VDM Dokumentation

von:
Marc Kevin Zenzen (11131724),
Luca Stamos (11132237),
Stefan Steinhauer (11132517)

Kubernetes Cluster **2**
- Minikube Installation und Erzeugen des Clusters 2
- Tunneling 2

Stateless **3**
- Use Case / Images 3
- Umsetzung in Docker 3
- Umsetzung mit Kubernetes (Minikube) 5
- Updates 10
- Autoscaling 10

Stateful **13**
- Use Case / Images 13
- Umsetzung in Docker 13
- Umsetzung mit Kubernetes (Minikube) 14
- Updates 24

Anwendung Absichern **25**
- Secrets 25
- Role Based Access Control (RBAC) 25
- Multifaktor Authentifizierung 25
- Service Mesh 25

Git Repository **26**

Kubernetes Cluster

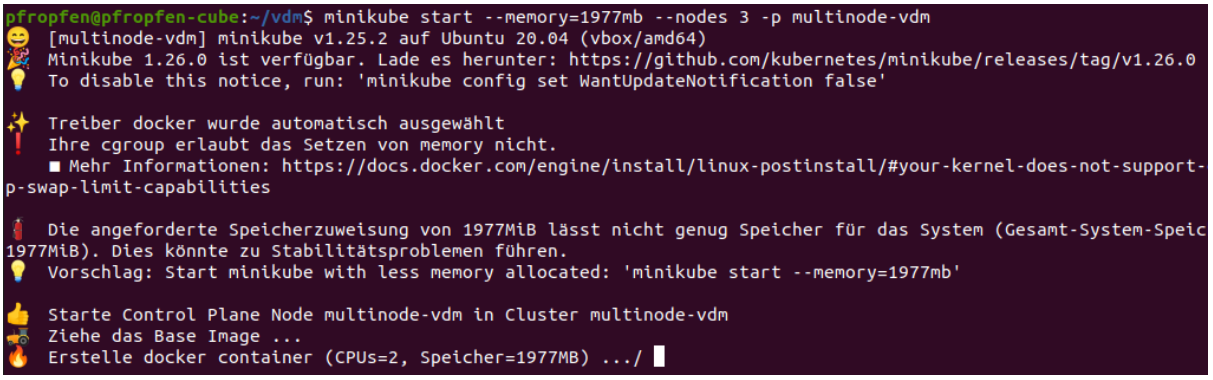
Aufgrund der vergleichsweise hohen Performance-Anforderungen von kubeadm wurde das Kubernetes Cluster in diesem Projekt mit Minikube aufgesetzt. Es besteht neben der Master/Control-Plane Node aus zwei weiteren Worker-Nodes.

Minikube Installation und Erzeugen des Clusters

Damit Minikube verwendet werden kann, muss zunächst der Installer geladen werden, welcher anschließend ausgeführt wird.

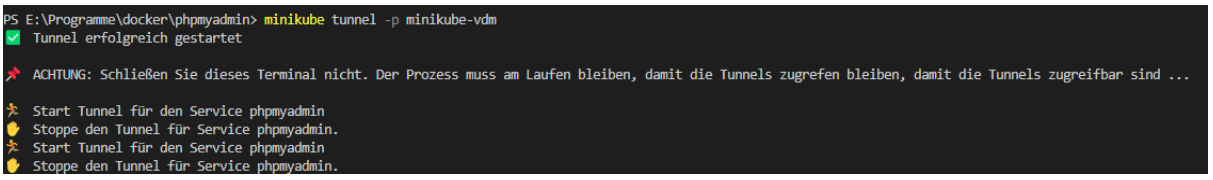
```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Nachdem Minikube erfolgreich installiert wurde kann mit "`minikube start`" das Cluster eingerichtet werden. Die gewünschte Anzahl an Nodes sowie der Name des Profils wird als Argument übergeben. Zusätzlich ist es auch möglich eine Begrenzung für den zugewiesenen Speicher anzugeben.



Tunneling

Im Laufe des Projekts sollen Dienste von außerhalb des Clusters erreichbar gemacht werden. Zu diesem Zweck kann die `tunnel`-Funktion von Minikube verwendet werden.



Stateless

Use Case / Images

Es werden insgesamt drei Images für die Applikation verwendet: Apache, Nginx und HA-Proxy. Anfragen an die Applikation werden von HA-Proxy entgegengenommen und per Round-Robin Verfahren an laufende Nginx- und Apache-Container (bzw. Pods) weitergeleitet.

Umsetzung in Docker

Um die Applikation mit Docker auszurollen, wird Docker-Compose verwendet. Die Docker-Compose File beinhaltet die Regeln für die 3 Services, die basierend auf den verwendeten Images erzeugt und gestartet werden sollen. Für den HA-Proxy-Service wird die Verwendung einer eigenen Dockerfile sowie eine Portweiterleitung festgelegt.

docker-compose.yml

```yaml
version: '3.7'

services:
  my_haproxy:
    image: my_haproxy
    build:
      context: .
      dockerfile: dockerfile_haproxy
      args:
        buildno: 1
    ports:
      - 8080:80
  nginx:
    image: nginx:latest
    restart: always
  apache:
    image: httpd:latest
    restart: always

```

Die Konfiguration von HA-Proxy wird in einer speziellen cfg-Datei beschrieben. Sie beinhaltet die verwendeten Frontend- und Backendserver, ihre Ports sowie das

Balance-Verfahren:

haproxy.cfg

