

## Cluster-Orchestrierung & Cluster Scheduling

Felix Kampfer

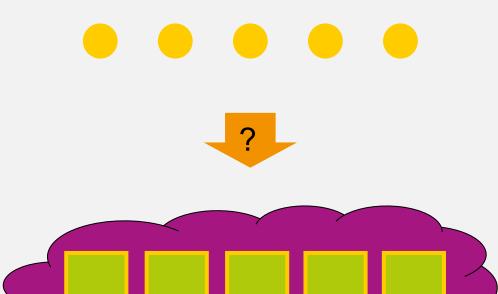
felix.kampfer@qaware.de



## **Cluster Scheduling**

#### **Das Problem**



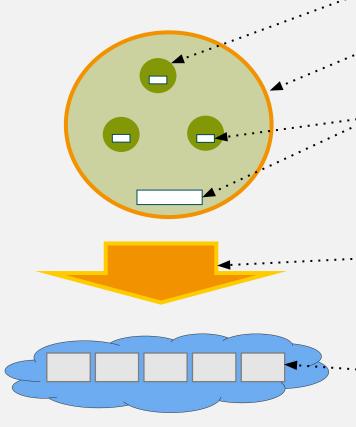


#### Rechenaufgaben

Rechen-Ressourcen (z.B. per laaS)

#### **Terminologie**





**Task**: Atomare Rechenaufgabe inklusive Ausführungsvorschrift.

**Job**: Menge an Tasks mit gemeinsamen Ausführungsziel. Die Menge an Tasks ist i.d.R. als DAG mit Tasks als Knoten und Ausführungsabhängigkeiten als Kanten repräsentiert.

**Properties**: Ausführungsrelevante Eigenschaften der Tasks und Jobs, wie z.B.:

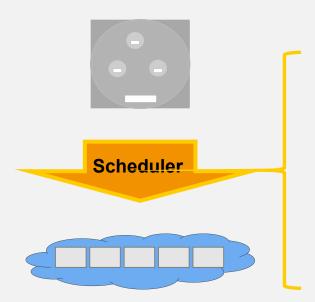
- Task: Priorität, Ressourcenverbrauch
- Job: Abhängigkeiten der Tasks, Ausführungszeitpunkt

Scheduler: Ausführung von Tasks auf den verfügbaren Resources unter Berücksichtigung der Properties und gegebener Scheduling-Ziele (z.B. Fairness, Durchsatz, Ressourcenauslastung). Ein Scheduler kann präemptiv sein, also die Ausführung von Tasks unterbrechen und neu aufsetzen können.

**Resources**: Cluster an Rechnern mit CPU-, RAM-, HDD-, Netzwerk-Ressourcen. Ein Rechner stellt seine Ressourcen temporär zur Ausführung eines oder mehrerer Tasks zur Verfügung (**Slot**). Die parallele Ausführung von Tasks ist isoliert zueinander.

#### Aufgaben eines Cluster-Schedulers:





**Cluster Awareness**: Die aktuell verfügbaren Ressourcen im Cluster kennen (Knoten inkl. verfügbare CPUs, verfügbarer RAM und Festplattenspeicher sowie Netzwerkbandbreite). Dabei auch auf Elastizität reagieren.

**Job Allocation**: Zur Ausführung eines Services die passende Menge an Ressourcen für einen bestimmten Zeitraum bestimmen und allokieren.

**Job Execution**: Einen Service zuverlässig ausführen und dabei isolieren und überwachen.

#### Die einfachste Form des Scheduling: Statische Partitionierung



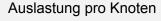
# Zeit Cluster-Zustand

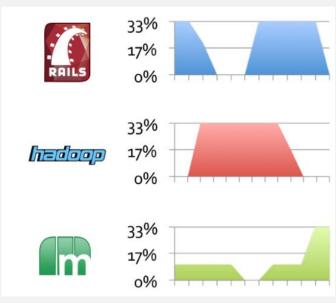
#### Vorteil:

Einfach zu realisieren

#### Nachteile:

- Nicht flexibel bei geänderten Bedürfnissen
- Geringere Auslastung
  - → hohe Opportunitätskosten

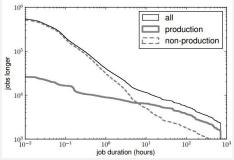




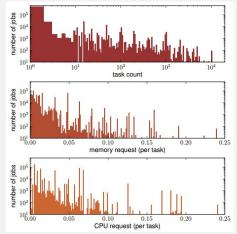
Bildquelle: Practical Considerations for Multi-Level Schedulers, Benjamin Hindman, 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015

#### Heterogenität im Scheduling

- In typischen Clustern ist die Workload an Jobs sehr heterogen.
- Charakteristische Unterschiede sind:
  - Ausführungsdauer: min, h, d, INF.
  - Ausführungszeit: sofort, später, zu einem Zeitpunkt.
  - Ausführungszweck: Datenverarbeitung, Request-Handling.
  - Ressourcenverbrauch: CPU-, RAM-, HDD-, NW-dominant.
  - Zustand: zustandsbehaftet, zustandslos.
- Zu unterscheiden sind mindestens:
  - **Batch-Jobs**: Ausführungszeit im Minuten- bis Stundenbereich. Eher niedrige Priorität und gut unterbrechbar. Müssen i.d.R. bis zu einem bestimmten Zeitpunkt abgeschlossen sein. Zustandsbehaftet.
  - **Service-Jobs**: Sollen auf unbestimmte Zeit unterbrechungsfrei laufen. Haben hohe Priorität und sollten nicht unterbrochen werden. Teilweise zustandslos.



Ausführungsdauer



Ressourcenverbrauch

Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis, Charles Reiss et al., 2012

## Bestehende Ressourcen einer Cloud können durch dynamische Partitionierung wesentlich effizienter genutzt werden.

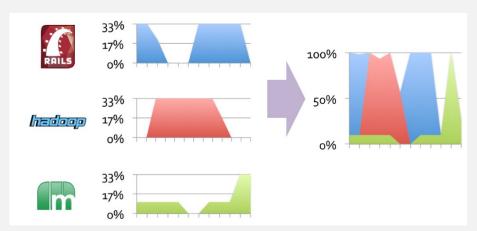


Cluster-Zustand



#### Statische Partitionierung

#### **Dynamische Partitionierung**



#### Vorteile der dynamischen Partitionierung:

- Höhere Auslastung der Ressourcen → weniger
   Ressourcen notwendig → geringere Betriebskosten
- Potenziell schnellere Ausführung einzelner Tasks, da Ressource opportun genutzt werden können.

