Politecnico di Torino

Authentication and Authorization in Kubernetes

Trad Gianfranco s323713

Networks, Cloud and Application Security

September 2024

Contents

Chapter 1

Introduction to Kubernetes

Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications. It abstracts the underlying infrastructure, allowing developers to focus on building applications without worrying about the complexities of managing the underlying hardware or cloud environment.

The name Kubernetes comes from the Greek word **κυβερνήτης**, which means helmsman or ship pilot. With this analogy in mind, we can think of Kubernetes as the pilot on a ship of containers, steering and managing them as they move through their lifecycle.

In recent years, Kubernetes has become a leading platform for orchestrating containerized applications due to its flexibility, scalability, and robust community support. However, with great power comes the responsibility to secure these environments, especially as they often manage critical infrastructure and sensitive data.

In a microservices architecture, applications are typically broken down into smaller, loosely coupled services that communicate with each other over a network. While this approach offers greater flexibility and scalability, it also introduces new security challenges. Multiple services need to communicate securely while ensuring that only authorized entities can access certain resources.

To address these challenges, Kubernetes provides several mechanisms for handling authentication and authorization, ensuring that only legitimate users and services can interact with the cluster and its resources. This lab will explore these mechanisms in detail, offering practical exercises to understand and implement them effectively.

1.1 Kubernetes Architecture

Kubernetes is built on a modular architecture that ensures flexibility, scalability, and ease of management. The architecture is composed of two main parts: the **control plane** and the **worker nodes**. The control plane is responsible for managing the entire cluster, while the worker nodes run the containerized applications.

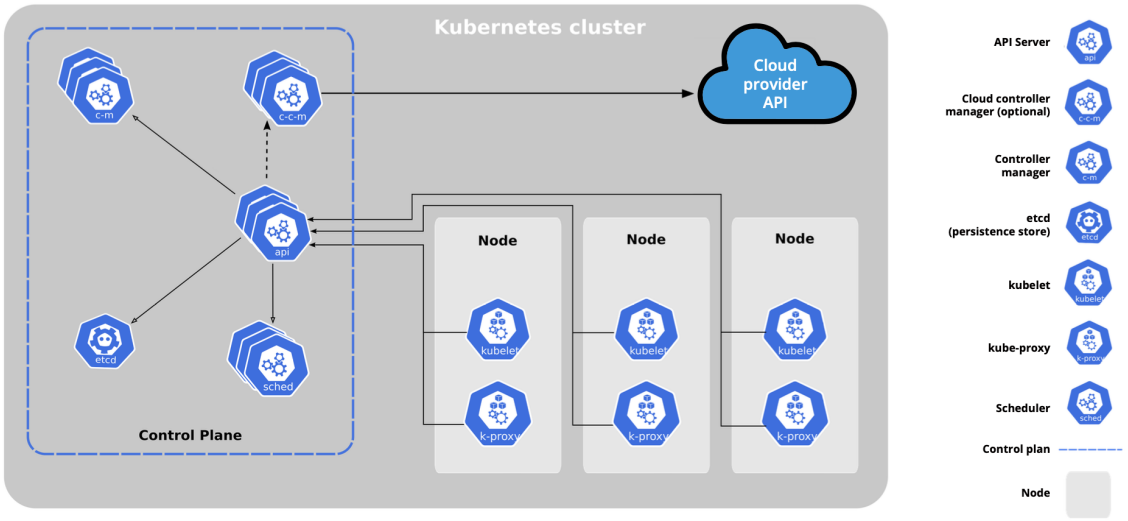

Figure 1.1: Kubernetes Architecture[1]

1.1.1 Control Plane Components

The control plane node provides a running environment for the control plane agents responsible for managing the overall state of the Kubernetes cluster, and it is the brain behind all operations inside the cluster. The control plane components are agents with very distinct roles in the cluster's management. In order to communicate with the Kubernetes cluster, users send requests to the control plane via a *Command Line Interface* (CLI) tool, a *Web User-Interface* (Web UI) Dashboard, or an *Application Programming Interface* (API).

The **key components** of the control plane include:

Kube-API Server

The Kubernetes API server is the central management component of the Kubernetes control plane, serving as the entry point for all administrative tasks within the cluster. It exposes the Kubernetes API, which is used by users, administrators, developers, operators, and external agents to interact with and manage the cluster. The API server intercepts RESTful calls from these entities, validates them, and processes the requests.

During the processing of API requests, the API server reads the current state of the Kubernetes cluster from the key-value store (**etcd**). After a request is executed, the API server updates the key-value store with the resulting state of the cluster, ensuring persistence.
It is worth noting that the **API server is the only control plane component that communicates directly with etcd**, acting as an interface for other control plane agents to inquire about or modify the cluster's state.

The API server is responsible for several critical functions:

- **Authentication and Authorization:** The API server manages authentication using different methods such as *client certificates*, *bearer tokens*, and HTTP Basic Authentication. It also evaluates authorization requests through mechanisms like *Attribute-Based Access Control* (ABAC) and *Role-Based Access Control* (RBAC).

[1] https://kubernetes.io/docs/concepts/overview/components/

- **API Management and Processing:** Once the API requests went through the authentication and authorization checks, if defined, the API server exposes the cluster API endpoint and handles all incoming API requests. Then, the API server processes all API requests and validates data for Kubernetes objects, such as pods and services.
- **Interaction with etcd:** The API server is the only component in the Kubernetes architecture that directly communicates with etcd, the distributed key-value store that holds the entire configuration and state of the cluster. It reads from and writes to etcd to persist the cluster's state.

AuthN & AuthZ Implications: Since the API server is the primary interface for interacting with the Kubernetes cluster, it is **critical to secure it with strong authentication and authorization mechanisms**. Unauthorized access to the API server could compromise the entire cluster, as it is responsible for processing all administrative operations and interacting with **etcd**. To reduce the attack surface of the cluster, securing the API server is essential.

Important Security Insight

A study by the Shadowserver Foundation in 2022 revealed that over 380,000 Kubernetes API servers were publicly accessible, underscoring the importance of securing this critical component[a].

[a] https://www.shadowserver.org/news/over-380-000-open-kubernetes-api-servers/

Scheduler

The kube-scheduler is responsible for assigning newly created workload objects, such as pods, to appropriate worker nodes within the cluster. During the scheduling process, the scheduler evaluates the current state of the Kubernetes cluster and the specific requirements of the new workload. It retrieves resource usage data and node information from the key-value store (etcd) via the API Server. Based on the scheduling parameters it selects the most suitable node for hosting the new workload. Once a decision is made, the scheduler communicates the selected node back to the API Server, which delegates the workload deployment to other control plane components.

AuthN & AuthZ Implications: Although the scheduler does not directly expose an API for external interaction, securing the interaction with it is essential for the cluster's stability and integrity. Authorization mechanisms ensure that only authorized control plane components can interact with the scheduler, **via the kube-API server**, preventing unauthorized modifications or disruptions in workload distribution.

