Kubernetes Mittelstufe: Stateless API mit Datenbank

In dieser Übung bauen wir eine API, die mit einer externen Datenbank spricht. Wir bewegen uns von der reinen Basis-Webseite zu einer interaktiveren Anwendung, bleiben aber noch überschaubarer als eine komplette Multi-Tier-Architektur.

Was wir lernen werden

In dieser Übung lernst du:

- Eine API mit Node.js in Kubernetes zu deployen
- Eine externe MySQL-Datenbank anzubinden
- Secrets für sensible Daten zu verwenden
- Umgebungsvariablen in Deployments zu nutzen
- Mit Ingress komplexere Routing-Regeln zu erstellen
- Mit Horizontal Pod Autoscaler zu skalieren

Vorbereitung

Was du brauchst:

- Eine laufende Kubernetes-Umgebung (wie Minikube)
- kubectl zum Steuern des Clusters
- Basic-Kubernetes-Kenntnisse (Pods, Deployments, Services)

Starte deinen Cluster

```
minikube start
```

Für Ingress brauchen wir das Addon:

```
minikube addons enable ingress
```

Teil 1: MySQL-Datenbank einrichten

Da unser Fokus auf Kubernetes und nicht auf Datenbank-Administration liegt, verwenden wir eine einfache, alleinstehende MySQL-Instanz. In echten Produktionsumgebungen würdest du wahrscheinlich eine verwaltete Datenbank-Lösung verwenden.

1.1 Namespace anlegen

```
kubectl create namespace meine-app
```

Was passiert hier? Wir trennen unsere Anwendung vom Rest des Clusters, das macht die Verwaltung übersichtlicher.

1.2 Secret für Datenbank-Passwörter erstellen

```
kubectl create secret generic db-credentials \
  --namespace meine-app \
  --from-literal=root-password=meinrootpasswort \
  --from-literal=user-password=meinuserpasswort
```

Was passiert hier? Wir erstellen ein Kubernetes-Secret, das unsere Datenbank-Passwörter sicher speichert. In einer echten Umgebung würdest du stärkere Passwörter verwenden!

1.3 PersistentVolume für MySQL erstellen

Speichere als `mysql-pv.yaml`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
  labels:
    type: local
spec:
  storageClassName: standard
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Was passiert hier? Wir erstellen einen permanenten Speicherplatz für die MySQL-Daten, damit diese nicht verloren gehen, wenn der Pod neu startet.

```
kubectl apply -f mysql-pv.yaml
```

1.4 PersistentVolumeClaim erstellen

Speichere als `mysql-pvc.yaml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
  namespace: meine-app
spec:
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Was passiert hier? Der PVC ist wie ein Anspruch auf Speicherplatz. MySQL wird diesen PVC nutzen, um Daten dauerhaft zu speichern.
kubectl apply -f mysql-pvc.yaml

1.5 MySQL Deployment erstellen

Speichere als `mysql-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: meine-app
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.7
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: root-password
        - name: MYSQL_DATABASE
          value: notizapp
        - name: MYSQL_USER
          value: notizuser
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: user-password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-storage
          mountPath: /var/lib/mysql
        resources:
          limits:
            memory: "512Mi"
            cpu: "500m"
```

```
    volumes:
    - name: mysql-storage
      persistentVolumeClaim:
        claimName: mysql-pvc
```

Was passiert hier?

- Wir deployen MySQL 5.7
- Die Passwörter kommen aus dem Secret
- Wir erstellen eine Datenbank "notizapp" und einen Benutzer "notizuser"
- Die Daten werden auf dem PVC gespeichert

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

1.6 MySQL Service erstellen

Speichere als `mysql-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: meine-app
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
```

Was passiert hier? Wir erstellen einen Headless-Service (clusterIP: None), der nur innerhalb des Clusters erreichbar ist - perfekt für eine Datenbank, die nicht von außen zugänglich sein soll.

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

1.7 Überprüfen, ob MySQL läuft

```
kubectl get pods -n meine-app
```

Warte, bis der Status "Running" ist.

1.8 Datenbank-Tabelle erstellen

Erstelle eine Datei `init.sql`:

```
CREATE TABLE IF NOT EXISTS notizen (
  id INT AUTO_INCREMENT PRIMARY KEY,
  titel VARCHAR(255) NOT NULL,
  inhalt TEXT,
  erstelltAm TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO notizen (titel, inhalt) VALUES
('Erste Notiz', 'Das ist meine erste Notiz in Kubernetes!'),
('Einkaufsliste', 'Milch, Brot, Eier');
```

Kopiere die SQL-Datei in den MySQL-Pod:

```
# Ersetze POD_NAME mit dem tatsächlichen Namen deines MySQL-Pods
POD_NAME=$(kubectl get pods -n meine-app -l app=mysql -o jsonpath="{.items[0].metadata.name}")
kubectl cp init.sql meine-app/$POD_NAME:/tmp/init.sql
```

Führe das SQL-Skript aus:

```
kubectl exec -it -n meine-app $POD_NAME -- mysql -u root -pmeinrootpasswort notizapp -e "source /tmp/init.sql"
```

Was haben wir gemacht? Wir haben die Datenbank mit einer Tabelle und zwei Beispieldatensätzen initialisiert, damit unsere API gleich etwas zum Anzeigen hat.