#### Cluster-Scheduler: Eingabe, Verarbeitung, Ausgabe

<u>Eingabe</u> eines Cluster-Schedulers ist Wissen über die Jobs und Tasks (Properties) und über die Ressourcen:

- Resource Awareness: Welche Ressourcen stehen zur Verfügung und wie ist der entsprechende Bedarf des Tasks?
- Data Awareness: Wo sind die Daten, die ein Task benötigt?
- QoS Awareness: Welche Ausführungszeiten müssen garantiert werden?
- **Economy** Awareness: Welche Betriebskosten dürfen nicht überschritten werden?
- Priority Awareness: Wie ist die Priorität der Task zueinander?
- Failure Awareness: Wie hoch ist die Wahrscheinlichkeit eines Ausfalls? (z.B. da ein Rack oder eine Stromvers.)
- **Experience** Awareness: Wie hat sich ein Tasks in der Vergangenheit verhalten?



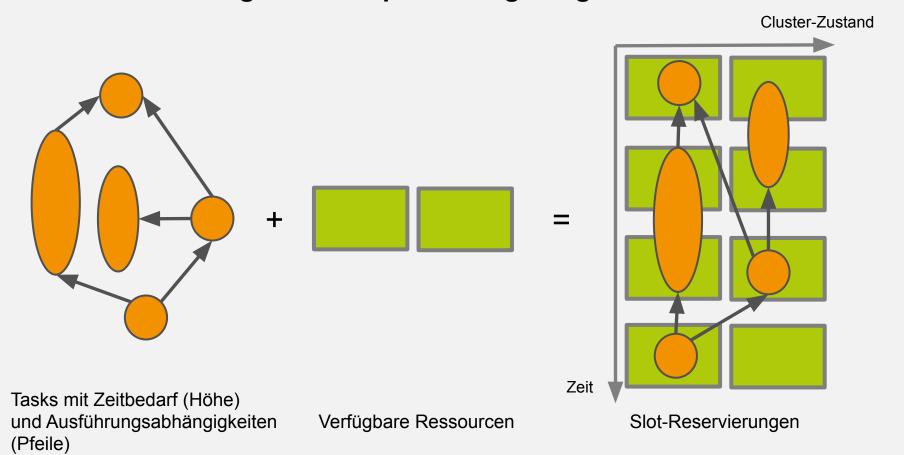
#### <u>Ausgabe</u> eines Cluster-Schedulers: Placement Decision als

- Slot-Reservierungen
- Slot-Stornierungen
   (im Fehlerfall, Optimierungsfall,
   Constraint-Verletzung)

<u>Verarbeitung</u> im Cluster-Scheduler: Scheduling-Algorithmen entsprechend der jeweiligen Scheduling-Ziele, wie z.B.:

- Fairness: Kein Task sollte unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird.
- Maximaler Durchsatz: So viele Tasks pro Zeiteinheit wie möglich.
- Minimale Wartezeit: Möglichst geringe Zeit von der Übermittlung bis zur Ausführung eines Tasks.
- Ressourcen-Auslastung: Möglichst hohe Auslastung aller Ressourcen.
- Zuverlässigkeit: Ein Task wird garantiert ausgeführt.
- Geringe End-to-End Ausführungszeit (z.B. durch Daten-Lokalität und geringe Kommunikationskosten, syn. Makespan)

#### Cluster-Scheduling ist eine Optimierungsaufgabe



#### Scheduling ist eine Optimierungsaufgabe...



#### ... und ist NP-vollständig.

Die Optimierungsaufgabe lässt sich auf das Traveling Salesman Problem zurückführen.

#### Das bedeutet:

- Es ist kein Algorithmus bekannt, der eine optimale Lösung garantiert in Polynomialzeit erzeugt.
- Algorithmus muss für tausende Jobs und tausende Ressourcen skalieren. Optimale Algorithmen, die den Lösungsraum komplett durchsuchen sind nicht praktikabel, da deren Entscheidungszeit zu lange ist für große Eingabemengen (|Jobs| x |Ressourcen|).
- Praktikable Scheduling-Algorithmen sind somit Algorithmen zur n\u00e4herungsweisen L\u00f6sung des Optimierungsproblems (Heuristiken, Meta-Heuristiken).

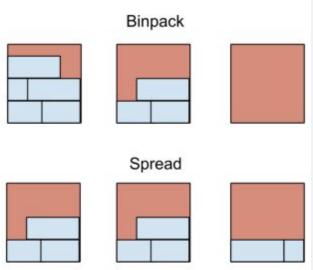
Darüber hinaus kommen Job-Anfragen kontinuierlich an, so dass selbst bei optimalem Algorithmus der Re-Organisationsaufwand pro Job unverhältnismäßig hoch werden kann.

#### **Einfache Scheduling-Algorithmen**



 Optimieren das Scheduling von Tasks oft in genau einer Dimension (z.B. CPU-Auslastung) bzw. wenigen Dimensionen (CPU-Auslastung und RAM).

- Populäre Algorithmen:
  - Binpack (Fit First)
  - Spread (Round Robin)





## Orchestrierung

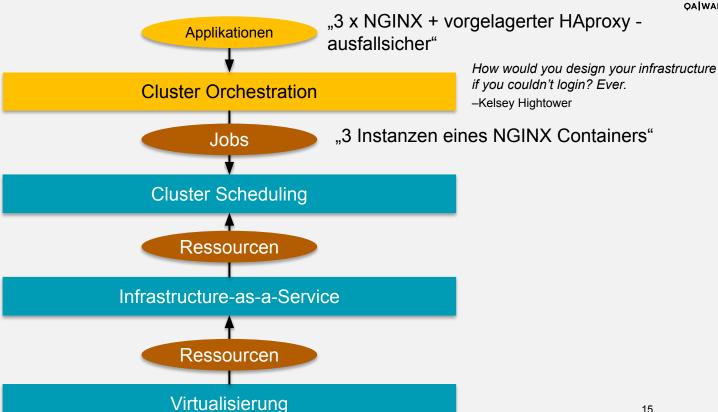
## Kubernetes engineers when they don't have a kitchen towel