Key-Value Data Store (etcd)

etcd is a strongly consistent, distributed key-value data store used to persist a Kubernetes cluster's state. It stores the entire configuration and state of the Kubernetes cluster, making it a critical component. As previously stated, out of all the control plane components, only the API Server is able to communicate with the etcd data store.

AuthN & AuthZ Implications: Since **etcd** contains the source of truth for the entire cluster, securing its interactions with the API server by providing Authentication and Authorization properties is of the utmost importance.

Controller Manager

The controller managers are components of the control plane node running controllers or operator processes to regulate the state of the Kubernetes cluster. Controllers are watch-loop processes continuously running and comparing the cluster's desired state (provided by objects' configuration data) with its current state (obtained from the key-value store via the API Server). In case of a mismatch, corrective action is taken in the cluster until its current state matches the desired state. Examples of controllers include the replication controller, which ensures the correct number of pod replicas, and the node controller, which monitors the status of nodes.

1.1.2 Worker Node Components

A worker node provides a running environment for client applications. These applications are microservices running as application containers. In Kubernetes the application containers are encapsulated in **Pods**, controlled by the cluster control plane agents running on the **control plane** node. Pods are scheduled on worker nodes, where they find required computing, memory and storage resources to run, and networking to communicate to each other and the outside world. A **Pod** is the smallest scheduling work unit in Kubernetes. It is a logical collection of one or more containers scheduled together, and the collection can be started, stopped, or rescheduled as a single unit of work. Each worker node includes the following components:

Kubelet

The kubelet is an agent running on each node, control plane and workers, and it communicates with the control plane. It receives Pod definitions, primarily from the API Server, and interacts with the container runtime on the node to run containers associated with the Pod. It also monitors the health and resources of Pods running containers.

AuthN & AuthZ Implications: The kubelet must authenticate requests from the API server to ensure that it only acts on valid and authorized instructions. Unauthorized access to the kubelet could allow an attacker to disrupt or control the container or image operations running on the pods.

Kube-proxy

Kube-proxy is a network proxy that runs on each worker node, maintaining network rules that allow communication between pods and services. It forwards traffic to the appropriate containers based on network policies.

Container Runtime

Although Kubernetes is described as a *container orchestration engine*, it lacks the capability to directly handle and run containers. In order to manage a container's lifecycle, Kubernetes requires a container runtime on the node where a Pod and its containers are to be scheduled. A runtime is required on each node of a Kubernetes cluster, both control plane and worker.
Kubernetes supports several container runtimes, such as *Docker*, *containerd*, and *CRI-O*.

1.2 Key Takeaways

Key Takeaways

- **API Server:** The Kubernetes API server is the central access point for all administrative tasks and manages communication with other control plane components. Since it processes and validates RESTful requests, securing the API server with robust **authentication and authorization** is crucial. Without these, unauthorized access could compromise the entire cluster's security, integrity and availability.
- **Why Authentication is needed:** Authentication ensures that only legitimate users, services, and components can communicate with the API server.
- **Why Authorization is needed:** Authorization controls what authenticated users and components can do within the cluster. Even after authentication, the API server must enforce **Role-Based Access Control (RBAC)** or other methods to ensure that users can only perform authorized actions, such as creating or deleting resources.
- **Etcd and Kubelet:** Both **etcd** and **kubelet** are managed via the **API server**. Securing API server requests with authentication and authorization is crucial for **etcd**, as it is the source of truth for the cluster, and for **kubelet**, as it receives pod definitions from the API server for workload execution.

Chapter 2

Authentication and Authorization in Kubernetes

As previously explained, Kubernetes is an API-centric platform where every request, whether from cluster components or users, must go through the API server. Hence, the API server is the central control point, acting as a gatekeeper, ensuring that all incoming requests are authenticated (AuthN) and authorized (AuthZ) before being processed.
When a request reaches the **Kube-API server**, it goes through 3 stages: **authentication**, **authorization** and **admission control**, which are illustrated in the following diagram:

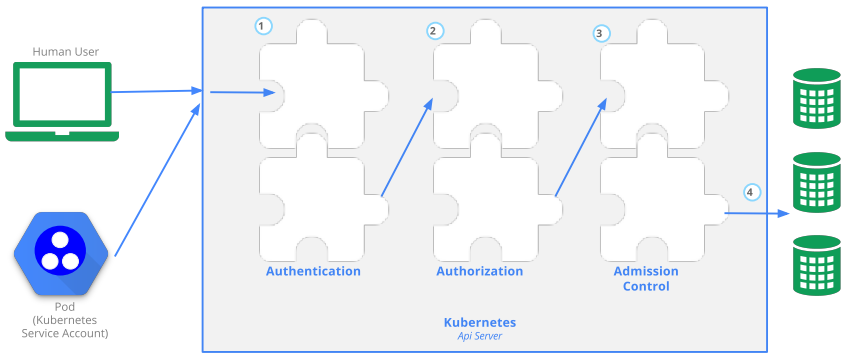

Figure 2.1: Kubernetes Authentication, Authorization, and Admission Control Process[2]

Kubernetes Users

In Kubernetes, there are two types of users:

- **Normal users**: Normal users are managed outside the cluster by independent services, such as user/client certificates, external identity providers (e.g., Google Accounts), or files with username-password pairs. Kubernetes does not store or manage normal user accounts internally, nor does it have API objects representing them. Instead, as a possible authentication mode, any user presenting a valid certificate signed by the **cluster**'s Certificate Authority (CA) is considered authenticated.
- **Service accounts**: Service accounts are managed within the Kubernetes cluster and are used to allow in-cluster processes, such as pods, to communicate with the API server. Each service account is tied to a specific *namespace*, and its credentials are mounted as Secrets in the corresponding pods, allowing authentication to the API server.

[2] https://kubernetes.io/docs/concepts/security/controlling-access/

2.1 Authentication in Kubernetes

Authentication is defined as the process of validating or verifying that a user or entity is who (or what) they claim to be.

2.1.1 Kubernetes Authentication Modes

Kubernetes supports multiple authentication modes[3], such as:

- **X.509 Client Certificates:** Users authenticate using X.509 client certificates that are validated by the API server against a certificate authority (CA). The API server is configured with the CA file using the `--client-ca-file=SOMEFILE` flag. Once validated, the subject's `commonName` is used as the username, and the organizations in the certificate are used as user groups. This method requires administrators to manage, distribute, and renew certificates for users.
- **Static Token File:** In this method, a static *csv* file containing predefined bearer tokens is passed to the API server using the `--token-auth-file=SOMEFILE` flag. Each entry in the *csv* file must contain a minimum of 3 columns: *token, user-name, user-id.*
  Currently with this approach, tokens last indefinitely, and the token list cannot be changed without restarting the API server.
- **Service Account Tokens:** Service accounts are tied to in-cluster processes, such as pods, and allow them to authenticate with the API server. Service account tokens are automatically issued and signed by the API server. These tokens are mounted into pods as secrets, enabling them to communicate with the API server. This method is widely used for automating in-cluster communications.
- **OpenID Connect (OIDC) Tokens:** OpenID Connect (OIDC) allows Kubernetes to integrate with external *OAuth2* identity providers such as Google, AWS, and Azure Active Directory. The API server uses ID tokens returned by these providers to authenticate users. The `ID token`, in JSON Web Token (JWT) format, acts as a bearer token for authentication.
- **Bootstrap Tokens:** Bootstrap tokens are used to authenticate nodes when creating new clusters or joining existing ones. These tokens are dynamically generated and managed, stored as secrets in the `kube-system` namespace.

