Kubernetes Grundlagen: Deine erste Web-App

In dieser Übung bauen wir eine einfache Web-App in Kubernetes. Ich zeige dir Schritt für Schritt, wie du die grundlegenden Kubernetes-Konzepte in der Praxis anwendest.

Warum Kubernetes lernen?

Kubernetes (oder K8s, weil 8 Buchstaben zwischen K und s) ist zum Standard für Container-Orchestrierung geworden. Es hilft dir dabei:

- Containerisierte Apps automatisch zu verteilen
- Deine Anwendungen zu skalieren, wenn viele Nutzer kommen
- Updates ohne Ausfallzeit zu machen
- Fehlerhafte Container automatisch neu zu starten

In den meisten Unternehmen läuft heute mindestens ein Teil der Software auf Kubernetes. Mit diesen Grundkenntnissen bist du gut vorbereitet für moderne DevOps-Jobs!

Was wir bauen werden

Wir erstellen eine einfache Webanwendung mit:

- Einer Nginx-Webseite (über ein Deployment)
- Einem Service, der die App erreichbar macht
- ConfigMaps für die Konfiguration
- Liveness Probes für Selbstheilung

Vorbereitung

Was du brauchst:

- Minikube oder eine andere lokale Kubernetes-Umgebung
- kubectl zum Steuern deines Clusters
- Ein Terminal/Kommandozeile

Starte deinen Cluster

```
minikube start
```

Was passiert hier? Dieser Befehl startet einen Mini-Kubernetes-Cluster auf deinem Computer. Perfekt zum Üben!
Überprüfe, ob alles läuft:

kubectl get nodes

Du solltest einen Node sehen, der "Ready" ist.

Teil 1: Dein erster Pod

Was ist ein Pod? Die kleinste Einheit in Kubernetes. Ein Pod enthält einen oder mehrere Container.

1.1 Pod-Manifest erstellen

Speichere als `mein-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: meine-webseite
  labels:
    app: nginx-demo
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

Was passiert hier?

- Wir definieren einen Pod namens "meine-webseite"
- Der Pod enthält einen Container mit Nginx (beliebter Webserver)
- Der Container stellt Port 80 bereit (Standard-HTTP-Port)

1.2 Pod erstellen

kubectl apply -f mein-pod.yaml

Was passiert hier? Kubernetes erstellt jetzt deinen Pod basierend auf der YAML-Datei. Es zieht das Nginx-Image und startet den Container.

1.3 Überprüfen, ob der Pod läuft

kubectl get pods

Du solltest etwa folgendes sehen:

```
NAME            READY   STATUS    RESTARTS   AGE
meine-webseite  1/1     Running   0          30s
```

Falls es nicht klappt, schau nach dem Problem:

kubectl describe pod meine-webseite

1.4 Die Webseite anschauen

Um auf den Pod zuzugreifen, leiten wir den Port weiter:
kubectl port-forward meine-webseite 8080:80

Was passiert hier? Dieser Befehl verbindet Port 8080 auf deinem Computer mit Port 80 im Pod.
Öffne jetzt deinen Browser und gehe zu: http://localhost:8080
Du solltest die Nginx-Willkommensseite sehen! Drücke Strg+C, um die Weiterleitung zu beenden.

1.5 Pod-Details anzeigen

kubectl describe pod meine-webseite

Schau dir die Ausgabe an - hier findest du viele Details wie:

- Welcher Node den Pod ausführt
- Wann er gestartet wurde
- IP-Adressen
- Events und Statusänderungen

1.6 Pod-Logs ansehen

kubectl logs meine-webseite

So siehst du die Log-Ausgaben des Containers - super nützlich bei Problemen!

Teil 2: Deployments für zuverlässige Apps

Einzelne Pods sind nicht sehr zuverlässig. Wenn einer abstürzt, wird er nicht automatisch neu gestartet. Deployments lösen dieses Problem!