```
defaults
    mode http

frontend http-in
        bind *:80
        default_backend servers

backend servers
    mode        http
    balance     roundrobin
    server      apache apache:80 maxconn 32
    server      nginx nginx:80 maxconn 32
```

Für HA-Proxy muss mit Hilfe einer Dockerfile ein eigenes Image erzeugt werden, welches die cfg-Datei enthält:

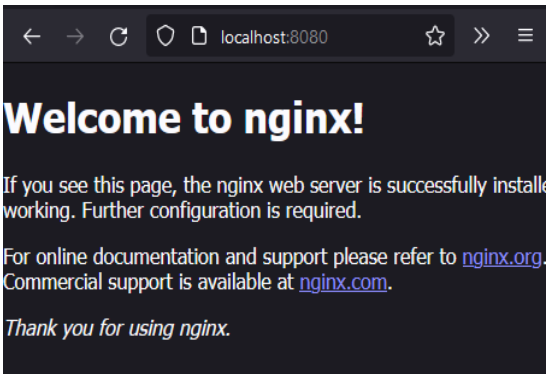
dockerfile_haproxy > ...

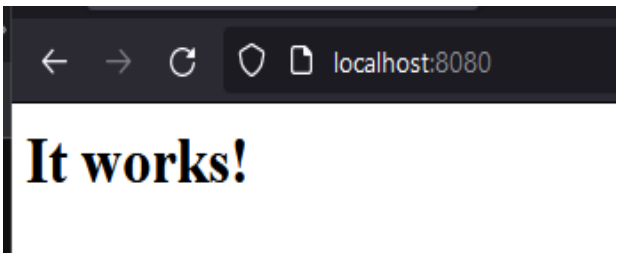
```
FROM haproxy:latest
COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
```

Nachdem die Docker-Services mit Hilfe der docker-compose Datei ausgerollt wurden, können die laufenden Container mit "`docker ps`" angezeigt werden:

```
PS E:\Programme\docker\Haproxy> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                    NAMES
0861d5ceaa27   my_haproxy     "docker-entrypoint.s…"   4 minutes ago    Up 4 minutes    0.0.0.0:8080->80/tcp     haproxy_my_haproxy_1
ca95da7d8a09   nginx:latest   "/docker-entrypoint.…"   19 minutes ago   Up 4 minutes    80/tcp                   haproxy_nginx_1
6c8af34372f4   httpd:latest   "httpd-foreground"       19 minutes ago   Up 4 minutes    80/tcp                   haproxy_apache_1
```

Sobald die benötigten Container gestartet sind, kann die Applikation über localhost:8080 aufgerufen werden. HA-Proxy leitet die Anfrage automatisch an einen der Dienste weiter, zum Beispiel Nginx. Durch das erneute Laden der Seite (refresh) wird die Anfrage nun auf Grund des round-robin-Verfahrens an den anderen Dienst geleitet.





Umsetzung mit Kubernetes (Minikube)

Um die docker-compose Datei in ein gültiges K8s-Manifest zu übersetzen kann das Programm “kompose” verwendet werden, welches separat installiert werden muss:

```
pfropfen@pfropfen-cube:~/vdm$ curl -L https://github.com/kubernetes/kompose/releases/download/v1.22.0/kompose-linux-amd64 -o kompose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
 70 23.8M   70 16.8M    0     0  1833k      0  0:00:13  0:00:09  0:00:04 2108k
```

Mit dem Befehl “`kompose convert`” kann anschließend die docker-compose Datei übersetzt werden. Mit “`--out`” wird der gewünschte Name des Manifests festgelegt.

```
pfropfen@pfropfen-cube:~/vdm$ kompose convert --out vdm.yaml
INFO Service name in docker-compose has been changed from "my_haproxy" to "my-haproxy"
```

Das Manifest unterteilt sich in den Bereich der Services und der Deployments. In diesem Projekt befinden sich alle Deployments und Services in der Selben Manifest-Datei da die Services nur in Verbindung zueinander verwendet werden und niemals ein Service alleine gestartet bzw. gestoppt werden soll. Das Manifest setzt sich aus den 3 Services HA-Proxy, Apache und Nginx sowie den zugehörigen Deployments zusammen.

HA-Proxy Service:
Damit der Dienst von außen erreichbar wird ist der Typ des Services auf “LoadBalancer” gesetzt. Die Portweiterleitung ist nach wie vor auf 8080 nach 80 festgelegt.

```yaml
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert --out vdm.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: myhaproxy
    name: myhaproxy
  spec:
    type: LoadBalancer
    ports:
      - name: "8080"
        port: 8080
        targetPort: 80
    selector:
      io.kompose.service: myhaproxy
  status:
    loadBalancer: {}
```

Während der HA-Proxy Service automatisch bei der Übersetzung des Manifests erstellt wurde, müssen die Services für Apache und Nginx manuell hinzugefügt werden, damit HA-Proxy diese über den Service-Namen findet.

Apache Service:
Bei diesem Service wird zusätzlich noch der Port 8081 auf 80 weitergeleitet und in der HA-Proxy Konfig dementsprechend angepasst.

```yaml
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert --out vdm.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: apache
    name: apache
  spec:
    ports:
      - name: "8081"
        port: 8081
        targetPort: 80
    selector:
      io.kompose.service: apache
  status:
    loadBalancer: {}
```

Nginx Service:
Bei dem Nginx Service wird zusätzlich der Port von 8082 auf 80 weitergeleitet.

```yaml
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert --out vdm.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: nginx
    name: nginx
  spec:
    ports:
      - name: "8082"
        port: 8082
        targetPort: 80
    selector:
      io.kompose.service: nginx
  status:
    loadBalancer: {}
```

HA-Proxy Deployment:
Das zu verwendende Image, welches vorher selbst erstellt und hochgeladen wurde, wird im Fall des HA-Proxy-Deployments aus dem eigenen Bereich des Docker-Hubs geladen.

