterRole to a user, group, or service account across the entire cluster.

How RBAC Works in the Kubernetes Architecture:

* The **API Server** (from the architecture diagram) enforces RBAC policies. When a request is made (e.g., via kubectl or the UI), the API Server checks the requester’s identity and their assigned roles to determine if the action is allowed.
* RBAC policies are stored in etcd and managed by the **Controller-Manager**, ensuring consistent enforcement across the cluster

**The structure of Role file:**

apiVersion: Specifies the API version of the RBAC resource being defined.

kind: Role : Defines the type of Kubernetes resource being created.

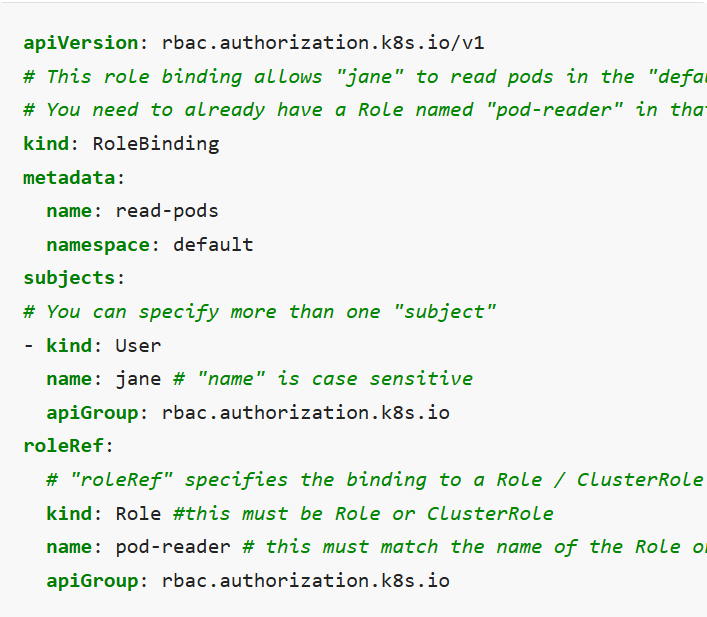
namespace: default : Specifies the namespace in which this Role is created.

name: pod-reader : The name of the Role, used to identify it within the namespace.

rules: Defines the permissions (rules) that this Role grants.

- apiGroups: [""] : Specifies the API group of the resources this rule applies to. An empty string [""] (refers to the core Kubernetes API group, which includes resources like Pods, Services, and ConfigMaps.)

resources: Lists the specific resources this rule applies to.

verbs: Specifies the actions (verbs) that are allowed on the resources.**** **The structure of RoleBuilding file:**

apiVersion: rbac.authorization.k8s.io/v1: Specifies the API version of the RBAC resource being defined.

kind: RoleBinding : Defines the type of Kubernetes resource being created.

name: read-pods : The name of the RoleBinding, used to identify it within the namespace.

namespace: default : Specifies the namespace in which this RoleBinding is created.

subjects: A list of entities (subjects) to whom the Role’s permissions will be granted.

- kind: User : Specifies the type of subject being bound to the Role.(User, group or service account)

name: jane : The name of the user being granted the permissions.

apiGroup: rbac.authorization.k8s.io : Specifies the API group for the subject.

roleRef: A section that references the Role (or ClusterRole) being bound to the subject. It specifies which Role’s permissions will be granted.

kind: Role : Specifies the type of role being referenced.

name: pod-reader : The name of the Role being referenced.

apiGroup: rbac.authorization.k8s.io :Specifies the API group of the Role being referenced.

* 1. **Demonstration**

In this demo, We will showcase Role-Based Access Control (RBAC) in Kubernetes by creating a namespace test, a ServiceAccount, and verifying permissions for resources in that namespace. I will then apply a Role and RoleBinding to grant specific permissions and confirm the changes.

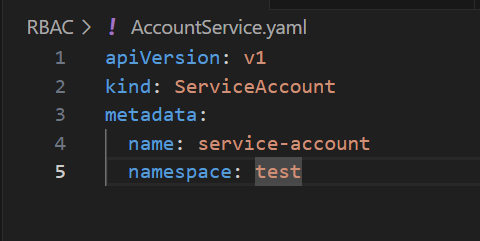
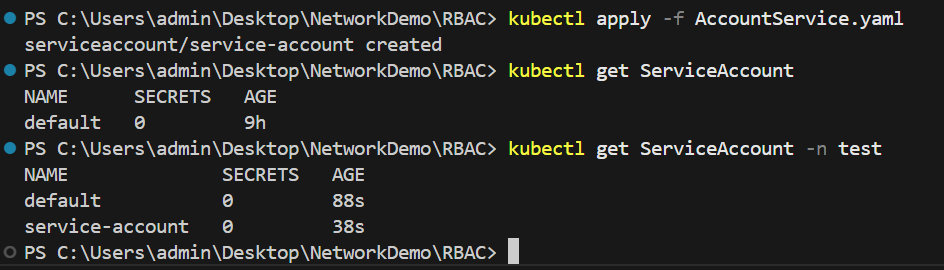
***Step 1:*** Create the Namespace “test”

I will create a namespace named test to isolate the resources for this demo via this command:

***“kubectl create namespace test”***

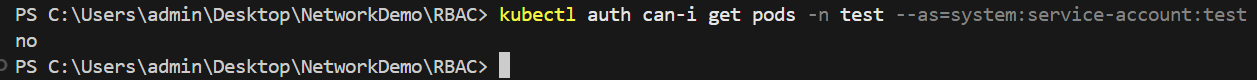
***Step 2:*** Create a ServiceAccount

Next, I will create a ServiceAccount named service-account in the test namespace via a YAML file and apply it to deploy our Service Account. With verification

***Step 3 :*** Verify Initial Permissions

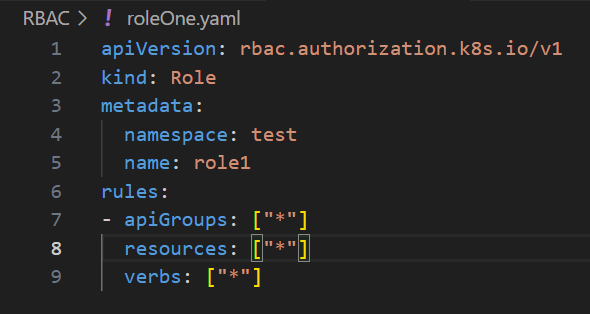
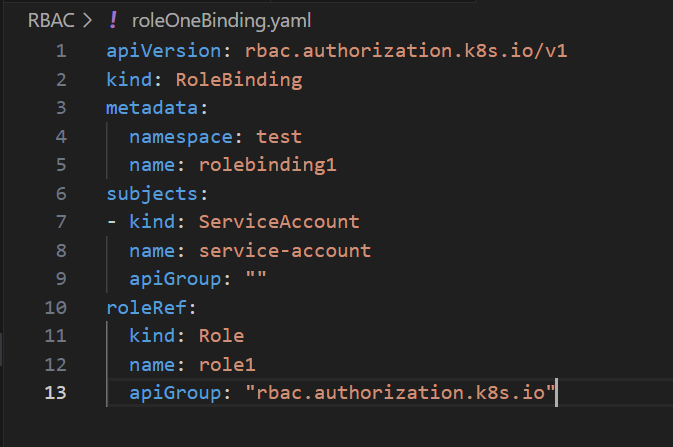
We will check if the test-user service-account has permissions to manage resources (e.g. Pods) in the test namespace using “kubectl auth can-i”



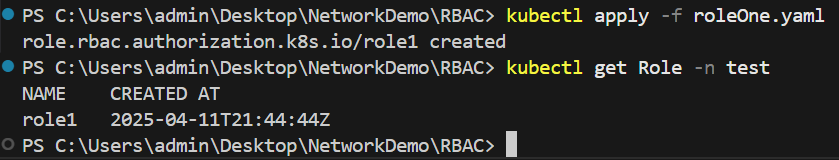
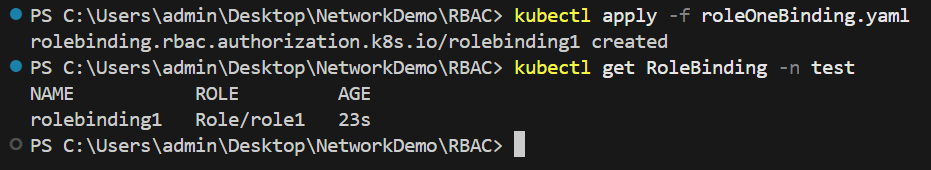
**Result**: The ServiceAccount has no permissions to get Pods ❌

***Step 4 :*** Create a Role and RoleBinding

To grant permissions, We will create a Role named role1 in the test namespace, allowing all actions on test namespace ressources. And bind this Role to the our ServiceAccount using a RoleBinding named rolebinding1:

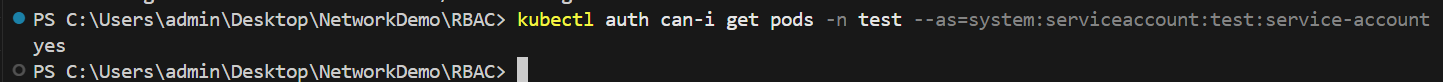
 

Apply the Role and RoleBinding :

***Step 6 :*** Verify Permissions After RBAC

We will recheck the ServiceAccount’s permissions to confirm it can now access to resources in the test namespace



**Result**: The ServiceAccount : service-account can now get Pods from test namespace (can do all actions to all resources in test namespace) ✅

1. **Conclusion :**

Over the years, containerization has transformed software deployment, with Kubernetes emerging as a leading platform for orchestrating containers. Its ability to manage complex workloads—such as microservices, auto-scaling, and dynamic orchestration—makes it a preferred choice for modern DevOps teams. However, as Kubernetes adoption grows, so do security challenges, particularly in managing network traffic and API access. Default configurations often leave clusters vulnerable, exposing pods, containers, and databases to risks. This report addresses these concerns by implementing security practices tailored to a Kubernetes cluster. we focus on two key areas: Network Policies to control pod-to-pod communication ( isolating busybox-prod from nginx while allowing busybox-dev) and RBAC to secure API interactions (granting a ServiceAccount in test specific permissions). Through this work, We aim to demonstrate how to secure a Kubernetes cluster effectively, providing insights that can guide further research and practical implementations in Kubernetes security.