Certified Kubernetes Application Developer (CKAD)

By Damian Igbe, PhD

Day 3

Domain 5.1: Demonstrate Basic Understanding of NetworkPolicies

NetworkPolicies in Kubernetes are used to control the communication between Pods. They define the rules and policies that control how groups of Pods are allowed to communicate with each other and with other network endpoints outside the Kubernetes cluster. By defining NetworkPolicies, you can restrict traffic to and from Pods based on various factors, such as namespace, IP address, and labels.
In this section, we'll cover the basics of NetworkPolicies:

- What NetworkPolicies are
- How they work in Kubernetes
- How to define and apply NetworkPolicies
- Common use cases and best practices for NetworkPolicies

1. What are NetworkPolicies?

A NetworkPolicy in Kubernetes is an API object that defines a set of rules governing how Pods communicate with each other and with external services. By default, Pods are open to communication from any other Pod within the same namespace, and they can initiate connections to external services. However, NetworkPolicies allow you to restrict this communication to only the Pods and external services you specify. NetworkPolicies are enforced by network plugins that support the NetworkPolicy API, such as Calico, Cilium, or Weave.

2. How Do NetworkPolicies Work?

NetworkPolicies work by specifying ingress (incoming traffic) and egress (outgoing traffic) rules for Pods.

These rules can specify:

- Selectors for Pods (e.g., by labels)
- Ports that are open or closed for communication
- Namespaces that are allowed or disallowed
- IP Blocks that control access to or from specific IP ranges

By default, if a NetworkPolicy is not applied, Pods are open to any communication. Once a NetworkPolicy is applied, it acts as a firewall, allowing only traffic that matches the rules defined in the policy and rejecting all other traffic.

3. Key Concepts

Ingress and Egress:

- Ingress refers to inbound traffic to a Pod.
- Egress refers to outbound traffic from a Pod.

You can define rules for both ingress and egress to control the flow of traffic into and out of a Pod.

- Pod Selectors: Pod selectors are used to identify which Pods the policy applies to. This can be based on labels assigned to Pods.
- Policy Types: You can define the ingress and/or egress rules in a NetworkPolicy:
- Ingress: Defines what inbound traffic is allowed.
- Egress: Defines what outbound traffic is allowed.

- Namespace Selectors: You can use namespace selectors to control communication between Pods in different namespaces.

4. Basic Example of a NetworkPolicy

Let's look at a simple example of a NetworkPolicy that restricts inbound traffic to a Pod:

NetworkPolicy Example: Allow HTTP Inbound Traffic
In this example, we will create a NetworkPolicy to allow only HTTP traffic (port 80) from other Pods in the same namespace.

```
apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: allow-http-ingress

spec:

  podSelector:

    matchLabels:

      app: my-app

  ingress:

  - ports:

    - protocol: TCP

      port: 80

  policyTypes:

  - Ingress
```

Explanation:

1. podSelector: This policy applies to Pods with the label app=my-app.
2. ingress: Specifies that only traffic on port 80 (HTTP) is allowed.
3. policyTypes: The policy is applied to ingress traffic (incoming traffic).

By applying this NetworkPolicy, Pods labeled app=my-app will only be able to accept traffic on port 80 (HTTP). Any other traffic is blocked.

5. Egress Example: Allowing Outbound Traffic

Next, let's look at a NetworkPolicy that allows a Pod to send outbound traffic only to a specific IP range.

```
apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: allow-egress-to-db

spec:

  podSelector:

    matchLabels:

      app: my-app

  egress:

  - to:

    - ipBlock:

        cidr: 192.168.0.0/16

  policyTypes:

  - Egress
```

Explanation:

- podSelector: This policy applies to Pods with the label app=my-app.
- egress: Specifies that traffic from the Pod is allowed only to the IP range 192.168.0.0/16.
- policyTypes: The policy is applied to egress traffic (outgoing traffic).

This policy ensures that Pods labeled app=my-app can only initiate connections to IPs within the range 192.168.0.0/16. Any other outbound traffic will be blocked.

6. Combining Ingress and Egress Policies

You can combine both ingress and egress rules in a single NetworkPolicy to control both inbound and outbound traffic for a set of Pods.

```
apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:
```

```
  name: allow-http-and-db-traffic
spec:
  podSelector:
    matchLabels:
      app: my-app
  ingress:
  - ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - ipBlock:
        cidr: 192.168.0.0/16
  policyTypes:
  - Ingress
  - Egress
```

This policy:

- Allows HTTP inbound traffic on port 80 (ingress).
- Allows outbound traffic to the 192.168.0.0/16 IP range (egress).

7. Default Deny Policy

A default deny policy can be created by not defining any ingress or egress rules in the policy. This implicitly denies all traffic by default. You can create a NetworkPolicy that denies all traffic to Pods by not specifying any rules.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-traffic
```

```
spec:

  podSelector: {}

  policyTypes:

  - Ingress

  - Egress
```

Explanation:

- The empty podSelector: {} applies the policy to all Pods in the namespace.
- No ingress or egress rules are defined, meaning all traffic (both inbound and outbound) is blocked.
- This policy effectively blocks all traffic to and from the Pods unless other NetworkPolicies allow specific traffic.

Lab Walkthrough:

Part 1: Deploy a Kind Cluster

For this lab, since Docker Desktop does not have a CNI that support network policies, by default, we will quickly create a cluster using KIND. KIND is a utility for setting up a multinode kubernetes cluster on a single node, like on a laptop. You can read more about KIND here.
https://kind.sigs.k8s.io/
Follow the steps below to deploy Kind. Full instructions for different platforms can be found here: https://kind.sigs.k8s.io/docs/user/quick-start

1. For windows, try below and wait for it to finish

```
winget install Kubernetes.kind
```

2. Test that kind is properly installed. You may need to create a new terminal to test.

```
kind
```

3. Expected output:

```
kind creates and manages local Kubernetes clusters using Docker container 'nodes'

Usage:

  kind [command]
```

```
Available Commands:
  build       Build one of [node-image]
  completion  Output shell completion code for the specified shell (bash, zsh or fish)
  create      Creates one of [cluster]
  delete      Deletes one of [cluster]
  export      Exports one of [kubeconfig, logs]
  get         Gets one of [clusters, nodes, kubeconfig]
  help        Help about any command
  load        Loads images into nodes
  version     Prints the kind CLI version

Flags:
  -h, --help              help for kind
      --loglevel string   DEPRECATED: see -v instead
  -q, --quiet             silence all stderr output
  -v, --verbosity int32   info log verbosity, higher value produces more output
      --version           version for kind

Use "kind [command] --help" for more information about a command.
```

3. Create a file called config.yaml with the following content. Take note that the CNI is disabled.

```
#kind-config.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
```

- role: worker

- role: worker

networking:

  disableDefaultCNI: true

4. Create a cluster with the file

kind create cluster --config config.yaml

5. Check the cluster nodes

kubectl get nodes

Expected output: Take note that the pods will be 'NOT READY' until the CNI is properly configured below.
NAME STATUS ROLES AGE VERSION

kind-control-plane Not Ready control-plane 40s v1.25.2

kind-worker Not Ready <none> 40s v1.25.2

kind-worker2 not Ready <none> 40s v1.25.2

6. Cilium supports Network Policies. Install Cilium, for your platform (e.g windows) by downloading it from here https://github.com/cilium/cilium-cli/releases
7. Install cillium. Double click it and locate the Cilium binary.
8. Install cilium CNI on the KIND cluster. It will detect kind cluster automatically.

cilium install

9. Test that Cilium is properly installed

cilium status --wait

Expected output:
/¯¯\

```
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    OK
 \__/¯¯\__/    Hubble Relay:       disabled
    \__/       ClusterMesh:        disabled