```
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert --out vdm.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: myhaproxy
    name: myhaproxy
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: myhaproxy
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: kompose convert --out vdm.yaml
          kompose.version: 1.22.0 (955b78124)
        creationTimestamp: null
        labels:
          io.kompose.service: myhaproxy
      spec:
        containers:
          - image: pfropfen/myhaproxy
            name: myhaproxy
            ports:
              - containerPort: 80
            resources: {}
        restartPolicy: Always
  status: {}
```

Apache Deployment:
Bei dem Apache Deployment wurde das Standard httpd Image in der aktuellen Version von Docker Hub verwendet.

```yaml
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert --out vdm.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: apache
    name: apache
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: apache
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: kompose convert --out vdm.yaml
          kompose.version: 1.22.0 (955b78124)
        creationTimestamp: null
        labels:
          io.kompose.service: apache
      spec:
        containers:
          - image: httpd:latest
            name: apache
            resources: {}
        restartPolicy: Always
  status: {}
```

Nginx Deployment:
Für das Nginx Deployment wird wie bei Apache auch die aktuelle Version des offiziellen Images verwendet.

```yaml
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert --out vdm.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: nginx
    name: nginx
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: nginx
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: kompose convert --out vdm.yaml
          kompose.version: 1.22.0 (955b78124)
        creationTimestamp: null
        labels:
          io.kompose.service: nginx
      spec:
        containers:
          - image: nginx:latest
            name: nginx
            resources: {}
        restartPolicy: Always
  status: {}
```

Für alle Deployments wird im Manifest jeweils 1 Replica festgelegt.

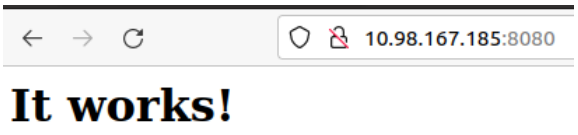
Die Applikation wird mit dem Befehl "`kubectl apply -f ./stateless.yaml`" auf Basis des Manifests ausgerollt. Nachdem die Pods gestartet wurden kann der Status mit "`kubectl get pods`" angezeigt werden:

```
pfropfen@pfropfen-cube:~/vdm$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
apache-85b5f48646-5hl9k      1/1     Running   0          63s
myhaproxy-66c74fcbb8-2lf4h   1/1     Running   0          63s
nginx-7775967764-ht4sc       1/1     Running   0          63s
```

Die externe IP des HA-Proxy-Dienstes kann mit "`kubectl get services`" ausgelesen werden. Der Dienst ist im Browser über diese IP erreichbar.

```
pfropfen@pfropfen-cube:~/vdm$ kubectl get services -o wide
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)          AGE
apache       ClusterIP      10.104.138.228   <none>          8081/TCP         18s
kubernetes   ClusterIP      10.96.0.1        <none>          443/TCP          117m
myhaproxy    LoadBalancer   10.98.167.185    10.98.167.185   8080:31384/TCP   18s
nginx        ClusterIP      10.99.186.100    <none>          8082/TCP         18s
```

Apache:



Wenn auf die entsprechende externe IP des Services mit dem vorher gesetzten Port zugegriffen wird, wird die Apache Seite geladen. Nach dem Neuladen der Seite wird durch HA-Proxy auf Nginx gewechselt.

Nginx:

Im Manifest kann unter Deployment (hier am Beispiel Apache) die Anzahl der Replicas geändert werden:

```yaml
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert --out vdm.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: apache
    name: apache
  spec:
    replicas: 4
    selector:
      matchLabels:
        io.kompose.service: apache
    strategy: {}
```

Mit 4 Replicas für Apache und 3 Replicas für Nginx:

```
pfropfen@pfropfen-cube:~/vdm$ kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP           NODE
apache-85b5f48646-5jdcn       1/1     Running   0          3m31s   10.244.1.5   multinode-vdm-m02
apache-85b5f48646-88hjm       1/1     Running   0          3m31s   10.244.0.4   multinode-vdm
apache-85b5f48646-dtvpn       1/1     Running   0          3m31s   10.244.2.3   multinode-vdm-m03
apache-85b5f48646-h5kdt       1/1     Running   0          3m31s   10.244.1.6   multinode-vdm-m02
myhaproxy-66c74fcbb8-gt4gj    1/1     Running   0          3m31s   10.244.2.5   multinode-vdm-m03
nginx-7775967764-6pjsm        1/1     Running   0          3m31s   10.244.2.4   multinode-vdm-m03
nginx-7775967764-t652z        1/1     Running   0          3m31s   10.244.1.4   multinode-vdm-m02
nginx-7775967764-vtj8j        1/1     Running   0          3m31s   10.244.0.3   multinode-vdm
```

Updates

Damit während eines Updates niemals ein Zustand eintritt in dem ein Dienst nicht mehr erreichbar ist, weil sich alle Instanzen gleichzeitig im Update-Prozess befinden, wird im Manifest für Updates die Update-Strategie vom Typ “rollingUpdate” ergänzt. So werden die Instanzen eines Deployments inkrementell geupdated, das bedeutet, die Updates der einzelnen Pods werden erst nach und nach durchgeführt. Der Eintrag “maxSurge” gibt dabei die maximale Anzahl der gleichzeitig durchgeführten Updates an, “maxUnavailable: 0” bewirkt dass es immer mindestens einen Pod gibt, welcher sich gerade nicht im Update-Prozess befinden darf.

