Gmail

Sandhya Babu <sandhya.bca7@gmail.com>

Day24 Challenge: Pod Mastery: Securing, Scaling, and Managing Kubernetes Pods
1 message

Sagar Utekar <getfitwithsagar2366@gmail.com> Thu, Dec 26, 2024 at 8:00 AM
Bcc: sandhya.bca7@gmail.com

Hello Learners,

Welcome back to another thrilling episode of the DevOps SRE Daily Challenge!

Congratulations! 🎉 We've already **covered 25% of the CKA syllabus** in just 6 days. This CKA Series is part of the larger Daily DevOps/SRE Challenge Series, a journey spanning 365 days of continuous learning and challenges. Today marks Day 24 overall in the series and Day 7 specifically for the CKA challenges.

Starting from today, we dive into the second section of the CKA syllabus: **Workloads and Scheduling**, where you'll begin to understand how Kubernetes manages Pods, resource allocation, and workload execution effectively.

Today, we're focusing on **Kubernetes Pods** - your application's most basic and fundamental unit of deployment. We're going to walk through the creation, configuration, and securing of Pods, while also integrating them with security mechanisms like Service Accounts, RBAC, and Network Policies.

Understanding Kubernetes Pods

What is a Pod?

- A Pod is the smallest deployable unit in Kubernetes.
- It encapsulates one or more containers (e.g., Docker containers), storage resources, and network configurations.

- Pods are designed to host tightly coupled application containers that need to share resources, such as:
  - **Network**: Containers in a Pod share the same IP address and port space.
  - **Storage**: They can share volumes for persistent or ephemeral storage.

Pod Lifecycle

- Pods are ephemeral by design. If a Pod dies, Kubernetes creates a new Pod (with a new IP) as part of a higher-level construct, like a Deployment or ReplicaSet.

Pod Networking

1. ### How Pods Get IPs

   - Each Pod is assigned a unique IP address by the Kubernetes network plugin (CNI, e.g., Calico, Flannel).
   - All containers in the Pod share this IP, allowing them to communicate with each other using `localhost`.
   - Pods communicate with other Pods or services through this IP or via DNS provided by Kubernetes.

2. ### Networking Within a Pod

   - Containers communicate using `localhost` and specific ports.
   - A shared network namespace ensures that all containers in the same Pod can access the same network interfaces and ports.

3. ### Networking Across Pods

   - Pods in different namespaces or clusters communicate using service DNS names or Pod IPs.
   - Network policies can restrict or allow communication between Pods based on labels, namespaces, or IP ranges.

4. ### Networking Across Namespaces

   - Namespaces isolate resources, including Pods, Services, and ConfigMaps.
   - Pods in different namespaces need explicit DNS names (e.g., `pod-name.namespace.svc.cluster.local`) for communication.

Pod Storage

1. **Volume Sharing**
   - Containers in a Pod can share storage volumes mounted at specified paths.
   - Volumes persist data between container restarts within the Pod.
2. **Ephemeral vs. Persistent Volumes**
   - Ephemeral: Data exists only as long as the Pod is alive (e.g., emptyDir).
   - Persistent: Data persists even after the Pod is deleted (e.g., PersistentVolume).

Pods and Namespaces

1. **Default Namespace**
   - If not specified, Pods are created in the default namespace.
2. **Custom Namespaces**
   - Namespaces provide logical separation of resources.
   - Pods in one namespace cannot directly access resources in another namespace unless allowed by network policies or RBAC.
3. **Pod DNS Naming**
   - Pods in the same namespace can be addressed by their name.
   - Pods in different namespaces use `pod-name.namespace.svc.cluster.local`.

Sharing Resources in a Pod

1. **Shared Network**
   - All containers share the same network namespace, allowing easy communication without exposing ports externally.

2. **Shared Storage**
   - Volumes enable containers to share data.
   - Use cases include:
     - Logs shared between containers.
     - Temporary data passed between containers.
3. **CPU and Memory**
   - Resource limits can be set for containers, but they still belong to the same cgroup.

How Pods Fit into Kubernetes

1. **Workloads**
   - Pods are managed by controllers like Deployments, StatefulSets, or DaemonSets for replication and self-healing.
2. **Ephemeral Nature**
   - Pods are not designed to be durable. Use PersistentVolumes or StatefulSets for stateful applications.