DaemonSet              cilium                   Desired: 3, Ready: 3/3, Available: 3/3
DaemonSet              cilium-envoy             Desired: 3, Ready: 3/3, Available: 3/3
Deployment             cilium-operator          Desired: 1, Ready: 1/1, Available: 1/1
Containers:            cilium                   Running: 3
                       cilium-envoy             Running: 3
                       cilium-operator          Running: 1
                       clustermesh-apiserver
                       hubble-relay
Cluster Pods:          14/14 managed by Cilium
Helm chart version:    1.17.0
Image versions         cilium
quay.io/cilium/cilium:v1.17.0@sha256:51f21bdd003c3975b5aaaf41bd21aee23cc08f44efaa27effc91c621bc9d8b1d: 3
                       cilium-envoy
quay.io/cilium/cilium-envoy:v1.31.5-1737535524-fe8efeb16a7d233bffd05af9ea53599340d3f18e@sha256:57a3aa6355a3223da360395e3a109802867ff635cb852aa0afe03ec7bf04e545: 3
                       cilium-operator
quay.io/cilium/operator-generic:v1.17.0@sha256:1ce5a5a287166fc70b6a5ced3990aaa442496242d1d4930b5a3125e44cccdca8: 1
```

10. Check that your cluster nodes are now ready:

```
kubectl get nodes
```

Expected output: Note that the nodes are now 'READY'

NAME STATUS ROLES AGE VERSION

kind-control-plane Ready control-plane 2m v1.25.2

kind-worker Ready <none> 2m v1.25.2

kind-worker2 Ready <none> 2m v1.25.2

11. Congratulations, you now have a multinode cluster created with KIND with Cilium CNI enabled for network policies to work.

Part 2: Network Policies

1. Create pod1 for the lab

kubectl run pod1 --image nginx:latest -l app=pod1

Expected output:
pod/pod1 created

2. Create pod2 for the lab
3.

kubectl run pod2 --image nginx:latest -l app=pod2

Expected output:
pod/pod1 created

3. See the details of your pod

kubectl get pods -o wide

Expected output: (yours may differ)
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

pod1 1/1 Running 0 59s 10.244.110.50 minikube <none> <none>

pod2 1/1 Running 0 57s 10.244.110.80 minikube <none> <none>

4. Before creating the network policies, test that this works

kubectl exec -it pod1 -- curl 10.244.110.80 --max-time 1

Expected output:
<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

5. Create Netwotk Policy

create a file called netpol.yaml with the following content:

```
apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: network-policy

  namespace: default

spec:

  podSelector:

    matchLabels:

      app: pod2

  policyTypes:

    - Ingress

    - Egress
```

6. Create the network poliy from the file

```
kubectl apply -f netpol.yaml
```

Expected output:
networkpolicy.networking.k8s.io/network-policy created

7. Test if the network policy is working. The policy above blocked all traffic to pod1 so you should not have access to the Nginx server.

kubectl exec -it pod1 -- curl 10.244.110.80 --max-time 1

Expected output.
curl: (28) Connection timed out after 1001 milliseconds

command terminated with exit code 28

8. Lets open the communication between pod1 and pod2

Create a file called netpol2.yaml and paste the following content.

```
apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: network-policy

  namespace: default

spec:

  podSelector:

    matchLabels:

      app: pod2

  policyTypes:

    - Ingress

    - Egress

  ingress:

    - from:

      - podSelector:

          matchLabels:

            app: pod1

  egress:
```

```
  - to:
    - podSelector:
        matchLabels:
          app: pod1
```

9. Create the network policy

```
kubectl apply -f netpol2.yaml
```

Expected output.
networkpolicy.networking.k8s.io/network-policy created

10. Test that the policy works. Now you should have access to the nginx as if there were no netpolicy.

```
kubectl exec -it pod1 -- curl 10.244.110.80 --max-time 1
```

Expected output:
<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

11. Lets test if another pod can access nginx in pod1. Create another pod and test as follows.

```
kubectl run pod3 --image nginx:latest
```

12. test from the new pod, pod3

```
kubectl exec -it pod3 -- curl 10.244.110.80 --max-time 1
```

Expected output.
curl: (28) Connection timed out after 1001 milliseconds

command terminated with exit code 28

Congratulations: Lab Ends

Relevant Network Policy Code Snippets

1. PodSelector

```
podSelector:
  matchLabels:
    app: app1
```

2. namespaceSelector

```
namespaceSelector:
  matchLabels:
    app: app1
```

3. ipBlock

```
ipBlock:
  cidr: 10.0.0.0/24
```

4. Multiple selectors with OR

```
ingress:
  - from:
      - namespaceSelector:
          matchLabels:
            app: demo
      - podSelector:
```

```
        matchLabels:
          app: demo-api
```

5. Multiple selectors with AND

```
ingress:
  - from:
      - namespaceSelector:
          matchLabels:
            app: demo
        podSelector:
          matchLabels:
            app: demo-api
```

6. AllowedPorts range

```
ingress:
  - from:
      - podSelector:
          matchLabels:
            app: demo
    ports:
      - protocol: TCP
        port: 1000
        endPort: 3000
```

7. Default Deny all traffic to a pod

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector:
    matchLabels:
      app: demo
  policyTypes:
    - Ingress
    - Egress
```

8. Default Deny all traffic to all pods

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

9. Default Deny all ingress traffic

```
apiVersion: networking.k8s.io/v1
```

```
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

10. Default Deny all egress traffic

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector: {}
  policyTypes:
    - Egress
```

11. Allow all traffic to a Pod

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector:
    matchLabels:
```

```
    app: demo
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - {}
  egress:
    - {}
```

8. Best Practices for NetworkPolicies

- Start with a Default Deny: Begin by creating a default deny policy to block all traffic. Then, incrementally allow only the traffic that is required.
- Apply NetworkPolicies Early: Apply policies early in the deployment cycle to avoid security holes.
- Label Pods Consistently: Use consistent labels on Pods to easily manage and target them with NetworkPolicies.
- Keep NetworkPolicies Simple: Start with simple, clear rules. Complex policies can be harder to debug and maintain.
- Use Namespace Selectors: For cross-namespace communication, use namespaceSelector to control which namespaces can communicate with each other.

9. Conclusion

NetworkPolicies are a critical tool in securing Kubernetes clusters by controlling traffic flow between Pods and other network endpoints. They enable you to implement the least privilege model by restricting unnecessary communication and reducing potential attack surfaces. By defining ingress and egress rules, you can manage communication between Pods, external services, and the network.

Key takeaways:

- NetworkPolicies control traffic flow between Pods.
- PodSelectors and NamespaceSelectors define which Pods are affected.
- You can control both ingress (incoming) and egress (outgoing) traffic.
- Use NetworkPolicies to minimize the attack surface and ensure secure communication.

Domain 3.2: Implement Probes and Health Checks

In Kubernetes, probes and health checks are essential components for maintaining the health and availability of applications running in your clusters. These mechanisms help Kubernetes determine whether an application is running correctly, and they play a vital role in ensuring high availability and reliability. As a Kubernetes Application Developer (CKAD), understanding and properly implementing probes and health checks in your application deployment is crucial for creating resilient applications that self-heal and remain stable under varying conditions.

1. What Are Probes and Health Checks in Kubernetes?

Kubernetes uses probes to periodically check the health of the containers running in Pods. If a container fails a health check, Kubernetes can automatically restart or replace it, ensuring minimal downtime and reducing the likelihood of failure. There are three primary types of health checks (or probes) in Kubernetes:

Startup Probe: The startup probe checks whether the application within the container has started properly. This probe is useful when containers require significant startup time. If the startup probe fails, Kubernetes will kill the container, assuming it has not started successfully. This typically has to be successful before Liveness and Readines Probes are used. This is useful for applications that requires time to start, especially legacy applications.

Liveness Probe: This probe checks whether the container is still running. If the liveness probe fails, Kubernetes will kill the container and restart it. This is useful for situations where the container is alive but stuck in a non-functional state.

Readiness Probe: This probe checks if the container is ready to serve traffic. A failing readiness probe will prevent traffic from being routed to the container, but it will not kill it. This is useful during startup or when the application is not yet ready to handle requests. This probe works with the service object. A pod that fails Readiness probe will be removed from the service endpoint.

Choose the right probe for the right purpose:

- Use readiness probes for containers that need time to initialize before receiving traffic.
- Use liveness probes to detect and restart containers that become unresponsive.
- Use startup probes to detect if containers take longer to start than expected

2. Why Are Probes Important?

1. Self-Healing: Probes allow Kubernetes to automatically restart failed containers, ensuring that your application remains available even when something goes wrong.
2. Service Discovery: By using readiness probes, Kubernetes ensures that only healthy and ready containers receive traffic, avoiding sending requests to containers that are not yet prepared to serve.
3. Resource Efficiency: Liveness probes help in identifying containers that are stuck or in a failed state, ensuring resources are not wasted on containers that are not functioning properly.
4. Improved User Experience: By ensuring that only healthy containers serve traffic, probes help avoid situations where users may experience downtime or degraded performance.

3. Health Check Methods

Health checks can be implemented in several ways:

- HTTP-based Health Checks: Kubernetes sends HTTP requests to a specified path (such as /healthz or /readiness) and expects a 200 OK response to determine if the application is healthy.
- TCP Socket Checks: Kubernetes can check if a specific port is open and accepting connections. If a connection can be established, the container is considered healthy.
- Command-based Health Checks: Kubernetes executes a specified command inside the container. If the command exits with a status of 0, the container is considered healthy. Otherwise, it's considered failed.

Example of Command-based Health Check:

```
livenessProbe:
  exec:
    command:
      - "/bin/sh"
      - "-c"
      - "cat /tmp/healthy"
  initialDelaySeconds: 5
  periodSeconds: 10
```

The command returns 0 if the file /tmp/healthy exists, which indicates the container is healthy.

4. Types of Health Checks (Probes)

1. Liveness Probe

The liveness probe determines whether your application is still running and healthy. If it fails, Kubernetes will restart the container.

Use cases:

1. If an application gets stuck and cannot recover on its own (e.g., a deadlock or infinite loop).
2. If the application is non-responsive but can still be running.

How it works:

1. Kubernetes will periodically call the endpoint you specify (e.g., an HTTP endpoint) to check if the application is healthy. If the probe fails,
2. Kubernetes restarts the container.

This pod will work because nginx's root directory / is created whenever nginx is started. If you modify the path to something that does not exist, like say /error.html, the test will fail, and the pod will keep restarting. You can describe the pod to see what it says. You will see something like:

Example of Liveness Probe:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: nginx
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
      failureThreshold: 3
```

Parameters:

1. initialDelaySeconds: How long to wait before performing the first probe.
2. periodSeconds: How often to run the probe.
3. failureThreshold: The number of consecutive failures before considering the container unhealthy.

2. Readiness Probe

The readiness probe determines whether the application inside the container is ready to serve traffic. Unlike the liveness probe, if the readiness probe fails, the container remains running but will not receive traffic until it passes.

Use cases:

1. If your application needs time to initialize (e.g., loading large datasets, establishing connections, etc.).

2. If your application becomes temporarily unresponsive due to high load or resource constraints.

This pod will work because nginx's root directory /index.html is created whenever nginx is started and ready. If you modify the path to something that does not exists, like say /error.html, the test will fail and the pod will not come on. You can describe the pod to see what it says. you will see somethibg like:

Readiness probe failed: HTTP probe failed with statuscode: 404

Example of Readiness Probe:

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-httpget
spec:
  containers:
  - name: readiness-httpget
    image: nginx:latest
    readinessProbe:
      httpGet:
        path: /index.html
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
      failureThreshold: 3
```

This pod will work because nginx runs on port 80 so the tcpsockert test will pass. If you modify the tcpSocker port to say 8080, the test will fail and the pod will never come online. You can describe the pod to see what it says. You will see something like: Readiness probe failed: dial tcp 10.1.5.1:8080: connect: connection refused

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: readiness-tcpsocket
spec:
  containers:
  - name: readiness-tcpsocket
    image: nginx
    ports:
    - containerPort: 80
    readinessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 15
      periodSeconds: 10
```

Combine both

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: combine-readiness-liveness
spec:
  replicas: 1
  selector:
    matchLabels:
      app: both
  template:
    metadata:
```

```
    labels:
      app: both
  spec:
    containers:
    - name: both-container
      image: nginx:latest
      ports:
      - containerPort: 80
      readinessProbe:
        httpGet:
          path: /index.html
          port: 80
          scheme: HTTP
        initialDelaySeconds: 5
        periodSeconds: 10
        timeoutSeconds: 5
      livenessProbe:
        httpGet:
          path: /
          port: 80
          scheme: HTTP
        initialDelaySeconds: 5
        periodSeconds: 10
        timeoutSeconds: 5
```

3. Startup Probe

The startup probe helps in scenarios where your container may take time to start (e.g., loading large configurations or running initial setup tasks). If the startup probe fails, Kubernetes assumes the container has failed to start and will kill it.

Use cases:

1. When the container takes a long time to start (e.g., databases, large applications).
2. When the application needs time to establish external dependencies, like database connections or external API calls.

Example of Startup Probe:

```
apiVersion: v1

kind: Pod

metadata:

  name: example-pod

spec:

  containers:

  - name: example-container

    image: my-app:latest

    startupProbe:

      httpGet:

        path: /startup

        port: 8080

      initialDelaySeconds: 10

      periodSeconds: 5

      failureThreshold: 5
```

Parameters:

1. initialDelaySeconds: How long to wait before performing the first probe.
2. failureThreshold: Number of failures before considering the container as failed.

5. Best Practices for Health Checks

Set appropriate initialDelaySeconds: Allow the application enough time to start and initialize before the probes begin. Setting an initial delay too short can result in false failures.
Choose the right probe for the right purpose:

- Use readiness probes for containers that need time to initialize before receiving traffic.
- Use liveness probes to detect and restart containers that become unresponsive.
- Use startup probes to detect if containers take longer to start than expected.
- Monitor probe results: Use Kubernetes' monitoring and logging tools to track the success or failure of health probes. This helps in identifying issues early.

- Test probes locally: Before deploying to Kubernetes, test your health check endpoints locally to ensure they work as expected.
- Avoid over-tightening probe thresholds: Set reasonable thresholds for failures and delays to prevent unnecessary restarts or blocking of traffic. Too aggressive health checks can cause unnecessary container restarts.

6. Conclusion

Implementing probes and health checks is essential for maintaining the health and availability of your applications in Kubernetes. As a Kubernetes Application Developer (CKAD), you must know how to use liveness, readiness, and startup probes effectively to ensure your applications remain resilient, self-healing, and able to serve traffic without downtime.
By correctly setting up health checks:

1. Your containers will be monitored and restarted automatically when necessary.
2. Kubernetes will route traffic only to ready containers.
3. You'll be able to identify and resolve issues early, improving your application's overall reliability.
4. Always test your probes, monitor the results, and adjust them based on the specific needs of your application!

Domain 4.2: Understand Authentication, Authorization, and Admission Control

As a Kubernetes Application Developer (CKAD), it's crucial to understand the foundational concepts of authentication, authorization, and admission control in Kubernetes. These are core components of Kubernetes' security architecture, ensuring that only authorized users and processes can access the cluster and that resources are deployed in a secure and controlled manner. Let's break down these concepts in the context of Kubernetes.

1. Authentication in Kubernetes

Authentication is the process of verifying the identity of a user, application, or service trying to access the Kubernetes API. Kubernetes supports several methods of authentication:
a) Common Authentication Methods in Kubernetes

- Certificates: Kubernetes can authenticate clients via TLS client certificates. When a user or service makes a request to the API server, the client presents a certificate, and the server verifies it.
- Bearer Tokens: A bearer token is typically used in service-to-service communication. When a client makes a request, the bearer token in the request header is validated by the API server.
- OpenID Connect (OIDC): Kubernetes can be integrated with external identity providers (e.g., Google, Azure AD) via OIDC for authentication. This allows you to leverage existing user directories to authenticate users.
- Static Password Files: Kubernetes also supports simple static password files for user authentication. This method is less secure but still available for basic setups.
- Webhook: Kubernetes can authenticate users via external systems through a webhook mechanism. When a request is made, the API server can send a request to an external authentication service, which can verify the identity and respond with a success or failure status.

b) How Authentication Works in Kubernetes When a user or service interacts with the Kubernetes API, the kube-apiserver is responsible for authenticating the client. Once authenticated, the request proceeds to authorization (covered next).

2. Authorization in Kubernetes

Authorization determines whether an authenticated user has the appropriate permissions to perform a specific action on a resource. Kubernetes uses Role-Based Access Control (RBAC) as its primary authorization mechanism.

a) Key Authorization Concepts in Kubernetes

- Role-Based Access Control (RBAC): RBAC allows Kubernetes administrators to define policies that specify what actions are allowed or denied on resources based on roles. RBAC has the following core components:
    - Role: A set of permissions within a specific namespace. A Role grants access to resources within that namespace.
    - ClusterRole: A set of permissions that can be applied across all namespaces or cluster-wide.
    - RoleBinding: A binding that assigns a Role to a user, group, or service account within a namespace.
    - ClusterRoleBinding: A binding that assigns a ClusterRole to a user, group, or service account across the entire cluster.

Example RBAC Role:

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: Role

metadata:

  namespace: default

  name: read-only

rules:

  - apiGroups: [""]

    resources: ["pods"]

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

- ServiceAccount: Kubernetes allows workloads (like Pods) to authenticate as a ServiceAccount. A ServiceAccount is associated with specific RBAC roles, determining its permissions.
- ABAC (Attribute-Based Access Control): Although Kubernetes mainly uses RBAC for authorization, it also supports ABAC, which allows decisions based on attributes such as user attributes, resource attributes, etc. ABAC is less commonly used compared to RBAC.

b) How Authorization Works in Kubernetes After authentication, the kube-apiserver checks whether the authenticated identity is authorized to perform the requested operation. If authorized, the request proceeds to the next stage; if not, the request is denied with an HTTP 403 Forbidden response.

3. Admission Control in Kubernetes

Admission Control is a set of plugins that govern and enforce policies on requests to the Kubernetes API server. These plugins operate after authentication and authorization but before the API server writes objects to etcd. Admission control is essential for enforcing security policies, ensuring that resources conform to organizational standards, and preventing misconfigurations or harmful deployments.
a) Common Admission Controllers in Kubernetes

- NamespaceLifecycle: Ensures that the creation or deletion of resources occurs only within active namespaces.
- LimitRanger: Enforces resource limits (like CPU or memory) for pods and containers.
- ServiceAccount: Ensures that every Pod is associated with a ServiceAccount, and that ServiceAccount is properly configured.
- PodSecurityPolicy (PSP): This deprecated controller enforces security policies at the Pod level, such as restricting privileged containers or enforcing security contexts.
- ValidatingAdmissionWebhook: Allows integration with external admission control systems to validate requests dynamically.
- MutatingAdmissionWebhook: Allows modification of Kubernetes objects before they are persisted (e.g., adding default labels or annotations).
- AlwaysPullImages: Enforces that every Pod in a namespace pulls images from a remote registry rather than using locally cached images, helping prevent the use of outdated images.

b) How Admission Control Works in Kubernetes

- After authentication and authorization checks, the admission controllers evaluate the request and can:
    - Allow the request to be processed by the API server.

- Reject the request with an error message (e.g., if a resource request exceeds limits or violates a policy).
- Mutate the request by modifying it before saving to the cluster (e.g., adding labels or annotations).

For example, the LimitRanger admission controller ensures that Pods can only request resources within a predefined range. If a user attempts to create a Pod requesting more than the allowed resources, the controller will reject the request.

4. Example Workflow: Authentication, Authorization, and Admission Control

1. Authentication: A user submits a request to the Kubernetes API to create a Pod.
The API server verifies the user's identity (e.g., using certificates or tokens). If the authentication is successful, the request proceeds to the authorization phase.
2. Authorization: Kubernetes checks if the user has the correct permissions to create a Pod.
If the user has the necessary permissions (via RBAC or other methods), the request moves to admission control. If the user lacks permissions, they are denied with a 403 Forbidden error.
3. Admission Control: The request to create the Pod is then evaluated by admission controllers.
If the Pod spec is valid and compliant with policies (such as resource limits), the request is allowed. If there are any violations (e.g., exceeding resource limits), the request is rejected.

5. Practical Steps to Configure Authentication, Authorization, and Admission Control

a) Enabling and Configuring Authentication
Kubernetes supports multiple authentication methods. When setting up a cluster, you can configure which methods to use (certificates, tokens, OIDC, etc.) in the API server's flags or config file.

b) Configuring RBAC for Authorization
RBAC roles are defined in YAML files. For example, you can create roles for users or service accounts and bind them to specific resources with RoleBinding and ClusterRoleBinding objects.

Example of a RoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: read-only-binding

  namespace: default

subjects:

  - kind: User

    name: "johndoe"

    apiGroup: rbac.authorization.k8s.io

roleRef:
```

kind: Role

name: read-only

apiGroup: rbac.authorization.k8s.io

c) Setting Up Admission Controllers

Admission controllers are enabled via the --enable-admission-plugins flag in the Kubernetes API server configuration. Some admission controllers come pre-enabled by default, but you may need to enable others manually depending on your needs. For example, enabling the PodSecurityPolicy (deprecated) admission controller:

kube-apiserver --enable-admission-plugins=PodSecurityPolicy

d) Viewing and Debugging Authentication and Authorization
You can view the roles and bindings in the cluster using:

kubectl get roles,rolebindings --all-namespaces

kubectl get clusterroles,clusterrolebindings

You can debug authentication and authorization issues by looking at the API server logs, which often contain detailed error messages.

Lab Walkthrough:

1. Check if RBAC is enabled

kubectl api-versions | grep rbac

Expected output:
rbac.authorization.k8s.io/v1

For information only: If not enabled, it can be enabled through configuration:

kube-apiserver --authorization-mode=RBAC

2. Create a Service Account

kubectl create serviceaccount demo-user

Expected output:
serviceaccount/demo-user created

3. Create an authorization token for your Service Account:

```
TOKEN=$(kubectl create token demo-user)
```

4. Configure kubectl with your Service Account

```
kubectl config set-credentials demo-user --token=$TOKEN
```

Expected output:
User "demo-user" set.

5. Set a new context

```
kubectl config set-context demo-user-context --cluster=docker-desktop --user=demo-user
```

Expected output:
Context "demo-user-context" created.

6. Check your current context so you can switch back. Write this down.

```
kubectl config current-context
```

Expected output. Yours may differ

```
docker-desktop
```

7. Switch to the demo-user-context

```
kubectl config use-context demo-user-context
```

Expected output.
Switched to context "demo-user-context".

8. check if you can see pods

kubectl get pods

Expected output.
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:default:demo-user" cannot list resource "pods" in API group "" in the namespace "default"

9. Switch back to the default context and test that it's working

kubectl config use-context docker-desktop

Expected output. Your output may differ

Switched to context "docker-desktop".

10. Create a role. Put the following in a file and name it role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: demo-role
  namespace: default
rules:
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
      - list
      - create
```

```
    - update
```

11. Create the role

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

Expected output.
role.rbac.authorization.k8s.io/demo-role created

12. Create a RoleBinding. Put the following in rolebinding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: demo-role-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: demo-role
subjects:
  - namespace: default
    kind: ServiceAccount
    name: demo-user
```

13. Create the rolebinding

```
kubectl apply -f rolebinding.yaml
```

Expected Output:
rolebinding.rbac.authorization.k8s.io/demo-role-binding created

14. Verify your service account has been granted the Role's Permissions

kubectl config use-context demo-user-context

Expected output:
Switched to context "demo-user-context".

15. Verify that the get pods command now runs successfully:

kubectl get pods

Expected output: Your output may vary
NAME READY STATUS RESTARTS AGE

nginx 1/1 Running 0 20s

16. Create a pod

kubectl run nginx --image=nginx:latest

Expected output: Your output may vary
pod/nginx created

17. Test that the pod is running

kubectl get pods

Expected output: Your output may vary
NAME READY STATUS RESTARTS AGE

nginx 1/1 Running 0 30s

18. delete will not be successful since this is not included in the role that was created.

kubectl delete pod nginx.