```yaml
minReadySeconds: 5
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 0
```

Autoscaling

Es ist möglich, Replicas nur bei Bedarf erzeugen zu lassen. Der Bedarf definiert sich durch vorher festgelegte Eigenschaften die sich auf einzelne Ressourcen wie Speicher oder CPU beziehen. In diesem Projekt wird HPA (HorizontalPodAutoscaler) verwendet. Damit HPA eingesetzt werden kann muss ein Metrics-Server aktiv sein, welcher Daten über die Auslastung einzelner Ressourcen liefert. Er ist Teil des Minikube Programms und kann mit "`minikube addons enable metrics-server -p multinode-vdm`" aktiviert werden.

Der Metrics-Server verwendet standardmäßig eine Zertifizierung, die während der Entwicklung umgangen werden kann, indem man die Datei deployments.apps mit den in der Abbildung markierten Zeilen erweitert:

```
spec:
  containers:
  - args:
    - --cert-dir=/tmp
    - --secure-port=4443
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --kubelet-use-node-status-port
    - --metric-resolution=15s
    command:
    - /metrics-server
    - --kubelet-insecure-tls
    - --kubelet-preferred-address-types=InternalIP
    image: k8s.gcr.io/metrics-server/metrics-server:v0.6.1
    imagePullPolicy: IfNotPresent
    livenessProbe:
      failureThreshold: 3
      httpGet:
        path: /livez
        port: https
```

Ist der metrics-server aktiv können mit "`kubectl top node`" die Informationen zur Auslastung der einzelnen Nodes abgerufen werden:

```
pfropfen@pfropfen-cube:~/vdm$ kubectl top node
NAME                CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
multinode-vdm       158m         7%     567Mi           28%
multinode-vdm-m02   35m          1%     180Mi           9%
multinode-vdm-m03   49m          2%     221Mi           11%
```

Jeder HPA wird im Manifest als zusätzliches Item angelegt. Die Spezifizierung enthält Informationen der zu beobachteten Ressource und dessen Zielauslastung (hier 2 bzw 10Mi (Mibibyte)) sowie den Namen des betreffenden Deployments und wie wieviele Replicas mindestens gestartet werden müssen und maximal gestartet werden können.

Horizontal Pod Autoscaler:

```yaml
- apiVersion: autoscaling/v2
  kind: HorizontalPodAutoscaler
  metadata:
    name: apache-autoscale
  spec:
    scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: apache
    minReplicas: 1
    maxReplicas: 5
    metrics:
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: 2Mi
```

```yaml
- apiVersion: autoscaling/v2
  kind: HorizontalPodAutoscaler
  metadata:
    name: nginx-autoscale
  spec:
    scaleTargetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: nginx
    minReplicas: 1
    maxReplicas: 5
    metrics:
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: 10Mi
```

Wenn die Zielauslastung nicht erreicht wird läuft der entsprechende Pod ohne Replicas:

```
pfropfen@pfropfen-cube:~/vdm$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
apache-85b5f48646-np6pt       1/1     Running   0          11s
myhaproxy-66c74fcbb8-cxt7g    1/1     Running   0          11s
nginx-7775967764-7rbp5        1/1     Running   0          11s
```

Überschreitet die Auslastung das Ziel werden zusätzliche Replicas erzeugt. Die aktuell beobachtete Auslastung kann in den HPAs eingesehen werden.

```
pfropfen@pfropfen-cube:~/vdm$ kubectl get horizontalpodautoscaler
NAME               REFERENCE             TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
apache-autoscale   Deployment/apache     7921664/2Mi     1         5         1          54s
nginx-autoscale    Deployment/nginx      5844992/10Mi    1         5         1          54s
```

In diesem Fall beobachtet der HPA beim Apache-Service eine Auslastung von 7,9/2 Mi, das bedeutet es werden insgesamt 4 Pods benötigt, demnach werden zusätzlich 3 Replikas erzeugt. Der Nginx-Service liegt mit 5,8/10 Mi unterhalb der Zielauslastung, es werden demnach keine Replikas erzeugt.

```
pfropfen@pfropfen-cube:~/vdm$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
apache-85b5f48646-2kbp6       1/1     Running   0          25s
apache-85b5f48646-8snf4       1/1     Running   0          25s
apache-85b5f48646-np6pt       1/1     Running   0          70s
apache-85b5f48646-sw4pq       1/1     Running   0          25s
myhaproxy-66c74fcbb8-cxt7g    1/1     Running   0          70s
nginx-7775967764-7rbp5        1/1     Running   0          70s
```

Stateful

Use Case / Images

Bei der Stateful Applikation werden drei Images gewählt. Einmal das phpMyAdmin Image und zweimal das MySQL Image. Es soll möglich sein, über phpMyAdmin auszuwählen, auf welche der beiden Datenbanken zugegriffen wird. Dabei ist es wichtig, dass die eine Datenbank unter der neuesten Version (aktuell 8) und die ander unter Version 5.7 läuft. Zusätzlich sollen beide Datenbanken mit einem Volume verbunden sein, damit die Daten nach einem Neustart des Clusters immer noch vorhanden sind.

Umsetzung in Docker

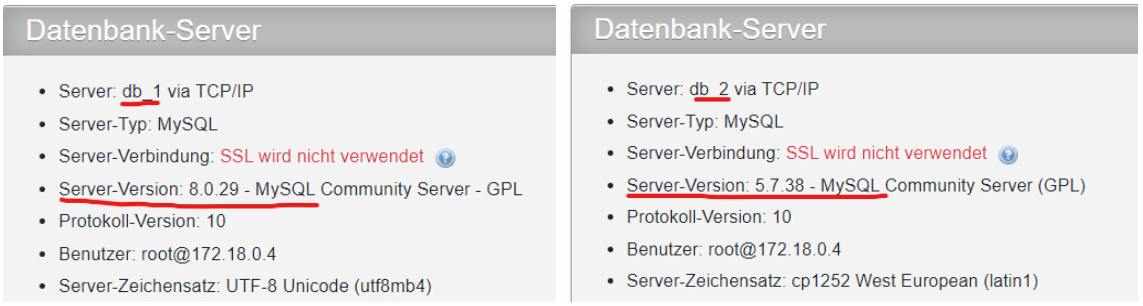
In diesem Abschnitt wird zunächst die Umsetzung in Docker-Compose dargestellt.
In der folgenden Abbildung ist die Docker-Compose.yml zu sehen. Zuerst wird die MySQL Datenbank in der neuesten Version angelegt. Anschließend (Zeile 9) wird die zweite MySQL Datenbank in Version 5.7 angelegt. Zuletzt wird der phpMyAdmin Service angelegt. Dieser enthält als Umgebungsvariable den PMA_HOST, in welchem die Servicenamen der Datenbanken (db_1 und db_2) angegeben werden. Damit kann später über die Oberfläche auf die jeweilige Datenbank zugegriffen werden. Außerdem wird der Service über Port 8080 freigegeben.

docker-compose.yml