Other possible authentication methods are **Webhook Token Authentication**, where the API server offloads token verification to a remote service via a webhook, and resorting to an external **Authenticating proxy** to handle authentication.

Recommended Authentication Methods

It is recommended to use at least two authentication methods in Kubernetes[3]:

- **Service Account Tokens** mostly deployed in automatic for service accounts, allowing in-cluster processes such as pods to communicate with the API server.
- At least one other method for user authentication.

For this laboratory, the user authentication method selected will be **X.509 client certificates**.

[3] https://kubernetes.io/docs/reference/access-authn-authz/authentication/#authentication-strategies

2.2 Authorization in Kubernetes

Authorization is defined as the process of verifying that a requested action or service is approved for a specific entity.
After a successful authentication, Kubernetes authorization of API requests takes place within the API server. All parts of an API request must be allowed by some authorization mechanism in order to proceed. In other words: access is denied by default.
It is worth noting, that the API request verbs for a resource are mapped to the HTTP methods as follows:

HTTP verb	request verb
POST	**create**
GET , HEAD	**get** (for individual resources), **list** (for collections, including full object content), **watch** (for watching an individual resource or collection of resources)
PUT	**update**
PATCH	**patch**
DELETE	**delete** (for individual resources), **deletecollection** (for collections)

Figure 2.2: Resource request verb mapping[4]

2.2.1 Kubernetes Authorization Modes

Kubernetes supports several authorization modes[4], which can be even used together in a single cluster:

- **Role-Based Access Control**: also known as **RBAC**, is the most widely used authorization mechanism in Kubernetes. It regulates access to resources based on the roles assigned to subjects, such as users and service accounts. Roles define specific operations that a subject is permitted to perform on resources, such as create, get, update, or patch. These operations are referred to as verbs. In RBAC, two types of roles can be created:
  - **Role:** Grants access to resources within a specific namespace.
  - **ClusterRole:** Similar to Role but grants permissions across the entire cluster.

  The following example demonstrates a Role definition:

  Example: Role Definition

  ```
  apiVersion: rbac.authorization.k8s.io/v1
  kind: Role
  metadata:
    namespace: ncas-lab
    name: pod-reader
  rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
  ```

  This manifest defines a role named `pod-reader`, which is granted read-only access verbs (get, watch and list) to pods within the `ncas-lab` namespace.

[4] https://kubernetes.io/docs/reference/access-authn-authz/authorization/

For comparison, below is an example of a ClusterRole:

Example: ClusterRole Definition

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-admin
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
- nonResourceURLs: ["*"]
  verbs: ["*"]
```

This manifest defines a fully permissive ClusterRole named `cluster-admin`, which grants full access to all resources and non-resource URLs across the entire cluster.

Once a role is created, it must be bound to users or service accounts using a `RoleBinding` or `ClusterRoleBinding`. There are two types of bindings:

- **RoleBinding:** Binds users or service accounts to a role within a specific namespace. It can also reference ClusterRoles to grant permissions to namespace resources.
- **ClusterRoleBinding:** Binds users or service accounts to ClusterRoles, granting access to resources at the cluster level and across all namespaces.

The following example demonstrates a RoleBinding:

Example: RoleBinding Definition

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-read-access
  namespace: ncas-lab
subjects:
- kind: User
  name: bob
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

This manifest defines a `RoleBinding` that binds the `pod-reader` role to the user `bob`, restricting the user's access to read pods in the `ncas-lab` namespace.

For comparison, a ClusterRoleBinding example is provided below:

Example: ClusterRoleBinding Definition

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:admins
```

This manifest defines a `ClusterRoleBinding` that grants full cluster-wide access to all members of the `system:admins` group.

Enabling RBAC Mode

To enable RBAC mode, the API server must be started with the `--authorization-mode=RBAC` flag.
Since RBAC is widely used and well-documented, it will be the chosen authorization method for this laboratory.

- **Attribute-Based Access Control (ABAC):** ABAC provides access control based on attributes such as the user, resource, and environment. Policies are written in JSON format and specify what operations a user can perform based on their attributes. ABAC is enabled using the `--authorization-mode=ABAC` flag and the `--authorization-policy-file=SOME_FILENAME` flag.

Example: ABAC policy

```
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
  "kind": "Policy",
  "spec": {
    "user": "bob",
    "namespace": "ncas-lab",
    "resource": "pods",
    "readonly": true
    }
}
```

In this policy, the user `bob` can only read `Pods` in the namespace `ncas-lab`.

- **Node Authorization:** Node authorization is a special-purpose authorization mode used for kubelets. It authorizes the kubelet to perform read and write operations for services, pods, and nodes. Kubelets authenticate as part of the `system:nodes` group and are assigned permissions based on their node identity, ensuring that each node can only access or modify resources that are relevant to it. This mode is enabled with the `--authorization-mode=Node` flag.

Other supported Authorization modes by Kubernetes are the **AlwaysAllow** mode, that allows all requests (to be used only if no authorization for API requests are required), the **AlwaysDeny** mode that blocks all requests (mostly used for testing), and the **WebHook** mode.

2.3 Admission Control in Kubernetes

Admission controllers[5] enforce **granular access control policies** after a request is authenticated and authorized.
These controllers can be **mutating** or **validating**:

- **Mutating controllers**: modify objects related to the API request
- **Validating controllers**: ensure the objects meet certain conditions, in order to fulfill the request.

Admission controllers can also restrict requests that create, delete, or modify objects.

To enable admission controllers, the API server must be started with the `--enable-admission-plugins` flag, which takes a comma-delimited list of controller names. For example:

Example: admission controller

```
--enable-admission-plugins=NamespaceLifecycle,ResourceQuota,PodSecurity
```

The admission control process occurs in two phases:

1. First, mutating controllers are executed
2. Then, validating controllers are applied

If any controller rejects the request in either phase, the request is **denied**.

The main reasons why **Admission Controllers** need to be enforced are:

- **Security**: Controllers can, for instance, validate container images upon deployment to ensure they adhere to predefined security standards. This validation can include checks for trusted sources and vulnerability scans.
- **Compliance**: Controllers can, for instance, validate incoming requests against predefined policies, ensuring that deployments, configurations, and resource allocations comply with organizational standards.