Error from server (Forbidden): pods "nginx" is forbidden: User "system:serviceaccount:default:demo-user" cannot delete resource "pods" in API group "" in the namespace "default"

1. Switch back to the previous context and user

kubectl config use-context docker-desktop

Testing RBAC permissions using kubectl auth can-i

You can test the permissions of the current context:

kubectl auth can-i --list --namespace=foo

kubectl auth can-i get pods --namespace=foo

kubectl auth can-i get pods

kubectl auth can-i get pods --all-namespaces

Check to see if service account "foo" of namespace "dev" can list pods in the namespace "prod" You must be allowed to use impersonation for the global option "--as"

kubectl auth can-i list pods --as=system:serviceaccount:dev:foo -n prod

kubectl auth can-i delete pods --as=system:serviceaccount:default:demo-user -n default

6. Conclusion

As a Kubernetes Application Developer, understanding authentication, authorization, and admission control is crucial to ensuring the security and proper functioning of your applications and the cluster. Here's a quick summary:

- Authentication ensures that only valid users or services can access the cluster.
- Authorization determines what an authenticated user or service is allowed to do, based on roles and permissions.
- Admission Control enforces policies and validates or modifies requests before they are applied to the cluster.

Together, these three mechanisms help secure the Kubernetes ecosystem by ensuring that only authorized users and processes can modify resources, while also enforcing best practices and organizational policies.

Domain 4.3: Understand Requests, Limits, and Quotas

In Kubernetes, managing resources effectively is critical for ensuring the stability, performance, and efficiency of applications running in the cluster. As a Kubernetes Application Developer (CKAD), it is essential to understand

- resource requests,
- limits, and
- quotas,

as these directly influence how workloads (Pods and containers) are

- scheduled,
- executed, and
- constrained within the cluster.

This section will cover the core concepts related to resource management and demonstrate how you can define and configure resource requests, limits, and quotas for containers and Pods to ensure that your applications are properly allocated resources.

1. Resource Requests and Limits

In Kubernetes, resources like CPU and memory (RAM) are allocated to containers within a Pod. Each container in a Pod can specify requests and limits for these resources.

a) Resource Requests
A request is the amount of a particular resource that the container is guaranteed to receive when it is scheduled onto a node. When a container is scheduled, Kubernetes ensures that enough resources are available on the node to meet the requested amount.

- CPU Requests:

The amount of CPU requested by the container. Kubernetes will schedule the container onto a node only if that node has enough available CPU.

- Memory Requests:

The amount of memory the container requests. Similarly, the container will only be scheduled onto a node if there is sufficient memory available.
Example:

```
apiVersion: v1

kind: Pod

metadata:

  name: resource-request-example

spec:

  containers:

  - name: my-container

    image: my-image
```

```
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
```

In this example, the container requests 64 MB of memory and 250 milli-CPU (0.25 CPU core).

Understanding CPU units

The CPU resource is measured in *CPU* units. One CPU, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal Intel processor with Hyperthreading

Fractional values are allowed. A Container that requests 0.5 CPU is guaranteed half as much CPU as a Container that requests 1 CPU. You can use the suffix m to mean milli. For example 100m CPU, 100 milliCPU, and 0.1 CPU are all the same. Precision finer than 1m is not allowed.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine.

b) Resource Limits

A limit is the maximum amount of a resource that a container can use. If a container tries to exceed its specified limit, Kubernetes may throttle the container's CPU usage or kill the container if it exceeds its memory limit (and it will be restarted, depending on the restart policy).

- CPU Limits: If a container tries to exceed the CPU limit, Kubernetes will throttle the container's CPU usage to prevent it from consuming too much.
- Memory Limits: If a container exceeds its memory limit, Kubernetes will terminate the container and restart it (OOMKill), as memory consumption is a critical resource for the node.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-limit-example
spec:
```

```
containers:

- name: my-container

  image: my-image

  resources:

    limits:

      memory: "128Mi"

      cpu: "500m"
```

In this example, the container can use up to 128 MB of memory and 500 milli-CPU (0.5 CPU core), but it is not allowed to consume more than that.

c) Requests vs. Limits:

- Requests ensure that the container has the required resources to run.
- Limits ensure that the container cannot use more resources than the specified maximum, preventing one container from starving other containers on the same node.

Important: If you don't specify limits, containers can use as much resource as is available on the node, which could lead to resource contention or instability.

2. Resource Quotas

Resource Quotas are a way to limit the amount of resources that can be consumed by a specific namespace within a Kubernetes cluster. This allows administrators to enforce policies that prevent one team or application from consuming all the available resources in the cluster.
A ResourceQuota object can define limits on resources like CPU, memory, number of Pods, number of Persistent Volume Claims (PVCs), and more.

a) Common ResourceQuota Constraints

- CPU: Limits the total CPU usage in the namespace.
- Memory: Limits the total memory usage in the namespace.
- Pods: Limits the number of Pods that can be created within the namespace.
- PersistentVolumeClaims: Limits the number of Persistent Volume Claims that can be created.

b) Example of a ResourceQuota

```
apiVersion: v1

kind: ResourceQuota

metadata:

  name: example-quota

  namespace: my-namespace

spec:
```

```
  hard:
    requests.cpu: "4"
    requests.memory: "8Gi"
    limits.cpu: "8"
    limits.memory: "16Gi"
    pods: "10"
    persistentvolumeclaims: "5"
```

In this example:

- The namespace my-namespace can request up to 4 CPUs and 8 GiB of memory in total.
- The namespace is allowed to create up to 10 Pods and 5 PersistentVolumeClaims.
- The maximum limits for CPU and memory are set at 8 CPUs and 16 GiB of memory.

c) ResourceQuota and Namespace Limits
When a ResourceQuota is defined for a namespace, Kubernetes will enforce the limits on the total resource usage for that namespace. This means if you exceed the quota, Kubernetes will prevent new resources from being created until resources are freed up or the quota is adjusted.

3. Best Practices for Resource Requests, Limits, and Quotas

As a Kubernetes Application Developer, understanding how to configure requests, limits, and quotas is essential for the stability and efficiency of your applications. Here are some best practices:
a) Always Define Requests and Limits
Always specify both requests and limits for containers. If only one is defined (e.g., only a request but no limit), it can lead to unexpected behaviors, such as excessive resource consumption or throttling.
Requests should reflect the minimum resources necessary for your application to run, while limits should reflect the maximum acceptable resources your application should consume.
b) Monitor Resource Usage
Continuously monitor your containers and Pods for resource usage to ensure that you are not over- or under-allocating resources. Use Kubernetes tools like kubectl top to view resource usage in your cluster.
Example:

```
kubectl top pod
```

c) Optimize Resource Requests
Setting resource requests too high can lead to overprovisioning, meaning your cluster may run out of resources. On the other hand, setting requests too low may lead to performance issues. Use monitoring tools like Prometheus to track resource usage and adjust requests and limits accordingly.
d) Use Resource Quotas to Control Resource Usage
Enforce ResourceQuotas to ensure that no single team or application uses more than its fair share of resources. For multi-tenant clusters, use ResourceQuotas to avoid resource contention between teams or applications.

4. How to View and Manage Resources

a) View Resources in the Cluster
You can check the resource requests and limits of containers and Pods using kubectl describe:
kubectl describe pod <pod-name>

This command will provide detailed information about the resource requests and limits for each container in the Pod.
b) View Resource Quotas in a Namespace
You can view the existing ResourceQuota objects in a specific namespace using the following command:
kubectl get resourcequota -n <namespace>

This will display the resource quotas in the specified namespace, along with the usage and available resources.

5. Troubleshooting Resource Issues

When resource issues arise, such as Pods being OOMKilled (Out of Memory) or Pods being throttled due to CPU limits, you can troubleshoot these problems with the following steps:
a) Check Resource Usage
Use kubectl top pod to monitor how much CPU and memory your Pods are using.
kubectl top pod

b) Check Pod Events
If Pods are being killed due to resource constraints, you can inspect their events to get more details:
kubectl describe pod <pod-name>

Look for events such as "OOMKilled" or "Memory Pressure".
c) Adjust Resource Requests and Limits
If your application is resource-intensive, consider increasing the resource requests and limits. Conversely, if your application is using fewer resources than expected, you can scale back the requests and limits.

6. Conclusion

Understanding resource requests, limits, and quotas is crucial for maintaining a healthy, efficient, and stable Kubernetes environment. As a Kubernetes Application Developer (CKAD), your role is to ensure that resources are allocated properly to your workloads, avoiding over- or under-provisioning.
To summarize:

- Requests: The minimum amount of resources that a container is guaranteed to receive.
- Limits: The maximum amount of resources a container can use.
- Resource Quotas: Limits on resources within a namespace to prevent one team or workload from consuming excessive resources.

By carefully managing these resources, you help ensure that Kubernetes clusters remain efficient, responsive, and predictable, while also preventing potential issues related to resource contention.

Domain 4.4: Understand ConfigMaps

In Kubernetes, ConfigMaps provide a way to manage configuration data for applications in a centralized and decoupled manner. As an Application Developer (CKAD), understanding how to use ConfigMaps is essential for managing configuration in a flexible and environment-agnostic way.

What is a ConfigMap?

A ConfigMap is an API object in Kubernetes that allows you to store configuration data in key-value pairs. You can then inject this data into your containers as:

- environment variables,
- command-line arguments, or
- configuration files.

By using ConfigMaps, you can separate configuration from your application code, making your applications more portable and easier to manage.
ConfigMaps are useful for storing configuration data that might change based on the environment (e.g., development, staging, production) but should remain consistent across deployments.
1. Creating a ConfigMap There are several ways to create a ConfigMap in Kubernetes:
a) Using kubectl from a File If you have a file containing configuration data, you can create a ConfigMap from that file using the kubectl create configmap command. Example:
kubectl create configmap my-config --from-file=config.txt

This command creates a ConfigMap named my-config using the contents of config.txt.
b) Using kubectl from Literal Values You can create a ConfigMap directly from the command line by specifying key-value pairs. Example:
kubectl create configmap my-config --from-literal=key1=value1 --from-literal=key2=value2

This creates a ConfigMap named my-config with two entries: key1=value1 and key2=value2.
c) Using a YAML Manifest You can define a ConfigMap in a YAML manifest and apply it using kubectl apply. Example ConfigMap YAML:

```
apiVersion: v1

kind: ConfigMap

metadata:

  name: my-config

data:

  key1: value1

  key2: value2
```

Apply it with:
kubectl apply -f configmap.yaml

2. Using ConfigMaps in Pods Once you've created a ConfigMap, you can inject its data into Pods in several ways:
a) As Environment Variables You can reference the keys from a ConfigMap as environment variables in your containers. Example:

```
apiVersion: v1

kind: Pod

metadata:

  name: configmap-env-example

spec:

  containers:

  - name: my-container

    image: nginx

    envFrom:

    - configMapRef:

        name: my-config
```

In this example, all keys in the my-config ConfigMap are injected into the container as environment variables.
b) As Environment Variables (Individual Keys) If you want to specify individual keys, you can reference them explicitly. Example:

```
apiVersion: v1

kind: Pod

metadata:

  name: configmap-env-example

spec:

  containers:

  - name: my-container

    image: nginx

    env:
```

```
    - name: KEY1
      valueFrom:
        configMapKeyRef:
          name: my-config
          key: key1
```

In this case, only the value of key1 from the my-config ConfigMap is set as the KEY1 environment variable in the container.

c) As Command-line Arguments You can pass the values from a ConfigMap as command-line arguments to the container's entrypoint. Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-args-example
spec:
  containers:
  - name: my-container
    image: nginx
    command:
    - "/bin/sh"
    - "-c"
    - "echo $(KEY1)"
    env:
    - name: KEY1
      valueFrom:
        configMapKeyRef:
          name: my-config
          key: key1
```

Here, the value of key1 in the my-config ConfigMap is passed as a command-line argument to the container, and the container will print it.
d) As Configuration Files You can mount the ConfigMap as a file inside the container. This is useful when the configuration is a set of files or needs to be structured in a specific way. Example:

```
apiVersion: v1

kind: Pod

metadata:

  name: configmap-volume-example

spec:

  containers:

  - name: my-container

    image: nginx

    volumeMounts:

    - name: config-volume

      mountPath: /etc/config

      readOnly: true

  volumes:

  - name: config-volume

    configMap:

      name: my-config
```

In this case, the my-config ConfigMap will be mounted as files in the /etc/config directory within the container. Each key in the ConfigMap becomes a separate file with the corresponding value as the file content.

3. Updating ConfigMaps You can update a ConfigMap at any time. However, updating a ConfigMap will not automatically trigger a Pod restart. You need to either manually restart the Pods or use strategies like rolling updates to ensure that the new configuration is applied.

a) Updating a ConfigMap Using kubectl You can update a ConfigMap by reapplying the new configuration using kubectl apply.

```
kubectl apply -f configmap.yaml
```

b) Triggering Pod Restarts After updating a ConfigMap, you may want to restart the Pods that use the ConfigMap to pick up the new configuration. You can do this by deleting the Pods, and Kubernetes will automatically recreate them.

kubectl delete pod <pod-name>

Alternatively, if you're using a Deployment, you can trigger a rolling update to apply the new configuration:
kubectl rollout restart deployment <deployment-name>

Lab Walkthrough

1. Put this information in cm.yaml

```
apiVersion: v1

kind: ConfigMap

metadata:

  name: demo-config

data:

  database_host: "192.168.0.1"

  debug_mode: "1"

  log_level: "verbose"
```

2. Create the configmaps

kubectl appply -f cm.yaml

Expected output:
configmap/demo-config created

3. List the configmaps

kubectl get configmaps

Expected output: (yours may differ)
NAME DATA AGE

demo-config 3 5m

4. Inspect the confimaps
kubectl describe configmap demo-config

Expected output:
Name: demo-config

Namespace: default

Labels: <none>

Annotations: <none>

Data

====

database_host:

192.168.0.1

debug_mode:

1

log_level:

----

verbose

BinaryData

====

Events: <none>

5. Get the information as a json

```
kubectl get configmap demo-config -o jsonpath='{.data}' | jq
```

Expected output:

```
{
  "database_host": "192.168.0.1",
  "debug_mode": "1",
  "log_level": "verbose"
}
```

6. Mount ConfigMaps as ENV. put the following in a file called cm-pod.yaml and apply it

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: app
      command: ["/bin/sh", "-c", "printenv"]
      image: busybox:latest
      envFrom:
        - configMapRef:
            name: demo-config
```

7. Apply it Expected output:

```
kubectl apply -f cm-pod.yaml
```

pod/demo-pod created

8. check logs

kubectl logs pod/demo-pod

Expected output.

...

database_host=192.168.0.1

debug_mode=1

log_level=verbose

...

9. Mount ConfigMaps as Command Line Arguments

Create a file called netpol2.yaml and paste the following content.

```
apiVersion: v1

kind: Pod

metadata:

  name: demo-pod

spec:

  containers:

    - name: app

      command: ["demo-app", "--database-host", "$(DATABASE_HOST)"]

      image: demo-app:latest

      env:

        - name: DATABASE_HOST

          valueFrom:

            configMapKeyRef:

              name: demo-config

              key: database_host
```

10. Mount ConfigMaps as Volumes. save to demo-pod.yaml

```
apiVersion: v1

kind: Pod

metadata:

  name: demo-pod

spec:

  containers:

    - name: app

      command: ["ls", "/etc/app-config"]

      image: demo-app:latest

      volumeMounts:

        - name: config

          mountPath: "/etc/app-config"

          readOnly: true

  volumes:

    - name: config

      configMap:

        name: demo-config
```

```
pod/demo-pod created
```

11. Check the values.