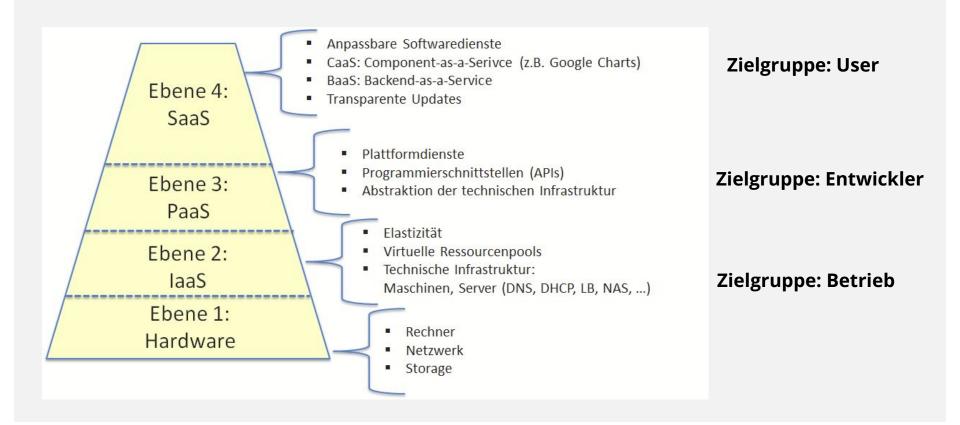


#### Das Big Picture: Wir sind nun auf Applikationsebene.

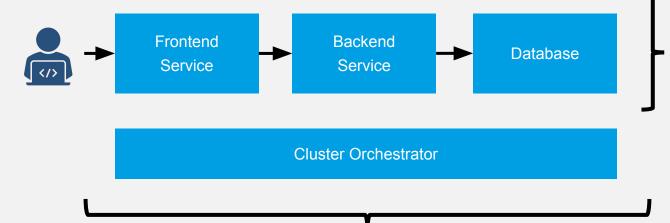




#### Das Big Picture: Wir sind nun auf Applikationsebene.



#### **Cluster-Orchestrierung: Beispiel**



Interessiert Dev

- Anwendungen
- Betriebssystem
- Dependencies

- Deployment
- Networking (Loadbalancing, Service Discovery, ...)
- Scaling
- "Insight" (Logging, Analytics, Rescheduling, Failure Recovery, …)



#### **Cluster-Orchestrierung**

- Ziel: Eine Anwendung, die in mehrere
   Betriebskomponenten (Container) aufgeteilt ist,
   auf mehreren Knoten laufen lassen.
- Führt Abstraktionen zur Ausführung von Anwendungen mit ihren Services in einem großen Cluster ein.
- Orchestrierung ist keine statische, einmalige Aktivität wie die Provisionierung, sondern eine dynamische, kontinuierliche Aktivität.
- Orchestrierung hat den Anspruch, alle Standard-Betriebsprozeduren einer Anwendung zu automatisieren.

Blaupause der Anwendung, die den gewünschten Betriebszustand der Anwendung beschreibt: Betriebskomponenten (Container), deren Betriebsanforderungen sowie die angebotenen und benötigten Schnittstellen.

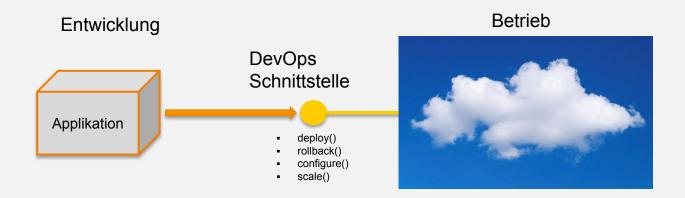


#### Steuerungsaktivitäten im Cluster:

- Start von Containern auf Knoten (→ Scheduler)
- Verknüpfung von Containern
- ..

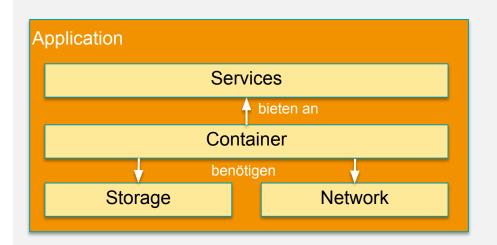
## Ein Cluster-Orchestrierer bietet eine Schnittstelle zwischen Betrieb und Entwicklung für ein Cluster an.

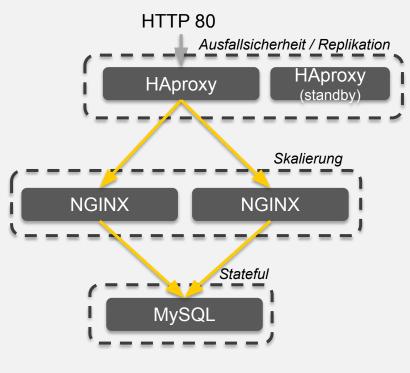




#### Blaupause einer Anwendung (vereinfacht)







Metamodell

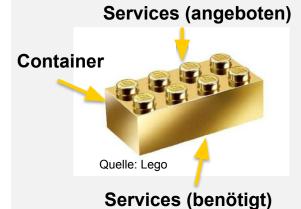
Modell

#### **Analogie 1: Lego Star Wars**

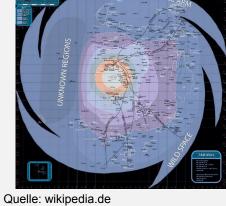


#### **Cluster-Orchestrierer**









**Cluster-Scheduler** 

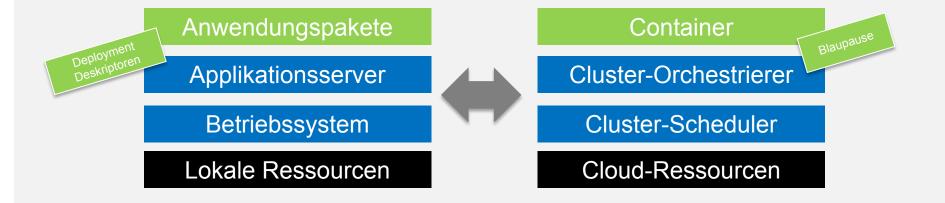
Quelle: Lego



Blaupause

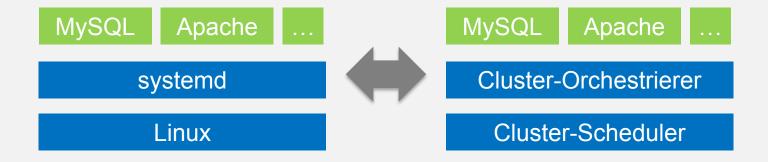
#### **Analogie 2: Applikationsserver**





#### **Analogie 3: Betriebssystem**





## Ein Cluster-Orchestrierer automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster (1 / 2):



- Scheduling von Containern mit applikationsspezifischen Constraints (z.B. Deployment- und Start-Reihenfolgen, Gruppierung, ...)
- Aufbau von notwendigen Netzwerk-Verbindungen zwischen Containern.
- Bereitstellung von persistenten Speichern für zustandsbehaftete Container.
- (Auto-)Skalierung von Containern.
- Re-Scheduling von Containern im Fehlerfall (Auto-Healing) oder zur Performance-Optimierung.
- Container-Logistik: Verwaltung und Bereitstellung von Containern.