2.3.1 Examples of Kubernetes Admission Controllers

- **NamespaceLifecycle:** Ensures that requests do not manipulate namespaces in invalid ways (e.g., preventing deletion of system namespaces like `default`, `kube-system`, etc.).
- **ResourceQuota:** Enforces quotas on resources (e.g., CPU, memory) for namespaces, ensuring that resource limits are not exceeded.
- **LimitRanger:** Automatically sets default resource requests and limits for pods if none are specified, helping control resource consumption.
- **PodSecurity:** Enforces Pod Security Standards (PSS) by validating pod security configurations such as privilege and container capabilities.
- **CertificateApproval:** Handles the approval of certificate signing requests (CSRs) within Kubernetes. It intercepts CSRs and determines whether they should be approved, denied, or left for manual intervention by a cluster administrator.

As of writing, in Kubernetes 1.31, the admission controllers enabled by default are[5]:
`CertificateApproval`, `CertificateSigning`, `CertificateSubjectRestriction`, `DefaultIngressClass`, `DefaultStorageClass`, `DefaultTolerationSeconds`, `LimitRanger`, `MutatingAdmissionWebhook`, `NamespaceLifecycle`, `PersistentVolumeClaimResize`, `PodSecurity`, `Priority`, `ResourceQuota`, `RuntimeClass`, `ServiceAccount`, `StorageObjectInUseProtection`, `TaintNodesByCondition`, `ValidatingAdmissionPolicy`, `ValidatingAdmissionWebhook`.

[5] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

2.4 Key Takeaways

Key Takeaways

Each request when reaching the Kube-API server goes through 3 stages: **authentication**, **authorization** and **admission control**.

1. Authentication:
Authentication verifies the identity of users or services interacting with the API server. Kubernetes supports multiple authentication methods such as:

- **X.509 Client Certificates:** Validates client certificates against a Certificate Authority (CA).
- **Service Account Tokens:** Used for in-cluster processes, like pods, to authenticate with the API server.
- **OpenID Connect (OIDC):** Allows integration with external OAuth2 identity providers like Google and AWS.

It is recommended to use at least two authentication methods, with service account tokens as the go-to solution for in-cluster processes.

2. Authorization:
Once authenticated, requests need to be authorized to ensure users or services have permissions for specific actions.
Kubernetes supports several authorization modes, including:

- **Role-Based Access Control (RBAC):** Regulates access to resources based on roles assigned to users and service accounts.
- **Attribute-Based Access Control (ABAC):** Grants access based on user attributes and policies.
- **Node Authorization:** Grants kubelets specific permissions to read and write pod, service, and node data.

3. Admission Control:
Admission controllers enforce granular policies after a request is authenticated and authorized. They can be:

- **Mutating Controllers:** Modify objects related to requests before persistence.
- **Validating Controllers:** Ensure objects meet certain conditions without modifying them.

If any controller rejects the request, even after passing authentication and authorization defined modes, the request is denied.

Chapter 3

Lab Activity: Kubernetes Authentication, Authorization, and Admission Control

3.1 Introduction to Minikube, kubectl, and KVM2

3.1.1 Minikube

Minikube implements a local Kubernetes cluster on macOS, Linux, and Windows systems. Minikube's primary goals are to be the best tool for local Kubernetes application development and to support all Kubernetes features that fit. In order to fully take advantage of all the features Minikube has to offer, a Type-2 Hypervisor or a *Container Runtime* should be installed on the local workstation, to run in conjunction with Minikube. The role of the hypervisor or container runtime is to offer an isolated infrastructure for the Minikube Kubernetes cluster components, that is easily reproducible, easy to use and tear down. This isolation of the cluster ensures that once no longer needed, the Minikube components can be safely removed leaving behind no configuration changes to personal workstation, thus no traces of their existence.
Hence, **Minikube** serves as an ideal tool for this laboratory session.

3.1.2 KVM (Kernel-based Virtual Machine)

KVM (Kernel-based Virtual Machine) is a virtualization module in the Linux kernel that allows the kernel to act as a hypervisor, enabling the creation and management of virtual machines.
In this tutorial, KVM will be the Type-2 Hypervisor driver selected for Minikube.

3.1.3 kubectl

Kubernetes provides a command line tool for communicating with a Kubernetes cluster's control plane, using the Kubernetes API. This tool is named **kubectl**. It enables users to deploy applications, manage cluster resources, and view logs. **kubectl** communicates with the Kubernetes API server to execute operations.

Other lab setups

In this lab, the setup chosen is Minikube on a **GNU/Linux x86_64** system (Debian-based) with **KVM** as the hypervisor.

For other setups, the following hypervisors and container runtimes are supported:

- **Linux:** VirtualBox, KVM2, QEMU hypervisors, or Docker and Podman runtimes.
- **macOS:** VirtualBox, HyperKit, VMware Fusion, Parallels, QEMU hypervisors, or Docker and Podman runtimes.
- **Windows:** VirtualBox, Hyper-V, VMware Workstation, QEMU hypervisors, or Docker and Podman runtimes.

For installation instructions and more details, refer to: Minikube's official installation guide

3.1.4 Dependencies for Minikube

To run Minikube with the KVM2 driver, the following dependencies are required:

- Virtualization enabled in BIOS

  Check virtualization on GNU/Linux

  Verify the virtualization support on your GNU/Linux OS in a terminal (a non-empty output indicates supported virtualization):
  `$ grep -E --color 'vmx|svm' /proc/cpuinfo`

- `libvirt`, `qemu`, `virt-manager`
- Minikube (with KVM2 driver support)
- kubectl (Kubernetes CLI)

3.2 Installation Instructions on Debian-based Distributions

3.2.1 Install KVM and Dependencies

Install KVM and dependencies

```
$ sudo apt update
$ sudo apt -y install bridge-utils cpu-checker libvirt-clients \
libvirt-daemon qemu qemu-kvm virt-manager
# Check KVM (KVM acceleration can be used?)
$ kvm-ok
$ sudo systemctl enable libvirtd
$ sudo systemctl start libvirtd
```

3.2.2 Install Minikube

Install Minikube and Test it

```
# Download and install Minikube
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64

# Start Minikube using KVM2 as the driver (to test it)
$ minikube start --driver=kvm2
$ minikube status
$ minikube stop
$ minikube delete
```

3.2.3 Install kubectl

Install kubectl

```
$ curl -LO "https://dl.k8s.io/release/ \
    $(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
$ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

3.3 Minikube: 2-Tier Web App Setup

This section describes the deployment of a simple 2-tier web application using Minikube with `kvm2` as the driver.
The application consists of a **React.js frontend** served by **NGINX** and a **Node.js backend**. Both the backend and frontend will be deployed in separate pods. Although minimal, this setup is representative of real-world scenarios.

3.3.1 Start Minikube

Start Minikube with the `kvm2` driver.

Start Minikube

```
$ minikube start --driver=kvm2
```

3.3.2 Create Namespace

Create a new namespace called `webapp` for the application.

Create Namespace

```
$ kubectl create namespace webapp
```

3.4 Deploying the 2-Tier Application

Local Test

First, navigate to the `demo-app` folder, which contains `frontend` and `backend` directories. Test the frontend and backend locally.

Test the Backend Locally

