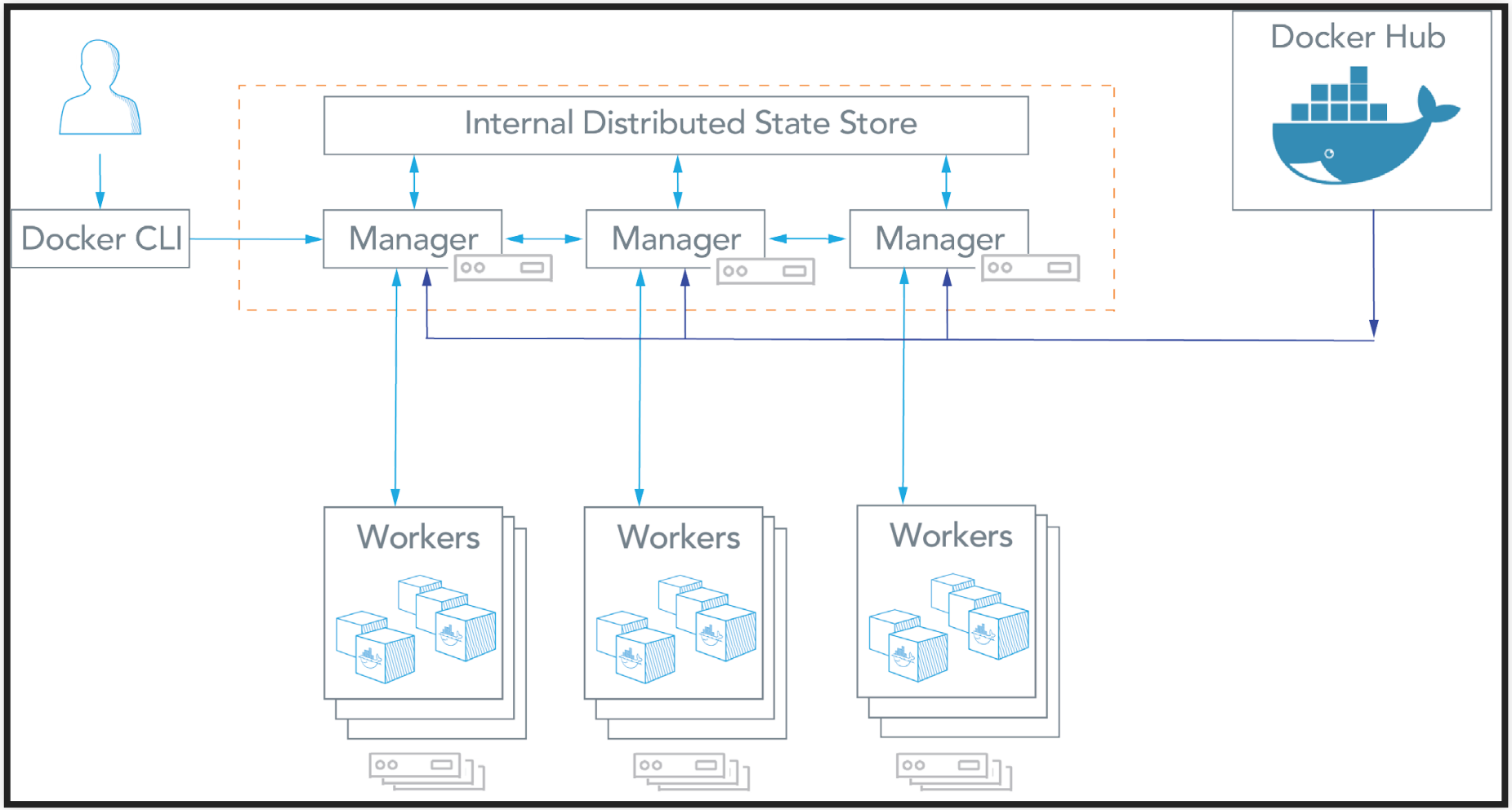
KAPITEL 5: ORCHESTRIERUNGSSYSTEME

LERNZIELE

- Kubernetes und Docker Swarm vergleichen
- Konzepte und Begriffe von Kubernetes erklären
- Grundelemente der Kubernetes CLI verwenden
- Kubernetes Manifeste erklären und anwenden
- Kubernetes Manifeste und Helm Charts differenzieren

5.1 DOCKER SWARM

- Ab Docker Engine v1.12.0 ist es möglich die Docker Engine im Swarm-Modus zu betreiben
- Ein Knoten übernimmt dabei den Manager, die zusätzlichen Knoten die dem Swarm beitreten arbeiten als Worker
- Jeder Knoten führt Tasks aus (i. a. Instanzen von Docker Images)
- Ermöglicht die Definition von Services, welche Tasks zusammenfassen, für Service Discovery dienen und Load Balancing übernehmen


Docker Hub
Internal Distributed State Store
Docker CLI
Manager
Manager
Manager
Workers
Workers
Workers

- **Node**: Eine Physische / Virtuelle Docker Engine Instanz. Können in eigener Infrastruktur oder extern betrieben werden.
- **Swarm**: Ein Cluster von Nodes (Docker Engines).
- **Manager Nodes**: Verwaltungsknoten erhält Dienstdefinitionen von Nutzern und verteilt Tasks an Arbeitsknoten. Dabei können
- **Worker Nodes**: Knoten, welche Tasks vom Manager zugewiesen bekommen und ausführen.
- **Service**: Ein Dienst bezieht sich auf ein Container Image und entsprechender Anzahl an Replikationen.
- **Task**: Atomare Aufgabe, welche auf einem Knoten ausgeführt wird. Kapselt die Container-Instanzen, welche aus den Images erzeugt werden. Tasks werden unabhängig voneinander ausgeführt.

GRUNDLAGEN

Aus vorangegangenen Beispielen haben wir eine `docker-compose.yaml` wie folgende gehabt

```yaml
version: "3"

services:
  app:
    image: marloto/simple-express-js
    ports:
     - 3000:3000
```

Init

Einmalig zu Beginn muss der Docker-Swarm initialisiert werden (von einem Knoten aus).

```
Local ~$ docker swarm init
```

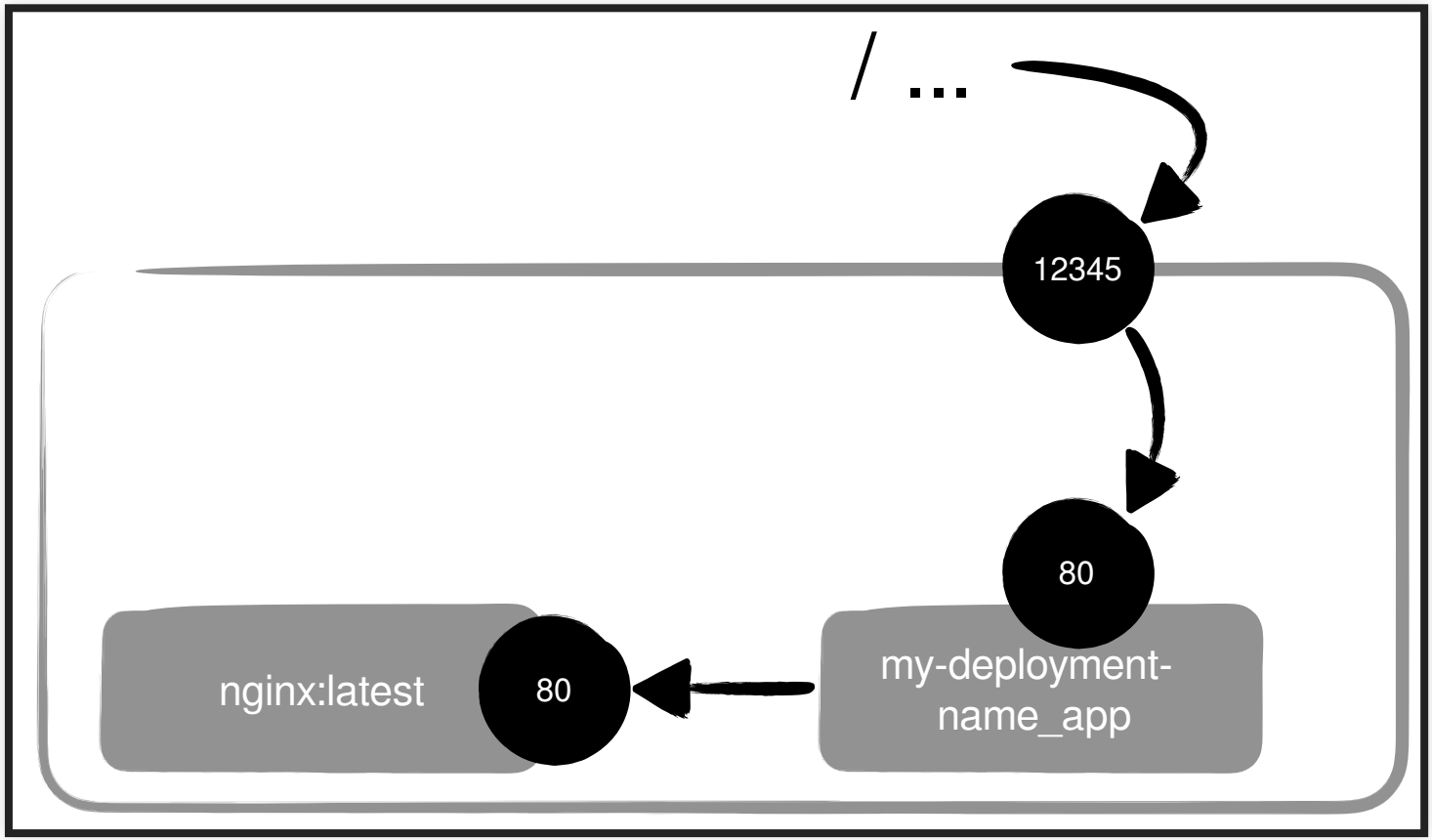
Man erhalt darauf hin einen Zugangsschlüssel und Informationen, wie weitere Docker Daemon diesem Swarm beitreten können.