2.1 Deployment-Manifest erstellen

Speichere als `mein-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx-demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-demo
  template:
    metadata:
      labels:
        app: nginx-demo
    spec:
```

```
    containers:
    - name: nginx
      image: nginx:1.20
      ports:
      - containerPort: 80
      resources:
        limits:
          memory: "128Mi"
          cpu: "100m"
      livenessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 3
        periodSeconds: 5
```

Was passiert hier?

- Wir erstellen ein Deployment namens "nginx-deployment"
- Es erstellt 3 identische Pods (replicas: 3)
- Wir begrenzen Ressourcen (CPU und RAM)
- Die livenessProbe prüft, ob die Pods gesund sind

2.2 Deployment erstellen

```
kubectl apply -f mein-deployment.yaml
```

2.3 Überprüfen, ob die Pods laufen

```
kubectl get pods
```

Du solltest 3 neue nginx-Pods sehen, alle mit ähnlichen Namen wie `nginx-deployment-7b4f54d6b9-xyzab`.

2.4 Deployment-Details ansehen

```
kubectl get deployments
kubectl describe deployment nginx-deployment
```

2.5 Pods beobachten

Lösche einen Pod und beobachte, wie Kubernetes einen neuen erstellt:

```
# Liste alle Pods
kubectl get pods

# Lösche einen der Pods (ersetze POD_NAME mit einem der angezeigten Namen)
kubectl delete pod POD_NAME

# Beobachte, wie ein neuer Pod gestartet wird
kubectl get pods -w
```

Drücke Strg+C, um die Beobachtung zu beenden.

Was ist passiert? Das Deployment bemerkt, dass ein Pod fehlt, und erstellt sofort einen neuen! Das ist die Selbstheilung von Kubernetes in Aktion.

Teil 3: Services - Zugriff auf deine Anwendung

Deployments lösen das Problem der Zuverlässigkeit, aber wie greift man auf die Anwendung zu? Dafür gibt es Services.

3.1 Service-Manifest erstellen

Speichere als `mein-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx-demo
  ports:
  - port: 80
    targetPort: 80
  type: NodePort
```

Was passiert hier?

- Wir erstellen einen Service namens "nginx-service"
- Er sucht alle Pods mit dem Label "app: nginx-demo"
- Er leitet Datenverkehr an Port 80 dieser Pods weiter
- Der Typ "NodePort" macht den Service von außerhalb des Clusters erreichbar

3.2 Service erstellen

```
kubectl apply -f mein-service.yaml
```

3.3 Service-Details ansehen

```
kubectl get services
kubectl describe service nginx-service
```

Beachte den "NodePort" - das ist der Port, über den du von außen zugreifen kannst.

3.4 Die Webseite über den Service ansehen

Bei Minikube:

```
minikube service nginx-service
```

Dieser Befehl öffnet automatisch deinen Browser mit der richtigen URL.

Alternativ:

```
# Finde den zugewiesenen Port
```

```
kubectl get service nginx-service

# Die Ausgabe zeigt etwas wie: 80:32145/TCP
# 32145 ist der externe Port

# Port-Weiterleitung (falls nötig)
kubectl port-forward service/nginx-service 8080:80
```

Dann öffne http://localhost:8080 im Browser.
Was ist cool an Services?

- Der Datenverkehr wird automatisch auf alle Pods verteilt (Load Balancing)
- Wenn Pods neu starten oder neue hinzukommen, aktualisiert sich der Service automatisch
- Du brauchst keine festen IP-Adressen mehr

Teil 4: ConfigMaps für die Konfiguration

Statt Konfigurationen fest in Container einzubauen, können wir ConfigMaps verwenden.

4.1 Eine angepasste Webseite erstellen

Speichere als `index.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Meine Kubernetes-Demo</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            text-align: center;
            margin-top: 50px;
            background-color: #f0f8ff;
        }
        h1 {
            color: #3080cc;
        }
    </style>
</head>
<body>
    <h1>Hallo Kubernetes!</h1>
    <p>Das ist meine erste Kubernetes-Anwendung.</p>
    <p>Erstellt am: 2023-09-30</p>
</body>
</html>
```

4.2 ConfigMap aus der HTML-Datei erstellen

kubectl create configmap website-content --from-file=index.html

Was passiert hier? Kubernetes erstellt eine ConfigMap, die den Inhalt unserer HTML-Datei enthält.

4.3 Überprüfen der ConfigMap

kubectl get configmaps
kubectl describe configmap website-content

4.4 Deployment mit ConfigMap aktualisieren

Speichere als `deployment-mit-config.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-configmap
  labels:
    app: nginx-demo-config
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx-demo-config
  template:
    metadata:
      labels:
        app: nginx-demo-config
    spec:
      containers:
      - name: nginx
        image: nginx:1.20
        ports:
        - containerPort: 80
        volumeMounts:
        - name: website-content
          mountPath: /usr/share/nginx/html
      volumes:
      - name: website-content
        configMap:
          name: website-content
```

Was passiert hier?