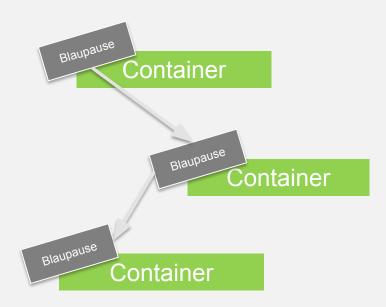
## Ein Cluster-Orchestrierer automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster (2 / 2):



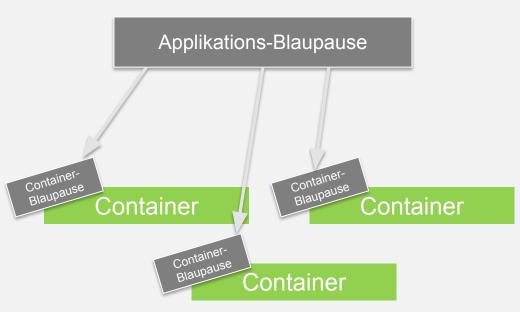
- Package-Management: Verwaltung und Bereitstellung von Applikationen.
- Bereitstellung von Administrationsschnittstellen (Remote-API, Kommandozeile).
- Management von Services: Service Discovery, Naming, Load Balancing.
- Automatismen für Rollout-Workflows wie z.B. Canary Rollout.
- Monitoring und Diagnose von Containern und Services.

#### 1-Level- vs. 2-Level-Orchestrierung





**1-Level-Orchestrierung** (Container-Graph)



#### 2-Level-Orchestrierung

(Container-Repository mit zentraler Bauanleitung)

#### 1-Level- vs. 2-Level-Orchestrierung

https://docs.docker.com/compose/compose-file weba: **QA** WARE image: qaware/nginx expose: FROM ubuntu - 80 ENTRYPOINT nginx EXPOSE 80 webb: image: qaware/nginx expose: - 80 haproxy: image: qaware/haproxy links: FROM ubuntu weba ENTRYPOINT haproxy webb EXPOSE 80 ports:

Plant
Docker
ENTRYPOINT nginx
EXPOSE 80

docker run -d --link
nginx:nginx

**1-Level-Orchestrierung** (Container-Graph)

#### 2-Level-Orchestrierung

- "80:80"

expose: - 80

(Container-Repository mit zentraler Bauanleitung)

#### **Kubernetes**

Josef Adersberger @adersberger · Jul 21

#### Google spares no effort to lauch

#kubernetes @ #OSCON







kubernetes Google

Manage a cluster of Linux containers as a single system to accelerate Dev and simplify Ops.

#### **Kubernetes**



- Cluster-Orchestrierer auf Basis von Containern, der eine Reihe an Kern-Abstraktionen für den Betrieb von Anwendungen in einem großen Cluster einführt. Die Blaupause wird über YAML-Dateien definiert.
- Open-Source-Projekt, das von Google initiiert wurde. Google will damit die jahrelange Erfahrung im Betrieb großer Cluster der Öffentlichkeit zugänglich machen und damit auch Synergien mit dem eigenen Cloud-Geschäft heben.
- Seit Juli 2015 in der Version 1.0 verfügbar und damit produktionsreif. Skaliert aktuell nachweislich auf 1000 Nodes großen Clustern.
- **Bei vielen Firmen im Einsatz** wie z.B. Google im Rahmen der Google Container Engine, Wikipedia, ebay. Beiträge an der Codebasis aus vielen Firmen neben Google u.A. Mesosphere, Microsoft, Pivotal, RedHat.
- Setzt den Standard im Bereich Cluster-Orchestrierung. Dafür wurde auch eigens die Cloud Native Computing Foundation gegründet (https://cncf.io).

#### **Architektur von Kubernetes**

























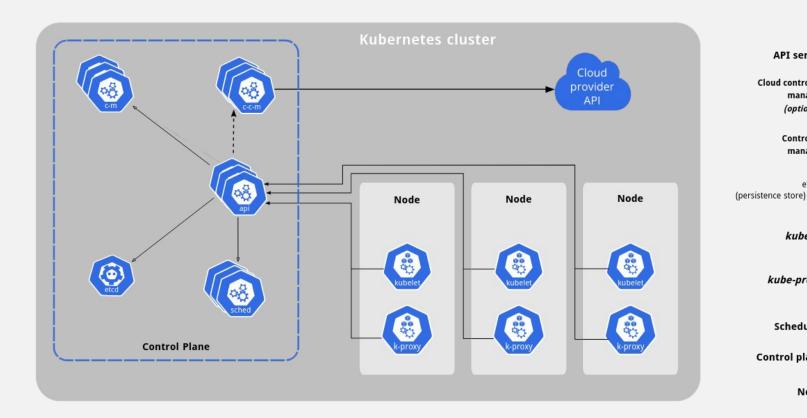






Node







### **Pods & Deployments**



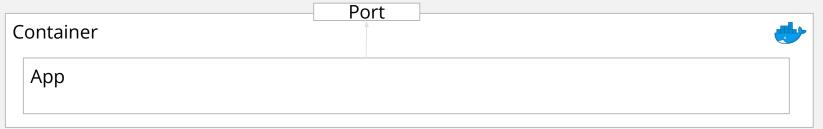
Der Grundbaustein ist eure **Anwendung**.

App



Der Grundbaustein ist eure Anwendung.

Die Anwendung steckt in einem Container (siehe Vorlesung "Virtualisierung"). Container öffnen Ports nach außen.

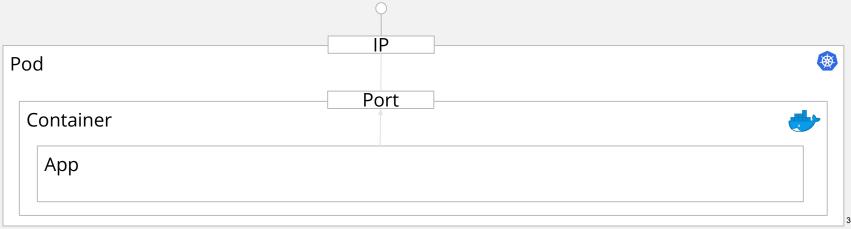




Der Grundbaustein ist eure Anwendung.

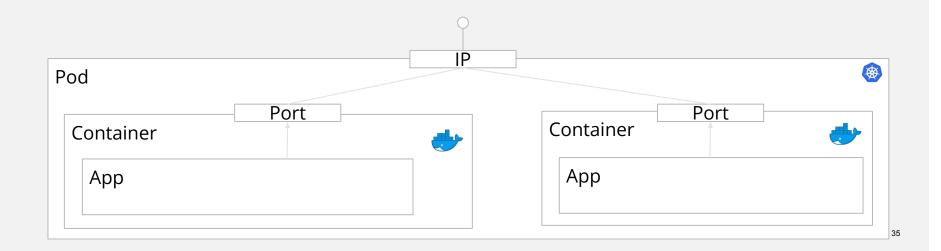
Die Anwendung steckt in einem Container (siehe Vorlesung "Virtualisierung"). Container öffnen Ports nach außen.

Container werden in Kubernetes zu Pods zusammengefasst. Pods haben nach außen hin eine IP-Adresse.