Teil 2: Node.js API-Server erstellen

2.1 ConfigMap für API-Konfiguration erstellen

Speichere als `api-config.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: api-config
  namespace: meine-app
data:
  config.js: |
    module.exports = {
      port: 3000,
      database: {
        host: 'mysql',
        port: 3306,
        database: 'notizapp',
        user: 'notizuser'
      }
    };
```

Was passiert hier? Wir erstellen eine Konfigurationsdatei für unsere Node.js-API. Die Datenbank-Verbindungsdaten (außer dem Passwort) werden hier gespeichert.

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

2.2 API-Deployment erstellen

Speichere als `api-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: notiz-api
  namespace: meine-app
spec:
  replicas: 2
  selector:
```

```
  matchLabels:
    app: notiz-api
template:
  metadata:
    labels:
      app: notiz-api
  spec:
    containers:
    - name: api
      image: node:14-alpine
      ports:
      - containerPort: 3000
      env:
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: user-password
      volumeMounts:
      - name: api-config-volume
        mountPath: /app/config
      - name: app-volume
        mountPath: /app
      command: ["/bin/sh", "-c"]
      args:
      - |
        cat > /app/server.js << 'EOF'
        const express = require('express');
        const mysql = require('mysql2/promise');
        const config = require('./config/config');

        const app = express();
        app.use(express.json());

        // Datenbankverbindung
        const dbConfig = {
          host: config.database.host,
          port: config.database.port,
          database: config.database.database,
          user: config.database.user,
          password: process.env.DB_PASSWORD
        };

        // GET /api/notizen - Alle Notizen abrufen
        app.get('/api/notizen', async (req, res) => {
          try {
            const connection = await mysql.createConnection(dbConfig);
            const [rows] = await connection.execute('SELECT * FROM notizen');
            await connection.end();
            res.json(rows);
          } catch (error) {
            console.error('Fehler beim Abrufen der Notizen:', error);
```

```
      res.status(500).json({ error: 'Datenbankfehler' });
    }
  });

  // GET /api/notizen/:id - Eine Notiz abrufen
  app.get('/api/notizen/:id', async (req, res) => {
    try {
      const connection = await mysql.createConnection(dbConfig);
      const [rows] = await connection.execute('SELECT * FROM notizen WHERE id = ?',
[req.params.id]);
      await connection.end();

      if (rows.length === 0) {
        return res.status(404).json({ error: 'Notiz nicht gefunden' });
      }

      res.json(rows[0]);
    } catch (error) {
      console.error('Fehler beim Abrufen der Notiz:', error);
      res.status(500).json({ error: 'Datenbankfehler' });
    }
  });

  // POST /api/notizen - Neue Notiz erstellen
  app.post('/api/notizen', async (req, res) => {
    const { titel, inhalt } = req.body;

    if (!titel) {
      return res.status(400).json({ error: 'Titel ist erforderlich' });
    }

    try {
      const connection = await mysql.createConnection(dbConfig);
      const [result] = await connection.execute(
        'INSERT INTO notizen (titel, inhalt) VALUES (?, ?)',
        [titel, inhalt || '']
      );
      await connection.end();

      res.status(201).json({
        id: result.insertId,
        titel,
        inhalt,
        message: 'Notiz erstellt'
      });
    } catch (error) {
      console.error('Fehler beim Erstellen der Notiz:', error);
      res.status(500).json({ error: 'Datenbankfehler' });
    }
  });

  // Gesundheitscheck
```

```
        app.get('/health', (req, res) => {
          res.json({ status: 'UP' });
        });

        // Server starten
        const port = config.port || 3000;
        app.listen(port, () => {
          console.log(`API-Server läuft auf Port ${port}`);
        });
        EOF

        cd /app && npm init -y &&
        npm install express mysql2 &&
        node server.js
      resources:
        limits:
          memory: "256Mi"
          cpu: "200m"
        requests:
          memory: "128Mi"
          cpu: "100m"
      readinessProbe:
        httpGet:
          path: /health
          port: 3000
        initialDelaySeconds: 10
        periodSeconds: 5
      livenessProbe:
        httpGet:
          path: /health
          port: 3000
        initialDelaySeconds: 20
        periodSeconds: 15
    volumes:
    - name: api-config-volume
      configMap:
        name: api-config
    - name: app-volume
      emptyDir: {}
```

Was passiert hier?