```
1  version: '3.7'
2
3  services:
4    db_1:
5      image: mysql:latest
6      environment:
7        MYSQL_ROOT_PASSWORD: nature
8      restart: always
9    db_2:
10     image: mysql:5.7
11     environment:
12       MYSQL_ROOT_PASSWORD: nature
13     restart: always
14   phpmyadmin:
15     image: phpmyadmin
16     environment:
17       PMA_HOSTS: db_1, db_2
18     ports:
19       - 8080:80
20     restart: always
```

Auf der Oberfläche des phpMyAdmin Containers kann zwischen zwei Datenbanken gewählt werden (db_1 und db_2). Nach Eingabe der Login Daten gelangt man auf die jeweilige Seite des Datenbank Containers.

In der linken Abbildung ist die Oberfläche des MySQL-Containers in Version 8 zu sehen. Auf der rechten Seite ist der Nutzer auf der Oberfläche des 2ten Datenbank Containers. Zusätzlich ist hier zu sehen, dass die Versionen stimmen (der Erste läuft in Version 8 und der Zweite läuft in Version 5.6).



Umsetzung mit Kubernetes (Minikube)

Zur Erzeugung des Manifests wurde wie bei der stateless-Anwendung zuvor ebenfalls “kompose” verwendet. Durch die Verwendung von “kompose” wurde ein Manifest mit 3 Deployments und 1 Service angelegt. Zu jedem Deployment wird ein Service benötigt, welcher das jeweilige Deployment erreichbar macht. Dafür müssen für die beiden Datenbank Deployments jeweils noch 1 Service angelegt werden. Um die Daten persistent zu speichern werden zusätzlich 2 Persistent Volumes, 2 Persistent Volume Claims sowie 2 Storage Classes benötigt.
Im finalen Stand enthält die Manifest-Datei insgesamt 3 Services, 3 Deployments, 2 Persistent Volumes, 2 Persistent Volume Claims sowie 2 Storage Classes. Dadurch, dass alles in einem Manifest vorhanden ist, entsteht der Vorteil, dass ein “`apply`” auf die YAML ausgeführt werden kann und alle Services, Deployments und Volumes automatisch erstellt und gestartet werden. Dasselbe gilt für das Stoppen des Manifests.

Manifest.yaml:
Die Services für die Datenbanken (db-1, db-2) wurden bei der Übersetzung durch “kompose“ nicht automatisch erstellt und müssen manuell hinzugefügt werden. Beide sind jeweils über den Port 3306 erreichbar.

Service db-1:

```yaml
apiVersion: v1
items:
  - apiVersion: v1
    kind: Service
    metadata:
      annotations:
        kompose.cmd: kompose convert --out statefull.yaml
        kompose.version: 1.22.0 (955b78124)
      creationTimestamp: null
      labels:
        io.kompose.service: db-1
      name: db-1
    spec:
      ports:
        - name: "3306"
          port: 3306
      selector:
        io.kompose.service: db-1
    status:
      loadBalancer: {}
```

Service db-2:

```yaml
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert --out statefull.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: db-2
    name: db-2
  spec:
    ports:
      - name: "3306"
        port: 3306
    selector:
      io.kompose.service: db-2
  status:
    loadBalancer: {}
```

Service phpmyadmin:
Der phpmyadmin-Service ist vom Typ “LoadBalancer” und somit von außen erreichbar. Für diesen Dienst ist eine Portweiterleitung von Port 8080 auf 80 eingerichtet.

```yaml
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert --out statefull.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: phpmyadmin
    name: phpmyadmin
  spec:
    type: LoadBalancer
    ports:
      - name: "8080"
        port: 8080
        targetPort: 80
        #nodePort: 30008
    selector:
      io.kompose.service: phpmyadmin
  status:
    loadBalancer: {}
```

Deployment db-1:
Die erste Datenbank verwendet ein Image der aktuellen Version von MySQL (mysql:latest). Im Deployment werden sowohl der Benutzername als auch das Passwort für den Zugang zur Datenbank eingetragen (siehe auch Abschnitt “Anwendung absichern - Secrets”). Für das Deployment wird eine Node-Affinität auf die Node “minikube-vdm-m02” eingetragen, dadurch wird sichergestellt, dass dieses Deployment immer auf der Selben Node gestartet wird. Es wird die Verwendung eines Persistent Volumes (db1-pv-v2) sowie dessen MountPath festgelegt, ebenso das zu dem verwendeten Volume gehörende PVC (Persistent Volume Claim).

```yaml
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert --out statefull.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: db-1
    name: db-1
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: db-1
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: kompose convert --out statefull.yaml
          kompose.version: 1.22.0 (955b78124)
        creationTimestamp: null
        labels:
          io.kompose.service: db-1
      spec:
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: kubernetes.io/hostname
                      operator: In
                      values:
                        - minikube-vdm-m02
        containers:
          - env:
              - name: MYSQL_ROOT_PASSWORD
                value: nature
            image: mysql:latest
            name: db-1
            volumeMounts:
              - name: db1-pv-v2
                mountPath: /var/lib/mysql
            resources: {}
        volumes:
          - name: db1-pv-v2
            persistentVolumeClaim:
              claimName: db1-pv-claim
        restartPolicy: Always
  status: {}
```

Deployment db-2:
Das Deployment der zweiten Datenbank ist analog zu db-1 aufgebaut. Verwendet wird ebenfalls ein MySQL-Image allerdings mit einer anderen Version (mysql:5.7). In diesem Fall ist die Node-Affinität auf “minikube-vdm-m03” festgelegt. Es wird jeweils ein eigenes Persistent Volume und Persistent Volume Claim verwendet.