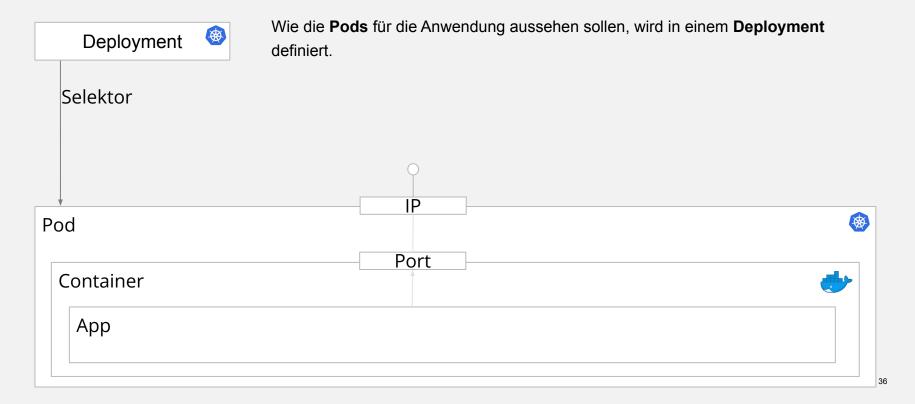




In einem **Pod** können auch mehrere **Container** laufen.

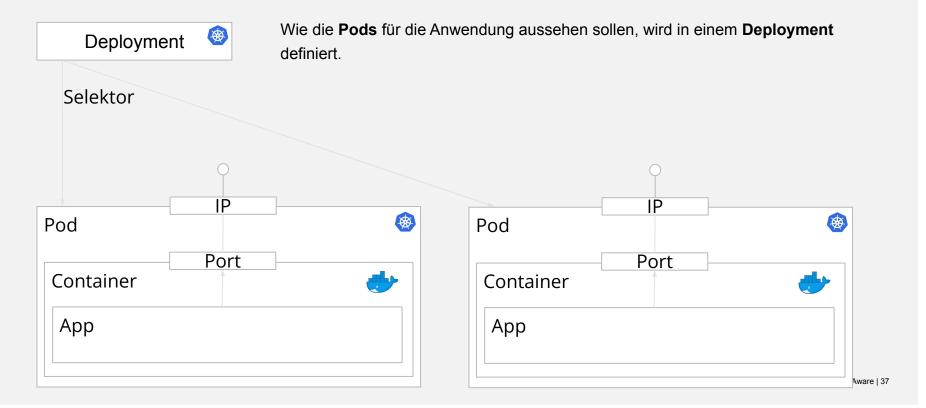






## Wichtige Kubernetes-Konzepte





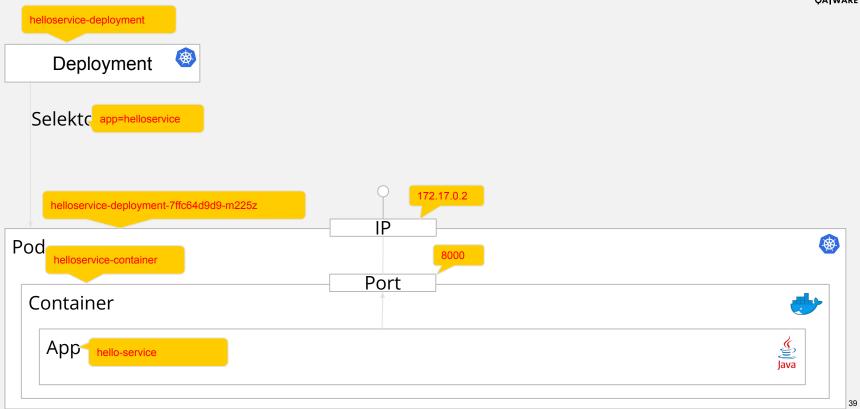
## **Deployment: Definition**



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-service
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloservice
    spec:
      containers:
      - name: hello-service
        image: "hitchhikersguide/zwitscher-service:1.0.1"
        ports:
        - containerPort: 8000
        env:
        - name:
          value: zwitscher-consul
```

## **Big Picture: Hello-Service**







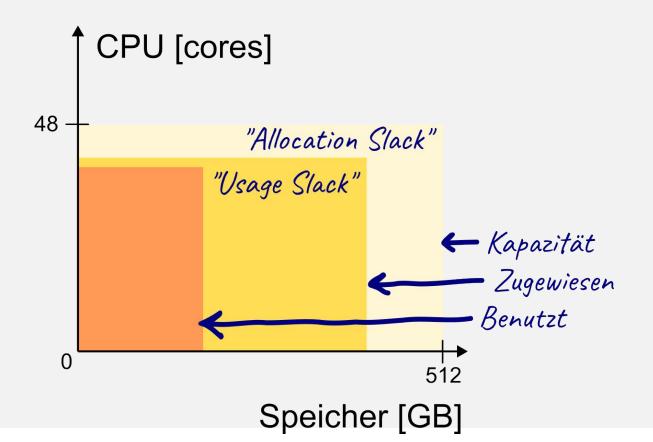
## **Probes & Resources**

#### **Resource Constraints**



```
resources:
 # Define resources to help K8S scheduler
 # CPU is specified in units of cores
 # Memory is specified in units of bytes
 # required resources for a Pod to be started
  requests:
   memory: "128M"
   cpu: "0.25"
 # the Pod will be (OOM-)killed or CPU-throttled
 # if limits are exceeded
 limits:
   memory: "192M"
   cpu: "0.5"
```

# Hauptziel ist es oft, die Ressourcen-Auslastung zu optimieren. Das spart Opportunitätskosten

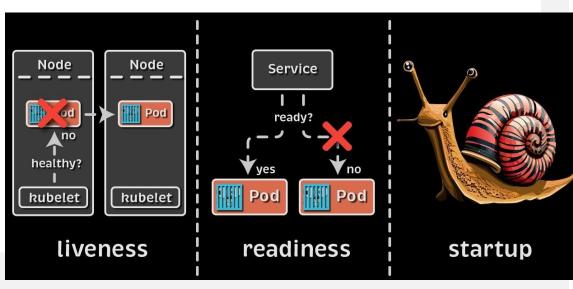


#### **Liveness und Readiness Probes**



