



Cluster-Orchestrierung

Simon Bäumlér

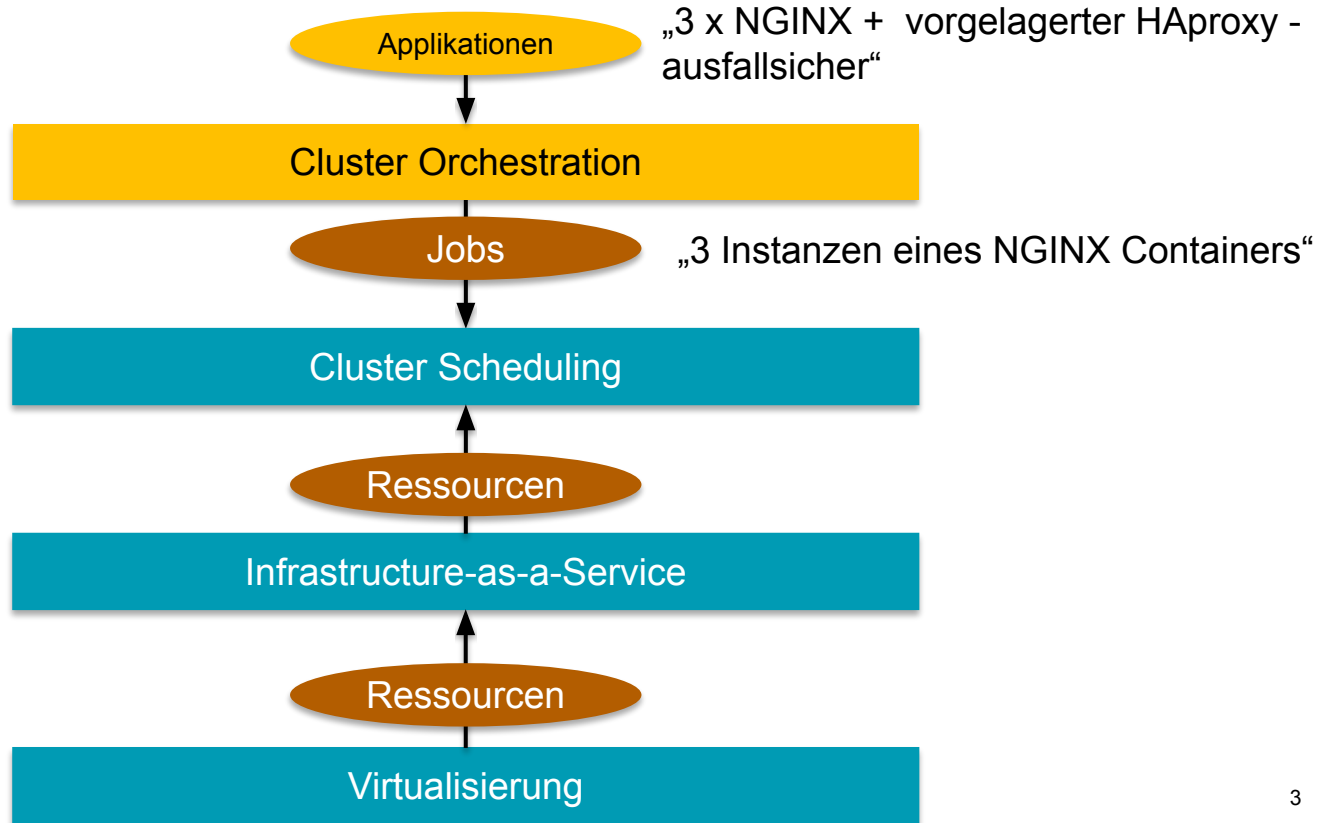
simon.baeumler@qaware.de

Kubernetes engineers when they don't have a kitchen towel

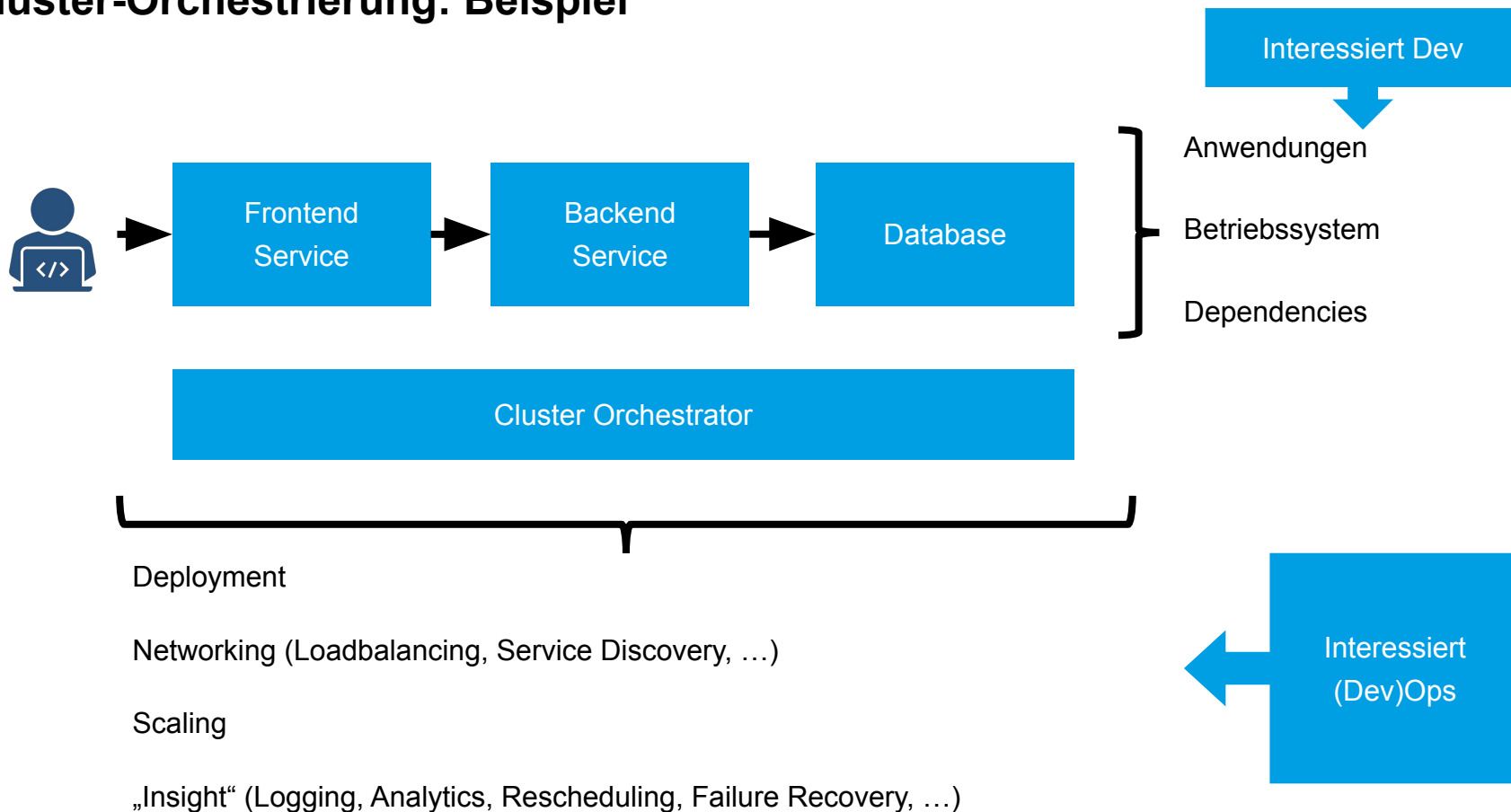


QA|WARE

Das Big Picture: Wir sind nun auf Applikationsebene.



Cluster-Orchestrierung: Beispiel



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.