```
$ cd demo-app/backend
$ npm install
$ node index.js
```

In another terminal, test the frontend:

Test the Frontend Locally

```
$ cd demo-app/frontend
$ npm install
$ npm run dev
```

3.4.1 Expected Output

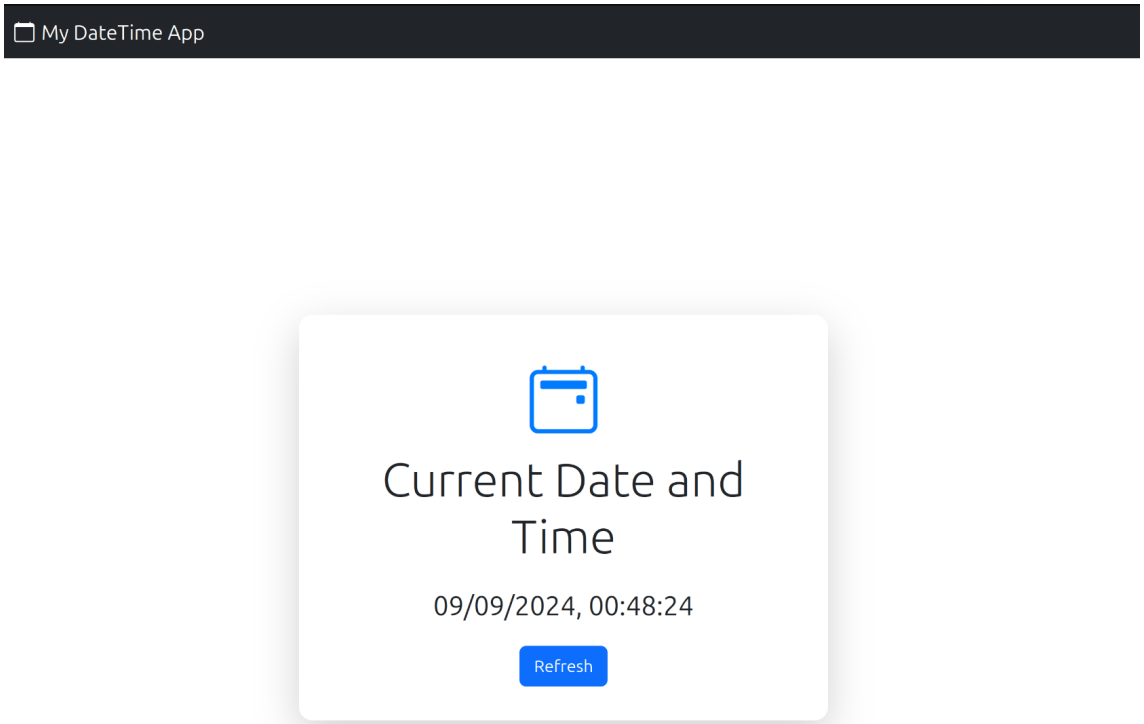

Figure 3.1: MyDateTime App

As you can see, it's a pretty simple web application. The front-end shows the current date and time received by the back-end. Nonetheless, this technology stack is representative of real-world case scenarios.

Dockerfile for Frontend

Create the Dockerfile for the frontend.

Frontend Dockerfile

```
FROM node:latest AS build

WORKDIR /app

COPY ./frontend/package*.json ./
RUN npm install
COPY ./frontend .

RUN npm run build

FROM nginx:alpine

COPY --from=build /app/dist /usr/share/nginx/html
COPY .frontend/nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 8080

CMD ["nginx", "-g", "daemon off;"]
```

Dockerfile for Backend

Create the Dockerfile for the backend.

Backend Dockerfile

```
FROM node:latest

WORKDIR /app

COPY ./backend/package*.json ./

RUN npm install

COPY ./backend .

EXPOSE 5000

CMD ["node", "index.js"]
```

3.4.2 Build Docker Images

Set Minikube's Docker environment and build the Docker images for both frontend and backend.

Set Docker Environment

```
$ eval $(minikube docker-env)
```

Build Docker Images

```
$ docker build -t my-react-app:latest -f frontend/Dockerfile .
$ docker build -t my-backend-app:latest -f backend/Dockerfile .
```

3.4.3 Deploy the Application

Create Kubernetes deployment YAML files for the frontend and backend, ensuring that both the backend and frontend will run in separate pods.

Frontend Deployment YAML

Frontend Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: webapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: my-react-app:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: frontend
  namespace: webapp
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
    nodePort: 30007
```

Notice how the *nodePort* specification was defined for the front-end Pod such that it can be reachable from outside the cluster.

Backend Deployment YAML

Backend Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  namespace: webapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: my-backend-app:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: backend
  namespace: webapp
spec:
  selector:
    app: backend
  ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000
```

Notice how the *nodePort* specification for the back-end was not defined.

Apply the deployment manifests.

Apply Deployment Files

```
$ kubectl apply -f frontend-deployment.yaml
$ kubectl apply -f backend-deployment.yaml
```

Verify that the pods are running.

List Pods

```
$ kubectl get pods -n webapp
```

Retrieve Minikube's IP address and access the application by visiting the NodePort for the frontend service.

Retrieve Minikube IP

```
$ minikube ip
```

Visit `http://<minikube-ip>:30007` in your browser to see the React application served by NGINX.

3.5 Authorization: Generating X.509 Certificates for Users

In this section, we will generate X.509 certificates for three users: `frontend-developer`, `backend-developer`, and `admin`. These certificates will authenticate users within the Kubernetes cluster.

3.5.1 CSR (Certificate Signing Request)

A *CSR* (Certificate Signing Request) is a block of encoded text containing a public key and identifying information. It is sent to a Certificate Authority (CA) to request a digital certificate. The *CA* reviews the *CSR* and, if everything is valid, signs it to generate the requested certificate. For more information consult: CSRs in Kubernetes.

3.5.2 Generate Private Keys and CSRs

We will now generate private keys, create *CSR*s for the front end developer , and sign them using Minikube's Certificate Authority (CA). Minikube automatically handled the creation of cluster's certificates, if you are curious about how to change that, more information is provided here: Minikube Certificates.

Step 1: Verify Minikube's CA Configuration

Firstly, check Minikube's *CA* certificate configuration to ensure certificates can be signed.

Verify CA Configuration

```
# Check the CA certificate locally
$ cat ~/.minikube/ca.crt

# Alternatively, SSH into Minikube VM
$ minikube ssh
$ sudo cat /var/lib/minikube/certs/ca.crt
```

Step 2: Generate Private Key and CSR for Frontend Developer

Frontend Developer CSR

```
# Generate private key for frontend-developer
$ openssl genrsa -out frontend.key 2048

Generating RSA private key, 2048 bit long modulus (2 primes)
.............................................+++++
.....................+++++

# Create CSR for frontend-developer
$ openssl req -new -key frontend.key \
  -out frontend.csr -subj "/CN=frontend-developer/O=webapp"
```

Step 3: Create CertificateSigningRequest Object

Once the *CSR* is generated, we need to encode it in *base64* and create a *Kubernetes CertificateSigningRequest* (CSR) object. Ensure that the *base64* string is copied in the `request` specification of the *CSR manifest*.

Create CSR YAML Manifest

```
# Encode the CSR in base64
$ cat frontend.csr | base64 | tr -d '\n'

LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFV...LS0tLQo=  # Example

# Create signing-request.yaml
$ vim signing-request.yaml
```

CSR Manifest

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: frontend-csr
spec:
  groups:
  - system:authenticated
  request: LS0tLS1CRUdJTiBDRVJUSUZJQ0...LS0tLQo= # Copy the base64 string here!
  signerName: kubernetes.io/kube-apiserver-client
  usages:
  - client auth
```

Step 4: Create and Approve the CSR in Kubernetes

Next, create the CertificateSigningRequest object and approve it.