```
# container will receive requests if probe succeeds
readinessProbe:
  httpGet:
    path: /admin/info
   port: 8080
  initialDelaySeconds: 30
  timeoutSeconds: 5
# container will be killed
# if probe fails
livenessProbe:
  httpGet:
   path: /admin/health
   port: 8080
  initialDelaySeconds: 90
```

timeoutSeconds: 10

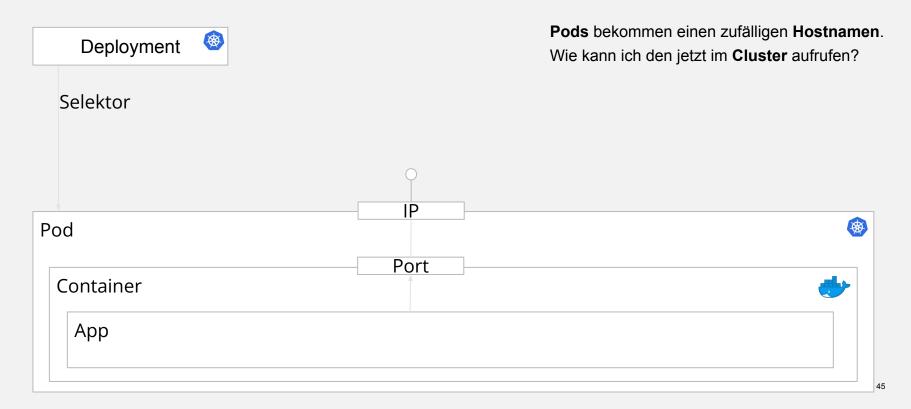




## **Services**

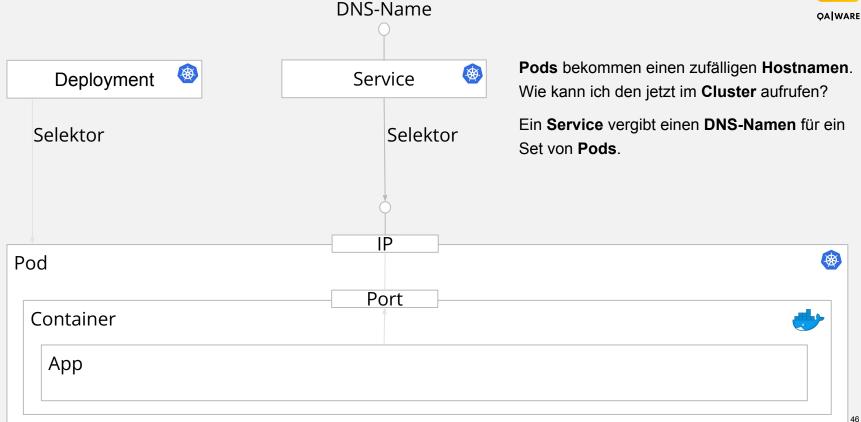
## Wichtige Kubernetes Konzepte





## Wichtige Kubernetes Konzepte

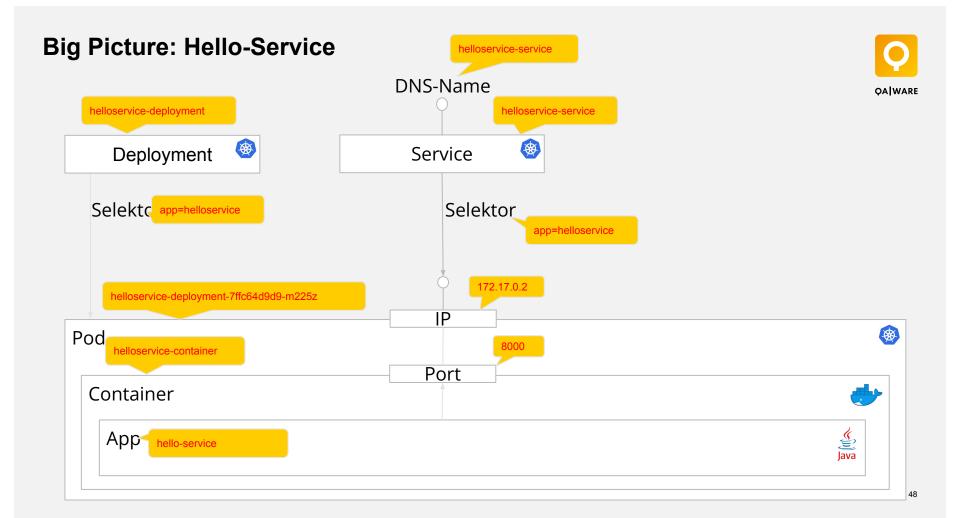




#### **Service: Definition**



```
apiVersion: \vee 1
kind: Service
metadata:
  name: hello-service
  labels:
    app: helloservice
spec:
  # use NodePort here to be able to access the port on each node
  # use LoadBalancer for external load-balanced IP if supported
  type: NodePort
  ports:
  - port: 8080
  selector:
    app: helloservice
```

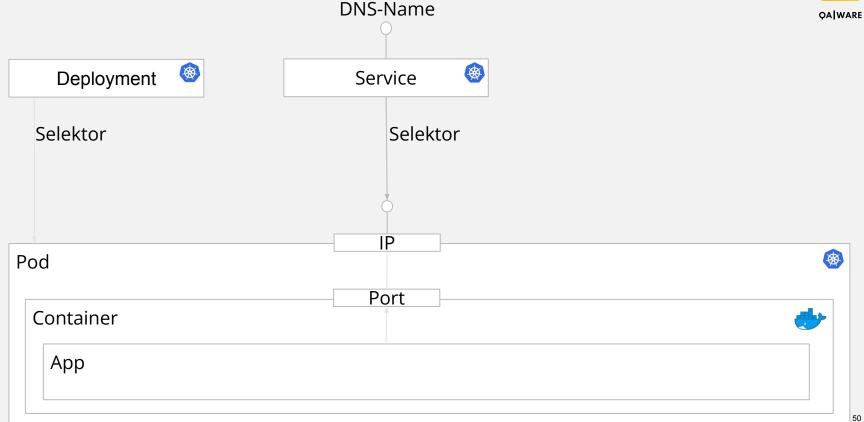




## **Config Maps**

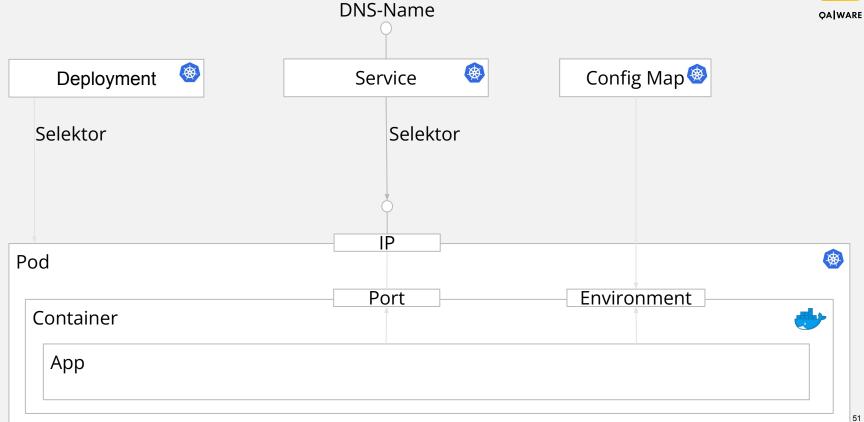
## Wichtige Kubernetes-Konzepte





## Wichtige Kubernetes-Konzepte





## **Konfiguration: Config Maps (1)**