```
kubectl logs pod/demo-pod
```

Expected output:

```
database_host

debug_mode
```

log_level

4. Best Practices for Using ConfigMaps
As a Kubernetes Application Developer (CKAD), you should consider these best practices when working with ConfigMaps:
a) Use ConfigMaps for Non-sensitive Data ConfigMaps are intended for non-sensitive configuration data (like feature flags, configuration parameters, etc.). For sensitive data (like passwords, API keys), use Secrets instead.
b) Version Your ConfigMaps For better management and to ensure that changes in the configuration are traceable, consider versioning your ConfigMaps. You can append version numbers to the ConfigMap names or store them in a source control system.
c) Use ConfigMap for Environment-Specific Configuration You can use different ConfigMaps for different environments (e.g., config-dev, config-prod) and inject the appropriate one into your Pods based on the environment.
d) Separate Configuration from Code Decouple your application's code from its configuration. By using ConfigMaps, you make your applications more portable, as configurations can be modified without changing the application code.
e) Avoid Hardcoding ConfigMap Names Whenever possible, avoid hardcoding ConfigMap names in your code. Instead, use environment variables or Helm charts to make it easier to manage configurations for multiple environments.

5. Troubleshooting ConfigMap Issues

If your application is not behaving as expected after configuring or updating a ConfigMap, consider these troubleshooting steps:
a) Check ConfigMap Content You can view the content of a ConfigMap using:
kubectl describe configmap <configmap-name>

b) Check Pod Logs If the container is not behaving as expected, check its logs to see if there are any errors related to the configuration:
kubectl logs <pod-name>

c) Verify Volume Mounts If you are mounting the ConfigMap as a file, ensure that the volume mount path is correct and that the files are available inside the container.

6. Conclusion

ConfigMaps are a powerful and flexible way to manage configuration data in Kubernetes. As a Kubernetes Application Developer (CKAD), you need to understand how to create, manage, and use ConfigMaps to inject configuration into your applications.
To summarize:

1. ConfigMaps store non-sensitive configuration data as key-value pairs.
2. They can be used in Pods as environment variables, command-line arguments, or mounted as files.

Best practices include versioning, separating configuration from code, and using ConfigMaps for environment-specific settings.

By understanding and using ConfigMaps effectively, you can make your Kubernetes applications more dynamic, flexible, and easier to manage across different environments.

Domain 4.6: Create & Consume Secrets

In Kubernetes, Secrets are a crucial way to store sensitive information such as passwords, API tokens, SSH keys, and other credentials in a secure and encrypted manner. These Secrets can be consumed by Pods and containers to access secure data without hardcoding it into your application code or configuration files.
In this section, we will cover the following topics:

- How to create Kubernetes Secrets.
- How to consume Secrets in a Pod.
- Best practices for handling Secrets securely.

1. What are Kubernetes Secrets?

Kubernetes Secrets store sensitive information, such as:

- Passwords
- OAuth tokens
- SSH keys
- TLS certificates

Secrets are stored in the cluster and can be used by Pods and other resources in the cluster to access sensitive information without exposing it in your configuration files. The key advantage of using Secrets is that they are encrypted at rest, and their values can be managed securely.

2. Creating Kubernetes Secrets

Secrets can be created in Kubernetes in several ways: manually via kubectl, through YAML files, or using external tools (e.g., Helm, Kustomize).
a) Create Secrets Using kubectl Command You can create Secrets directly from the command line using kubectl create secret. There are various ways to create Secrets depending on how you want to provide the secret data.
Example 1: Create a Secret from Literal Values
kubectl create secret generic my-secret \

--from-literal=username=admin \

--from-literal=password=secretpassword

This creates a Secret named my-secret with two keys: username and password, and their respective values.
Example 2: Create a Secret from a File
If you have a file containing sensitive information, you can create a Secret from the file content.
kubectl create secret generic my-secret \

--from-file=my-ssh-key=/path/to/ssh/keyfile

This creates a Secret named my-secret with a key my-ssh-key containing the contents of the file /path/to/ssh/keyfile.
Example 3: Create a Secret from Multiple Files
kubectl create secret generic my-secret \

```
  --from-file=ssh-private-key=/path/to/private_key \

  --from-file=ssh-public-key=/path/to/public_key
```

This creates a Secret with two files, ssh-private-key and ssh-public-key, containing the respective SSH keys.
b) Create Secrets Using YAML Secrets can also be defined in a YAML manifest. This is useful for version control and automation.
Example:

```
apiVersion: v1

kind: Secret

metadata:

  name: my-secret

type: Opaque

data:

  username: YWRtaW4=    # Base64 encoded string for 'admin'

  password: c2VjcmV0cGFzc3dvcmQ=    # Base64 encoded string for 'secretpassword'
```

- type: Opaque: This indicates the secret is of a general type and not associated with any specific Kubernetes use case (e.g., Docker registry secrets).
- data: The sensitive data in a Secret must be base64 encoded. In this example, the values for username and password are encoded strings for admin and secretpassword.

To apply the YAML manifest:

```
kubectl apply -f secret.yaml
```

3. Consuming Secrets in a Pod

Once the Secret is created, it can be consumed by a Pod in two primary ways:

1. Environment Variables
2. Mounted as Volumes

a) Consume Secrets as Environment Variables
You can expose a Secret as an environment variable inside a container. This method is useful when your application needs to access the Secret as a simple environment variable.
Example: Consume a Secret as an environment variable in a Pod

```
apiVersion: v1

kind: Pod

metadata:

  name: secret-demo
```

```
spec:
  containers:
  - name: my-container
    image: nginx
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: password
```

In this example:

1. The username and password keys from the Secret my-secret are injected into the container as environment variables (SECRET_USERNAME and SECRET_PASSWORD).
2. The application inside the container can now access the values through environment variables.

b) Consume Secrets as Mounted Volumes

Secrets can also be mounted as volumes, which allows your application to access them as files within the container. This is useful when you need to pass a file-based Secret (like an SSL certificate, SSH key, etc.).

Example: Consume a Secret as a mounted volume

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-demo
spec:
  containers:
```

```
  - name: my-container
    image: nginx
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/secrets
  volumes:
  - name: secret-volume
    secret:
      secretName: my-secret
```

In this example:

1. The Secret my-secret is mounted as a volume at /etc/secrets inside the container.
2. The container will have access to all the keys in the Secret as files under /etc/secrets, e.g., /etc/secrets/username and /etc/secrets/password.

Lab Walkthrough

1. Encode the secrets

```
echo -n 'admin' | base64

echo -n 'password' | base64
```

2. Create the secrets.yaml with this content

```
apiVersion: v1

kind: Secret

metadata:

  name: demo-secret

type: Opaque

data:

  username: YWRtaW4=

  password: cGFzc3dvcmQ=
```

or

```
apiVersion: v1

kind: Secret

metadata:

  name: demo-secret

type: Opaque

stringData:

  username: admin

  password: password

kubectl -n secrets-demo apply -f demo-secret.yaml
```

Expected output:
secret/demo-secret created

3. Mount Secret as EWnv Variables

```
apiVersion: v1

kind: Pod

metadata:

  name: env-pod

spec:

  containers:

    - name: secret-test

      image: nginx

      command: ['sh', '-c', 'echo "Username: $USER" "Password: $PASSWORD"']

      env:

        - name: USER

          valueFrom:

            secretKeyRef:

              name: database-credentials
```

```
      key: username.txt
    - name: PASSWORD
      valueFrom:
        secretKeyRef:
          name: database-credentials
          key: password.txt
```

4. Create the file

```
kubectl -n secrets-demo apply -f secret-test-evn-pod.yaml
```

Expected output:
secret/demo-secret created

5. Describe the secret

```
kubectl -n secrets-demo describe pod env-pod
```

Expected output:

6. Mount the volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test-pod
spec:
  containers:
  - name: secret-test
    image: nginx
    volumeMounts:
```

```
    - name: secret-volume

      mountPath: /etc/config/secret

  volumes:

  - name: secret-volume

    secret:

      secretName: database-credentials
```

7. Apply it

```
kubectl -n secrets-demo apply -f secret-test-volume-pod.yaml
```

8. check logs

```
kubectl -n secrets-demo logs env-pod
```

9. Describe the secrets

```
kubectl -n secrets-demo describe pod env-pod
```

10. See the volumes

```
kubectl -n secrets-demo exec volume-test-pod -- cat /etc/config/secret/username.txt

kubectl -n secrets-demo exec volume-test-pod -- cat /etc/config/secret/password.txt

kubectl -n secrets-demo exec volume-test-pod -- ls /etc/config/secret
```

4. Best Practices for Managing Secrets

- Avoid Hardcoding Secrets: Never hardcode sensitive data like passwords, API tokens, or certificates directly into your code or Kubernetes YAML files.
- Use kubectl to Read Secrets: Avoid printing Secrets in plain text to the console. Use kubectl get secret my-secret -o yaml or kubectl describe secret my-secret to view Secrets in a secure way.
- Limit Access: Only allow the Pods or services that need to access a particular Secret to do so. Use Role-Based Access Control (RBAC) to control access to Secrets.
- Encrypt Secrets at Rest: Kubernetes supports encrypting Secrets at rest. Ensure that your Kubernetes cluster has this enabled.
- Use External Secrets Management: If you need to integrate with enterprise-grade secrets management solutions, consider using tools like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault.

Base64 Encoding Is Not Encryption: Secrets are base64 encoded, not encrypted. Ensure that your Kubernetes cluster uses encryption at rest to protect Secret data.

5. Deleting a Secret

To delete a Secret that is no longer required:

```
kubectl delete secret my-secret
```

This removes the Secret from the cluster. Make sure that any applications or resources depending on it are updated or deleted before removing a Secret.

6. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to securely create and consume Secrets is vital for building safe and efficient applications. Whether you choose to pass Secrets as environment variables or mount them as volumes, ensuring that they are handled securely and not exposed in your code is crucial for maintaining the confidentiality of sensitive data.

1. Creating Secrets: Use kubectl, YAML manifests, or external tools.
2. Consuming Secrets: Inject them as environment variables or mount them as volumes in your Pods.
3. Best Practices: Implement security measures like encryption at rest, and limit access to Secrets using RBAC.

By mastering these concepts, you can securely manage sensitive data in your Kubernetes clusters and build more secure, reliable applications.

Domain 3.5: Debugging in Kubernetes

Debugging is a critical skill for a Kubernetes Application Developer (CKAD). Kubernetes is a powerful platform for deploying and managing containerized applications, but it introduces complexity, especially when issues arise. Whether it's a misconfigured deployment, a failed Pod, or a networking issue, knowing how to debug and troubleshoot effectively is essential. This section will cover the tools and techniques you can use to debug applications and Kubernetes resources.

1. Key Debugging Concepts in Kubernetes

When debugging in Kubernetes, it's important to understand the following key concepts: Pod Lifecycle: Pods can go through various states like Pending, Running, Succeeded, or Failed. Understanding the lifecycle helps identify where a problem might be occurring.

1. Kubernetes Resources: Debugging Kubernetes resources (like Pods, Deployments, Services, and ConfigMaps) involves checking their configurations, status, and logs.
2. Cluster Context: Debugging often requires understanding the cluster context, including node status, kubelet logs, and network configuration.
3. Container Behavior: Problems with containers themselves (such as crashes, memory limits, or network timeouts) often manifest in the container logs.

2. Common Debugging Scenarios in Kubernetes

As an application developer, you'll face several common issues when working with Kubernetes. Here are some of the most frequent ones: Pods Not Starting: Pods that are stuck in the Pending or CrashLoopBackOff state. Service Unavailability: Pods are running but cannot be accessed via the Kubernetes Service. Configuration Issues: Incorrect ConfigMaps, Secrets, or environment variables affecting the behavior of the application. Resource Constraints: Pods or containers being terminated or restarted due to memory or CPU constraints.

3. Debugging Tools and Techniques

Kubernetes provides a variety of tools and commands to debug these issues. Here's a breakdown of the most useful debugging techniques:

1. kubectl describe

The kubectl describe command provides detailed information about Kubernetes resources. It's extremely useful for debugging issues with Pods, Services, Deployments, and more. Pods: To describe a Pod, use the following command:
kubectl describe pod <pod-name>

This will give you details about the Pod, including events, container status, resource usage, and any error messages.
Example output: If a Pod is stuck in Pending or CrashLoopBackOff, kubectl describe pod will often provide insight into what's going wrong (e.g., resource allocation, failed pull image, etc.).

2. kubectl logs

Logs are one of the most effective ways to understand what's happening inside a container. If a Pod is crashing or behaving unexpectedly, check its logs: Basic logs:

kubectl logs <pod-name> -c <container-name>

Stream logs in real-time:
kubectl logs -f <pod-name> -c <container-name>

Previous logs: If the container crashed and was restarted, you can view logs from the previous instance:
kubectl logs <pod-name> -c <container-name> --previous

Logs often provide the most direct insight into why a container is failing, such as uncaught exceptions, resource issues, or missing dependencies.

3. kubectl get

The kubectl get command is used to retrieve information about Kubernetes resources. It can be helpful to view the current state of resources and identify any issues. Pods: To get information about the status of Pods in a namespace:
kubectl get pods -n <namespace>

Deployment: To check the status of a deployment:
kubectl get deployment <deployment-name> -n <namespace>

Services: To check the Services running in the cluster:
kubectl get svc -n <namespace>

The output will show the current state, such as whether the Pods are running, and if there are any issues with resource allocation.

4. kubectl top

The kubectl top command is used to monitor resource usage (CPU and memory) for nodes and Pods. It can help identify if your application is running out of resources. Monitor nodes:

kubectl top nodes

Monitor Pods:
kubectl top pods -n <namespace>

If a Pod is experiencing memory or CPU issues, it may be terminated or fail to start. kubectl top helps identify these problems.

5. kubectl port-forward

If you need to debug an application from inside a Pod, the kubectl port-forward command allows you to access an application running inside a Kubernetes cluster without exposing it externally. Example command:

kubectl port-forward pod/<pod-name> 8080:80

This forwards port 8080 on your local machine to port 80 inside the container. This is useful for debugging an application's behavior from your local machine without exposing it to the public internet.

6. kubectl exec

The kubectl exec command allows you to execute commands directly inside a running container. This is useful for inspecting the state of an application or running debugging tools in the container. Example command:

kubectl exec -it <pod-name> -c <container-name> -- /bin/bash

This opens an interactive terminal session inside the container, allowing you to run commands such as ps, top, curl, etc., to investigate the issue.

7. Checking Events

Kubernetes events provide insights into what's happening in the cluster, including Pod creation, failure, and resource allocation issues. You can view events using:

kubectl get events -n <namespace>

Events help identify why a Pod might be stuck in the Pending state, why it was evicted, or why a deployment failed to rollout.

4. Troubleshooting Common Issues

Here are some common issues and how to debug them:

1. Pod Stuck in Pending

Check the Pod's events: Often, Pods stuck in the Pending state are waiting for resources (like CPU or memory). Use

kubectl describe pod <pod-name>

to see if there are any warnings about resource availability.
Check resource requests/limits: If your Pod is requesting more resources than are available, it may not be scheduled. Check if the Pod has resource requests/limits set appropriately.

2. Pod Crashes or Restarts (CrashLoopBackOff)

View logs:

Use kubectl logs <pod-name>

to see what's happening in the container. Look for errors like missing files, connection issues, or configuration problems.

- Check for resource issues: Make sure the container isn't being killed due to resource limits (memory/CPU). Use kubectl top pods to monitor usage.
- Check readiness/liveness probes: A misconfigured probe can cause Kubernetes to restart the Pod frequently. Review the health check configurations.

3. Service Not Accessible

- Verify Service: Use kubectl describe svc to ensure the Service is correctly pointing to the Pods and has the correct ports.
- Check Network Policies: Ensure there are no network policies blocking access to the service.
- Check Pod Logs: If the application inside the Pod is failing to respond to requests, check the logs of the container to ensure the application is running and listening on the expected port.

4. Misconfigured Environment Variables or ConfigMaps

- Inspect ConfigMap/Secret: Use kubectl describe configmap or kubectl describe secret to ensure that the correct environment variables are being set.
- Check Pods' Environment: Use kubectl exec to inspect the environment inside the container to ensure that variables are correctly passed.

5. Debugging Network Issues

Networking problems often occur when Pods are unable to communicate with each other, a service, or the outside world. Common network issues include DNS resolution problems, service misconfiguration, or blocked ports.

- Test Connectivity: Use kubectl exec to ping other Pods or services to verify connectivity.
- Check Network Policies: Ensure that Kubernetes network policies are not unintentionally blocking traffic.
- Check DNS Resolution: If Pods cannot resolve domain names, it might indicate a DNS issue. Check the CoreDNS logs using kubectl logs -n kube-system .

6. Conclusion

Debugging is a crucial skill for a Kubernetes Application Developer (CKAD). Kubernetes provides several powerful tools and commands like

kubectl describe, kubectl logs, kubectl exec, and kubectl top

to help you troubleshoot and identify issues with Pods, containers, and other resources.
By familiarizing yourself with these tools and techniques, you will be better equipped to identify, diagnose, and resolve issues in your Kubernetes-based applications, ensuring smooth operation in a cloud-native environment.