Here's a **basic YAML file** for a Kubernetes Pod, along with explanations for each line:

```
apiVersion: v1                    # Specifies the API version for the resource. 'v1' is
for core Kubernetes resources like Pods.
kind: Pod                         # Specifies the type of resource being created. Here,
it's a Pod.
metadata:                         # Metadata contains information about the Pod.
  name: my-pod                    # The name of the Pod, must be unique within the
namespace.
  namespace: default              # (Optional) Specifies the namespace where the Pod
will reside. Default is 'default' if not specified.
  labels:                         # Key-value pairs to categorize the Pod.
```

```
    app: my-app                    # A label to identify the application this Pod belongs
to.
spec:                              # Specification of the desired state of the Pod.
  containers:                      # Lists all the containers that will run within this
Pod.
  - name: my-container             # The name of the container within the Pod.
    image: nginx:latest            # Specifies the container image to use. Here, it's
'nginx' with the 'latest' tag.
    ports:                         # Defines the ports the container will expose.
    - containerPort: 80            # Specifies that the container listens on port 80.
    resources:                     # Specifies resource requests and limits for the
container.
      requests:                    # Minimum resources the container requires.
        cpu: "100m"                # 100 milliCPU (0.1 CPU core).
        memory: "128Mi"            # 128 MiB of memory.
      limits:                      # Maximum resources the container can use.
        cpu: "250m"                # 250 milliCPU (0.25 CPU core).
        memory: "256Mi"            # 256 MiB of memory.
    volumeMounts:                  # Specifies how volumes are mounted into the
container.
    - name: shared-data            # The name of the volume to mount.
      mountPath: /data             # Path inside the container where the volume will be
mounted.
  volumes:                         # Defines the volumes available to the Pod.
  - name: shared-data              # Volume name.
    emptyDir: {}                   # A temporary directory that shares data between
containers in the Pod.
  restartPolicy: Always            # Specifies the restart policy for containers. Default
is 'Always'.
```

Tasks:

The Theory Challenge: Understanding Pods and Security

1. **What is a Pod in Kubernetes?**
   - Define a Pod and explain its role in Kubernetes.
   - Discuss the difference between a Pod and a container in Kubernetes.
   - What are static pods, init containers, multi-container pods in Kubernetes?
2. **Managing Access to Pods:**
   - Discuss how Kubernetes manages access to Pods through RBAC.
   - What are the different levels of access control for Pods (e.g., namespaces, roles, service accounts)?
3. **Security Context in Pods:**
   - Define "Security Context" in Kubernetes.
   - Explain how Security Contexts are used to enhance pod security (e.g., running as a specific user, controlling privileges).
4. **Pod Networking and Security:**
   - Discuss Kubernetes networking policies and how they help secure communication between Pods.
   - What is the role of Network Policies in Kubernetes, and why are they critical for Pod security?
5. **Resource Requests and Limits for Pods:**
   - Explain the concept of resource requests and limits for pods in Kubernetes.
   - Why is it important to define resource limits for your Pods?

The practical challenge:

Step 1: Pod Basics

Read about the Pod definition, lifecycle, and uses from the documentation.

Create a Pod named `basic-pod` running `nginx` using:

Imperative method (Using command):

```
kubectl run basic-pod --image=nginx --restart=Never
```

Declarative method (YAML manifest):

```
apiVersion: v1
kind: Pod
metadata:
  name: basic-pod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Step 2. Multi-Container Pods

Understand the concept of multi-container Pods and how they share networking and storage.

Create a Pod named `multi-pod` with:

1. `nginx` container exposing port 80.
2. `busybox` container running `sleep 3600`.

Use a shared `emptyDir` volume to exchange data between containers.

Imperative Approach:

```
kubectl run ecommerce-pod --image=nginx --restart=Never --port=80
```

Note: Add second container by editing the pod yaml file and applying the changes

Declarative Approach:

Create the following YAML file ecommerce-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
name: ecommerce-pod
labels:
      app: nginx
run: busybox
spec:
containers:
- name: nginx
image: nginx
ports:
- containerPort: 80
- name: busybox
image: busybox
command: ["sleep", "3600"]
```

Deploy the Pod

Run the following command to deploy the pod:

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

Verify the Pod Status

```
kubectl get pods
```

Describe the Pod

> kubectl describe pod ecommerce-pod

Access the Pod Logs
View the logs of the NGINX Pod to ensure it's running correctly:

> kubectl logs ecommerce-pod

Step 3. Add Init Containers

Add an init container to ensure the nginx container starts only after necessary dependencies are initialized.

YAML Update:

```
initContainers:
- name: init-busybox
  image: busybox
  command: ["sh", "-c", "echo Initializing frontend; sleep 5"]
```

Step 4. Secure the Pod

Add securityContext features to the Pod:

- Ensure all containers run as non-root users.
- Drop unnecessary Linux capabilities.

YAML Update:

```
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
  containers:
  - name: nginx
    image: nginx
```

```
    securityContext:
      capabilities:
        drop:
        - ALL
  - name: busybox
    image: busybox
    securityContext:
      capabilities:
        drop:
        - ALL
```

Securing Pods with Service Accounts

Pods can inherit permissions through Service Accounts. Let’s secure the Pod by using a Service Account for access management.