```
apiVersion: v1
kind: ConfigMap
metadata:
    name: game-demo
data:
    # property-like keys; each key maps to a simple value
    player_initial_lives: "3"
    ui_properties_file_name: "user-interface.properties"
```

## **Konfiguration: Config Maps (2)**

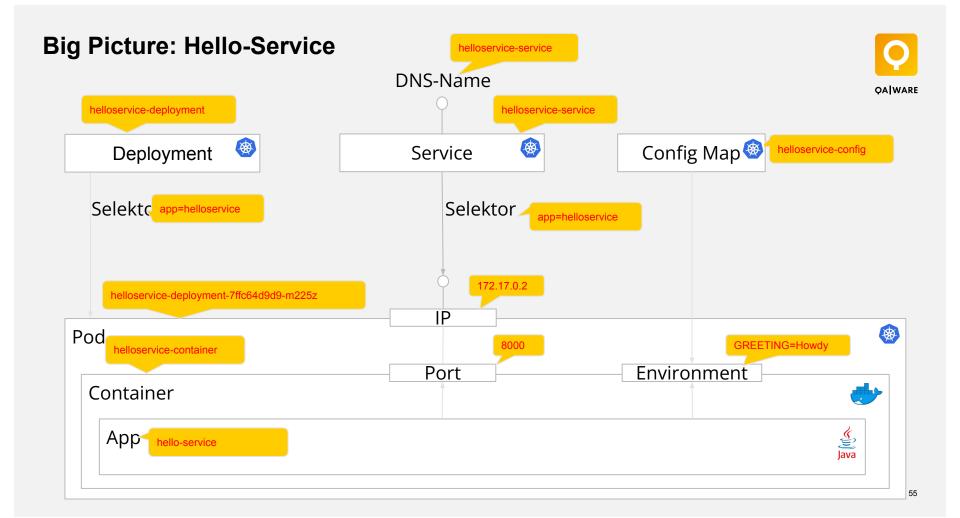


```
apiVersion: ∨1
kind: Pod
metadata:
 name: configmap-demo-pod
spec:
 containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
         valueFrom:
                                       # from the key name in the ConfigMap.
           configMapKeyRef:
              name: game-demo # The ConfigMap this value comes from.
              key: player_initial_lives # The key to fetch.
        - name: UI PROPERTIES FILE NAME
         valueFrom:
           configMapKeyRef:
              name: game-demo
              key: ui_properties_file_name
```

## **Konfiguration: Config Maps (2)**



```
apiVersion: ∨1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      # Import all environment variables from ConfigMap.
      # Keys should already be in screaming snake case.
      envFrom:
        - configMapRef:
            name: game-demo
```



#### **Deklarativer Ansatz mit Kustomize**



Normales Vorgehen:

```
$ kubectl apply -f deployment.yaml
$ kubectl apply -f service.yaml
$ kubectl apply -f configmap.yaml
```

- Für eine Sammlung von vielen Dateien nicht geeignet
- Deshalb: Die yaml-Dateien mit kustomize gruppieren!