Deploy Service

```
Local ~$ docker stack deploy --compose-file=docker-compose.yml my-deployment
Creating network my-deployment-name_default
Creating service my-deployment-name_app
Local ~$ docker stack ls
NAME                SERVICES
my-deployment-name   1
```

- Für das Deployment im Docker Swarm wird mit `docker stack deploy --compose-file=docker-compose.yml my-deployment-name` gearbeitet.
- Anschließend wird wie bereits bei Docker-Compose alles notwendige initialisiert.
- Die laufenden Dienste können mittels `docker stack ls` angezeigt werden

Es entstehen Dienste und nicht nur Container

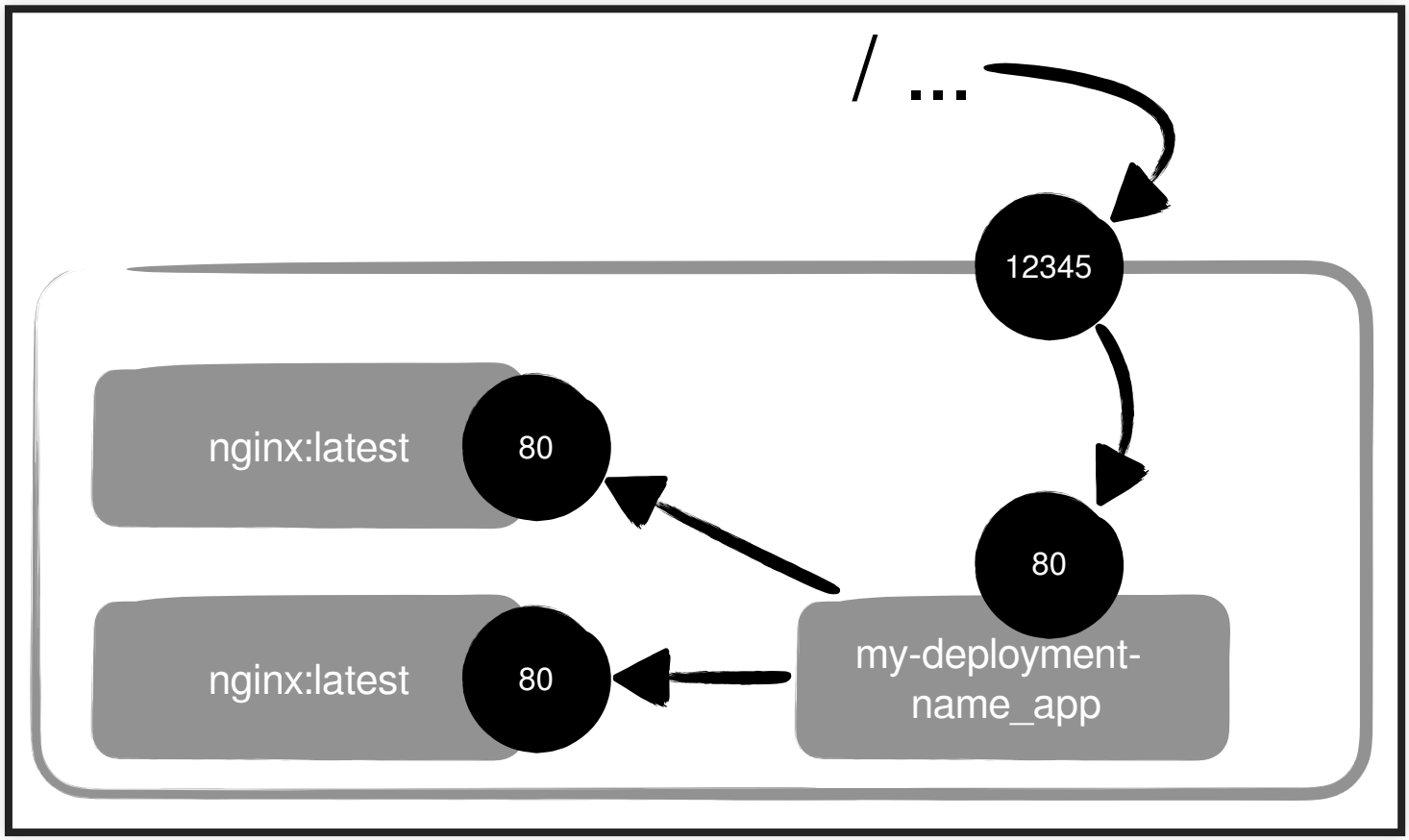


Update Service

```
Local ~$ docker service ls
ID                  NAME                          MODE                REPLICAS
ug2imu0j7e35        my-deployment-name_app        replicated          1/1
Local ~$ docker service scale my-deployment-name_app=2
my-deployment-name_app scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

Steigt die Last und ein Knoten ist nicht mehr ausreichend, lassen sich die Dienste skalieren mit `docker service scale my-deployment-name_app=2`

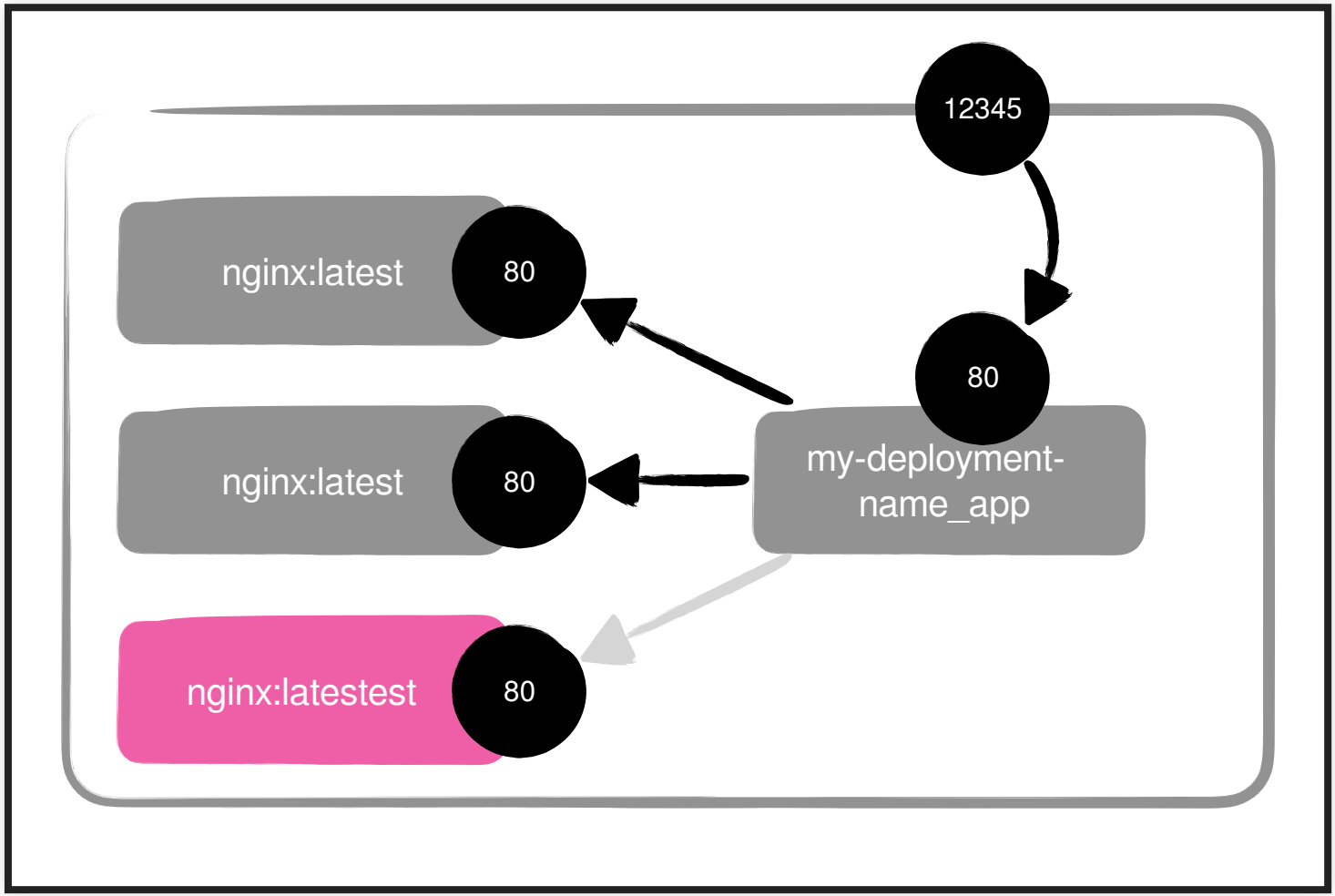
Einem Dienst sind zwei Container zugeordnet

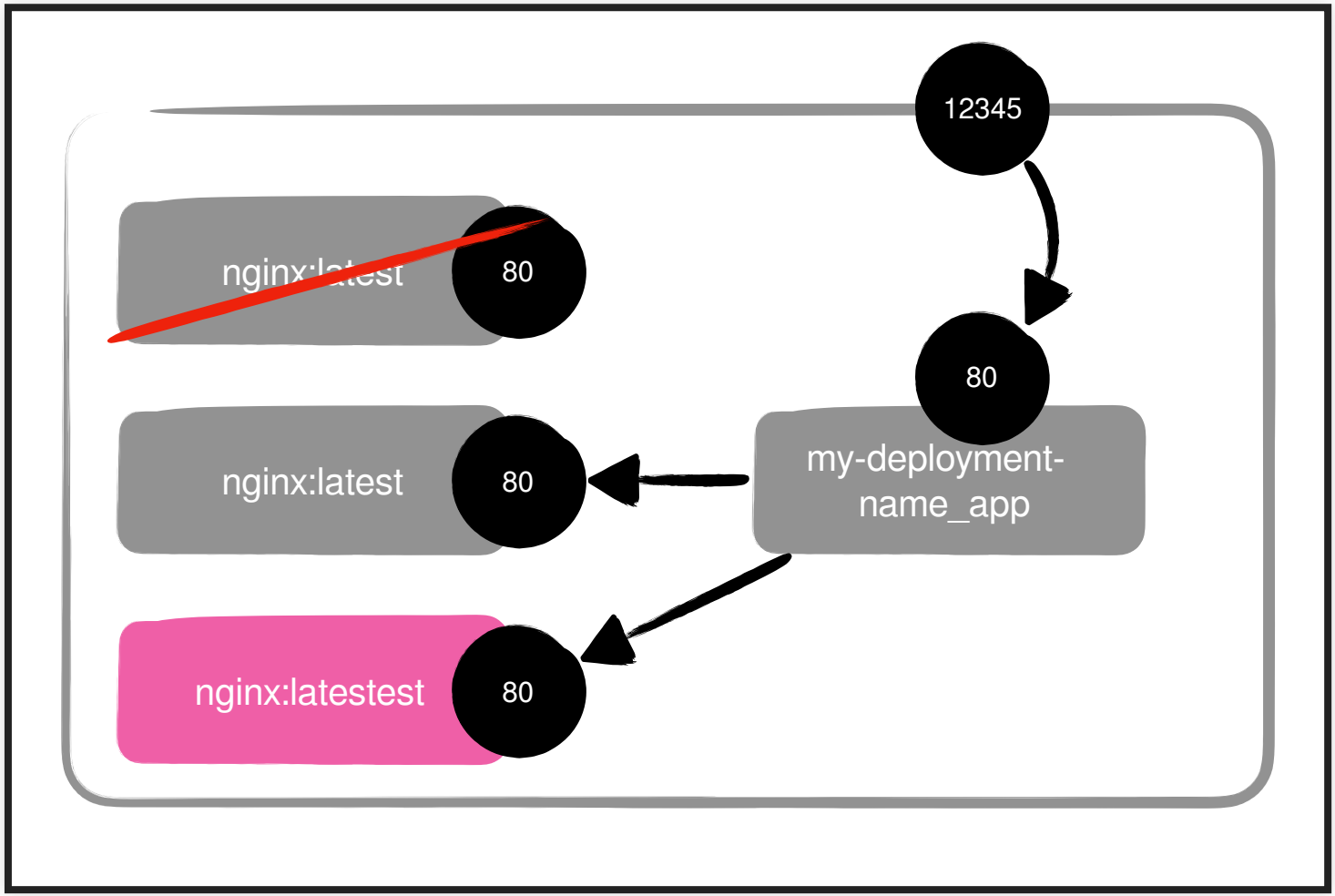