Create and Approve CSR

```
# Create CSR object in Kubernetes
$ kubectl create -f signing-request.yaml

certificatesigningrequest.certificates.k8s.io/frontend-csr created

# List CSRs (should show pending state)
$ kubectl get csr

NAME          AGE   SIGNERNAME                            REQUESTOR       CONDITION
frontend-csr  12s   kubernetes.io/kube-apiserver-client   minikube-user   Pending

# Approve CSR
$ kubectl certificate approve frontend-csr

certificatesigningrequest.certificates.k8s.io/frontend-csr approved

# List CSRs again (should show approved and issued states)
$ kubectl get csr

NAME          AGE   SIGNERNAME                            REQUESTOR       CONDITION
frontend-csr  57s   kubernetes.io/kube-apiserver-client   minikube-user   Approved
```

Step 5: Retrieve and Store the Approved Certificate

Finally, retrieve the approved certificate and store it as `frontend.crt`.

Retrieve Approved Certificate

```
# Extract the approved certificate
$ kubectl get csr frontend-csr \
  -o jsonpath='{.status.certificate}' | base64 -d > frontend.crt

# View the certificate
$ cat frontend.crt

-----BEGIN CERTIFICATE-----
MIIDGzCCA...
...
...NOZRRZBVunTjK7A==
-----END CERTIFICATE----- # Example
```

Step 6: Configure Kubectl for Frontend Developer

Now, configure `kubectl` to use the frontend developer's certificate and key for authentication.

Configure Kubectl

```
# Set frontend-developer's credentials
$ kubectl config set-credentials frontend-developer \
  --client-certificate=frontend.crt --client-key=frontend.key

User "frontend-developer" set.

# Create a context for frontend-developer
$ kubectl config set-context frontend-context \
  --cluster=minikube --namespace=webapp --user=frontend-developer

Context "frontend-context" created.

# View the kubectl configuration (check frontend user and context)
$ kubectl config view
```

Step 7: Generate Private Keys for remaining users

For simplicity, for the remaining users we will directly sign their certificates with the cluster's *CA*. If instead you wish to use *CSR*, you can apply the same commands as previously did with the `frontend-developer` user.

Generate Private Keys and Directly sign the Certificate

```
# Backend Developer
$ openssl genrsa -out backend.key 2048
$ openssl req -new -key backend.key -out backend.csr
-subj "/CN=backend-developer/O=webapp"

# Create a context for backend-developer
$ kubectl config set-context backend-context \
--cluster=minikube --namespace=webapp --user=backend-developer
Context "backend-context" created.

# Admin (Cluster Admin)
$ openssl genrsa -out admin.key 2048
$ openssl req -new -key admin.key -out admin.csr -subj "/CN=admin/O=admin"
```

Step 8: Sign the Certificates using Minikube CA

Sign the *CSR*s using Minikube's *CA*:

Sign Certificates

```
# Sign backend-developer CSR
$ openssl x509 -req -in backend.csr
-CA ~/.minikube/ca.crt -CAkey ~/.minikube/ca.key -out backend.crt -days 10

# Sign admin CSR
$ openssl x509 -req -in admin.csr
-CA ~/.minikube/ca.crt -CAkey ~/.minikube/ca.key -out admin.crt -days 10
```

Step 9: Configure `kubectl` for remaining users

Now, configure `kubectl` to use the certificates and private keys for the remaining user.

Configure kubectl

```
# Set credentials for backend-developer
$ kubectl config set-credentials backend-developer \
  --client-certificate=backend.crt --client-key=backend.key

# Set credentials for admin
$ kubectl config set-credentials admin \
  --client-certificate=admin.crt --client-key=admin.key
```

Step 10: Check all users in `kubectl`

Check that all users are defined and present in `kubectl`'s client's configuration manifest.

Configure kubectl

```
$ kubectl config view
```

3.6 Authorization: Role-Based Access Control (RBAC)

This section defines Role-Based Access Control (RBAC) for three users: `frontend-developer`, `backend-developer`, and `admin`, assigning them specific roles and bindings within the Kubernetes cluster.

3.6.1 Frontend Developer Role

We will create a role for the `frontend-developer` user, granting access to manage only the frontend pod in the `webapp` namespace.

Step 1: Create the Role YAML

Specify the pod name in the `resourceNames` field for the `frontend-developer` role.

Find frontend pod-id

```
$ kubectl get pods -n webapp
```

RBAC Manifest: Frontend Developer Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: frontend-developer
  namespace: webapp
rules:
- apiGroups: [""]
  resources: ["pods"]
  resourceNames: ["frontend-<pod-id>"] # Paste the pod id here.
  verbs: ["get", "watch", "list", "create", "update"]
```

Save this file as `frontend-role.yaml`.

Step 2: Apply the Role

Apply the role in the Kubernetes cluster using the following command:

Apply Frontend Developer Role

```
$ kubectl create -f frontend-role.yaml
```

Verify the role has been created:

Verify Frontend Developer Role

```
$ kubectl -n webapp get roles

NAME                CREATED AT
frontend-developer  ...
```

Step 3: Create the RoleBinding YAML

RBAC Manifest: Frontend Developer RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: frontend-dev-binding
  namespace: webapp
subjects:
- kind: User
  name: frontend-developer
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: frontend-developer
  apiGroup: rbac.authorization.k8s.io
```

Save this file as `frontend-rolebinding.yaml`.

Step 4: Apply the RoleBinding

Apply the rolebinding to bind the role to the user:

Apply Frontend Developer RoleBinding

```
$ kubectl create -f frontend-rolebinding.yaml
```

Verify the rolebinding has been created:

Verify Frontend Developer RoleBinding

```
$ kubectl -n webapp get rolebindings

NAME                 ROLE                      AGE
frontend-dev-binding Role/frontend-developer  10s
```

3.6.2 Backend Developer Role

Next, we create a role for the `backend-developer`, granting access only to manage the backend pod in the `webapp` namespace.

Step 1: Create the Role YAML

Specify the pod name in the `resourceNames` field for the `backend-developer` role.