```
$ kubectl apply -k kustomization.yaml
```

# Kustomize Wenn man nicht jede Datei einzeln ins Cluster schieben will



#### commonLabels:

app: pizza-service

namespace: pizzeria

#### resources:

- deployment.yaml
- service.yaml
- pod-disruption-budget.yaml

#### configMapGenerator:

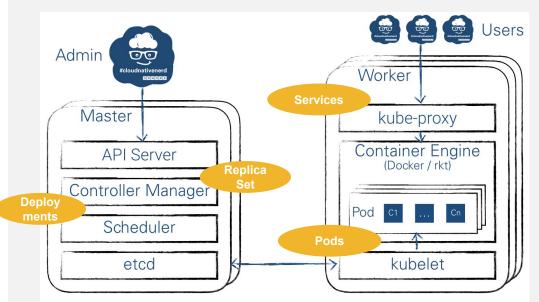
- name: pizza-config
   behavior: create
   literals:
  - TOPPING="garlic"

#### patchesStrategicMerge:

deployment-patch.yaml

### Aufgaben der Kubernetes-Bausteine.





API Server: Stellt die REST API von Kubernetes zur Verfügung (Admin-Schnittstelle)

Controller Manager: Verwaltet die Replica Sets / Replication Controller (stellt Anzahl Instanzen sicher) und Node Controller (prüfen Maschine & Pods)

Scheduler: Cluster-Scheduler.

etcd: Stellt einen zentralen Konfigurationsspeicher zur Verfügung.

Kubelet: Führt Pods aus.

Container Engine: Betriebssystem-Virtualisierung.

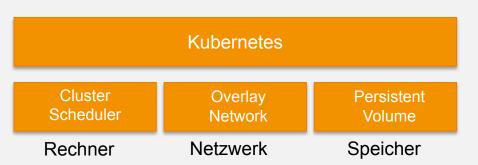
kube-proxy: Stellt einen Service nach Außen zur Verfügung.

# Neben einem Cluster-Scheduler setzt Kubernetes auch noch auf Netzwerk- und Storage-Virtualisierungen auf.



Netzwerk-Virtualisierung (Overlay Network)

- OpenVSwitch
- Flannel
- Weave
- Calico



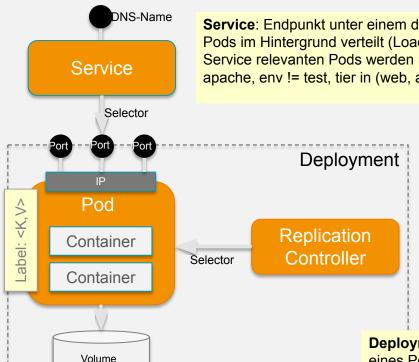
https://kubernetes.io/docs/concepts/cluster-administration/networking/

Storage-Virtualisierung (Persistent Volume), insbesondere zur Behandlung von zustandsbehafteten Containern.

- GCE / AWS Block Store
- NFS
- iSCSI
- Ceph
- GlusterFS

#### Die Kern-Abstraktionen von Kubernetes.





**Service**: Endpunkt unter einem definierten DNS-Namen, der Aufrufe an die Pods im Hintergrund verteilt (Load Balancing, Failover). Die für einen Service relevanten Pods werden über ihre Labels selektiert, z.B.: role = apache, env != test, tier in (web, app)

**Pod**: Gruppe an Containern, die auf dem selben Knoten laufen und sich eine Netzwerk-Schnittstelle inklusive einer dedizierten IP, persistente Volumes und Umgebungsvariablen teilen. Ein Pod ist die atomare Scheduling-Einheit in K8s. Ein Pod kann über sog. *Labels* markiert werden, das sind frei definierbare Schlüssel-Wert-Paare.

**Replication Controller**: stellen sicher, dass eine spezifizierte Anzahl an Instanzen pro Pod ständig läuft. Ist für Reaktionen im Fehlerfall (Re-Scheduling), Skalierung und Rollouts (Canary Rollouts, Rollout Tracks, ..) zuständig.

**Deployment**: Klammer um einen gewünschten Zielzustand im Cluster in Form eines Pods mit dazugehörigem Replication Controller. Ein Deployment bezieht sich nicht auf Services, da diese in der K8s-Philosophie einen von Pods unabhängigen Lebenszyklus haben.

### Weitere Konzepte von Kubernetes



#### Anwendungen

- **StatefulSet:** Wenn eine Anwendung doch einen Zustand braucht
- Persistent Volumes: Zugriff auf persistenten Speicher

#### Global

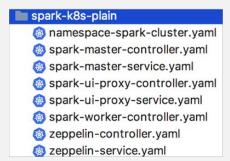
- DaemonSet: Wenn ein Dienst auf jedem Node im Cluster laufen muss
- Job: Wenn eine Aufgabe regelmäßig ausgeführt werden muss

#### Security

• Network Policies: Wer darf mit wem kommunizieren?

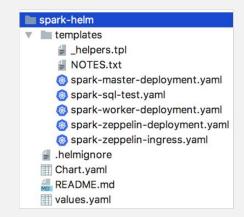
## Helm: Verwaltung von Applikationspaketen für Kubernetes.





- kubectl, kubectl, kubectl, ...
- Konfiguration?
- Endpunkte?

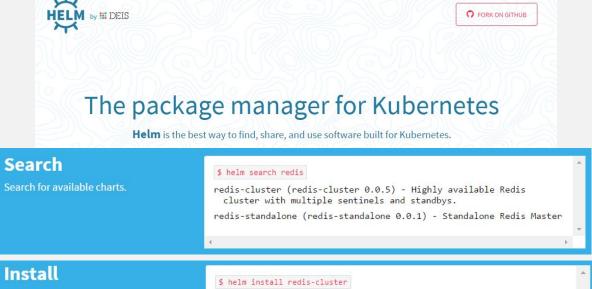




- Chart suchen auf <a href="https://hub.kubeapps.com">https://hub.kubeapps.com</a>
- Doku dort lesen (README.md)
- Konfigurationsparameter lesen:
   helm inspect stable/spark
- Chart starten mit überschriebener Konfiguration:
   helm install --name my-release
   -f values.yaml stable/spark

## Helm: Verwaltung von Applikationspaketen für Kubernetes.





Deploy the chart to your kubernetes

\$ helm install redis-cluster
---> Running `kubectl create -f` ...
services/redis-sentinel
pods/redis-master
replicationcontrollers/redis
replicationcontrollers/redis-sentinel
---> Done

Quelle: https://github.com/helm/helm



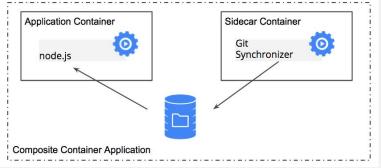
# Orchestrierungsmuster

## Orchestrierungsmuster – Separation of Concerns mit modularen

Q

OAIWARE

Containern



PHP app

redis proxy

Redis Shards

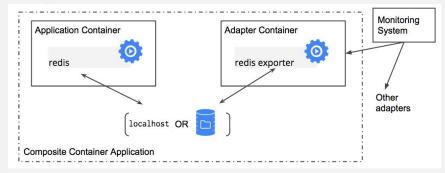
localhost

Composite Container Application

Ambassador Container

Sidecar Container

**Ambassador Container** 



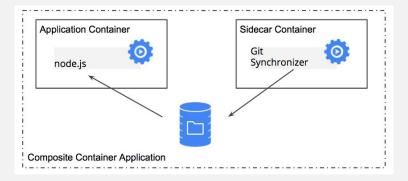
**Application Container** 

**Adapter Container** 

#### **Sidecar Containers**



Sidecar containers extend and enhance the "main" container, they take existing containers and make them better. As an example, consider a container that runs the Nginx web server. Add a different container that syncs the file system with a git repository, share the file system between the containers and you have built Git push-to-deploy. But you've done it in a modular manner where the git synchronizer can be built by a different team, and can be reused across many different web servers (Apache, Python, Tomcat, etc). Because of this modularity, you only have to write and test your git synchronizer once and reuse it across numerous apps. And if someone else writes it, you don't even need to do that.

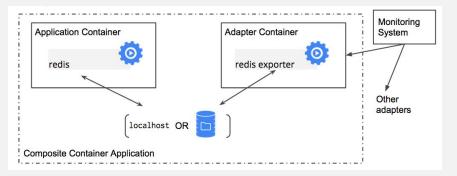


Quelle: <a href="https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/">https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/</a>

### Adapter containers



Adapter containers standardize and normalize output. Consider the task of monitoring N different applications. Each application may be built with a different way of exporting monitoring data. (e.g. JMX, StatsD, application specific statistics) but every monitoring system expects a consistent and uniform data model for the monitoring data it collects. By using the adapter pattern of composite containers, you can transform the heterogeneous monitoring data from different systems into a single unified representation by creating Pods that groups the application containers with adapters that know how to do the transformation. Again because these Pods share namespaces and file systems, the coordination of these two containers is simple and straightforward.

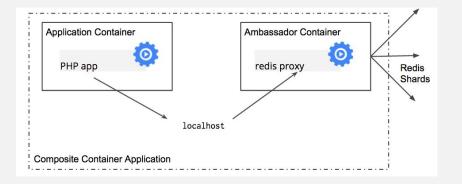


Quelle: https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/

#### **Ambassador containers**



Ambassador containers proxy a local connection to the world. As an example, consider a Redis cluster with read-replicas and a single write master. You can create a Pod that groups your main application with a Redis ambassador container. The ambassador is a proxy is responsible for splitting reads and writes and sending them on to the appropriate servers. Because these two containers share a network namespace, they share an IP address and your application can open a connection on "localhost" and find the proxy without any service discovery. As far as your main application is concerned, it is simply connecting to a Redis server on localhost. This is powerful, not just because of separation of concerns and the fact that different teams can easily own the components, but also because in the development environment, you can simply skip the proxy and connect directly to a Redis server that is running on localhost.



Quelle: <a href="https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/">https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/</a>

#### **Service Mesh**

QAIWARE

**Control Plane**: Konfiguration und Network Management

**Data Plane**: Traffic Management

Service Meshes beinhalten viel Funktionalität:

- Advanced Traffic Management (L4/L7)
- Sicherheit: mTLS, RBAC, Cert Mgmt
- "Deep" Observability

#### ... aber auch einige Probleme:

- Zusätzlichen Ressourcenverbrauch; Latenz
- Neue APIs, Logging/Security Konzepte
- Konfiguration ^ 2