```yaml
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert --out statefull.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: db-2
    name: db-2
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: db-2
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: kompose convert --out statefull.yaml
          kompose.version: 1.22.0 (955b78124)
        creationTimestamp: null
        labels:
          io.kompose.service: db-2
      spec:
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: kubernetes.io/hostname
                      operator: In
                      values:
                        - minikube-vdm-m03
        containers:
          - env:
              - name: MYSQL_ROOT_PASSWORD
                value: nature
            image: mysql:5.7
            #image: mysql:latest
            name: db-2
            volumeMounts:
              - name: db2-pv-v2
                mountPath: /var/lib/mysql
            resources: {}
        volumes:
          - name: db2-pv-v2
            persistentVolumeClaim:
              claimName: db2-pv-claim
        restartPolicy: Always
  status: {}
```

Deployment phpMyAdmin:
Bei dem phpMyAdmin Deployment ist wird das Standard phpMyAdmin Image vom Docker Hub runtergeladen. Hierbei ist es wichtig, dass die Umgebungsvariable “PMA_HOST” angegeben wird. Bei dieser werden die Services db-1 und db-2 angegeben, damit phpMyAdmin diese kennt und auf diese zugreifen kann. Zusätzlich wird über den Parameter “PMA_PORT” der Port für die Datenbanken angegeben. Das Deployment wird mit einem Replica gestartet.
Bei diesem Deployment spielt es keine Rolle auf welchem Node es gestartet wird, aus dem Grund wird keine Node-Affinity angegeben.

```yaml
- apiVersion: apps/v1
  kind: Deployment
  metadata:
    annotations:
      kompose.cmd: kompose convert --out statefull.yaml
      kompose.version: 1.22.0 (955b78124)
    creationTimestamp: null
    labels:
      io.kompose.service: phpmyadmin
    name: phpmyadmin
  spec:
    replicas: 1
    selector:
      matchLabels:
        io.kompose.service: phpmyadmin
    strategy: {}
    template:
      metadata:
        annotations:
          kompose.cmd: kompose convert --out statefull.yaml
          kompose.version: 1.22.0 (955b78124)
        creationTimestamp: null
        labels:
          io.kompose.service: phpmyadmin
      spec:
        containers:
          - env:
              - name: PMA_HOSTS
                value: db-1, db-2
              - name: PMA_PORT
                value: "3306"
            image: phpmyadmin
            name: phpmyadmin
            resources: {}
        restartPolicy: Always
  status: {}
```

Persistent Volumes können einer Storage Class zugeordnet werden. Dazu müssen die Storage Classes zunächst angelegt werden. In ihnen werden der provisioner, die reclaim policy sowie der binding mode festgelegt. Storage Classes dienen dem Administrator dazu, die Verwendung von Persistent Volumes zu ermöglichen ohne dem Benutzer eine Einsicht auf die Details der Implementierung zu geben.

Storage Class 1:

```yaml
- apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    name: sc-local-storage-db1
  provisioner: kubernetes.io/no-provisioner
  reclaimPolicy: Retain
  volumeBindingMode: WaitForFirstConsumer
```

Storage Class 2:

```yaml
- apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    name: sc-local-storage-db2
  provisioner: kubernetes.io/no-provisioner
  reclaimPolicy: Retain
  volumeBindingMode: WaitForFirstConsumer
```

Db1-pv:

Für beide persistent Volumes wird eine Gesamtgröße von 10 MiByte festgelegt. Zusätzlich wird der Modus “Filesystem” festgelegt. Dieser Modus ist der default. Als “accessMode” wurde ReadWriteOnce gewählt. Dies besagt, dass die Volumes jeweils nur an ein Node gemountet werden können. Das Persistent Volume (db1-pv-v2) verweist auf die Storage Class sc-local-storage-db1 und das zweite Persistent Volume (db2-pv-v2) verweist auf die Storage Class sc-local-storage-db2. Als Typ für die Volumes wurde “local” verwendet. Zusätzlich wurde für beide Persistent Volumes eine Node-Affinity festgelegt. Das db1-pv-v2 läuft ausschließlich auf dem Node minikube-vdm-m02 und das db2-pv-v2 wird ausschließlich auf dem minikube-vdm-m03 Node ausgeführt.

```yaml
- apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: db1-pv-v2
  spec:
    capacity:
      storage: 10Mi
    volumeMode: Filesystem
    accessModes:
      - ReadWriteOnce
    persistentVolumeReclaimPolicy: Delete
    storageClassName: sc-local-storage-db1
    local:
      path: /mnt/w01/sata
    nodeAffinity:
      required:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                  - minikube-vdm-m02
```

Db2-pv:

```yaml
- apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: db2-pv-v2
  spec:
    capacity:
      storage: 10Mi
    volumeMode: Filesystem
    accessModes:
      - ReadWriteOnce
    persistentVolumeReclaimPolicy: Delete
    storageClassName: sc-local-storage-db2
    local:
      path: /mnt/w01/sata2
    nodeAffinity:
      required:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                  - minikube-vdm-m03
```

Das Bindeglied zwischen Pod und PV bildet jeweils das PVC (Persistent Volume Claim). Da beide Datenbanken jeweils ihr eigenes PV verwenden, werden ebenfalls zwei PVCs benötigt. Die PVCs verweisen auf die verwendete Storage Class und beinhalten Informationen zum Access Mode sowie zur Menge des zu beanspruchenden Speichers.

Db1-pv-claim:

```yaml
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: db1-pv-claim
  spec:
    storageClassName: sc-local-storage-db1
    accessModes:
      - ReadWriteOnce
    resources:
      requests:
        storage: 20Gi
```

Db2-pv-claim:

```yaml
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: db2-pv-claim
  spec:
    storageClassName: sc-local-storage-db2
    accessModes:
      - ReadWriteOnce
    resources:
      requests:
        storage: 20Gi
```

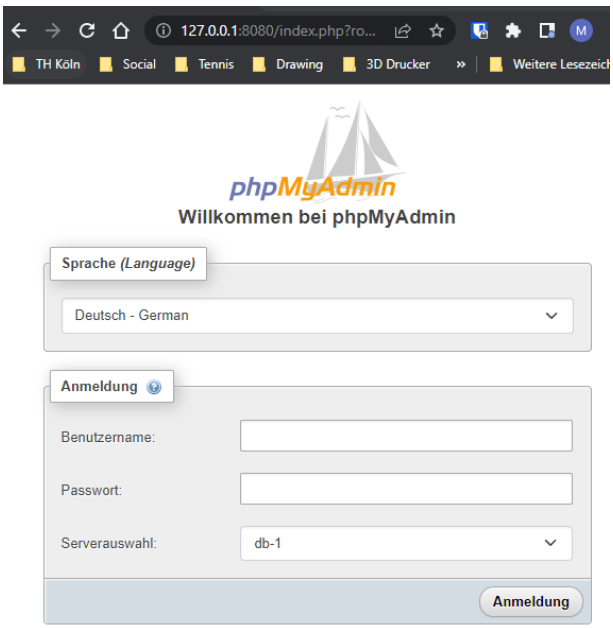
Die verwendeten Volumes sind durch die festgelegten Regeln des Manifests an bestimmte Nodes gebunden. Die Pods, welche auf das jeweilige Volume zugreifen, werden immer auf der entsprechenden Node gestartet. Während phpmyadmin auf einer beliebigen Node laufen kann, läuft db-1 daher immer auf Node m03 und db-2 immer auf Node m02.

```
PS E:\Programme\docker\phpmyadmin> kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE   IP           NODE              NOMINATED NODE   READINESS GATES
db-1-79b894f89-qlcc6          1/1     Running   0          9s    10.244.1.8   minikube-vdm-m02  <none>           <none>
db-2-68876cbc7d-gnmln         1/1     Running   0          9s    10.244.2.3   minikube-vdm-m03  <none>           <none>
phpmyadmin-6b64c99f57-dbrp7   1/1     Running   0          9s    10.244.1.7   minikube-vdm-m02  <none>           <none>
```

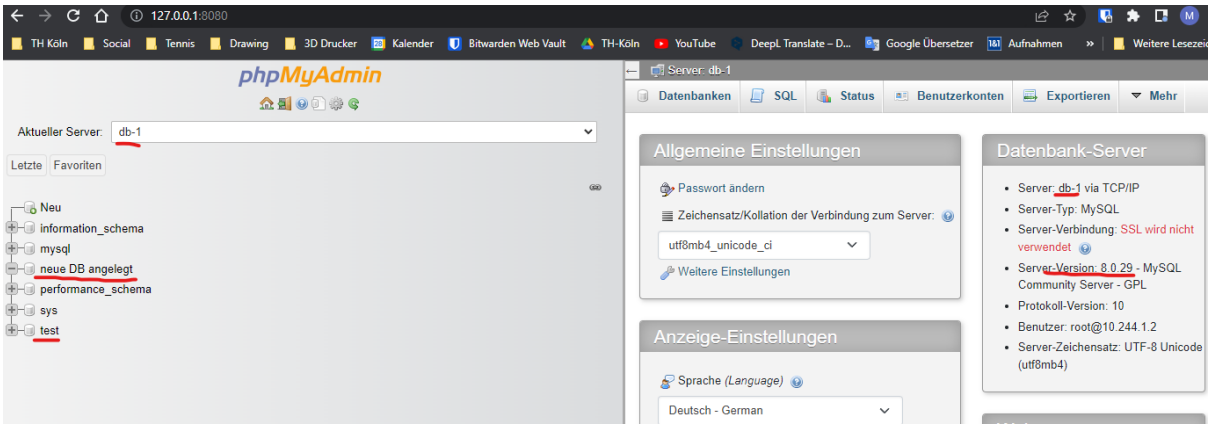
Änderungen der Datenbanken werden auf den entsprechenden Volumes persistent abgelegt und sind auch nach neu erstellten Pods weiterhin verfügbar. Der phpmyadmin Service verfügt über eine externe IP und ist von außen über diese erreichbar.