```
Local ~$ docker service update --force my-deployment-name_app
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

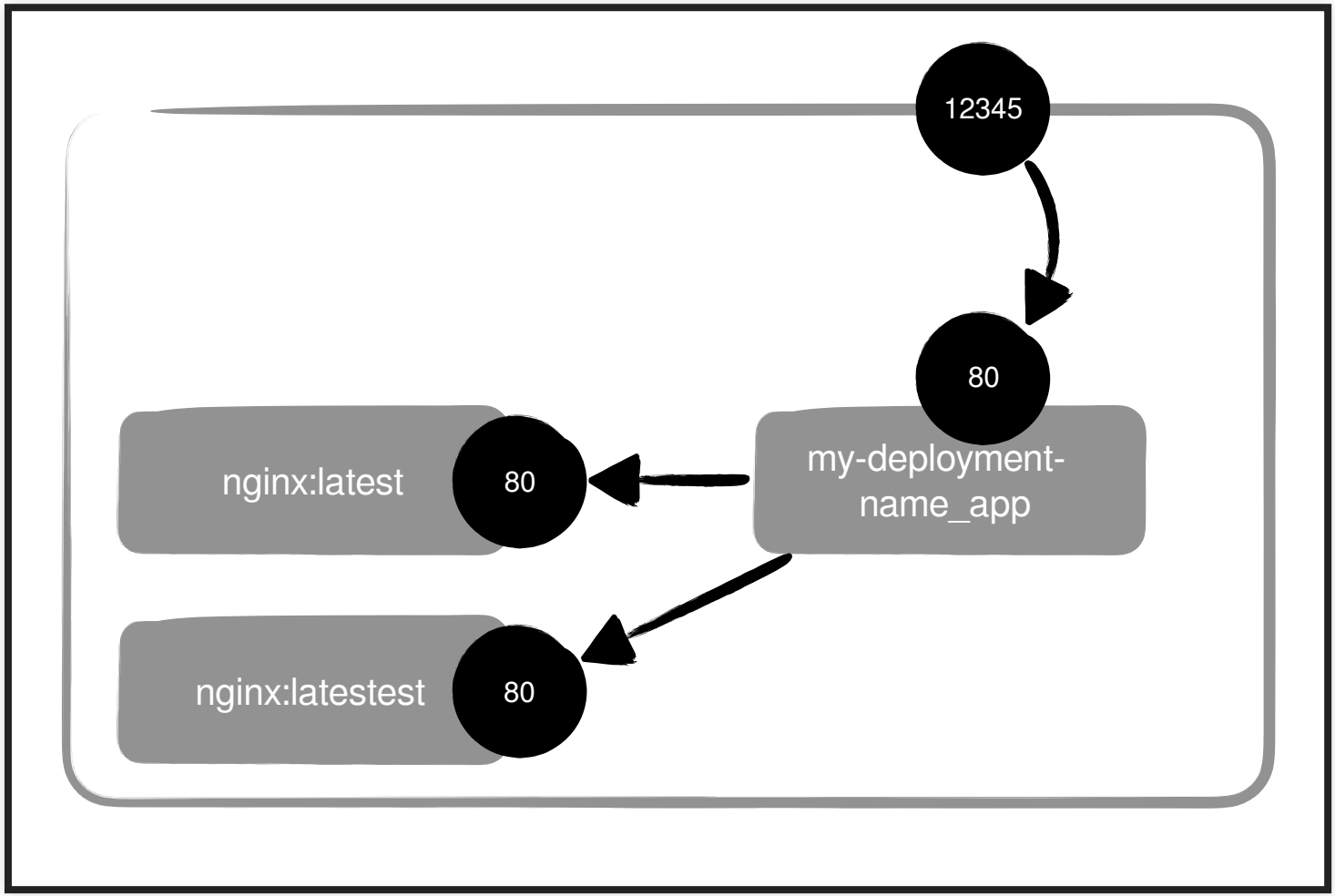
- Aktualisierungsprozesse sind über zusätzliche Attribute im `docker-compose.yml` Detailliert beschreibbar
- Mittels `docker service update --force my-deployment-name_app` lässt der Aktualisierungsprozess sich starten
- Update mit `--force` startet auch ohne Änderungen am Image