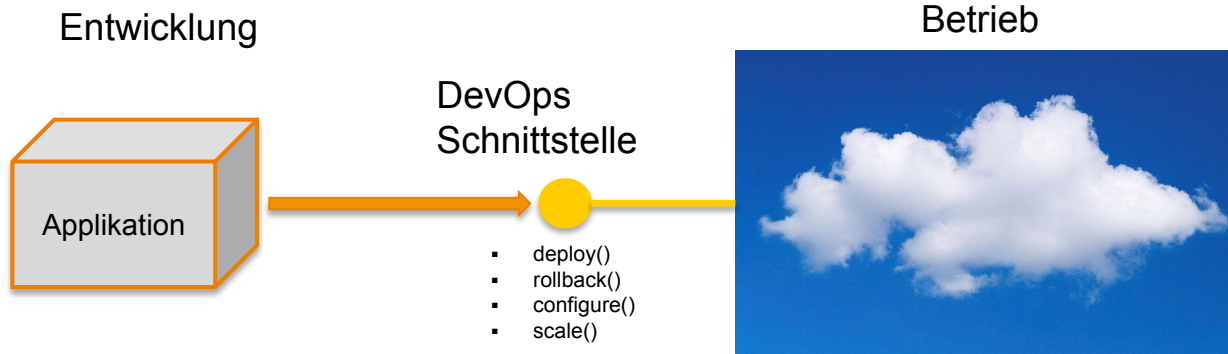
Cluster-Orchestrierer



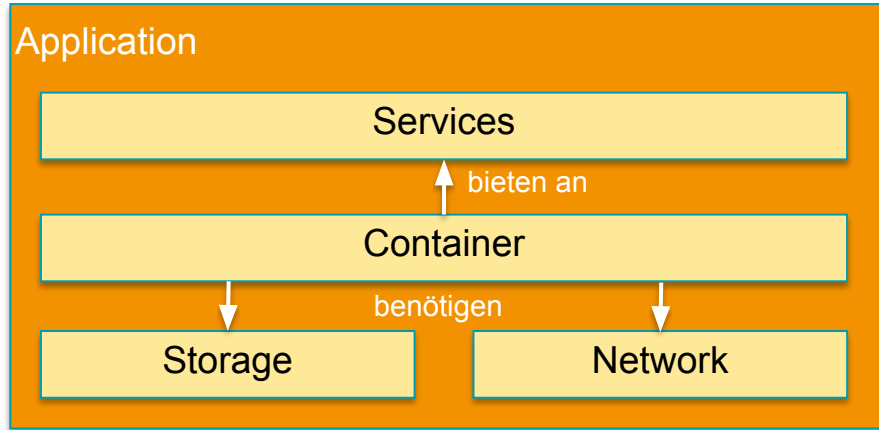
Steuerungsaktivitäten im Cluster:

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

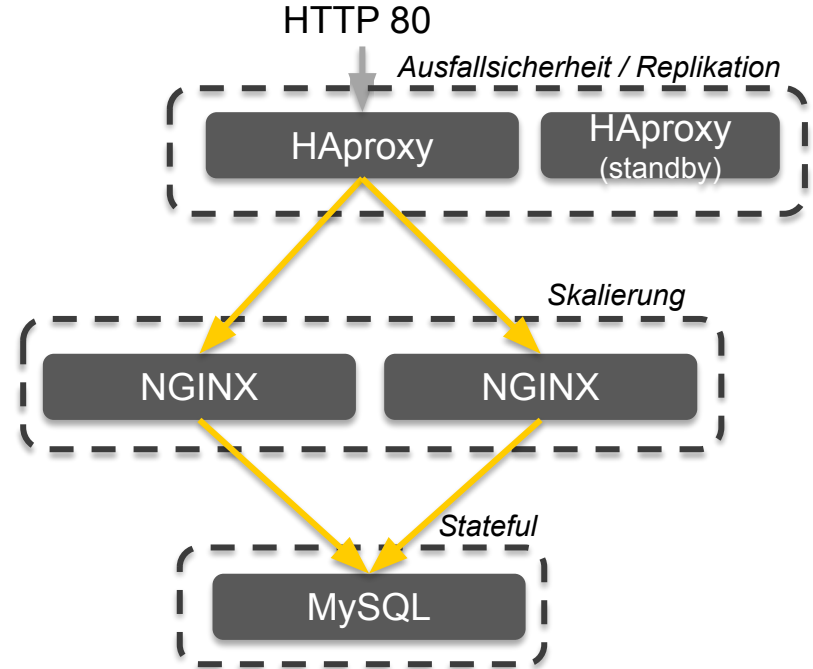
Ein Cluster-Orchestrierer bietet eine Schnittstelle zwischen Betrieb und Entwicklung für ein Cluster an.



Blaupause einer Anwendung (vereinfacht)