Find backend pod-id

```
$ kubectl get pods -n webapp
```

RBAC Manifest: Backend Developer Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: backend-developer
  namespace: webapp
rules:
- apiGroups: [""]
  resources: ["pods"]
  resourceNames: ["backend-<pod-id>"] # Copy the pod id here.
  verbs: ["get", "watch", "list", "create", "update"]
```

Save this file as `backend-role.yaml`.

Step 2: Apply the Role

Apply the role in the Kubernetes cluster:

Apply Backend Developer Role

```
$ kubectl create -f backend-role.yaml
```

Verify the role:

Verify Backend Developer Role

```
$ kubectl -n webapp get roles

NAME                CREATED AT
backend-developer   ...
```

Step 3: Create the RoleBinding YAML

RBAC Manifest: Backend Developer RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: backend-dev-binding
  namespace: webapp
subjects:
- kind: User
  name: backend-developer
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: backend-developer
  apiGroup: rbac.authorization.k8s.io
```

Save this file as `backend-rolebinding.yaml`.

Step 4: Apply the RoleBinding

Apply the rolebinding:

Apply Backend Developer RoleBinding

```
$ kubectl create -f backend-rolebinding.yaml
```

Verify the rolebinding:

Verify Backend Developer RoleBinding

```
$ kubectl -n webapp get rolebindings

NAME                 ROLE                      AGE
backend-dev-binding  Role/backend-developer  15s
```

3.6.3 Cluster Admin Role

Lastly, we create a cluster-wide admin role for the `admin` user with full access to all resources in the cluster.

Step 1: Create the ClusterRole YAML

RBAC Manifest: Cluster Admin Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-adm
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

Save this file as `cluster-admin-role.yaml`.

Step 2: Apply the Role

Apply the cluster role in Kubernetes:

Apply Cluster Admin Role

```
$ kubectl create -f cluster-admin-role.yaml
```

Verify the cluster role:

Verify Cluster Admin Role

```
$ kubectl get clusterroles

NAME              CREATED AT
cluster-adm       ...
```

Step 3: Create the ClusterRoleBinding YAML

RBAC Manifest: Cluster Admin RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-binding
subjects:
- kind: User
  name: admin
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-adm
  apiGroup: rbac.authorization.k8s.io
```

Save this file as `cluster-admin-binding.yaml`.

Step 4: Apply the ClusterRoleBinding

Apply the cluster role binding:

Apply Cluster Admin RoleBinding

```
$ kubectl create -f cluster-admin-binding.yaml
```

Verify the rolebinding:

Verify Cluster Admin RoleBinding

```
$ kubectl get clusterrolebindings

NAME                  ROLE                 AGE
cluster-admin-binding ClusterRole/cluster-adm 30s
```

Now that we have succesfully set the authentication mode and authorization policies for each user we can test them.

3.7 AuthN & AuthZ: Testing API Access for Users

We will now test access for each user using specific certificates and keys. We will test one authorized API (successful) and one unauthorized API (failure) for each user.

3.7.1 Frontend Developer Access

Per the authorization policies we defined the `frontend-developer` can only perform *CRUD* operations on the `frontend` pode.

Test Frontend Developer Access (Authorized)

```
# Test access to the specific frontend pod (authorized)
$ kubectl --client-certificate=frontend.crt \
        --client-key=frontend.key \
        --context=frontend-context \
        get pod frontend--<pod-id> --namespace=webapp
# Output: Details of the pod if access is authorized
```

Test Frontend Developer Access (Unauthorized)

```
# Test access to backend resources (unauthorized)
$ kubectl --client-certificate=frontend.crt \
        --client-key=frontend.key \
        --context=frontend-context \
        get pod backend--<pod-id> --namespace=webapp
# Output: "Error from server (Forbidden): pods "backend--<pod-id>" is forbidden
```

3.7.2 Backend Developer Access

Per the authorization policies we defined the `backend-developer` can only perform *CRUD* operations on the `backend` pode.

Test Backend Developer Access (Authorized)

```
# Test access to the specific backend pod (authorized)
$ kubectl --client-certificate=backend.crt \
        --client-key=backend.key \
        --context=backend-context \
        get pod backend-<pod-id> --namespace=webapp
# Output: Details of the pod if access is authorized
```

Test Backend Developer Access (Unauthorized)

```
# Test access to frontend resources (unauthorized)
$ kubectl --client-certificate=backend.crt \
        --client-key=backend.key \
        --context=backend-context \
        get pod frontend-<pod-id> --namespace=webapp
# Output: "Error from server (Forbidden): pods "frontend-<pod-id>" is forbidden
```

3.7.3 Cluster Admin Access

Test Namespace Admin Access (Authorized)

```
# Test full access as cluster-admin (authorized)
$ kubectl --client-certificate=admin.crt --client-key=admin.key \
get pods --namespace=webapp
# Output: "yes"
```

Test Cluster Admin Access (Authorized)

```
# Admin has access to the all namespaces in the cluster (default also)
kubectl --client-certificate=admin.crt --client-key=admin.key get pods
```

3.8 Change Backend Developer Authorization

We will now modify the `backend-developer` role to allow this role to only see (**get, list**) all pods in the `webapp` namespace, therefore, also the `frontend` pod.

RBAC Manifest: Updated Backend Developer Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: webapp
  name: backend-developer
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

Reapply the modified role:

Apply Updated Backend Role

```
$ kubectl apply -f backend-role.yaml
```

3.8.1 Test Updated Backend Developer Access

After changing the permissions, retest the access to all pods in the `webapp` namespace.

Test Updated Frontend Developer Access (Authorized)

```
# Test access to all pods in the namespace is now unauthorized
$ kubectl --client-certificate=backend.crt \
        --client-key=backend.key \
        --context=backend-context \
        get pods --namespace=webapp
# Output: informations on all pods in the webapp namespace.
```

Test Updated Backend Developer Access (Unauthorized)

```
# Test update operation is not authorized anymore
$ kubectl --client-certificate=backend.crt \
        --client-key=backend.key \
        --context=backend-context \
        auth can-i update pods --namespace=webapp
# Output: "No" as we set only get and list operations.
```

3.9 Admission Controllers Exercise: Resource Quota

In this exercise, we will work with the `ResourceQuota` admission controller. The goal is to observe how admission controllers enforce granular access control.

3.9.1 Activate ResourceQuota Admission Controller

The `ResourceQuota` admission controller ensures that pods in a namespace do not exceed the defined CPU and memory limits. It is important to verify that this controller is already active in the cluster. For Minikube, `ResourceQuota` is enabled by default.

Start Minikube if needed

```
# Start Minikube IF not started already
$ minikube start --driver=kvm2
```

You can confirm that the `ResourceQuota` admission controller is active by checking the running API server configuration:

Check Active Admission Controllers

```
$ kubectl -n kube-system describe pod kube-apiserver-minikube | grep -i admission
```

3.9.2 Create a Namespace and Apply Resource Quota

Next, create a new namespace called `admission-test` and apply a resource quota that limits the total CPU and memory usage for pods in this namespace.

Create Namespace

```
$ kubectl create namespace admission-test
```

Define the resource quota in a YAML file. The following YAML specifies that the `total CPU` and `memory` requests for all pods in this namespace cannot exceed 1 CPU and *1Gi of memory*, while the `total CPU` and `memory` limits cannot exceed *2 CPU* and *2Gi of memory*:

Resource Quota

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: test-quota
  namespace: admission-test
spec:
  hard:
    requests.cpu: "1"
    requests.memory: "1Gi"
    limits.cpu: "2"
    limits.memory: "2Gi"
```

Save this as `resource-quota.yaml` and apply it to the `admission-test` namespace:

Apply Resource Quota

```
$ kubectl apply -f resource-quota.yaml
```

You can verify the applied quota by checking the resource quota in the namespace:

Check Resource Quota

```
$ kubectl get resourcequota -n admission-test
```

3.9.3 Deploy a Pod that Violates Resource Quota

Now, let's create a pod that violates the resource quota by requesting and limiting more CPU and memory than allowed by the `test-quota`.