1. **Create a Service Account**
   Define the service account in your YAML file:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-sa
```

2. **Bind the Service Account to the Pod**
   Update the Pod definition to use the created Service Account:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  serviceAccountName: nginx-sa
  containers:
```

```
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```

3. **Deploy the Pod with the Service Account**
   Re-deploy the Pod with the new configuration:

   ```
   kubectl apply -f nginx-pod-sa.yaml
   ```

4. **Verify the Service Account Binding**
   Run the following to check if the Service Account is properly assigned:

   ```
   kubectl get pod nginx-pod -o=jsonpath='{.spec.serviceAccountName}'
   ```

Implementing RBAC for Pod Access

Control who can access the resources in your Pods using Kubernetes RBAC. In this task, you'll configure RBAC to allow only authorized users to access the Pod.

1. **Define a Role**
   Create a Role in a specific namespace (e.g., `default`) that allows reading Pod details:

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

2. **Bind the Role to a User or Service Account**
   Define a RoleBinding to associate the user/service account with the Role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: nginx-sa
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

3. **Apply the Role and RoleBinding**
   Run the following commands to apply the configurations:

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

Step 5. Configure Resource Limits

Configure resource requests and limits for the containers:

- nginx: Request 100m CPU and 128Mi memory; Limit 250m CPU and 256Mi memory.
- busybox: Request 50m CPU and 64Mi memory; Limit 100m CPU and 128Mi memory.

YAML Update:

```
resources:
  requests:
    memory: "128Mi"
    cpu: "100m"
  limits:
```

```
    memory: "256Mi"
    cpu: "250m"
```

Step 6. Implement Probes

Add liveness and readiness probes for the nginx container:

- Liveness probe: HTTP GET `/healthz` on port 80.
- Readiness probe: TCP check on port 80.

YAML Update:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 80
  initialDelaySeconds: 3
  periodSeconds: 5
readinessProbe:
  tcpSocket:
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 10
```

Step 7. Static Pod Implementation

Deploy the same Pod as a static Pod by placing the YAML file in `/etc/kubernetes/manifests/`.

Steps:

1. Copy the YAML file to the directory:

```
sudo cp ecommerce-pod.yaml /etc/kubernetes/manifests/
```

2. Verify the kubelet automatically creates the Pod:

> kubectl get pods

Step 8. Enable or control Networking

Deploy a second Pod named `backend` running busybox.
Verify communication between `ecommerce-pod` and `backend` using DNS.

Steps:

1. **Create the backend Pod:**

```
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["sleep", "3600"]
```

2. **Test communication:**

> kubectl exec ecommerce-pod -- ping backend

Now limit the communication between Pods to ensure that only authorized Pods can communicate with each other.

1. **Create a Network Policy**
   Define a simple Network Policy that allows only Pods with a specific label to communicate with your Pod:

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

```
metadata:
  name: allow-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: nginx
```

2. **Apply the Network Policy**
   Apply the policy using:

   ```
   kubectl apply -f network-policy.yaml
   ```

3. **Test Communication Between Pods**
   Deploy another Pod (e.g., `curl-pod`) in the same namespace and test if it can access the ecommerce Pod.

   ```
   kubectl run curl-pod --rm -it --image=curlimages/curl -- /bin/sh
   curl ecommerce-pod:80
   ```

Step 9. Test Pod Behavior

1. Simulate a resource conflict by deploying the Pod on a node with limited resources.
2. Deploy pod with the same name in same and different namespaces
3. Test the liveness and readiness probes by making `/healthz` unavailable temporarily.

References:

- Kubernetes Pods Documentation

Essential `kubectl` Commands for Pods

1. List Pods in the Current Namespace

   ```
   kubectl get pods
   ```

2. List Pods Across All Namespaces

   ```
   kubectl get pods --all-namespaces
   ```

3. Describe Pod Details

   ```
   kubectl describe pod <pod-name>
   ```

4. View Logs of a Pod

   ```
   kubectl logs <pod-name>
   ```

   For a specific container in a multi-container Pod:

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

5. Stream Logs in Real Time

   ```
   kubectl logs -f <pod-name>
   ```

6. Exec into a Running Pod

   ```
   kubectl exec -it <pod-name> -- /bin/bash
   ```

   For a specific container in a multi-container Pod:

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

7. List Pod Events

   ```
   kubectl get events --field-selector involvedObject.name=<pod-name>
   ```

8. Delete a Pod

   > kubectl delete pod <pod-name>

9. Get Pod YAML Configuration

   > kubectl get pod <pod-name> -o yaml

10. Edit a Pod Configuration (use with caution; not for static Pods or deployments)

    > kubectl edit pod <pod-name>

11. List Pods in a Specific Namespace

    > kubectl get pods -n <namespace>

12. Describe Pod in a Specific Namespace

    > kubectl describe pod <pod-name> -n <namespace>

Submission Guidelines

1. **Answers to the Theory Section**
   - Provide responses to the questions about Pods, networking, storage sharing, and namespaces.
2. **Screenshots or Outputs of:**
   - Multi-container Pod creation steps (both imperative and YAML-based).
   - Verifications of shared `emptyDir` volume functionality.
   - `kubectl get pod`, `describe`, `logs`, and `exec` commands.
   - Successful validations of liveness and readiness probes.
   - Observations of resource limits being applied.
3. **Documentation**
   - Your approach and learnings from handling shared volumes and multi-container setups.

- Notes on differences observed across namespaces and other configurations.

4. **Social Media Post**

- Share your progress on social media with the hashtags: #getfitwithsagar, #SRELife, #DevOpsForAll #ckawithsagar

If you missed any previous challenges, you can catch up by reviewing the problem statements on GitHub.

Best regards,
Sagar Utekar