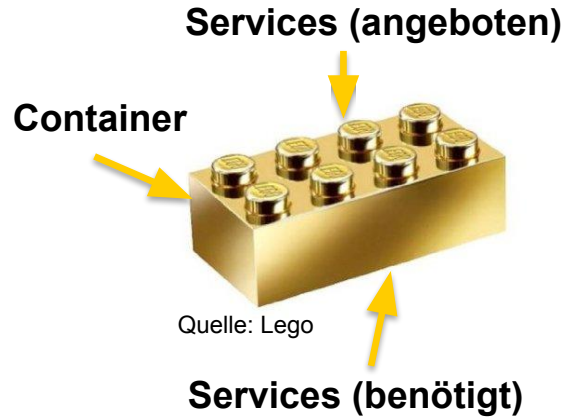


Metamodell

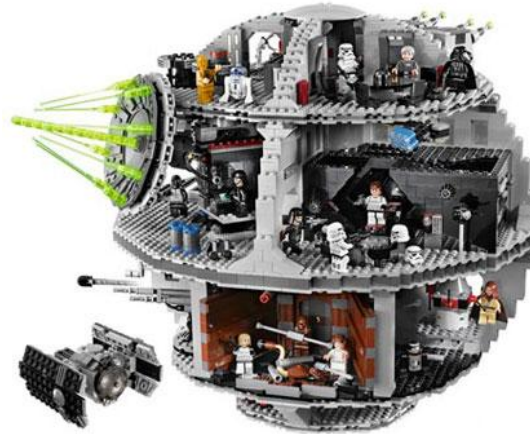


Modell

Analogie 1: Lego Star Wars



Cluster-Orchestrierer

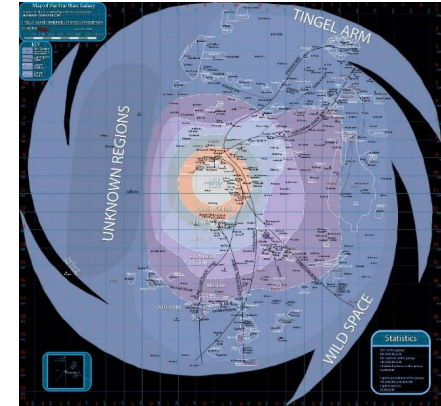


Quelle: Lego



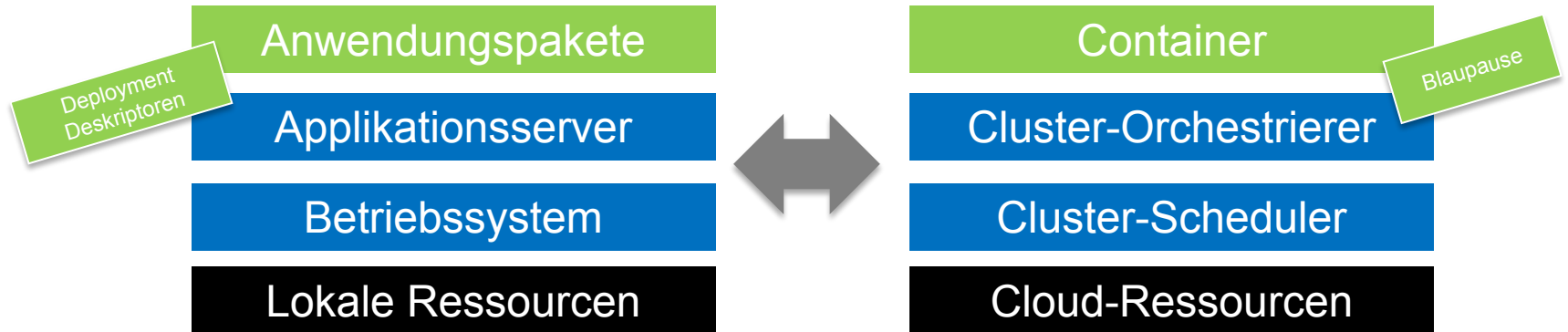
Blaupause

Cluster-Scheduler

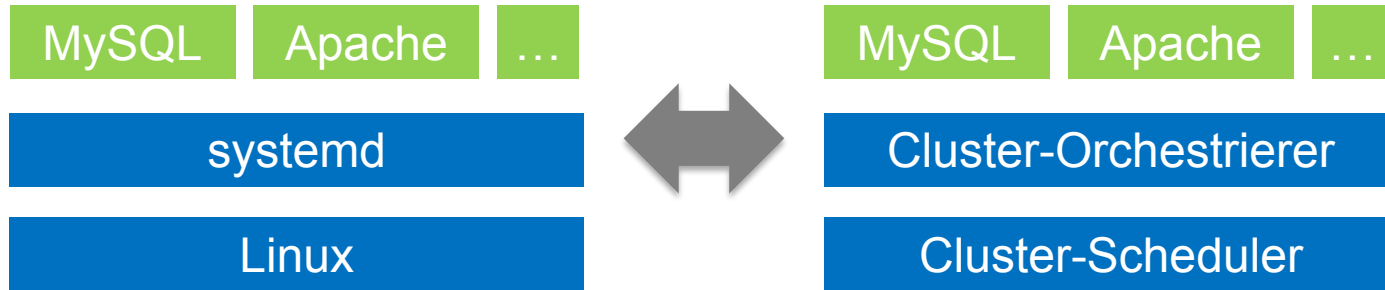


Quelle: wikipedia.de

Analogie 1: Applikationsserver



Analogie 2: Betriebssystem



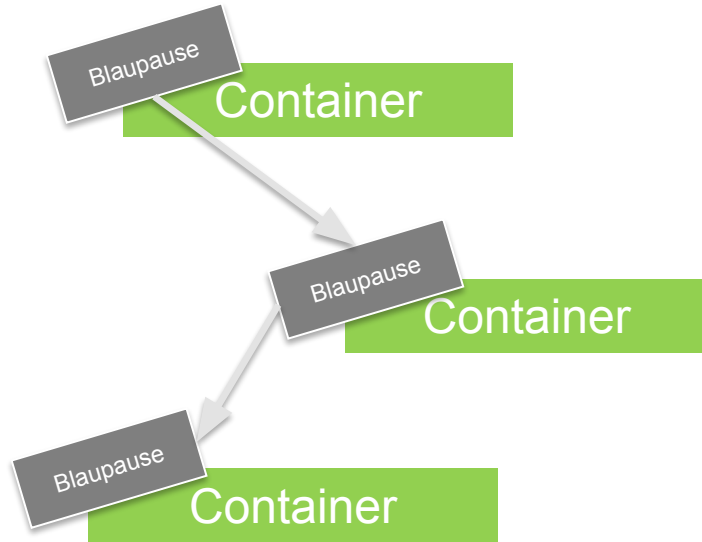
Ein Cluster-Orchestrierer automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster (1 / 2):

- 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.

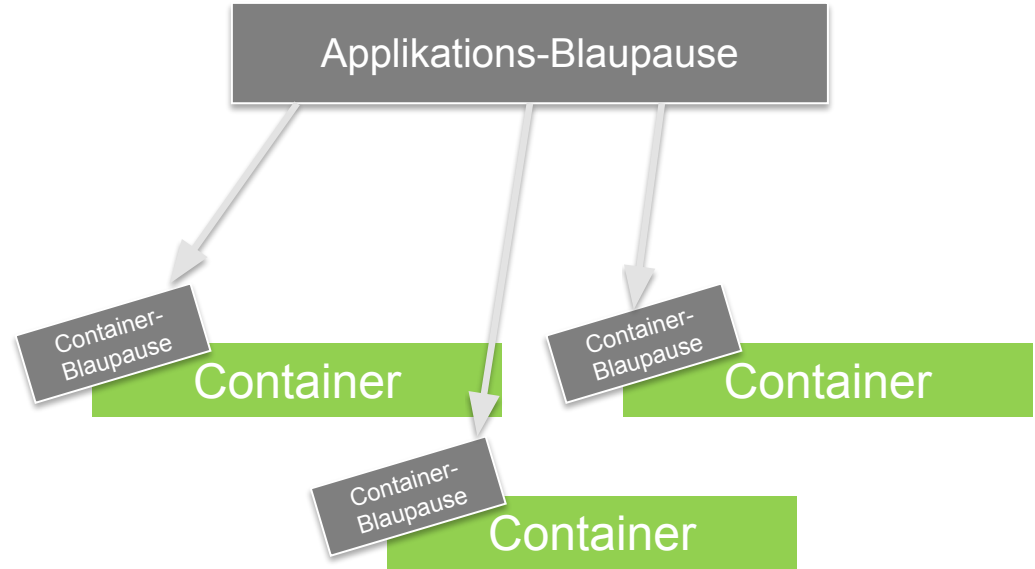
Ein Cluster-Orchestrierer automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster (2 / 2):

- 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

Plain Docker

```
FROM ubuntu
ENTRYPOINT nginx
EXPOSE 80
```

```
docker run -d --link
nginx:nginx
```

1-Level-Orchestrierung
(Container-Graph)

Docker Compose

<https://docs.docker.com/compose/compose-file>

weba:

```
image: qaware/nginx
expose:
  - 80
```

webb:

```
image: qaware/nginx
expose:
  - 80
```

haproxy:

```
image: qaware/haproxy
links:
  - weba
  - webb
ports:
  - „80:80“
expose:
  - 80
```

FROM ubuntu
ENTRYPOINT nginx
EXPOSE 80

FROM ubuntu
ENTRYPOINT haproxy
EXPOSE 80

2-Level-Orchestrierung

(Container-Repository mit zentraler Bauanleitung)

Kubernetes



kubernetes by Google

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

Josef Adersberger @adersberger · Jul 21

Google spares no effort to launch
#kubernetes @ #OSCON

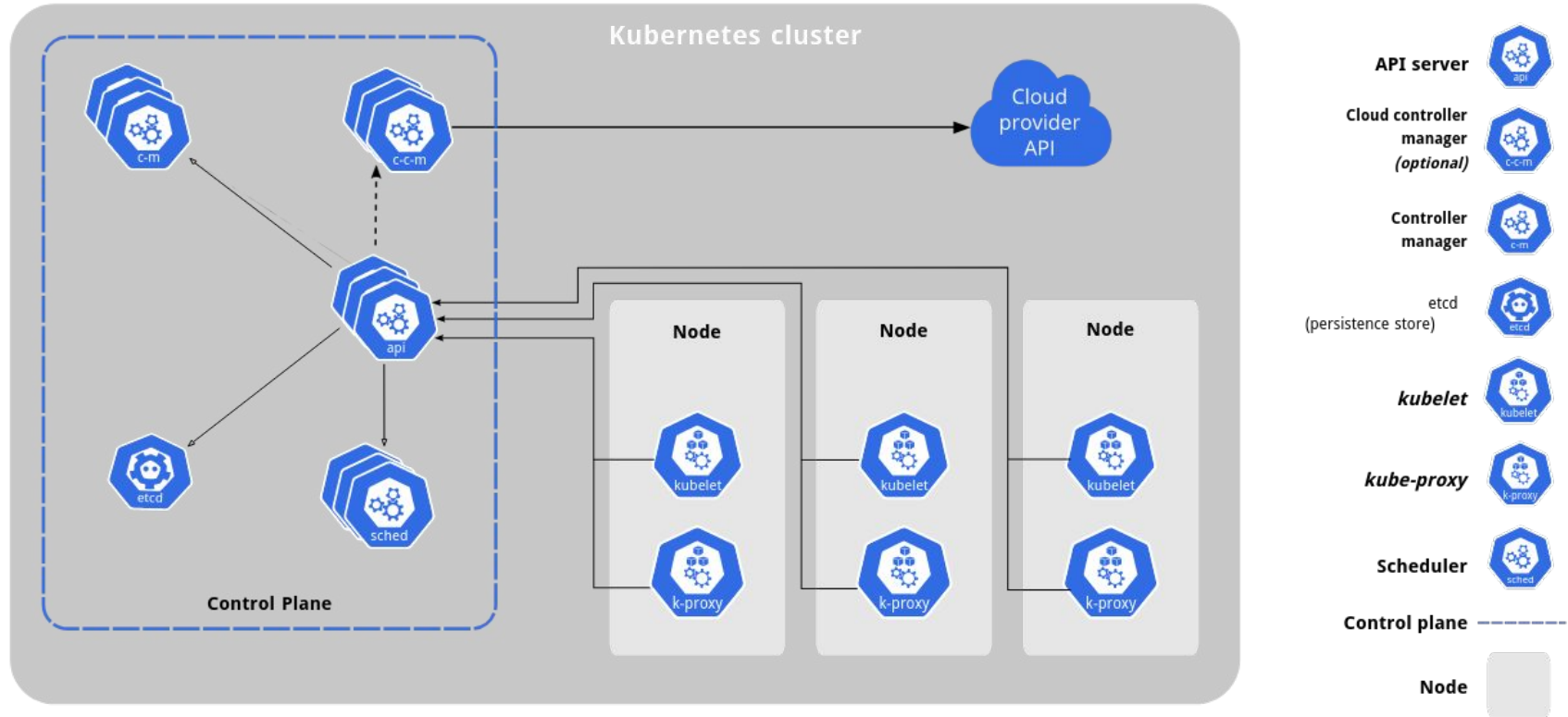
2015



Kubernetes

- Cluster-Orchestrierer auf Basis von Docker-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





QA|WARE

Pods & Deployments

Wichtige Kubernetes-Konzepte



QA|WARE

Der Grundbaustein ist eure **Anwendung**.

App

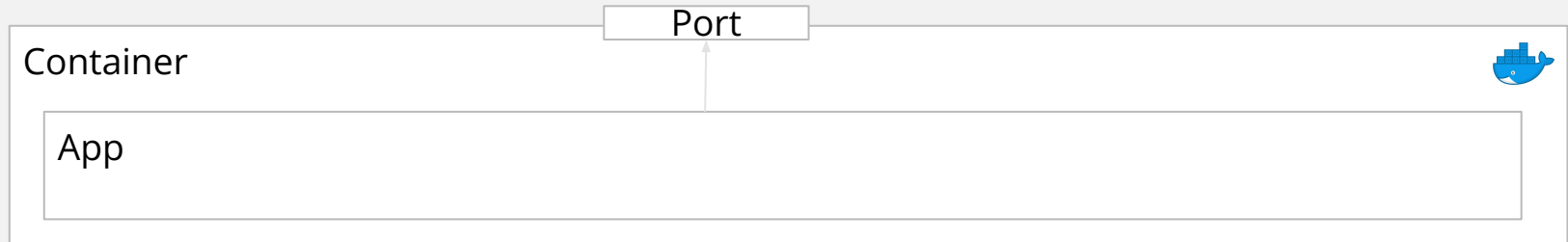
Wichtige Kubernetes-Konzepte



QA|WARE

Der Grundbaustein ist eure **Anwendung**.