```
PS E:\Programme\docker\phpmyadmin> kubectl get services -o wide
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE    SELECTOR
db-1         ClusterIP      10.98.169.255   <none>        3306/TCP         7m11s  io.kompose.service=db-1
db-2         ClusterIP      10.106.46.254   <none>        3306/TCP         7m11s  io.kompose.service=db-2
kubernetes   ClusterIP      10.96.0.1       <none>        443/TCP          23h    <none>
phpmyadmin   LoadBalancer   10.103.89.38    127.0.0.1     8080:30463/TCP   7m11s  io.kompose.service=phpmyadmin
```

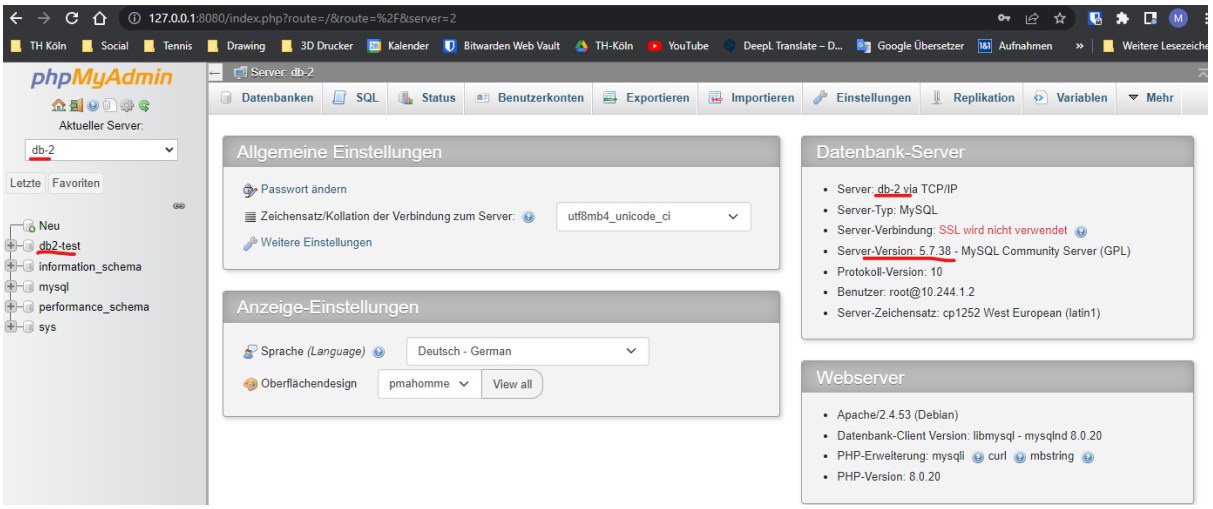
Über die externe IP des phpmyadmin-Services ist der Dienst im Browser erreichbar und führt dementsprechend auf die phpMyAdmin Anmeldeseite. Mit den im Manifest festgelegten Zugangsdaten kann der Benutzer sich wahlweise bei db-1 oder db-2 anmelden.



Ist der Benutzer angemeldet, zeigt phpMyAdmin auf der linken Seite den Inhalt der Datenbank, im rechten Bereich des Fensters besteht die Möglichkeit, Datensätze einzufügen oder zu bearbeiten (hier am Beispiel von db-1).



Folgende Abbildung zeigt die entsprechende Ansicht der Seite für db-2.



Updates

Damit die Anwendung während eines Updates weiterhin jederzeit erreichbar ist, wird, wie in der stateless Applikation auch, das “Rolling Update” verwendet (siehe auch Abschnitt “Stateless - Update”). Allerdings wird dies in diesem Fall nur für den phpMyAdmin Dienst eingerichtet, da von den Datenbanken nicht beliebige Replikas erstellt werden können.

Anwendung Absichern

Secrets

Damit empfindliche Daten wie Passwörter nicht als Klartext im Manifest auftauchen, können sogenannte Secrets verwendet werden. Dabei werden die Daten in einer eigenen Datei gespeichert, auf die Benutzer keinen Zugriff haben. Im Manifest wird dann die Benutzung der in der Datei liegenden Daten festgelegt.

Role Based Access Control (RBAC)

Bei der rollenbasierten Zugriffskontrolle handelt es sich um einen Autorisierungsmechanismus zur Einschränkung und Kontrolle des Benutzerzugriffs auf eine Ressource oder ein System auf der Grundlage von standardmäßigen oder benutzerdefinierten Rollen. Alle Kubernetes RBAC-Rollenautorisierungsentscheidungen in einem Cluster werden von der API-Gruppe rbac.authorization.k8s.io gesteuert. Als Best Practice sollten RBAC-Rollen im Cluster immer aktiviert sein, um sicherzustellen, dass Projekte - und die Organisation als Ganzes - die Best Practices für Sicherheit besser einhalten.

Eine Sammlung von “good practices” wird auf der offiziellen Kubernetes-Seite angeboten: https://kubernetes.io/docs/concepts/security/rbac-good-practices/

Multifaktor Authentifizierung

Eine in der heutigen Zeit häufig verwendete Methode um Daten vor fremdem Zugriff zu schützen ist die sogenannte Multifaktor-Authentifizierung, häufig 2-Faktor-Authentifizierung (2FA). Dabei ist eine Anmeldung auf einem System nicht mehr durch bloße Zugangsdaten möglich. Während des Anmeldevorgangs wird vom System eine Anfrage über einen zusätzlichen Weg erzeugt, häufig Smartphone oder E-Mail. Nur wenn der Anwender auch über diesen Zugriff verfügt kann er sich beim System anmelden.

Service Mesh

Service Mesh verwaltet den Netzwerkverkehr zwischen den Diensten. Dies geschieht auf eine sehr viel elegantere und skalierbare Art und Weise, als dies sonst mit viel manueller, fehleranfälliger Arbeit und einem langfristig nicht tragbaren Betriebsaufwand verbunden wäre.
Im Allgemeinen setzt Service Mesh auf Ihrer Kubernetes-Infrastruktur auf und macht die Kommunikation zwischen den Diensten über das Netzwerk sicher und zuverlässig.

Ein Service Mesh kann sich ähnlich wie ein Tracking-Service für Pakete vorgestellt werden. Es verfolgt die Routing-Regeln und leitet den Datenverkehr und die Paketroute dynamisch, um die Zustellung zu beschleunigen und den Empfang sicherzustellen. (https://www.techtarget.com/searchitoperations/tutorial/Be-selective-with-Kubernetes-RBAC-permissions)

Git Repository

Die verwendeten Manifeste stehen in einem Git Repository online zur Verfügung: https://github.com/Zenska11/VDM-Kubernetes.git