- Wir erstellen einen Node.js-Server, der eine RESTful API bereitstellt
- Die Konfiguration kommt aus der ConfigMap, das Datenbankpasswort aus dem Secret
- Die API kann Notizen auflisten, einzelne Notizen anzeigen und neue Notizen erstellen
- Wir haben Readiness und Liveness Probes für bessere Zuverlässigkeit

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

2.3 API-Service erstellen

Speichere als `api-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: notiz-api
  namespace: meine-app
spec:
  selector:
    app: notiz-api
  ports:
  - port: 80
    targetPort: 3000
  type: ClusterIP
```

Was passiert hier? Wir erstellen einen Service, der auf die API-Pods zeigt. Wir verwenden ClusterIP, da wir später einen Ingress für den externen Zugriff erstellen.

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

2.4 HorizontalPodAutoscaler für die API erstellen

Aktiviere zuerst den Metrics Server in Minikube:

```
minikube addons enable metrics-server
```

Speichere als `api-hpa.yaml`:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: notiz-api-hpa
  namespace: meine-app
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: notiz-api
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Was passiert hier? Wir erstellen einen HorizontalPodAutoscaler, der die Anzahl der API-Pods zwischen 2 und 5 automatisch anpasst, basierend auf der CPU-Auslastung.

```
kubectl apply -f api-hpa.yaml
```

2.5 Ingress erstellen

Speichere als `notiz-ingress.yaml`:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: notiz-ingress
  namespace: meine-app
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
  - host: notiz-app.local
    http:
      paths:
      - path: /api/?(.*)
        pathType: Prefix
        backend:
          service:
            name: notiz-api
            port:
              number: 80
```

Was passiert hier? Wir erstellen einen Ingress, der externe Anfragen an `/api/` an unseren API-Service weiterleitet.

```
kubectl apply -f notiz-ingress.yaml
```

2.6 Host-Eintrag hinzufügen

Füge folgenden Eintrag zu deiner Hosts-Datei hinzu:

```
127.0.0.1 notiz-app.local
```

- Windows: `C:\Windows\System32\drivers\etc\hosts`
- Linux/Mac: `/etc/hosts`

2.7 Minikube-Tunnel starten (in einem separaten Terminal)

```
minikube tunnel
```

Dieser Befehl ermöglicht den Zugriff auf den Ingress über localhost.

Teil 3: Testen der Anwendung

Wir können jetzt unsere API testen!

3.1 Alle Notizen abrufen

```
curl http://notiz-app.local/api/notizen
```

Oder besuche die URL in deinem Browser.

3.2 Eine einzelne Notiz abrufen

```
curl http://notiz-app.local/api/notizen/1
```

3.3 Eine neue Notiz erstellen

```
curl -X POST http://notiz-app.local/api/notizen \
  -H "Content-Type: application/json" \
  -d '{"titel":"Kubernetes lernen", "inhalt":"Heute habe ich eine API in Kubernetes deployed!"}'
```

3.4 Überprüfen, ob die neue Notiz erstellt wurde

```
curl http://notiz-app.local/api/notizen
```

Teil 4: Überwachung und Problembehandlung

Kubernetes bietet viele Möglichkeiten, deine Anwendungen zu überwachen.

4.1 Deployment-Status überwachen

```
kubectl get deployments -n meine-app
```

4.2 Pod-Status überwachen

```
kubectl get pods -n meine-app
```

4.3 Pod-Logs anzeigen

```
# Ersetze POD_NAME durch den Namen eines API-Pods
kubectl logs -n meine-app POD_NAME
```

4.4 Beschreibung eines Pods anzeigen

```
kubectl describe pod -n meine-app POD_NAME
```

4.5 API-Skalierung beobachten

```
kubectl get hpa -n meine-app -w
```

Um Last zu erzeugen und die Autoskalierung zu testen, kannst du viele Anfragen an die API senden:

```
for i in {1..1000}; do
  curl http://notiz-app.local/api/notizen &
done
```

Beobachte, wie der HPA die Anzahl der Pods erhöht.

Herausforderungen

Wenn du mehr lernen möchtest, probiere folgende Erweiterungen:

1. Frontend hinzufügen: Erstelle eine einfache Webseite, die die API nutzt, um Notizen anzuzeigen und zu erstellen.
2. Löschfunktion: Erweitere die API um eine Methode zum Löschen von Notizen.
3. Aktualisierungsfunktion: Füge eine Methode zum Aktualisieren von Notizen hinzu.
4. Authentifizierung: Füge eine einfache API-Schlüssel-Authentifizierung hinzu.
5. Ressourcenbegrenzungen verfeinern: Experimentiere mit verschiedenen Ressourcenlimits und beobachte das Verhalten.

Aufräumen

Wenn du fertig bist, kannst du alles wieder löschen:

```
kubectl delete namespace meine-app
kubectl delete pv mysql-pv
```

Geschafft!

Herzlichen Glückwunsch! Du hast erfolgreich:

1. Eine MySQL-Datenbank in Kubernetes deployed
2. Eine Node.js-API erstellt, die mit der Datenbank kommuniziert
3. Secrets für sensible Daten verwendet
4. Einen Ingress für den externen Zugriff eingerichtet
5. Automatische Skalierung implementiert

Mit diesen Kenntnissen bist du gut gerüstet, um komplexere Anwendungen in Kubernetes zu entwickeln und zu betreiben!