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



Wichtige Kubernetes-Konzepte

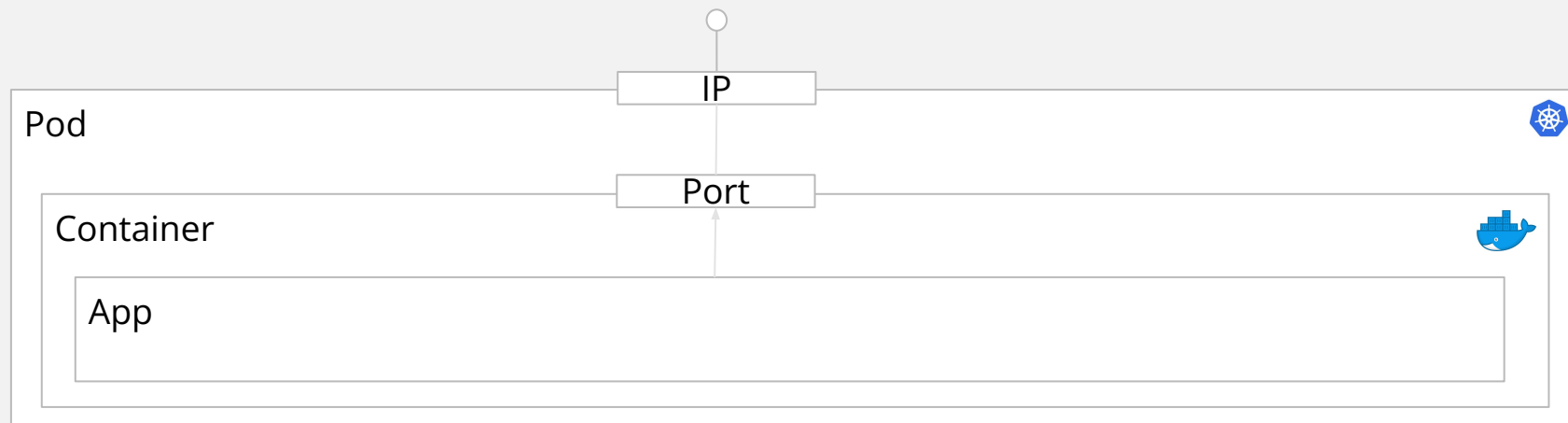


QA|WARE

Der Grundbaustein ist eure **Anwendung**.

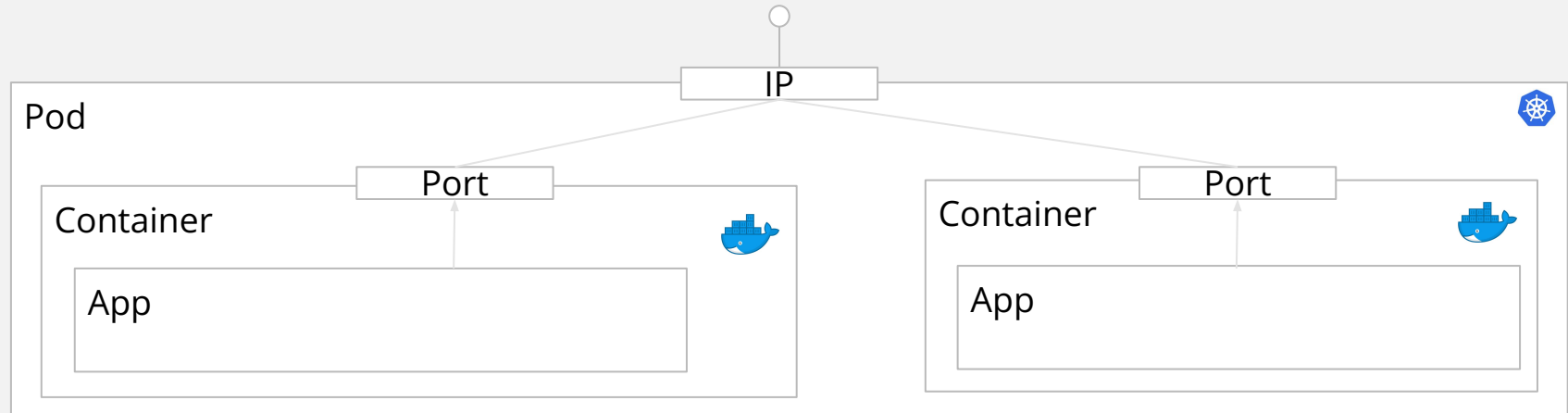
Die **Anwendung** steckt in einem **Container** (siehe Vorlesung “Virtualisierung”). **Container** öffnen **Ports** nach außen.

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



Wichtige Kubernetes-Konzepte

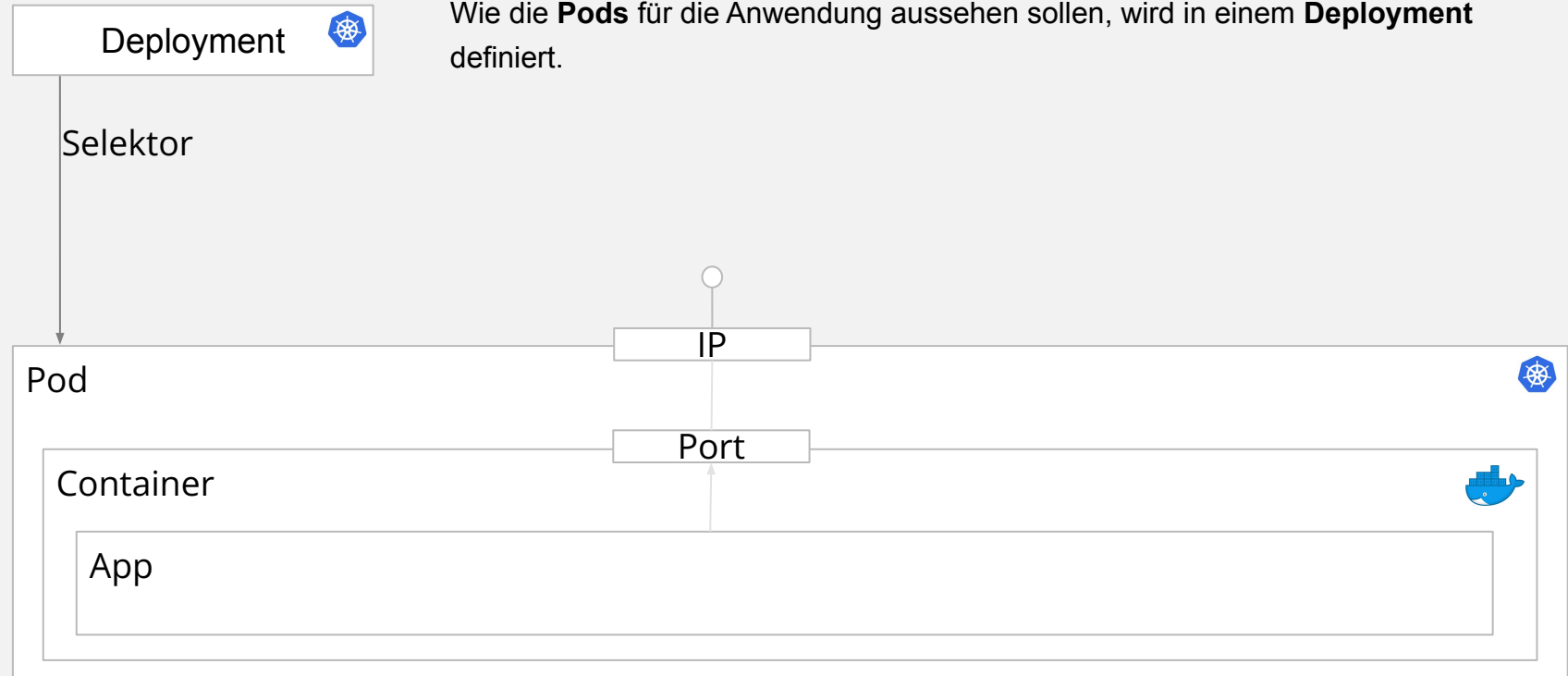
In einem **Pod** können auch mehrere **Container** laufen.



Wichtige Kubernetes-Konzepte



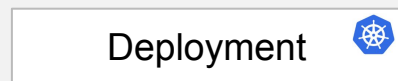
QA|WARE



Wichtige Kubernetes-Konzepte

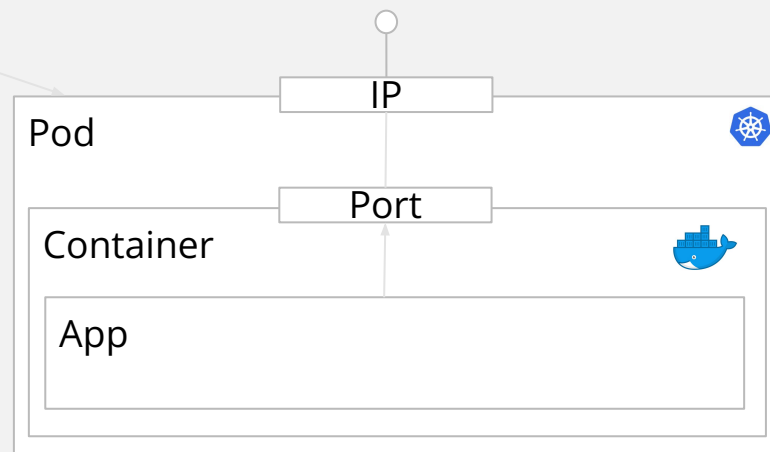
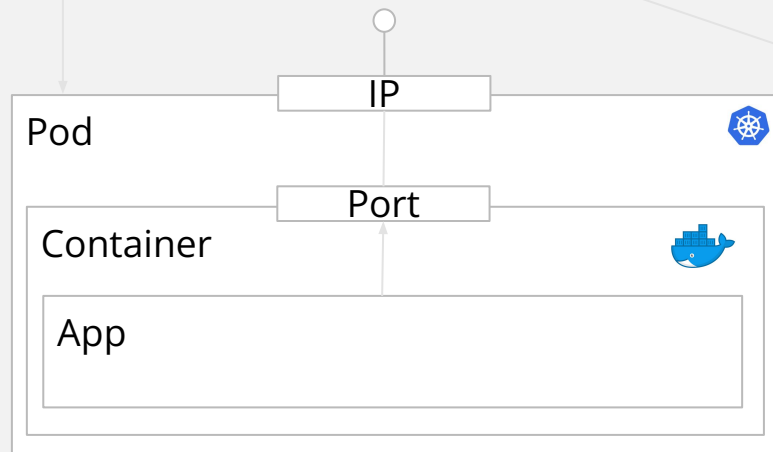


QA|WARE



Wie die **Pods** für die Anwendung aussehen sollen, wird in einem **Deployment** definiert.

Selektor



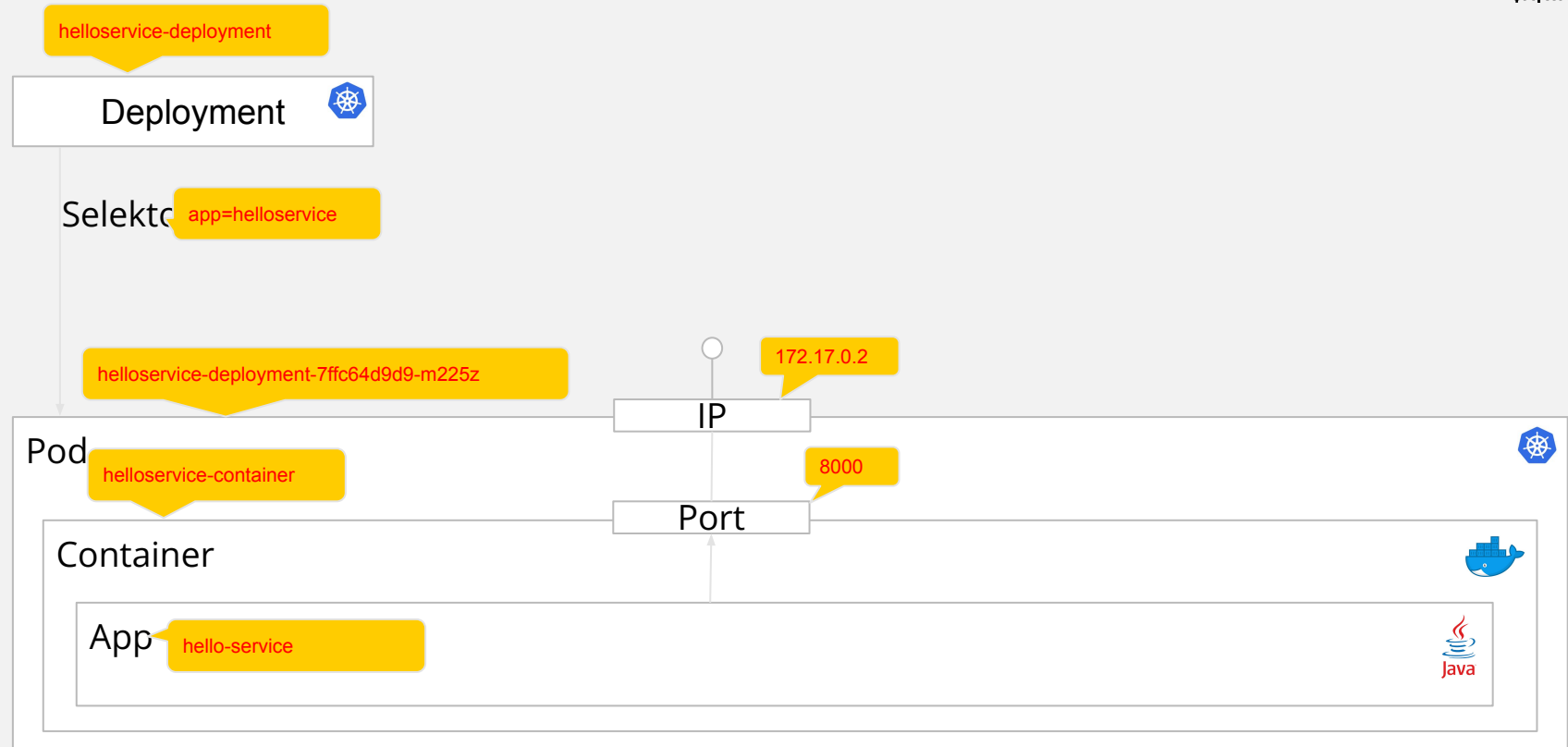
Deployment: Definition



QA|WARE

```
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: 8080
          env:
            - name:
              value: zwitscher-consul
```

Big Picture: Hello-Service





QA|WARE

Probes & Resources

Resource Constraints



Q|WARE

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 restarted if limits are exceeded

limits:

memory: "192M"

cpu: "0.5"

Liveness und Readiness Probes



QA|WARE

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



QA|WARE

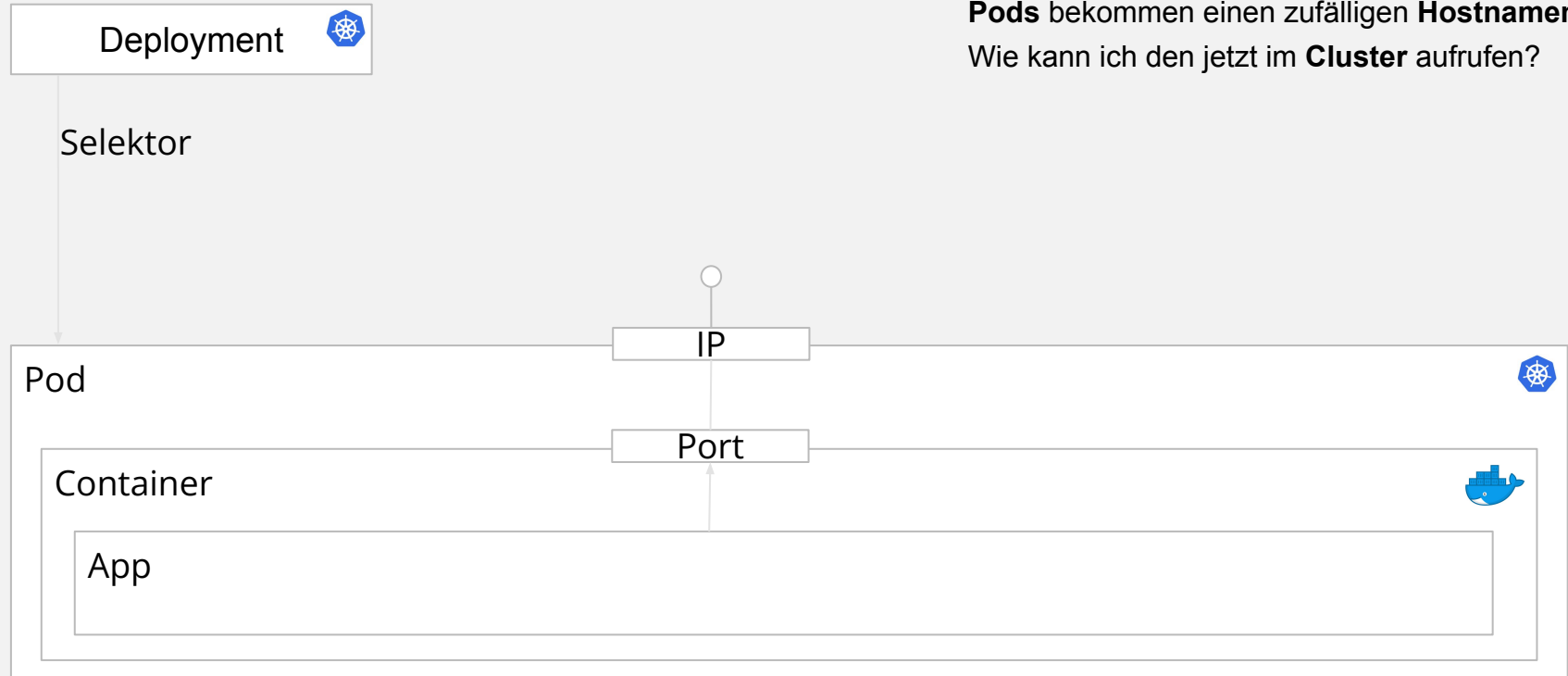
Services

Wichtige Kubernetes Konzepte



QA|WARE

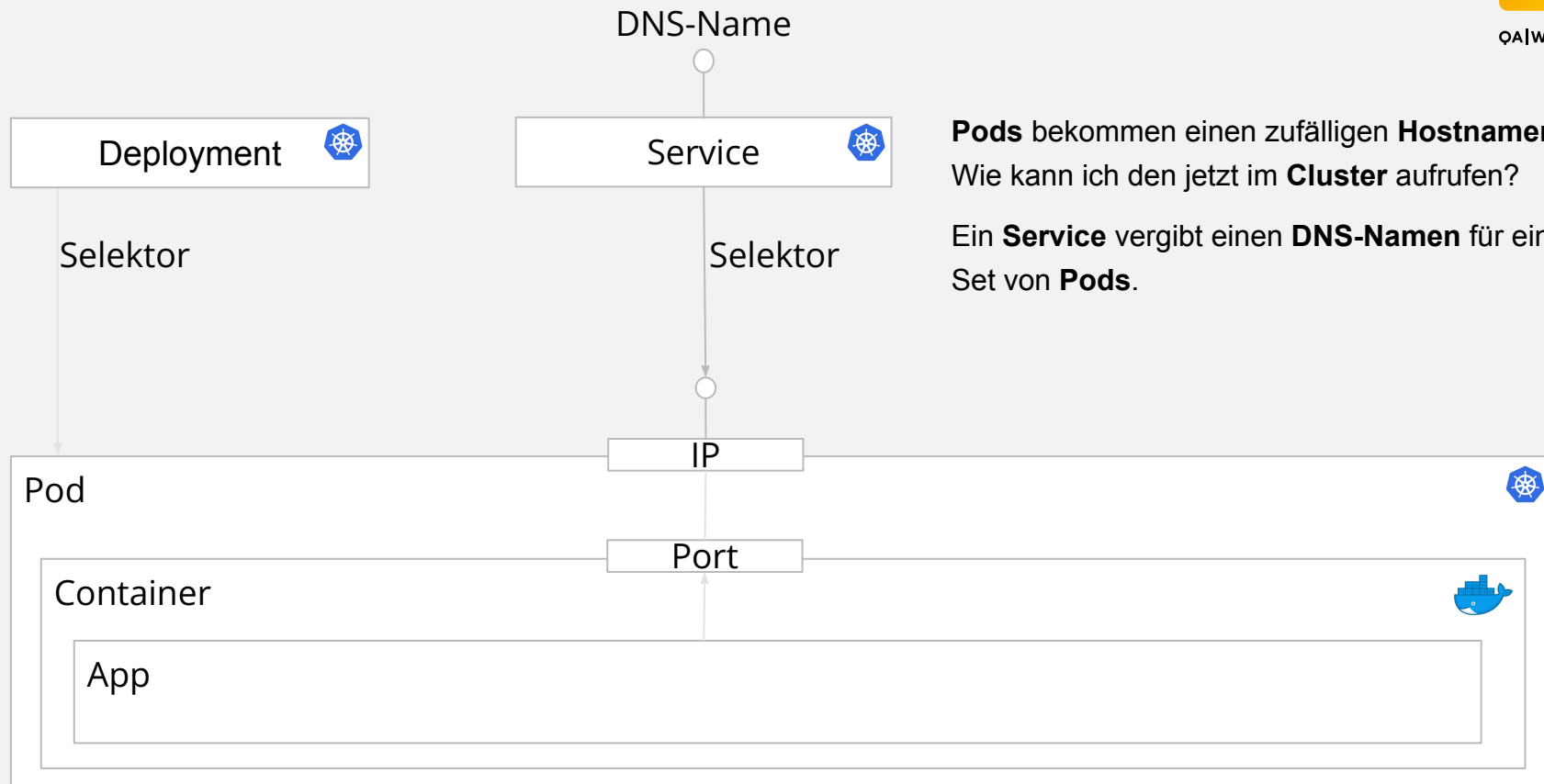
Pods bekommen einen zufälligen **Hostnamen**.
Wie kann ich den jetzt im **Cluster** aufrufen?



Wichtige Kubernetes Konzepte



QA|WARE



Service: Definition



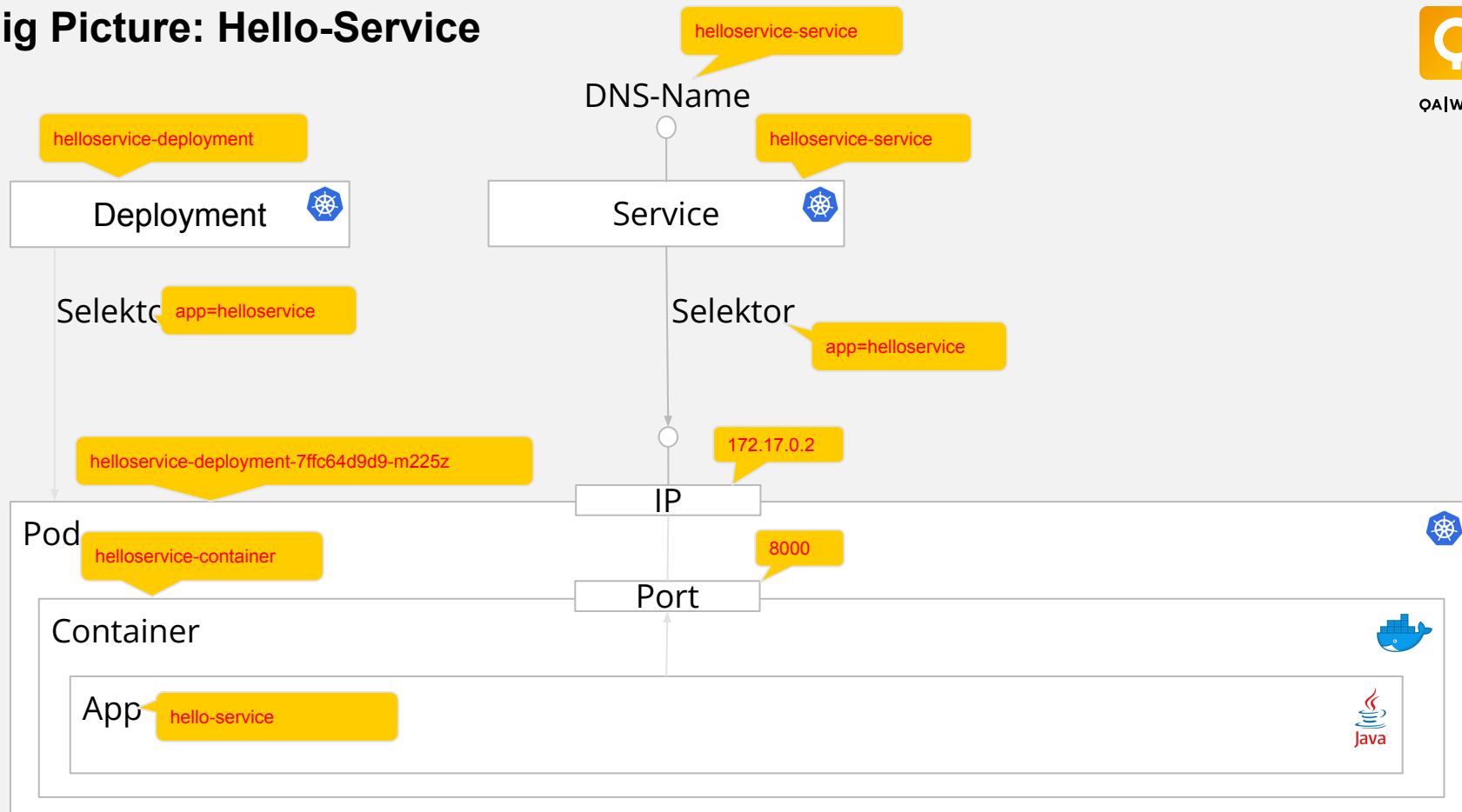
QA|WARE

```
apiVersion: v1
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
```

Big Picture: Hello-Service



QA|WARE





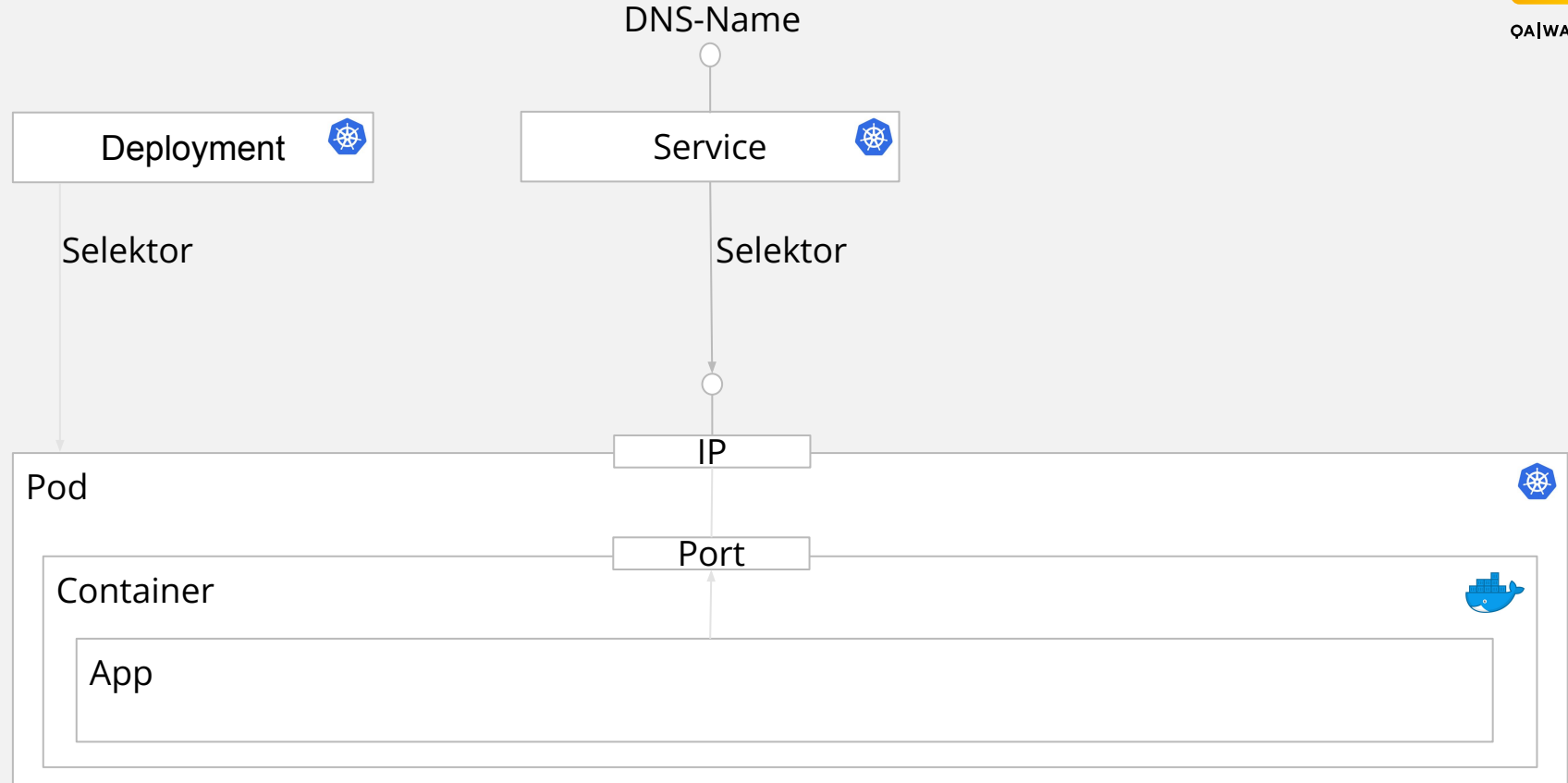
QA|WARE

Config Maps

Wichtige Kubernetes Konzepte



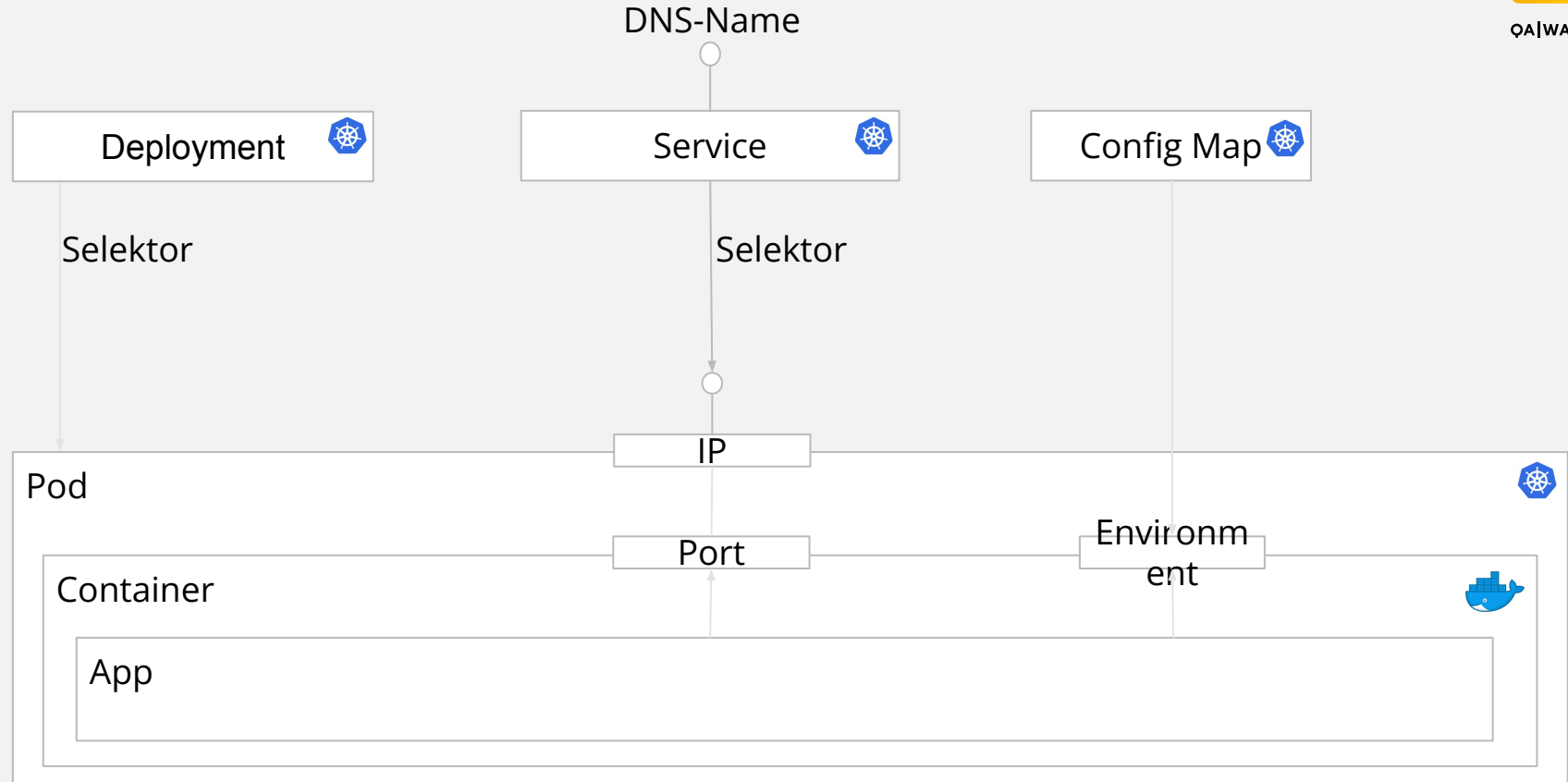
QA|WARE



Wichtige Kubernetes Konzepte



QA|WARE



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)

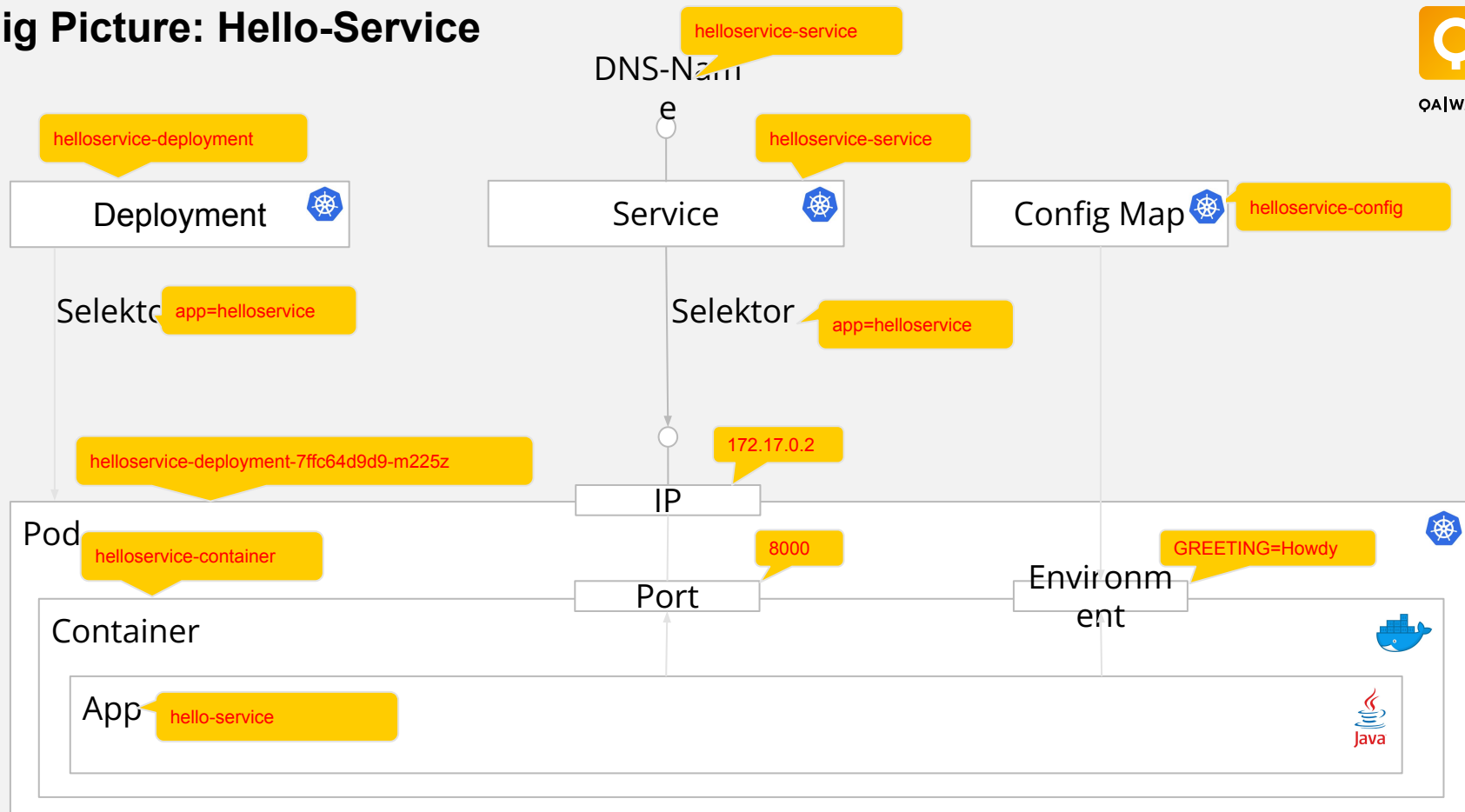


```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
                                          # from the key name in the ConfigMap.
          valueFrom:
            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
```

Big Picture: Hello-Service



QA|WARE



Weitere Konzepte von Kubernetes



QA|WARE

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?



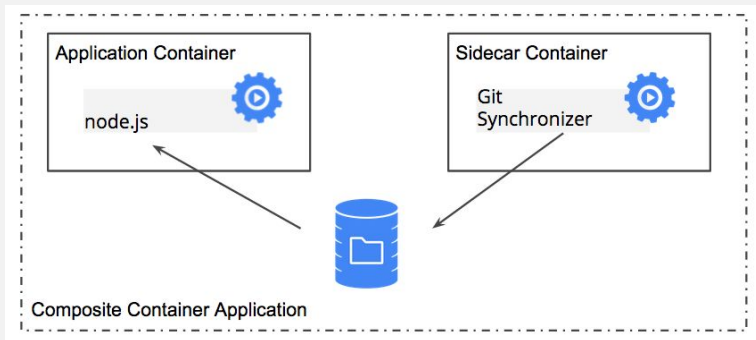
QA|WARE

Orchestrierungsmuster

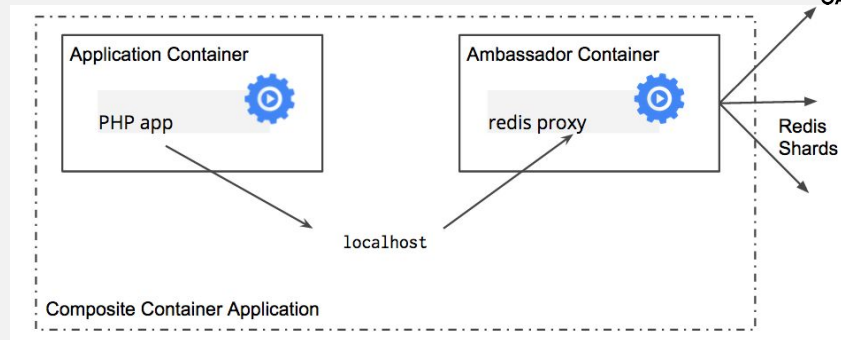
Orchestrierungsmuster – Separation of Concerns mit modularen Containern



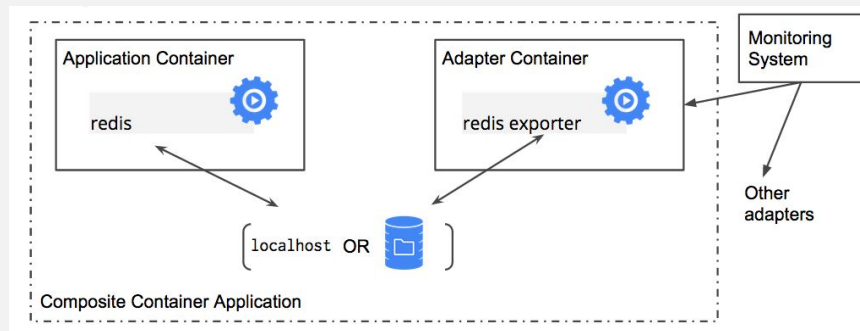
QO!WARE



Sidecar Container



Ambassador Container



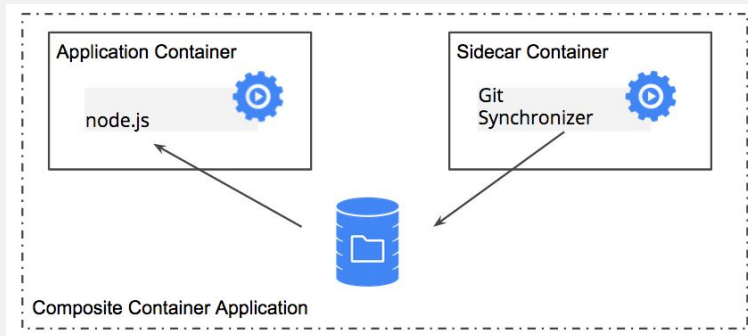
Adapter Container

Sidecar Containers



QA|WARE

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.

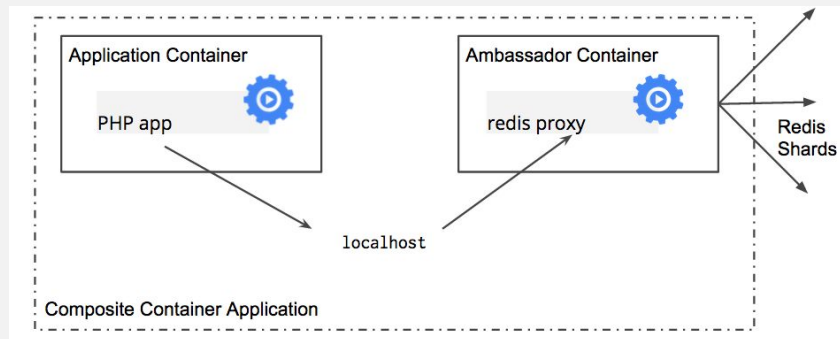


Ambassador containers



QAIWARE

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.

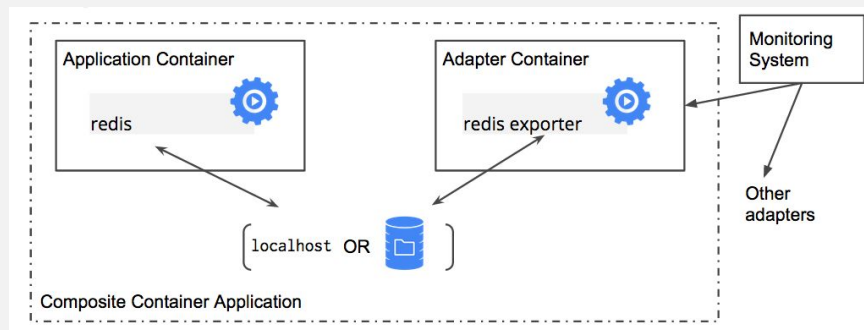


Adapter containers



QA|WARE

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/>