Ein neuer Container mit der neuen Version wird erzeugt

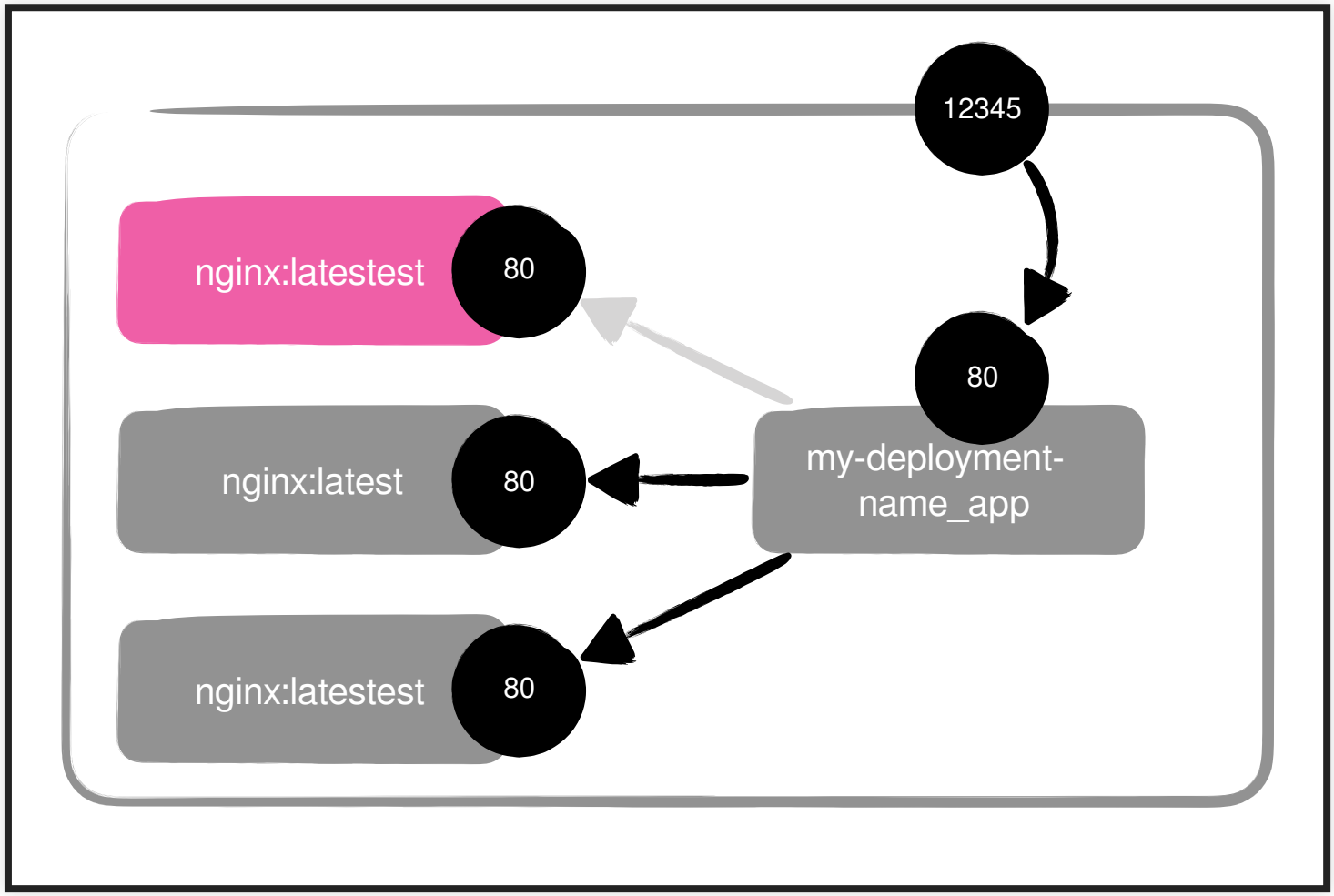



Erst wenn der neue Container läuft wird ein alter gestoppt
12345
nginx:latest
80
80
nginx:latest
80
my-deployment-
name_app
nginx:latestest
80

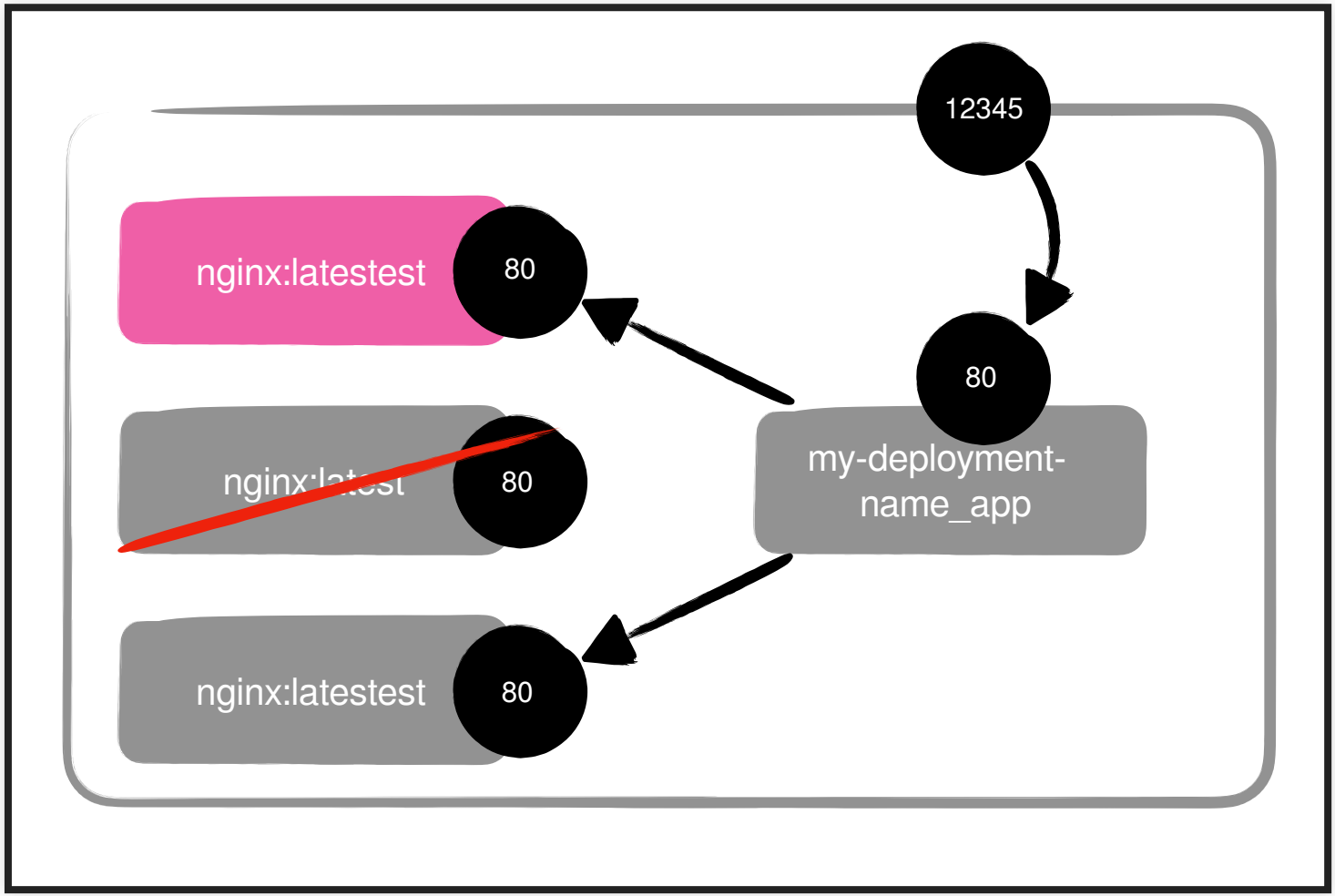
Es laufen dann zwei unterschiedliche Versionen



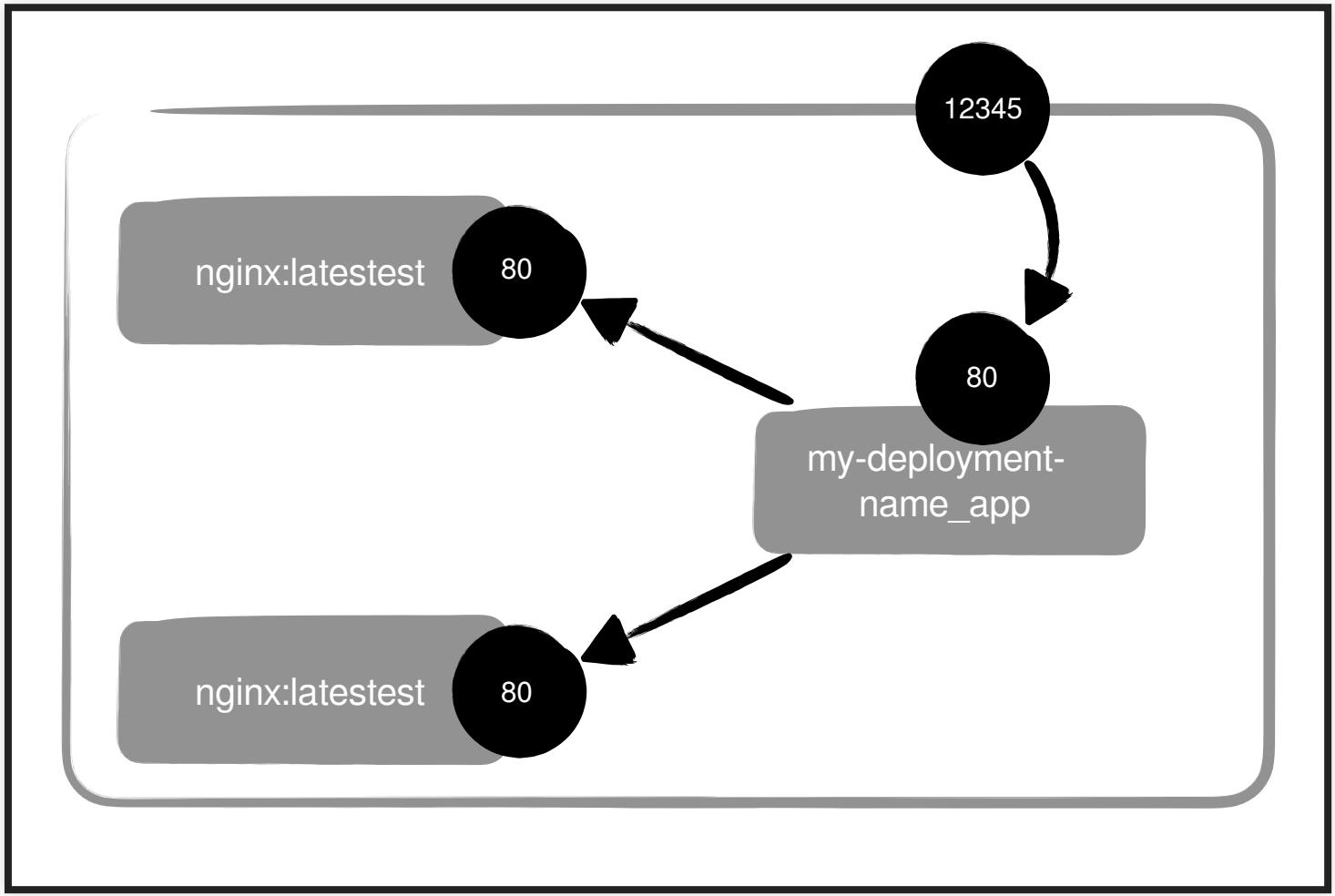
Anschließend entsteht ein weiterer neuer Container



... welcher den verbleibenden Container ersetzt



... und den Aktualisierungsprozess beendet



Remove Service

```
Local ~$ docker stack rm my-deployment-name
Removing service my-deployment-name_app
Removing network my-deployment-name_default
```

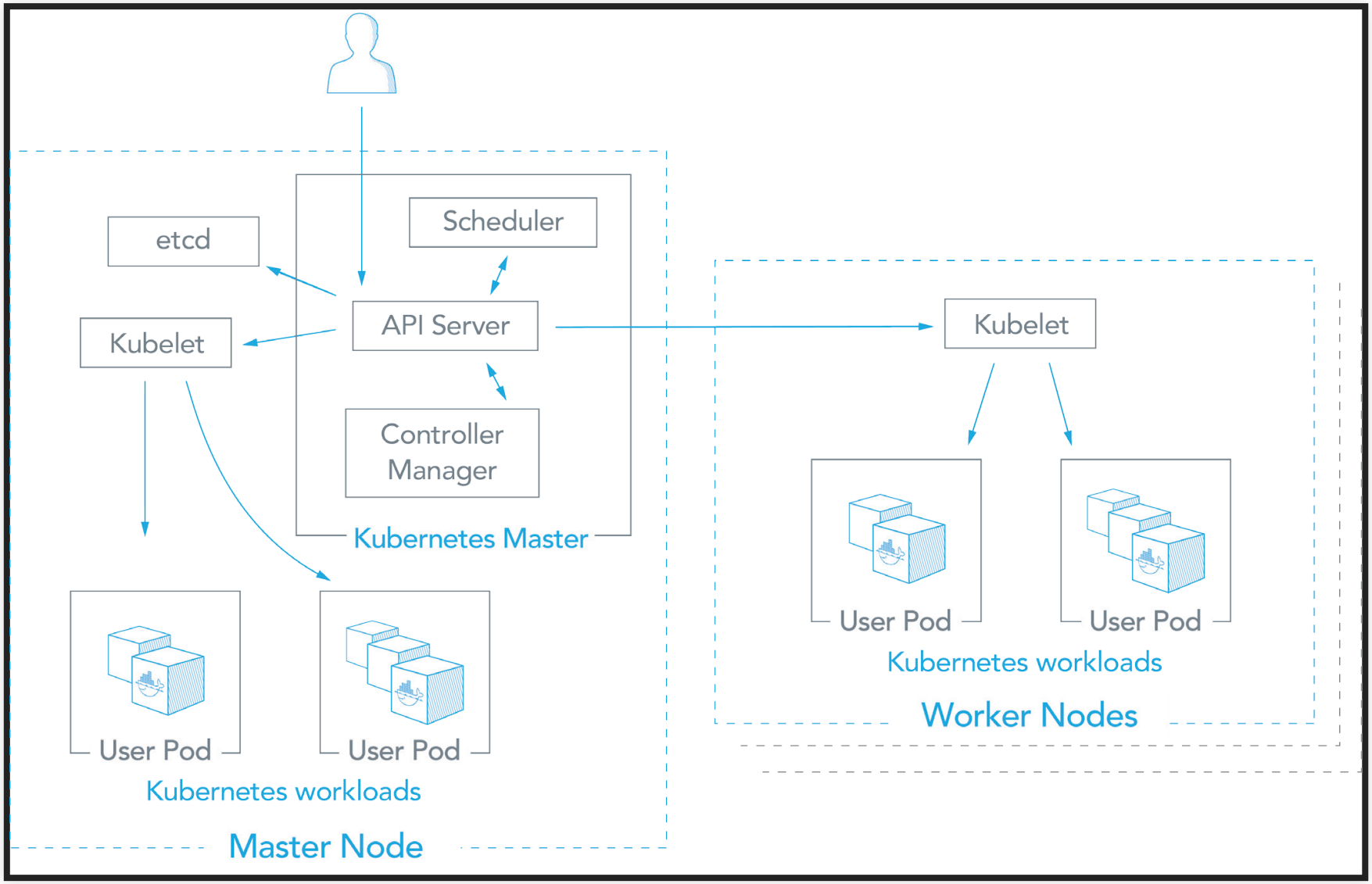
Stacks können genauso wie im Fall von Docker-Compose mit einem Befehl weggeräumt werden: `docker stack rm my-deployment-name`

5.2 KUBERNETES

- Kubernetes (vom griechischen Wort κυβερνήτης, »Steuermann« oder »Pilot«, abgeleitet) ist ein Open-Source System zum (automatisierten) Veröffentlichen, Skalieren und Verwalten von in Containern verpackten Anwendungen
- Kubernetes wurde von Google gebaut, basierend auf deren Erfahrungen bei der Ausführung von Containern im Produktivbetrieb und deren internen Cluster Management System "Borg"

> *So gut wie alle Services bei Google laufen in Containern: Gmail, Google Search, Google Maps, Google App Engine und so weiter.*

John Arundel und Justin Domingus: Cloud Native DevOps mit Kubernetes



Quelle: https://platform9.com/blog/kubernetes-docker-swarm-compared/

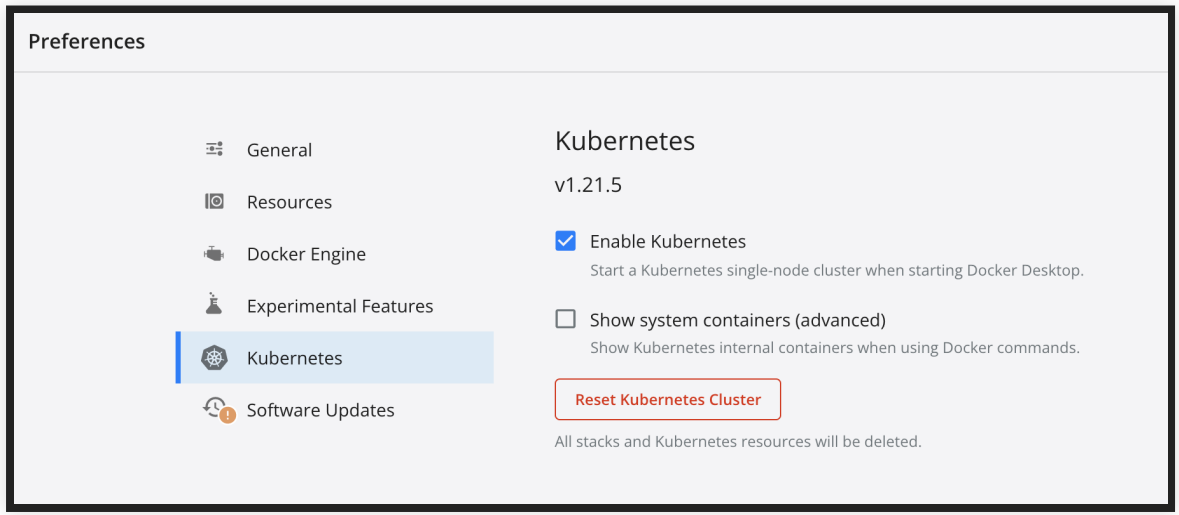
- **Pods:** Kubernetes veröffentlicht und plant Container in Gruppen, sogenannte Pods. Container in einem Pod werden auf dem selben Knoten ausgeführt und teilen sich Ressourcen wie Dateisystem, Kernel Namespace und IP-Adresse.
- **Deployments:** Ein Deployment beschreibt einen Bauzustand und kann genutzt werden um eine Menge von Pods zu verwalten / erzeugen.
- **Services:** Endpunkte für den namens-basierten Zugriff auf Pods, wobei selbige durch einen Label-Selektor zusammengefasst werden. Anfragen werden im Round-Robin an Pods verteilt. Ein DNS Server für den Zugriff wird von Kubernetes verwaltet und behält die Übersicht über neue, aktualisierte oder entfernte Pods.
- **Labels:** Key-Value-Paare welche z. B. Pods angehängt werden um die Suche zu vereinfachen oder dienen als Selektor für die Beschreibung einer Menge an Pods.

- **etcd:** Konsistenter und hochverfügbarer Schlüsselwertspeicher, der als Kubernetes-Backing-Store für alle Clusterdaten verwendet wird.
- **API-Server:** Der API-Server ist eine Komponente der Kubernetes-Kontrollebene, die die Kubernetes-API offenlegt. Der API-Server ist das Frontend für die Kubernetes-Kontrollebene.
- **Controller Manager:** Komponente der Steuerungsebene, die die Steuerungsprozesse ausführt. Umfasst (in einzelnen Prozessen) den Node Controller, Job Controller, Endpunkt Controller und Service Account & Token controller.
- **Scheduler:** Komponente der Steuerungsebene, die auf neu erstellte Pods ohne zugewiesenen Knoten achtet und einen Knoten auswählt, auf dem sie laufen sollen.
- **Kubelet:** Ein Agent, der auf jedem Knoten des Clusters läuft. Er stellt sicher, dass die Container in einem Pod laufen. Kommuniziert mit dem API Server.

Quelle: https://kubernetes.io/docs/concepts/overview/components/

CONTAINER IN KUBERNETES

Als erstes Beispiel, in Anlehnung an Docker, folgt der Start eines Containers. Hierfür ist eine Kubernetes-Umgebung notwendig. Diese wird mit dem *Docker Desktop* ausgeliefert und kann über die Einstellungen aktiviert werden.



Wird kein Docker Desktop verwendet ist der sogenannte *Minikube* eine Einstiegsmöglichkeit.

Vorbereitung

Nutzen Sie das bereitgestellte Projekt-Archiv simple-express-js.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Das Beispiel muss anschließend in einer Registry bereitgestellt werden, z.B. dem Docker Hub:

```
Local ~$ docker build -t express-example .
Local ~$ docker image tag express-example marloto/express-example
Local ~$ docker image push marloto/express-example
```

Pod erzeugen und Port-Forward

```
Local ~$ kubectl run demo --image=marloto/express-example --port=3000 \
    --labels app=demo
pod/demo created
Local ~$ kubectl port-forward pod/demo 9999:3000
Forwarding from 127.0.0.1:9999 -> 3000
Forwarding from [::1]:9999 -> 3000
Handling connection for 9999
```

- Zuerst wird aus dem Beispiel-Image ein Pod erzeugt und der interne Port 3000 bekannt gegeben
- Anschließend wird der externe Port 9999 auf den internen Port 3000 gemappt
- Sobald `http://localhost:9999/hello` aufgerufen wird, ist dies in der Konsole sichtbar

Details zum Pod

```
Local ~$ kubectl get pods --selector app=demo
NAME   READY   STATUS    RESTARTS   AGE
demo   1/1     Running   0          3m33s
Local ~$ kubectl describe pod/demo
Name:         demo
Namespace:    default
Priority:     0
Node:         docker-desktop/192.168.65.4
Start Time:   Wed, 22 Sep 2021 13:25:20 +0200
Labels:       app=demo
Annotations:  <none>
...
```

- Mittels `get` lassen sich Ressourcen in Kubernetes auflisten, z.B. `pods` und über den Selector kann mittels Label eine Eingrenzung vorgenommen werden
- Mittels `describe` können Details angezeigt werden

Pod entfernen

```
Local ~$ kubectl delete pods --selector app=demo
pod "demo" deleted
Local ~$ kubectl get pods --selector app=demo
No resources found in default namespace.
```

- Pods lassen sich mittels `delete` entfernen, erneut kann hierfür der Selektor genutzt werden
- Anschließend sollte der Pod nicht mehr existieren

Deployments und Services

Pods werden in der Praxis in Deployments bereitgestellt und über Services der Zugriff realisiert.

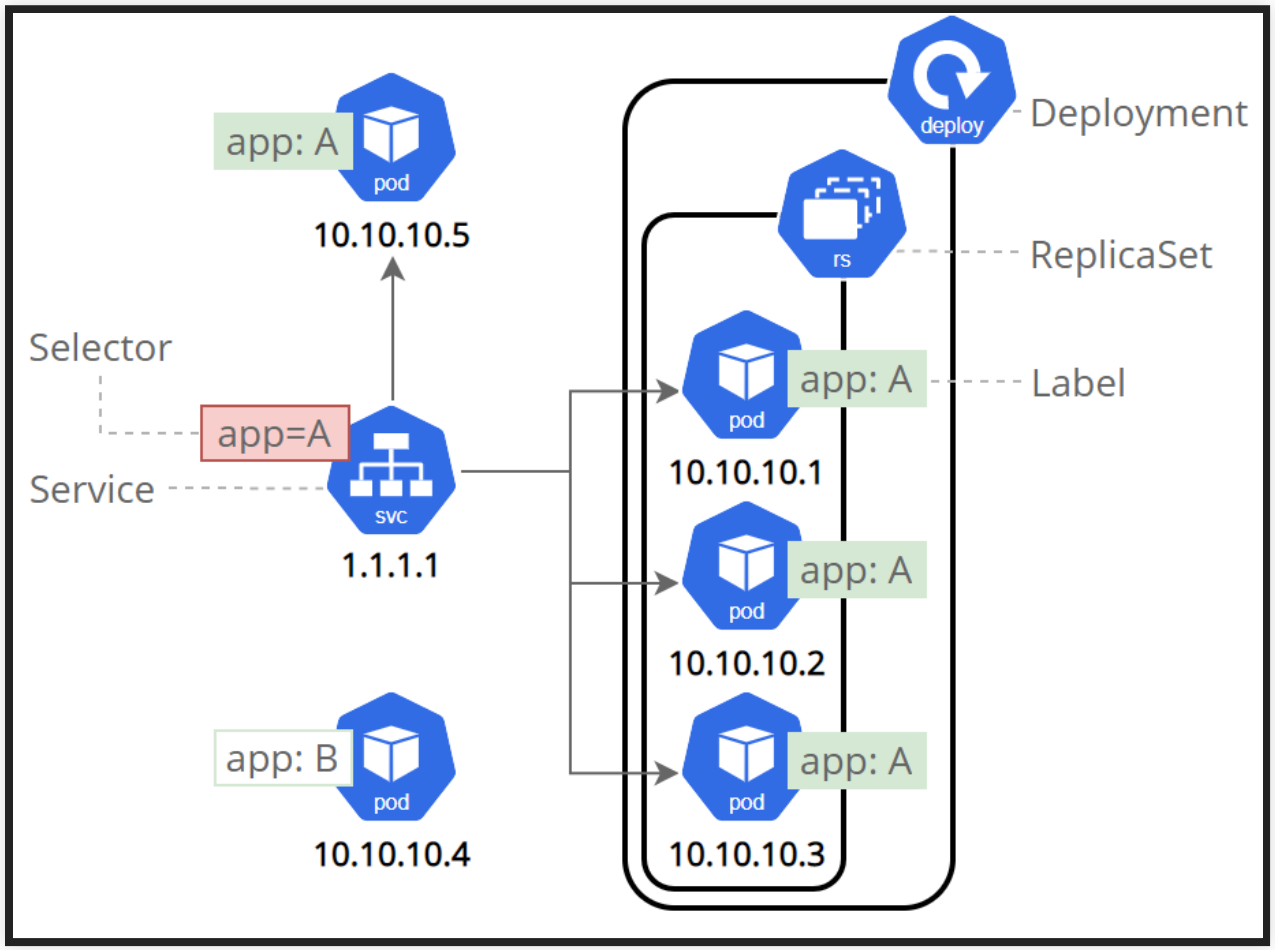
```
Local ~$ kubectl create deployment --image=marloto/express-example demo
deployment.apps/demo created
Local ~$ kubectl expose deployment demo --port=9999 --name=demo-http
service/demo-http exposed
kubectl port-forward deployment/demo 9999:3000
Forwarding from 127.0.0.1:9999 -> 3000
Forwarding from [::1]:9999 -> 3000
Handling connection for 9999
```

Deployments, Services und Pods anzeigen

```
Local ~$ kubectl get deployments
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
demo   1/1     1            1           5m8s
Local ~$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-b55d85cd8-tp2hv    1/1     Running   0          5m9s
Local ~$ kubectl get services
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
demo-http    ClusterIP   10.105.127.220   <none>        3000/TCP   5m10s
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    26h
```

- Anzeigen von Deployments
- Anzeigen von Pods
- Anzeigen von Services
- Ressourcen mit einmal abfragen, z.B. `kubectl get pods,deployments`

Services, Deployments und Pods



Quelle : Verwendung eines Services

Erklärung:

- Jeder Pod hat im Kubernetes Netzwerk eine eindeutige Adresse
- Pods die über Deployments und ReplicaSets entstanden sind als auch Pods die direkt gestartet wurden
- Services leiten Anfragen an Pods weiter und verwenden hierfür den Selector, welcher die Menge an Pods die die Verbindungsanfragen erhalten sollen, auswählt

Exec

```
Local ~$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-b55d85cd8-tp2hv    1/1     Running   0          6m34s
Local ~$ kubectl exec -it demo-b55d85cd8-tp2hv -- /bin/sh
# cat /etc/hostname
```

Wie auch in Docker ist es möglich *in* den Container zu wechseln und in dessen Kontext zu arbeiten. Hier wird eine Shell in dem gestarteten Pod erzeugt, mit der dann im folgenden gearbeitet werden kann.

Logs

```
Local ~$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
demo-b55d85cd8-tp2hv    1/1     Running   0          6m34s
Local ~$ kubectl logs -f demo-b55d85cd8-tp2hv

> simple-express@1.0.0 start /usr/src/app
> DEBUG=example:* node app.js

2021-09-22T12:07:15.422Z example:app Server has started on port 3000
```

Mittels `logs` lassen sich die stdout und stderr Ausgaben die in bei der Ausführung des Containers aufgelaufen sind anzeigen.

Deployments entfernen

```
Local ~$ kubectl delete deployment demo
deployment.apps "demo" deleted
Local ~$ kubectl delete service demo-http
service "demo-http" deleted
```

- Deployment und Service müssen einzeln entfernt werden.
- Das Entfernen des Deployments entfernt automatisch alle mit dem Deployment verbundenen Pods weg

RESSOURCEN-MANIFESTE

- Kubernetes ist ein inhärentes deklaratives System
- Es wird ständig der tatsächliche Status mit dem gewünschten Status abgeglichen
- Ziel: Status aktualisieren und Kubernetes führt die Anpassungen durch
- Verwendung von Manifesten zu Beschreibung des gewünschten Status

Beispiel Deployment-Manifest: deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
# ...
```

```yaml
# ...
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: marloto/express-example
          ports:
            - containerPort: 3000
```

Anwenden des Deployment-Manifests

```
Local ~$ kubectl apply -f deployment.yaml
deployment.apps/demo created
Local ~$ kubectl get pods --selector app=demo
NAME                    READY   STATUS    RESTARTS   AGE
demo-776484f85b-w7glb   1/1     Running   0          7m23s
Local ~$ kubectl get deployments --selector app=demo
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
demo   1/1     1            1           7m47s
```

- Mittels `apply` das Manifest anwenden
- Bestätigung zeigt, dass `deployment.apps/demo` gestartet wurde
- Details zum gestarteten Container lassen sich mit `get pods` anzeigen
- Details zum Deployment lassen sich mit `get deployments` anzeigen

Beispiel Service-Manifest: service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
    - port: 9999
      protocol: TCP
      targetPort: 3000
  selector:
    app: demo
  type: ClusterIP
```

Anwenden des Service-Manifest

```
Local ~$ kubectl apply -f service.yaml
service/demo created
Local ~$ kubectl get services --selector app=demo
NAME   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
demo   ClusterIP   10.99.247.189   <none>        9999/TCP   4m54s
Local ~$ kubectl port-forward service/demo 9999:9999
Forwarding from 127.0.0.1:9999 -> 3000
Forwarding from [::1]:9999 -> 3000
Handling connection for 9999
```

- Erneut mittels `apply` das Manifest anwenden
- Mittels `get services` kann überprüft werden ob der Service entstanden ist
- Für den Test kann erneut mittels `port-forward` geprüft werden ob Anfragen ankommen
- Rufen Sie anschließend http://localhost:9999/hello auf

Manifest entfernen

```
Local ~$ kubectl delete -f service.yaml
service "demo" deleted
Local ~$ kubectl delete -f deployment.yaml
deployment.apps "demo" deleted
```

- Zu erst den Service mit `delete` entfernen
- ... anschließend das Deployment

Mehrere Manifest-Dateien anwenden

Es lassen sich alle notwendigen Resource-Manifest-Dateien in einem Ordner sammeln und den gesamten Ordner in Kubernetes anwenden.

```
└── example/
    ├── deployment.yaml
    └── service.yaml
```

```
Local ~$ kubectl apply -f example/
deployment.apps/demo created
service/demo created
Local ~$ kubectl delete -f example/
deployment.apps "demo" deleted
service "demo" deleted
```

In folgenden Beispiel finden sich die dargestellten Manifeste.

Nutzen Sie das bereitgestellte Projekt-Archiv kubernetes-manifest.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

RESSOURCEN MANAGEMENT

Damit der Kubernetes-Scheduler Entscheidungen treffen kann, ist es notwendig zu definieren, wieviel Ressourcen einzelne Container benötigen.

Hierfür bietet Kubernetes die Möglichkeit Ressourcen-Anforderungen und -Grenzen zu definieren.

Im wesentlichen betrifft dies zwei Arten von Ressourcen: CPU und Speicher.

CPU

Wird in CPU-Einheiten ausgedrückt, die sich an einer vCPU in der AWS, einem Google Cloud Core, einem Azure vCore oder einem Hyperthread orientiert. In Kubernetes werden die CPU-Anforderungen und -Grenzen in Millicpus ausgedrückt, z.B. 100m sind 10% einer CPU.

Speicher

Wird in Bytes angegeben, praktischerweise ehr in Mebibytes (MiB).

Hinweis: Kubernetes erlaubt es Ressourcen zu überbuchen, also die Summe aller Container kann die Ressourcen des Hosts übersteigen

Beispiel für Container-Anforderungen

```yaml
spec:
  containers:
    - name: demo
      image: marloto/express-example:hello
      ports:
        - containerPort: 8888
      resources:
        requests:
          memory: "10Mi"
          cpu: "100m"
```

Definiert die Anforderung von 10 MiB für den Speicher sowie 100 Millicpus.

Beispiel für Container-Grenzen

```yaml
spec:
  containers:
    - name: demo
      image: marloto/express-example:hello
      ports:
        - containerPort: 8888
      resources:
        limits:
          memory: "10Mi"
          cpu: "100m"
```

Geben Sie immer Ressourcen-Anforderungen und -Grenzen für Ihre Container an. Das hilft Kubernetes dabei, Ihre Pods gut zu schedulen und zu managen.

Quelle: John Arundel und Justin Domingus: Cloud Native DevOps mit Kubernetes

Liveness-Probe

- Von außen betrachtet ist es nicht ersichtlich, ob ein Container *noch läuft* und Requests bearbeiten kann
- Hierfür ist es erforderlich zu definieren, wie das Funktionieren der Anwendung geprüft werden kann
- Mittels Liveness-Probe kann diese Angabe im Manifest erfolgen, zum Beispiel indem eine HTTP-Anfrage gesendet wird

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 3000
  initialDelaySeconds: 3
  periodSeconds: 3
```

Bei eine Antwort mit HTTP-Statuscode 2xx oder 3xx wird angenommen, dass der Container noch *lebt*. Andernfalls wird der Container als tot angesehen und neu gestartet.

Readiness-Probe

Ähnlich wie Liveness jedoch mit anderem Fokus. Mittels Readiness wird geprüft, wann der Dienst bereit ist.

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 3000
  initialDelaySeconds: 3
  periodSeconds: 3
```

Wird ein Pod ohne Readiness-Probe gestartet, wird direkt Traffic an den Pod geleitet. Existiert die Probe wartet Kubernetes bis die Probe erfolgreich war.

AUTOMATISIERTE SKALIERUNG

Wie lassen sich Deployments skalieren?

Horizontaler Pod Autoscaler

- Verwendung von Grenzen und Limits
- Empfehlenswert Liveness- und Readiness-Probe
- Automatische Skalierung basierend auf einem überwachten Wert

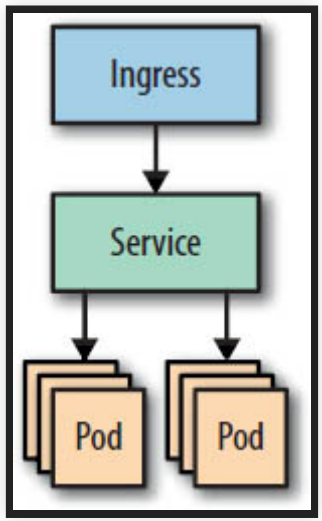
```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
    name: demo-hpa
    namespace: default
spec:
    scaleTargetRef:
        apiVersion: extensions/v1beta1
        kind: Deployment
        name: demo
    minReplicas: 1
    maxReplicas: 10
    metrics:
      - type: Resource
        resource:
            name: cpu
            targetAverageUtilization: 80
```

- Erzeugt einen Autoscaler
- Verknüpft den Autoscaler mit einem Deployment
- Erlaubt die Skalierung von 1 bis 10 Replicas
- Orientiert sich an der CPU, wobei 80% Ausnutzung im Durchschnitt über die aktuellen Replicas vorliegen muss

INGRESS-RESSOURCEN

Entspricht einer Art Load-Balancer vor den Services in einem Kubernetes-Cluster

Ingress-Ressourcen empfangen Anfragen, senden diese an den Service und dieser verteilt die Anfragen auf die Pods basierend auf den definierten Selektoren.



Einfaches Beispiel eines Manifests für eine Ingress-Ressource

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
    name: demo-ingress
spec:
    backend:
        serviceName: demo
        servicePort: 9999
```

Ingress Regeln für die Verteilung von Traffic

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
    name: fanout-ingress
spec:
    rules:
      - http:
            paths:
              - path: /hello
                backend:
                    serviceName: hello
                    servicePort: 80
              - path: /goodbye
                backend:
                    serviceName: goodbye
                    servicePort: 80
```

Quelle: John Arundel und Justin Domingus: Cloud Native DevOps mit Kubernetes

VOLUMES

Wie in einer Docker-Engine sind Modifikationen auf Dateiebene in Containern flüchtig. Wird ein Container beendet (z.B. durch einen Crash), wird der Container in einem neuen sauberen Zustand gestartet.

Kubernetes unterstützt viele Arten von Volumes. Ein Pod kann eine beliebige Anzahl von Volumetypen gleichzeitig verwenden. *Flüchtige* Volumes haben eine Lebensdauer von einem Pod, *persistente* Volumes existieren über die Lebensdauer eines Pods hinaus.

Typen für Volumes

- `awsElasticBlockStore`: bindet ein Amazon Web Services (AWS) EBS Volume
- `azureDisk`: bindet ein Microsoft Azure Data Disk
- `configMap`: bietet eine Möglichkeit, Konfigurationsdaten in Pods zu injizieren; in einer ConfigMap gespeicherten Daten können in einem Volume des Typs configMap referenziert und dann von containerisierten Anwendungen, die in einem Pod laufen, konsumiert werden

- `emptyDir`: existiert so lange, wie der Pod auf diesem Knoten läuft; das emptyDir-Volume ist zunächst leer; alle Container des Pods können dieselben Dateien im emptyDir-Volume lesen und schreiben, wobei dieses Volume in jedem Container unter demselben oder einem anderen Pfad gemountet sein kann
- `nfs`: kann eine bestehende NFS-Freigabe (Network File System) in einen Pod einhängen; der Inhalt eines nfs-Volumes bleibt erhalten und das Volume wird lediglich ausgehängt wenn der Pod beendet wird
- `local`: repräsentiert ein gemountetes lokales Speichergerät wie eine Festplatte, eine Partition oder ein Verzeichnis

Verwendung von Volumes

- PersistentVolume beschreibt den Datenspeicher (local, nfs oder andere)
- PersistentVolumeClaim werden von Pods genutzt, um physischen Speicher anzufordern
- Pods verwenden PersistentVolumeClaim und binden Sie in Containern im Dateisystem

Beispiel PersistentVolume

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

```
kubectl apply -f pv-volume.yaml
kubectl get pv task-pv-volume
```

Quelle: Kubernetes Dokumentation

Für das folgende Beispiel bietet es sich an, in den Ordner eine HTML-Datei abzulegen.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Test</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

Hinweis: In der Praxis sind Local-Volumes nicht geeignet, hier kommen entsprechende Typen zum Einsatz, die in einem Cluster einen dezentralen Zugriff erlauben (nfs, azureDisk, awsElasticBlockStore, usw.)

Beispiel PersistentVolumeClaim

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

```
kubectl apply -f pv-claim.yaml
kubectl get pv task-pv-volume # Zeigt Status = Bound
kubectl get pvc task-pv-claim
```

Quelle: Kubernetes Dokumentation

Beispiel Verwendung in Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

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

Quelle: Kubernetes Dokumentation

Ausprobieren

```
kubectl port-forward pod/task-pv-pod 9999:80
```

Anschließend http://localhost:9999/index.html im Browser öffnen.

Aufräumen

```
kubectl delete pod task-pv-pod
kubectl delete pvc task-pv-claim
kubectl delete pv task-pv-volume
```

Nutzen Sie das bereitgestellte Projekt-Archiv kubernetes-volumes.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

KONFIGURATION

Trennung von Anwendung und Konfiguration in Kubernetes Setups.

Dabei umfassen Konfigurationen Werte bezüglich umgebungsspezifische Einstellungen, DNS-Adressen anderer Services und Authentifizierungs-Credentials. Ein wichtiges Element zum Speichern von Konfigurationen in Kubernetes sind **ConfigMaps**.

ConfigMaps erlauben benannte Schlüssel/Wert-Paare als Datei im Pod zu erzeugen oder in die Umgebung zu injizieren.

Als Beispiel soll folgende Konfiguration dienen:

```
autoSaveInterval: 60
batchSize: 128
protocols:
  - http
  - https
```

Diese Konfiguration könnte über ein ConfigMap-Manifest bereitgestellt werden.

```
apiVersion: v1
data:
  config.yaml: |
    autoSaveInterval: 60
    batchSize: 128
    protocols:
      - http
      - https
kind: ConfigMap
metadata:
  name: demo-config
  namespace: demo
```

- Die Werte aus dem Beispiel sind hier in dem Abschnitt `data` eingebettet.
- Das Manifest kann anschließend wie bereits bekannt mittels `kubectl apply -f config-manifest.yaml` angewendet werden

Alternativ kann die Konfiguration auch mit einer Kurzform bereitgestellt werden

```
Local ~$ kubectl create configmap demo-config --namespace=demo --from-file=c
configmap "demo-config" created
```

Export der Konfiguration anschließend möglich mittels:

```
Local ~$ kubectl get configmap/demo-config --namespace=demo --export -o yaml
```

Verwendung in Containern als ENV-Variable

Die Beispiel-Anwendung in Express und NodeJS erlaubt die Konfiguration des verwendeten Begrüßung über eine Umgebungsvariable: vgl. `const greeting = process.env.GREETING || 'Hello, World!';`

Eine Konfiguration über eine ConfigMap könnte wie folgt aussehen:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  greeting: 'Hello, Universe!'
```

Das zugehörige Deployment des Containers kann dann entsprechend angepasst werden:

```yaml
# ...
spec:
  containers:
    - name: demo
      image: marloto/express-example
      ports:
        - containerPort: 3000
      env:
        - name: GREETING
          valueFrom:
            configMapKeyRef:
              name: demo-config
              key: greeting
```

Kurzform, zur direkten Übernahme aller Werte aus der Konfiguration:

```yaml
# ...
spec:
  containers:
    - name: demo
      image: marloto/express-example
      ports:
        - containerPort: 3000
      envFrom:
        - configMapKeyRef:
            name: demo-config
```

Alle Werte werden mit dem verwendeten Schlüssel übernommen, in dem Fall heißt die ENV-Variable `greeting` (Kleinbuchstaben).

Verwendung in Containern als Argumente

```yaml
# ...
spec:
  containers:
    - name: demo
      image: marloto/express-example
      args:
        - "${GREETING2}"
      ports:
        - containerPort: 3000
      env:
        - name: GREETING2
          valueFrom:
            configMapKeyRef:
              name: demo-config
              key: greeting
```

Konfigurationsdateien ableiten

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  config: |
    {"greeting": "Hello, Mars!"}
```

Im Beispiel wird ein Datenblock erzeugt, hier `{"greeting": "Hello, Mars!"}` mit JSON, welcher die Konfigurationsdaten enthält. Wichtig ist der `|` in YAML für Datenblöcke.

Verwendung in Container-Deployment

```yaml
spec:
  containers:
    - name: demo
      image: marloto/express-example
      ports:
        - containerPort: 3000
      volumeMounts:
        - mountPath: /usr/src/app/config/
          name: demo-config-volume
          readOnly: true
  volumes:
    - name: demo-config-volume
      configMap:
        name: demo-config
      items:
        - key: config
          path: config.json
```

GEHEIMNISSE

Geheimnisse sollten nicht in Konfigurationen platziert werden

```
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
stringData:
  magicWord: xyzzy
```

Verwendung in einem Container

```yaml
# ...
spec:
  containers:
    - name: demo
      image: marloto/express-example
      ports:
        - containerPort: 3000
      env:
        - name: GREETING
          valueFrom:
            secretKeyRef:
              name: demo-secret
              key: magicWord
```

Vergleich Secrets vs. ConfigMap

- ConfigMap und Secrets lassen sich ausgeben mittels `kubectl describe ...`
- Im Fall von Secrets werden Informationen nicht direkt ausgegeben (allerhöchstens als Base64-String)
- Der Zugriff auf Secrets lässt sich durch RBAC steuern, womit Containern und Nutzern nur eingeschränkter Zugriff erlaubt ist

5.3 HELM

Packetmanager für Kubernetes: Helm

Helm ist ein Open-Source-Werkzeug zum Bereitstellen, Teilen und Nutzen von für Kubernetes entwickelte Software.

Details zu Helm

- Entwickelt 2015 von Deis und später von Microsoft übernommen
- Helm umfasst eine CLI mit der der Packetmanager genutzt wird
- Erlaubt die Installation von Anwendungen in einem Kubernetes Cluster
- Erfordert die Installation von Kubernetes-Ressourcen, damit Helm mit Kubernetes kommunizieren kann
- Graduated Status innerhalb der Cloud Native Computing Foundation

Installation

Helm CLI wird eigenständig installiert (z.B. über Apt, Chocolatey oder Homebrew), entsprechende Anleitungen je System lassen sich in der Dokumentation finden.

Anschließend ist es notwendig ein Repository zu ergänzen, z.B. mittels:

```
Local ~$ helm repo add bitnami https://charts.bitnami.com/bitnami
```

Wichtige Begriffe

- Ein **Chart** ist ein Helm-Paket mit allen Ressourcen-Definitionen, die zum Ausführen einer Anwendung in Kubernetes notwendig sind.
- Ein **Repository** ist ein Ort, an dem Charts gesammelt und zur Verfügung gestellt werden.
- Ein **Release** ist eine bestimmte Instanz eines Charts, die in einem Kubernetes-Cluster läuft.

John Arundel und Justin Domingus: Cloud Native DevOps mit Kubernetes.

CLI VERWENDUNG

Charts auflisten

```
Local ~$ helm search repo bitnami
NAME                    CHART VERSION   APP VERSION   DESCRIPTION
...
bitnami/mongodb         10.28.1         4.4.10        NoSQL document-oriented databas
bitnami/mysql           8.8.8           8.0.26        Chart to create a Highly availa
...
```

Installation eines Charts

```
Local ~$ helm install bitnami/mysql --generate-name
NAME: mysql-1634312136
LAST DEPLOYED: Fri Oct 15 17:35:40 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
...
```

Start überwachen

```
Local ~$ kubectl get pods -w --namespace default
NAME                    READY   STATUS              RESTARTS   AGE
mysql-1634312136-0      0/1     ContainerCreating   0          31s
```

Chart entfernen

```
Local ~$ helm delete mysql-1634312136
release "mysql-1634312136" uninstalled
```

ÜBUNG: CHART DEPLOYEN

Installieren Sie Helm über einen für Sie passenden Weg. Ergänzen Sie das Bitnami-Repo und installieren Sie `wordpress` als Chart wie beschrieben.

ÜBUNG: CHART REVIEW

Führen Sie einen Review der Template-Dateien des Wordpress-Charts im Bitnami-Repo durch, identifizieren Sie bekannte Merkmale in den Kubernetes Manifesten.

https://github.com/bitnami/charts/tree/master/bitnami/wordpress

5.4 REFERENZEN

- Arundel, J., & Domingus, J. (2019): Cloud Native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud. O'Reilly Media.