- Wir erstellen ein neues Deployment
- Die ConfigMap wird als Volume eingebunden
- Der Inhalt wird im Standard-Webverzeichnis von Nginx bereitgestellt

4.5 Neues Deployment erstellen

kubectl apply -f deployment-mit-config.yaml

4.6 Service für das neue Deployment erstellen

Speichere als `service-config.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-config-service
spec:
  selector:
    app: nginx-demo-config
  ports:
  - port: 80
    targetPort: 80
  type: NodePort
```

```
kubectl apply -f service-config.yaml
```

4.7 Die neue Webseite ansehen

```
minikube service nginx-config-service
```

Oder über Port-Forwarding:

```
kubectl port-forward service/nginx-config-service 8081:80
```

Dann öffne http://localhost:8081 im Browser.
Was ist hier passiert? Du hast deine Anwendungskonfiguration (HTML-Inhalt) von der Anwendung selbst (Nginx) getrennt. So kannst du Änderungen an der Webseite vornehmen, ohne das Container-Image neu zu erstellen!

Teil 5: Die Selbstheilung in Aktion

Jetzt schauen wir uns an, wie die Liveness-Probe funktioniert.

5.1 Pod mit defekter Webseite erstellen

Speichere als `fehler-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: fehler-webseite
  labels:
    app: fehler-demo
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
```

```
    path: /nicht-existent
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 5
```

Was passiert hier? Wir erstellen einen Pod mit einer Liveness-Probe, die nach einer Seite sucht, die nicht existiert. Die Probe wird nach 5 Sekunden zum ersten Mal ausgeführt und dann alle 5 Sekunden wiederholt.

5.2 Fehler-Pod erstellen

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

5.3 Beobachte, was passiert

```
kubectl get pod fehler-webseite -w
```

Du wirst sehen, wie der Pod startet, dann aber mehrmals neu startet, weil die Liveness-Probe fehlschlägt.

5.4 Details des Pods ansehen

```
kubectl describe pod fehler-webseite
```

In den Events siehst du, dass Kubernetes den Container neu startet, weil die Liveness-Probe fehlschlägt.
Warum ist das wichtig?

- Kubernetes erkennt automatisch fehlerhafte Anwendungen
- Es versucht, sie selbst zu reparieren (durch Neustart)
- Du musst nicht ständig deine Anwendungen überwachen - Kubernetes tut das für dich!

Geschafft!

Super! Du hast gerade die Grundlagen von Kubernetes kennengelernt:

1. Pods: Die kleinste Einheit in Kubernetes
2. Deployments: Für zuverlässige, selbstheilende Anwendungen
3. Services: Für den Zugriff auf deine Anwendungen
4. ConfigMaps: Für externe Konfigurationen
5. Liveness-Probes: Für automatische Fehlererkennung

Was als Nächstes?

- Experimentiere mit verschiedenen Container-Images
- Ändere die HTML-Datei in der ConfigMap und aktualisiere sie
- Erhöhe die Anzahl der Replicas im Deployment
- Lerne mehr über Persistent Volumes für Datenspeicherung

- Schau dir Kubernetes Dashboard an für eine grafische Oberfläche

Aufräumen

Wenn du fertig bist, kannst du alle erstellten Ressourcen löschen:

```
kubectl delete deployment nginx-deployment nginx-configmap
kubectl delete pod meine-webseite fehler-webseite
kubectl delete service nginx-service nginx-config-service
kubectl delete configmap website-content
```

Oder Minikube stoppen/löschen:

```
minikube stop
# oder
minikube delete
```