Write a YAML file for a pod that exceeds the resource quota limits:

Pod Specification: Large Pod (Violates ResourceQuota)

```
apiVersion: v1
kind: Pod
metadata:
  name: large-pod
  namespace: admission-test
spec:
  containers:
  - name: large-container
    image: busybox
    resources:
      requests:
        memory: "2Gi"   # Exceeds the 1Gi limit
        cpu: "2"        # Exceeds the 1 CPU limit
      limits:
        memory: "4Gi"   # Exceeds the 2Gi limit
        cpu: "4"        # Exceeds the 2 CPU limit
    command: ["sh", "-c", "sleep 3600"]
```

Save it as `large-pod.yaml` and apply it:

Apply Large Pod Configuration

```
$ kubectl apply -f large-pod.yaml
```

The `ResourceQuota` admission controller should reject this pod due to exceeding the quota limits.

Expected Output

```
Error: pods "large-pod" is forbidden: exceeded quota: test-quota,
requested: limits.cpu=4, limits.memory=4Gi, used: limits.cpu=0,
limits.memory=0, limited: limits.cpu=2, limits.memory=2Gi
```

Chapter 4

Advanced Concept: Security Contexts and Pod Security Admission

In Kubernetes, Security Contexts provide a way to define the security settings and privileges for Pods and Containers. These settings control key security features such as user and group IDs, file system permissions, and the ability to escalate privileges within a container.
Additionally, the Kubernetes Pod Security Admission mechanism enforces security standards cluster-wide by automating security context enforcement at the namespace level.

4.1 Security Contexts

A **Security Context** defines privilege and access control settings for a Pod or Container. These can include options such as:

- **User/Group IDs**: Define the user and group IDs under which the processes inside the container run.
- **File System Groups (fsGroup)**: Define the group ID applied to file system operations, such as mounted volumes.
- **Privilege Escalation**: Controls whether a process in a container can gain additional privileges (via `setuid` binaries, etc.).

Security contexts can be applied at two levels:

1. **Pod Level**: These security settings apply to all containers within the Pod.
2. **Container Level**: These settings override or add to the Pod-level context for specific containers.

4.1.1 Pod Manifest Example

Below is a sample Pod manifest with security contexts:

Security Context Pod Manifest example

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
```

Security Context Pod Manifest example (Cont'd)

```
  volumes:
  - name: vol
    emptyDir: {}
  containers:
  - name: busy
    image: busybox:1.28
    command: [ "sh", "-c", "sleep infinity" ]
    volumeMounts:
    - name: vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

4.1.2 Breakdown of the Pod Manifest

- **Pod-level Security Context**:
  - `runAsUser: 1000`: All containers within this Pod will run processes as the user with ID 1000.
  - `runAsGroup: 3000`: All processes in the containers will run with the group ID 3000.
  - `fsGroup: 2000`: This group ID (2000) will be applied to file system operations, meaning that any files created by containers in this Pod on mounted volumes will have the group ownership of 2000.
- **Volume Configuration**:
  - A volume named `vol` is created as an empty directory and mounted at `/data/demo` inside the container. Files created here will inherit the `fsGroup` setting from the Pod-level security context.
- **Container-level Security Context (busy container)**:
  - `allowPrivilegeEscalation: false`: The `busy` container is prevented from escalating its privileges, meaning that even if a child process within this container tries to gain more privileges than its parent (e.g., through `setuid`), it will be denied.

4.2 Pod Security Admission

While security contexts can be applied on a per-Pod or per-Container basis, Kubernetes also provides a scalable way to enforce security settings across the cluster using **Pod Security Admission**. This is an admission controller that applies security policies to Pods at the namespace level.

The **Pod Security Admission** controller enforces three Pod Security Standards:

1. **Privileged**: Allows unrestricted access and is used for trusted workloads that need elevated permissions.
2. **Baseline**: Provides a moderate level of security by blocking known privilege escalation paths, suitable for most applications.
3. **Restricted**: Enforces the highest level of security by limiting access to dangerous privileges, appropriate for critical applications and environments with untrusted users.

The controller automates the enforcement of these standards by configuring security context restrictions on Pods when they are created or modified. Each standard ensures that Pods adhere to specific sets of security controls, such as:

- Restricting root privileges.
- Disallowing privilege escalation.

When a namespace is configured to enforce the **restricted** profile, any Pod that does not conform to security best practices will be rejected at deployment time.

Chapter 5

NSA Guidelines on Kubernetes Authentication and Authorization

The *NSA's Kubernetes Hardening Guidance*[6] (version 1.2, August 2022) outlines several important practices to fortify Kubernetes clusters against malicious access, focusing on authentication, authorization, and enforcing least privilege access.

Authentication in Kubernetes

Authentication ensures that only verified users or services can interact with the Kubernetes API server. As previously explained, Kubernetes clusters primarily use two types of user accounts:

- **Service Accounts**: These are used by pods to interact with the API server. Managed by the ServiceAccount Admission Controller, service accounts automatically generate bearer tokens for API access. If misconfigured, these tokens can be exploited if left unsecured. The NSA advises restricting access to pod secrets and managing service account permissions carefully.
- **Normal User Accounts**: Kubernetes relies on external services (e.g., X.509 client certificates, OpenID tokens) for user and admin account authentication. The NSA strongly advises against using weak methods like static password files, as they are vulnerable to exploitation.

The NSA recommends using at least one secure authentication method, such as OpenID Connect (OIDC) or client certificates, and strongly advises disabling anonymous requests to reduce the attack surface.

Authorization in Kubernetes

Authorization mechanisms control which actions authenticated users and services can perform. The NSA encourages the use of **Role-Based Access Control (RBAC)** to manage permissions. With RBAC, roles are defined to specify actions users can perform within namespaces (Roles) or across the entire cluster (ClusterRoles). These roles are assigned to users or service accounts via RoleBindings or ClusterRoleBindings.

The NSA emphasizes:

- Ensuring that **RBAC is enabled** and properly configured.
- Disabling permissive authorization modes, such as `AlwaysAllow`, to enforce least privilege access.

[6] https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF