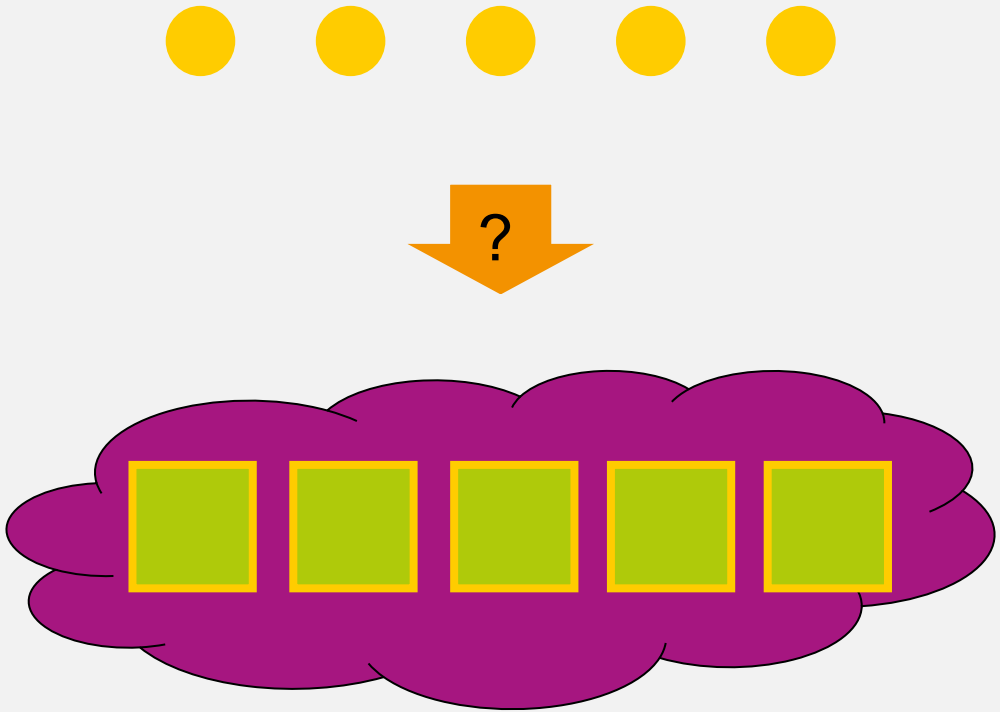
Cluster Orchestration

Franz Wimmer
franz.wimmer@qaware.de

Cluster Scheduling

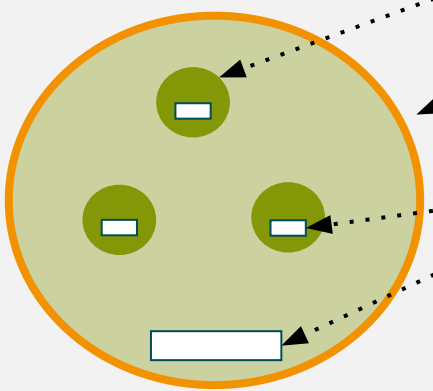
The Problem



Computational tasks

Computing resources (e.g. via IaaS)

Terminology



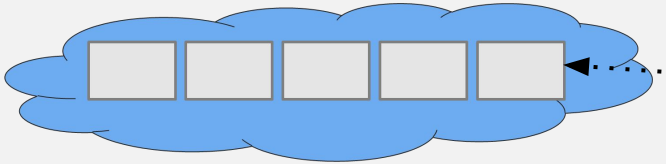
Task: Atomic calculation including execution rule.

Job: Set of tasks with common execution goal. The set of tasks is usually represented as a DAG with tasks as nodes and execution dependencies as edges.

Properties: Properties of the tasks and jobs that are relevant for execution, such as:

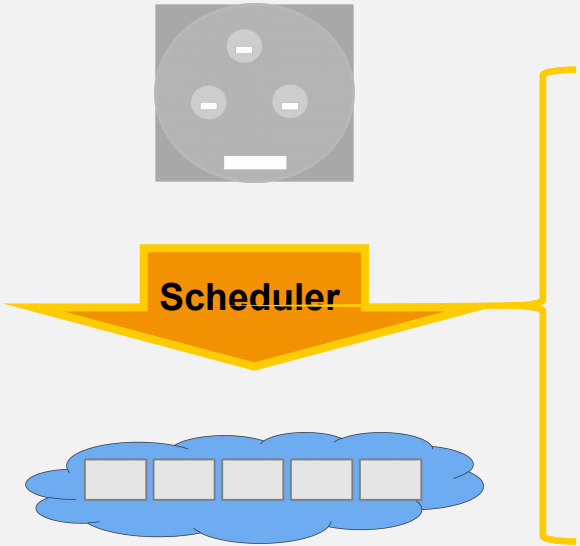
- Task: execution time, priority, resource consumption
- Job: task dependencies, execution time

Scheduler: Execution of tasks on the available resources, taking into account the properties and given **scheduling objectives** (e.g. fairness, throughput, resource utilization). A scheduler can be **preemptive**, i.e. it can interrupt the execution of tasks and restart them.



Resources: Clusters of computers with CPU, RAM, HDD and network resources. A computer makes its resources temporarily available for the execution of one or more tasks (slot). The parallel execution of tasks is isolated from each other.

Duties of a cluster scheduler:



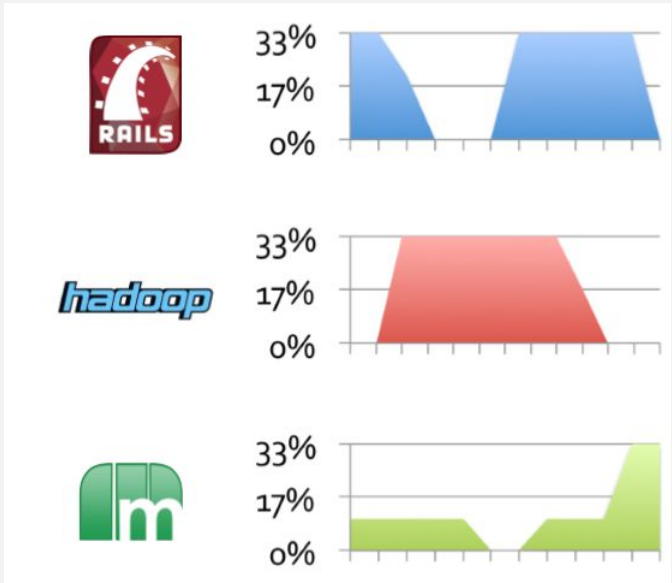
Cluster Awareness: Know the currently available resources in the cluster (nodes including available CPUs, available RAM and hard disk space, and network bandwidth). Also respond to elasticity.

Job Allocation: To determine and allocate the appropriate amount of resources for a specific period of time to perform a service.

Job Execution: Reliably perform a service while isolating and monitoring it.

The simplest form of scheduling: static partitioning

Utilization per node



Advantage:

- Easy to implement

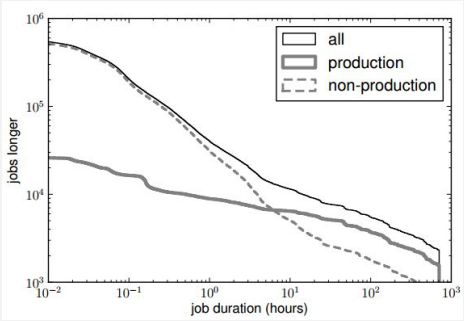
Disadvantages:

- Not flexible enough to adapt to changing needs
- Lower capacity utilization
- high opportunity costs

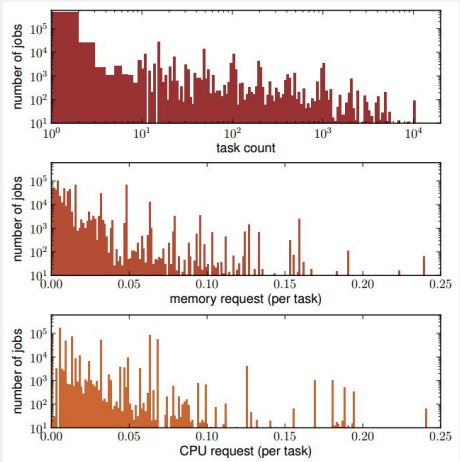
Image source: Practical Considerations for Multi-Level Schedulers, Benjamin Hindman, 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015

Heterogeneity in scheduling

- In typical clusters, the workload of jobs is very heterogeneous.
  - Characteristic differences are:
  - Execution duration: min, h, d, INF.
  - Execution time: immediately, later, at a point in time.
  - Execution purpose: data processing, request handling.
  - Resource consumption: CPU-, RAM-, HDD-, NW-dominant.
  - State: stateful, stateless.
- At least the following must be distinguished:
  - Batch jobs: execution time in the minute to hour range. Rather low priority and easily interruptible. Usually have to be completed by a certain point in time. Stateful.
  - Service jobs: should run indefinitely without interruption. Have high priority and should not be interrupted. Partially stateless.



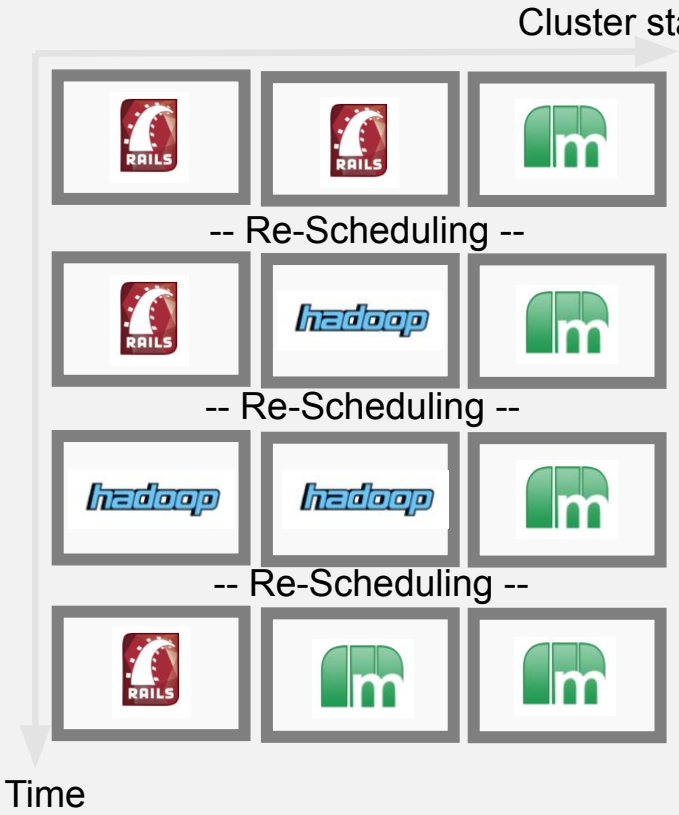
Ausführungsdauer



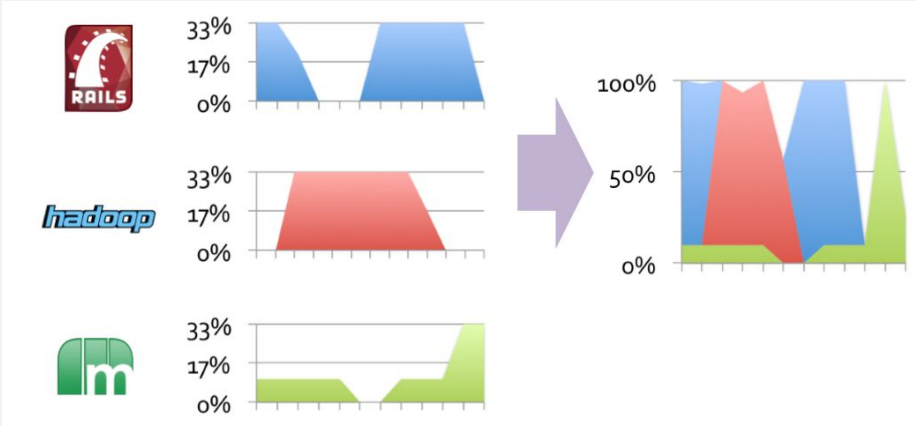
Ressourcenverbrauch

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

The existing resources of a cloud can be used much more efficiently through dynamic partitioning.



Static Partitioning **dynamic partitioning**



Advantages of dynamic partitioning:

- **Higher resource utilization => fewer resources required => lower operating costs**
- **Potentially faster execution of individual tasks, since resources can be used opportunely.**

A cluster scheduler: **input**, processing, output.

The **input** of a cluster scheduler is knowledge about the jobs and tasks (properties) and about the resources:

- Resource awareness: Which resources are available and what are the corresponding needs of the task?
- Data awareness: Where is the data that a task needs?
- QoS awareness: Which execution times must be guaranteed?
- Economy awareness: Which operating costs must not be exceeded?
- Priority awareness: What is the priority of the tasks in relation to each other?
- Failure awareness: What is the probability of a failure? (e.g. because of a rack or a power supply)
- Experience awareness: How has a task behaved in the past?

A cluster scheduler: input, **processing**, output.

Processing in the cluster scheduler: Scheduling algorithms according to the respective scheduling goals, such as:

- Fairness: No task should have to wait disproportionately long while another is favored.
- Maximum throughput: As many tasks per unit of time as possible.
- Minimum latency: The shortest possible time from submission to execution of a task.
- Resource utilization: The utilization of available resources should be as high as possible.
- Reliability: A task is guaranteed to be executed.
- Low end-to-end execution time (e.g. through data locality and low communication costs, syn. makespan)
- Processing in the cluster scheduler: Scheduling algorithms according to the respective scheduling objectives, such as:
  - Fairness: No task should have to wait an unreasonably long time while another is given preferential treatment.
  - Maximum throughput: As many tasks as possible per unit of time.
  - Minimum latency: The shortest possible time from transmission to execution of a task.
  - Resource utilization: The highest possible utilization of available resources.
  - Reliability: A task is guaranteed to be executed.
  - Low end-to-end execution time (e.g. through data locality and low communication costs, syn. makespan)

A cluster scheduler: input, processing, **output**.

Output of a cluster scheduler:

- Placement decision as
- Slot reservations
- and slot cancellations (in the event of an error, optimization or constraint violation)

The main objective is often to optimize resource utilization. This saves opportunity costs.

Scheduling is an optimization task...

... and is NP-complete.

The optimization task can be traced back to the Traveling Salesman Problem.

This means:

- There is no known algorithm that guarantees an optimal solution in polynomial time.
- The algorithm must be scalable for thousands of jobs and thousands of resources. Optimal algorithms that completely search the solution space are not practical, since their decision time is too long for large input sets (|jobs| x |resources|).
- Practical scheduling algorithms are therefore algorithms for approximately solving the optimization problem (heuristics, meta-heuristics).

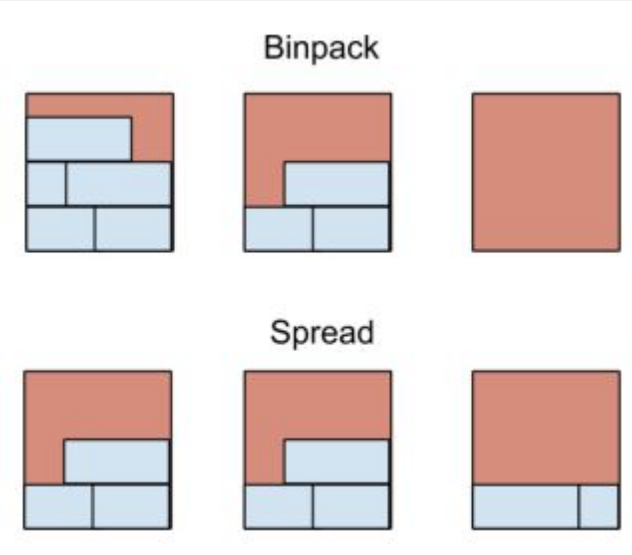
Furthermore, job requests are coming in continuously, so that even with an optimal algorithm, the reorganization effort per job can become disproportionately high.

Simple scheduling algorithms

- Optimize the scheduling of tasks often in exactly one dimension (e.g. CPU utilization) or a few dimensions (CPU utilization and RAM).

Popular algorithms:

- Binpack (Fit First)
- Spread (Round Robin)

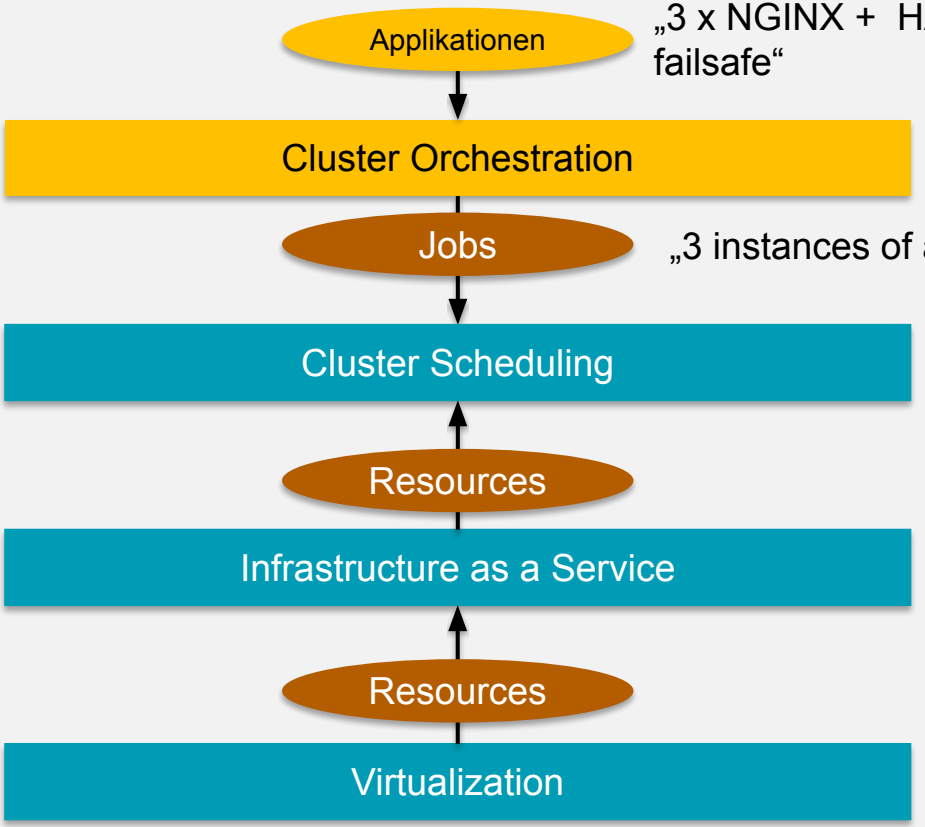


http://www.binpacking.4fan.cz
http://container-solutions.com/using-binpack-with-docker-swarm

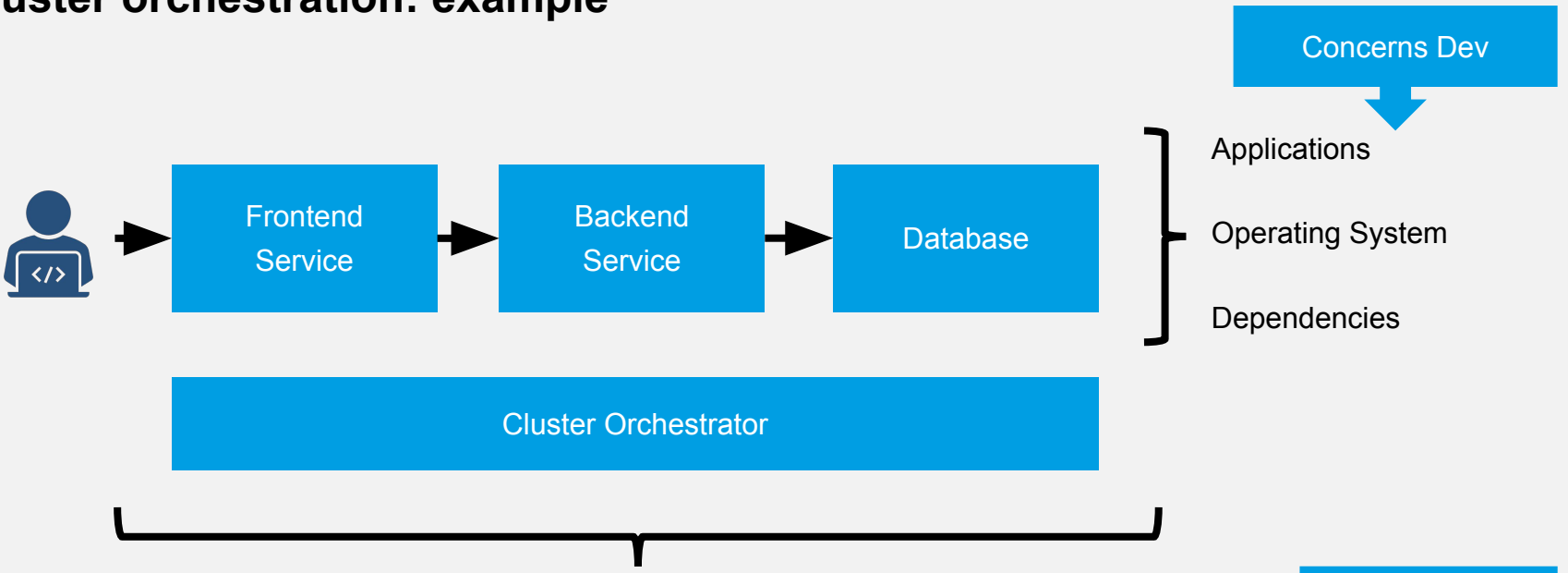
Kubernetes engineers when they don't have a kitchen towel

QA|WARE

The Big Picture: We are now at the application level.



Cluster orchestration: example



Cluster Orchestration

Objective: Run an application that is distributed across multiple operational components (containers) on multiple nodes.

Introduces abstractions for running applications with their services in a large cluster.

Orchestration is not a static, one-time activity like provisioning, but a dynamic, continuous activity.

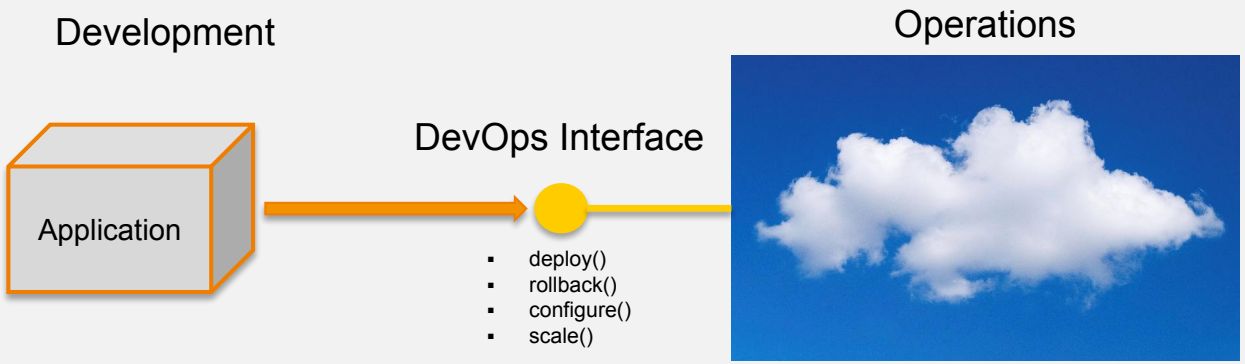
Orchestration has the potential to automate all standard operational procedures of an application.

Blueprint of the application that describes the desired operating state of the application: operating components (containers), their operating requirements, and the offered and required interfaces.

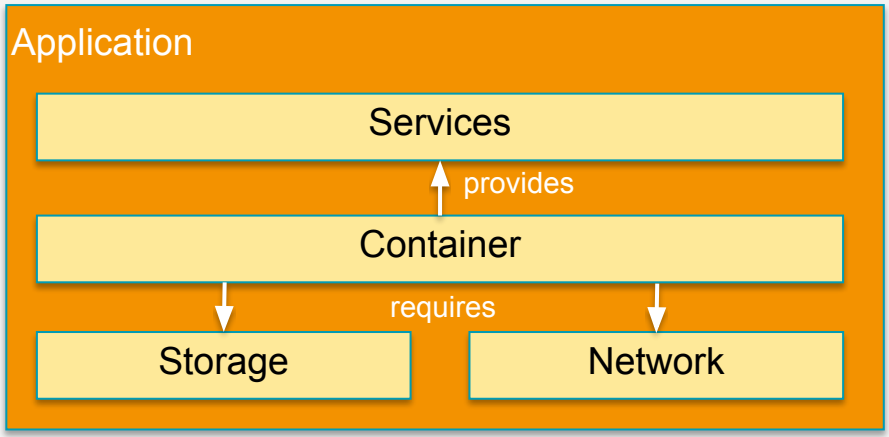
Cluster Orchestrator

- **Control activities in the cluster:**
- **Starting containers on nodes (scheduler)**
- **Linking containers**
- **...**

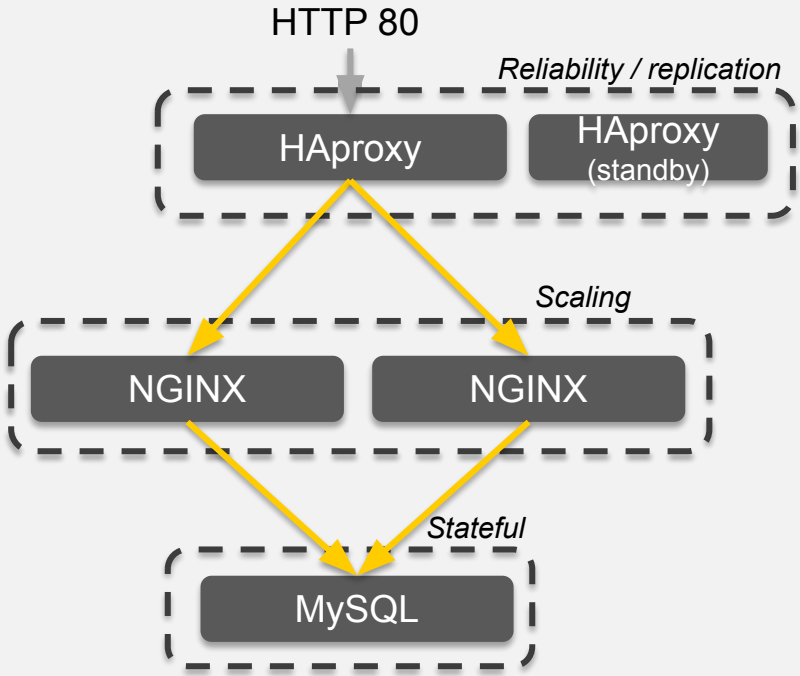
A cluster orchestrator provides an interface between operations and development for a cluster.



Simplified blueprint of an application



Meta model



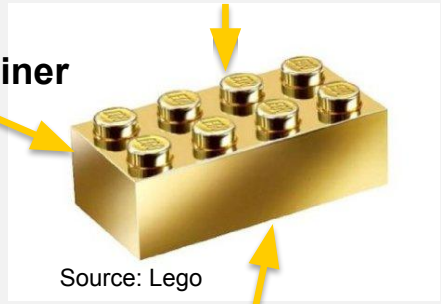
Model

Analogy 1: Lego Star Wars

QA|WARE

Services (provided)

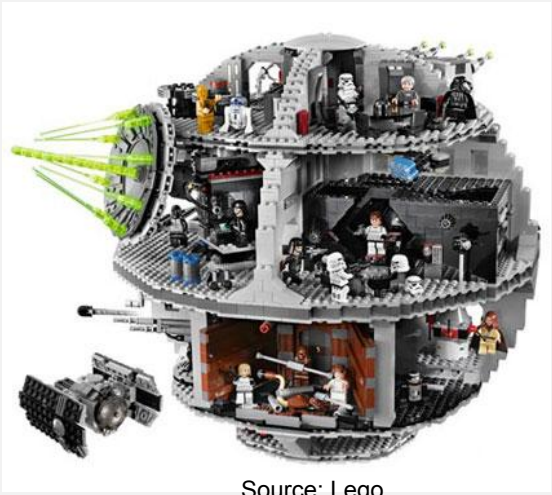
Container



Source: Lego

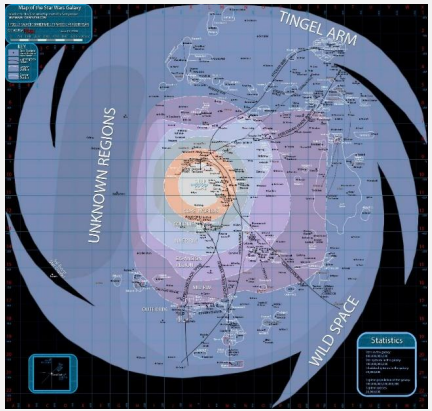
Services (required)

Cluster Orchestrator



Source: Lego

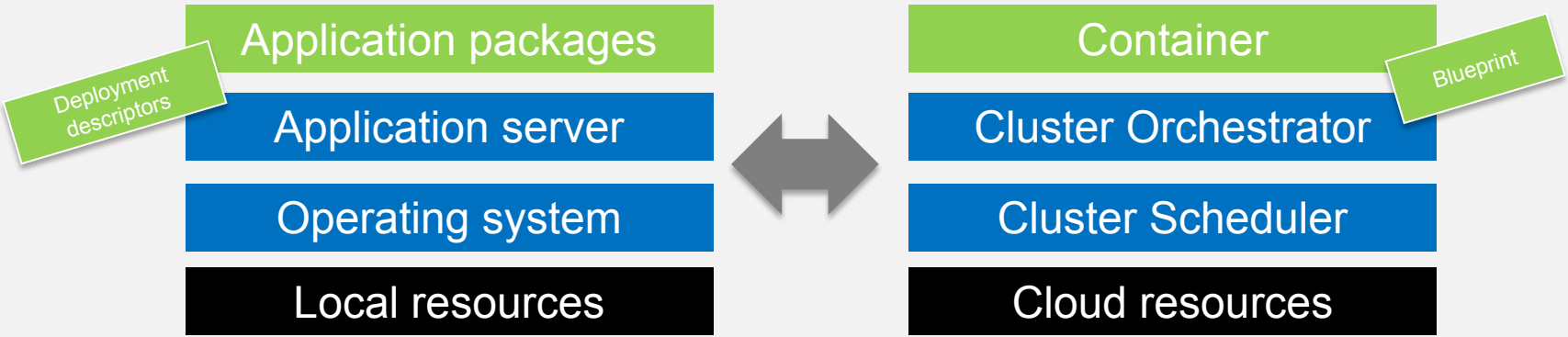
Cluster Scheduler



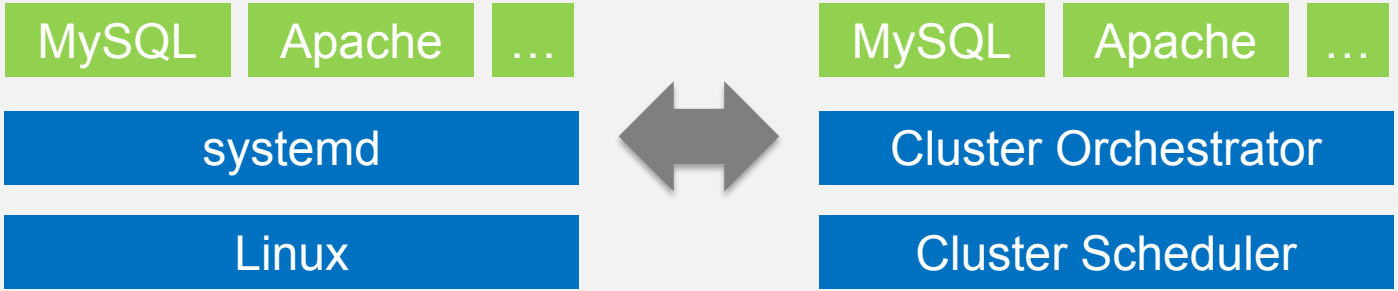
Quelle: wikipedia.de

Blueprint

Analogy 2: Application server



Analogie 3: Operating system



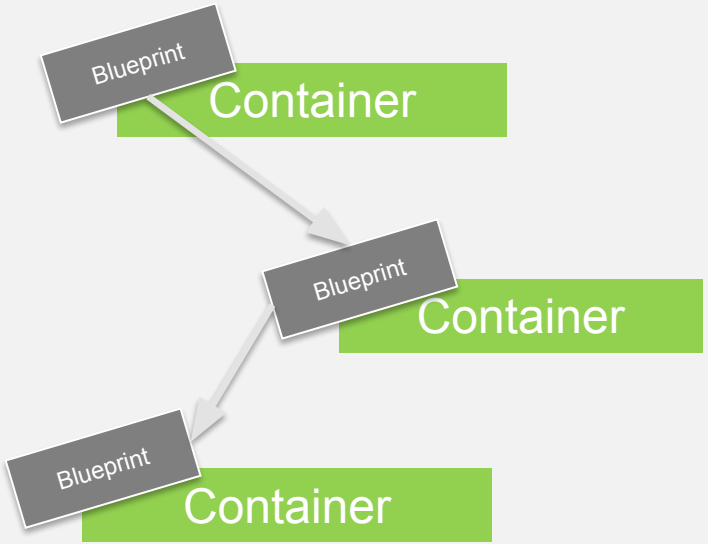
A cluster orchestrator automates many operational tasks for applications on a cluster. (1 / 2):

- Scheduling of containers with application-specific constraints (e.g. deployment and start orders, grouping, ...)
- Establishing the necessary network connections between containers.
- Providing persistent storage for stateful containers.
- (Auto-)scaling of containers.
- Rescheduling of containers in the event of an error (auto-healing) or to optimize performance.
- Container logistics: management and provision of containers.

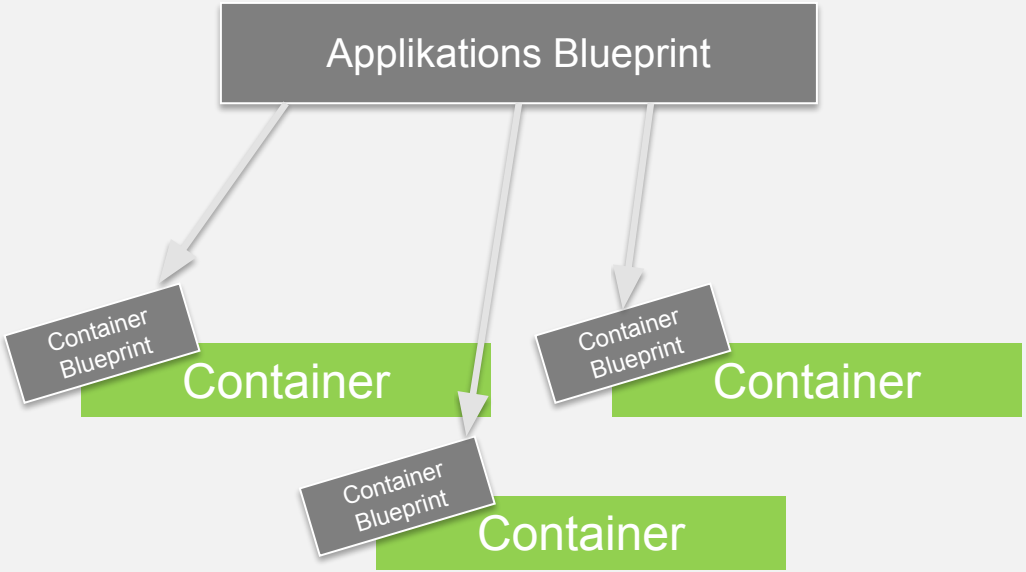
A cluster orchestrator automates many operational tasks for applications on a cluster. (2 / 2):

- Package management: administration and provision of applications.
- Provision of administration interfaces (remote API, command line).
- Service management: service discovery, naming, load balancing.
- Automation for rollout workflows such as Canary Rollout.
- Monitoring and diagnosis of containers and services.

1-level vs. 2-level orchestration



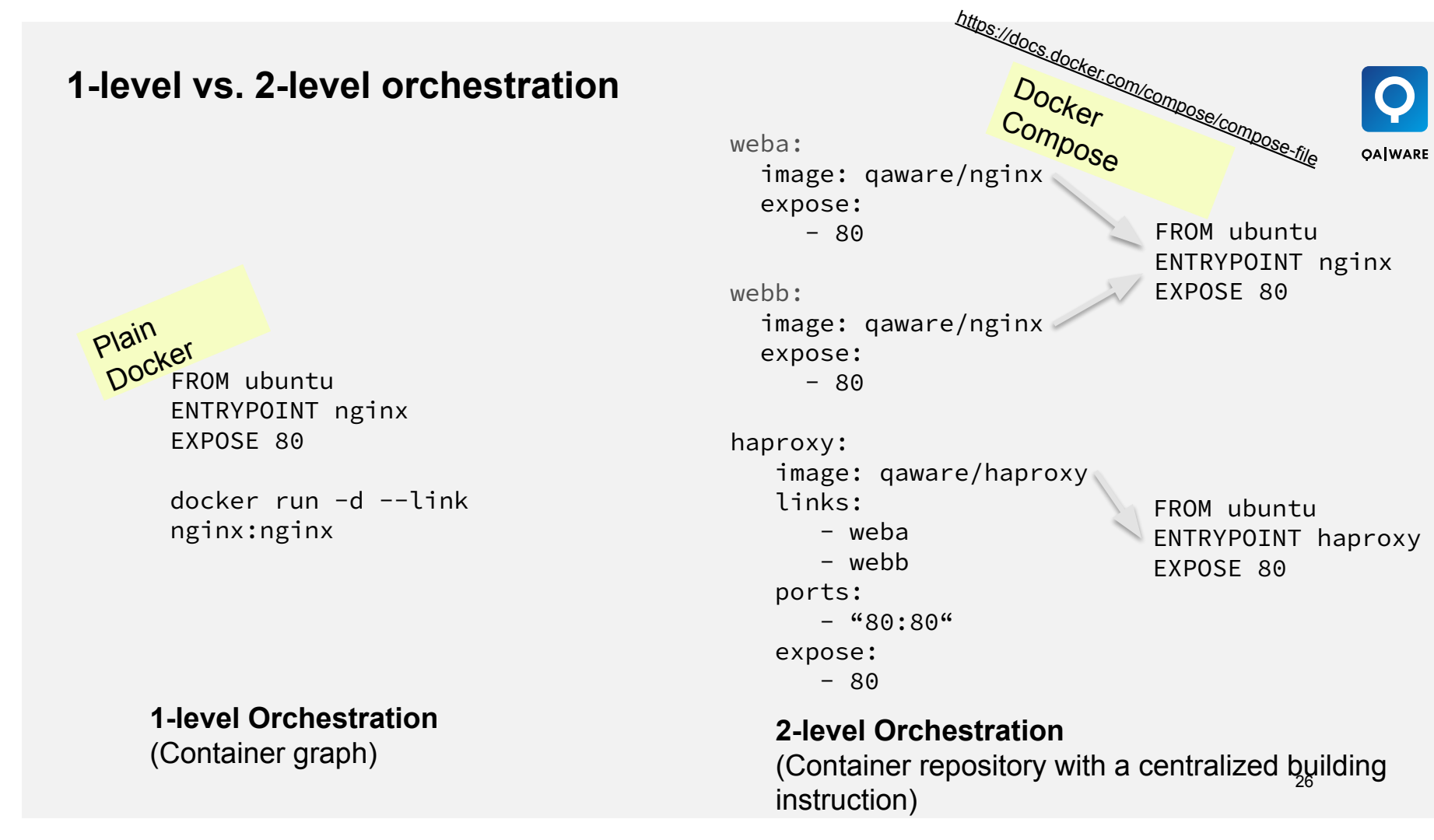
1-level Orchestration
(Container Graph)



2-level Orchestration
(Container repository with a centralized building instruction)

1-level vs. 2-level orchestration

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



1-level Orchestration
(Container graph)

2-level Orchestration
(Container repository with a centralized building instruction)

Kubernetes

kubernetes by Google

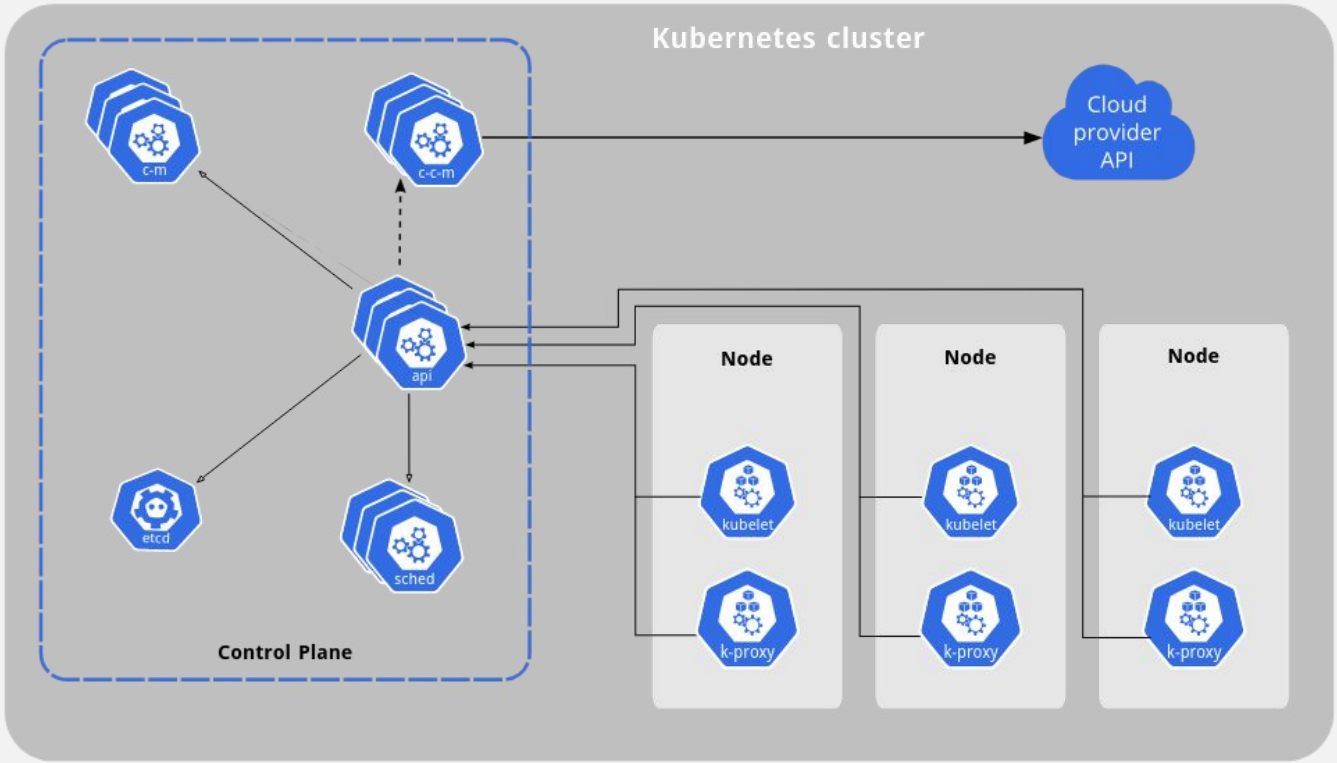
Manage a cluster of Linux containers as a single system to accelerate Dev and simplify Ops.

http://kubernetes.io

Kubernetes

- Container-based cluster orchestrator that introduces a set of core abstractions for running applications in a large cluster. The blueprint is defined using YAML files.
- Open-source project initiated by Google. Google wants to make its years of experience in operating large clusters available to the public and thus also leverage synergies with its own cloud business.
- Version 1.0 has been available since July 2015 and is therefore ready for production. It currently scales demonstrably to clusters of 1000 nodes.
- Used by many companies, such as Google as part of the Google Container Engine, Wikipedia, ebay. Contributions to the code base from many companies besides Google – including Mesosphere, Microsoft, Pivotal, RedHat.
- Sets the standard in the area of cluster orchestration. The Cloud Native Computing Foundation was founded specifically for this purpose (https://cncf.io).

Kubernetes architecture



API server

Cloud controller manager *(optional)*

Controller manager

etcd (persistence store)

kubelet

kube-proxy

Scheduler

Control plane

Node

Pods & Deployments

Important Kubernetes Concepts

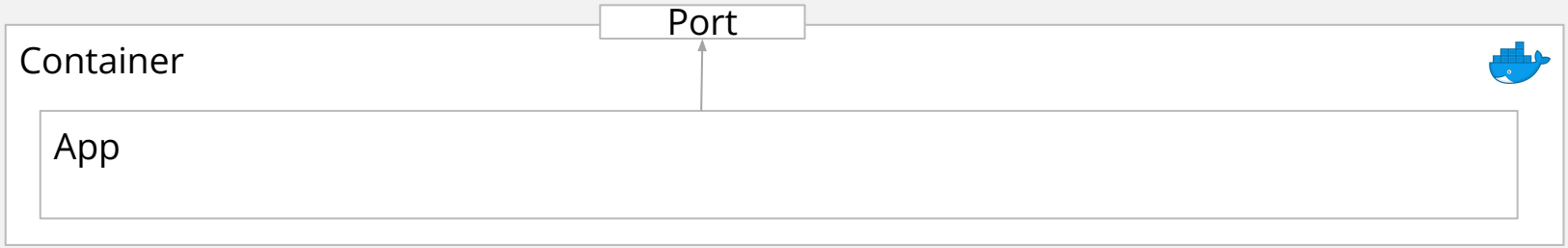
The basic building block is your **application**.

App

Important Kubernetes Concepts

The basic building block is your **application**.

The **application** is in a **container** (see lecture “Virtualization”). **Containers** open **ports** to the outside.



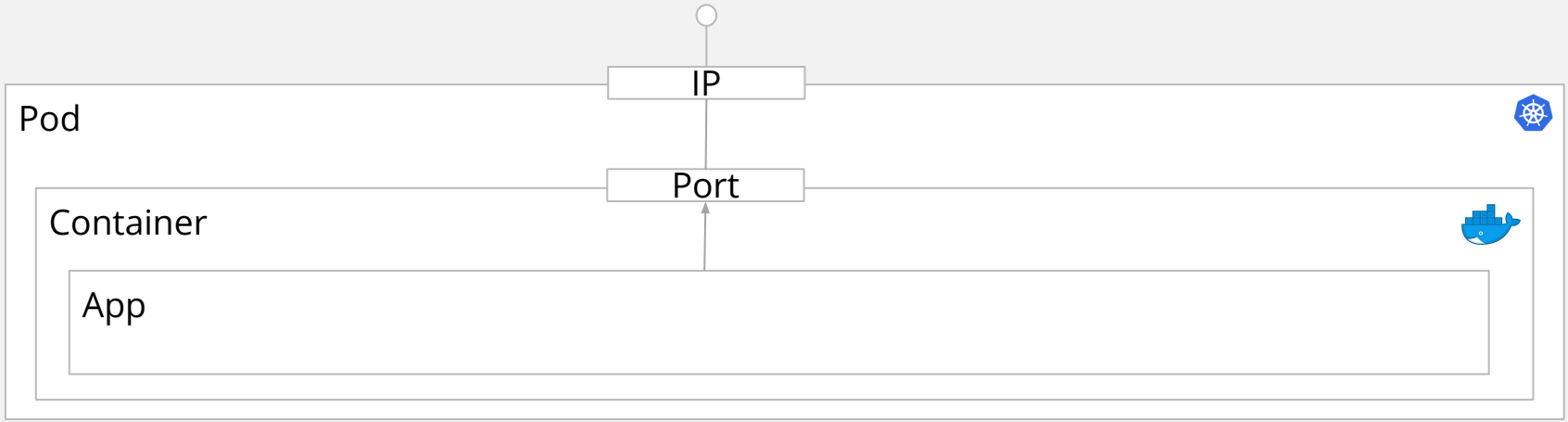
Important Kubernetes Concepts

QA|WARE

The basic building block is your **application**.

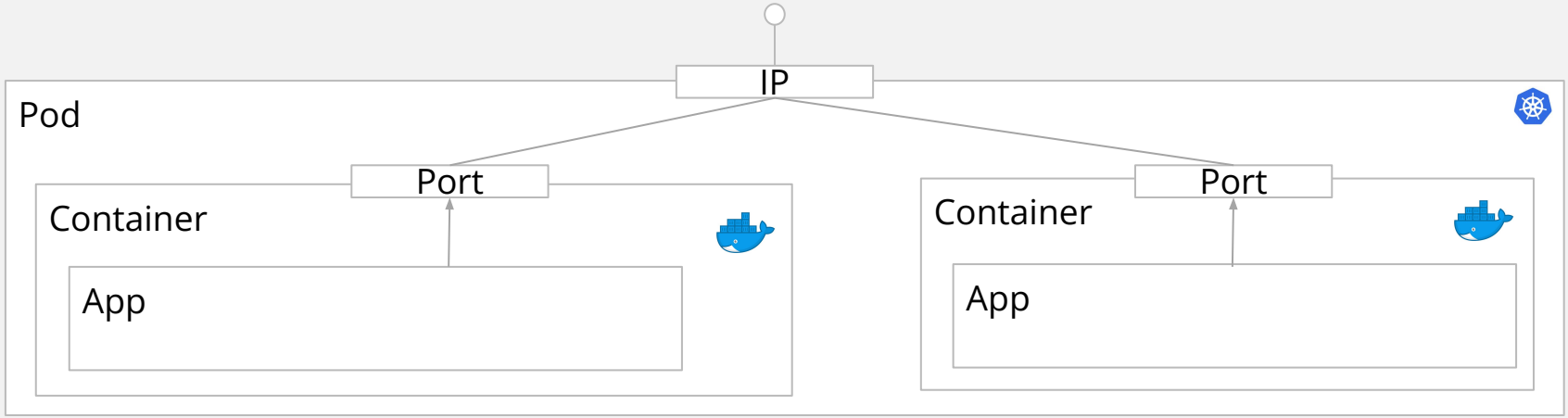
The **application** is in a **container** (see lecture “Virtualization”). **Containers** open **ports** to the outside.

In Kubernetes, **containers** are grouped into **pods**. **Pods** have one **IP address** facing outwards.

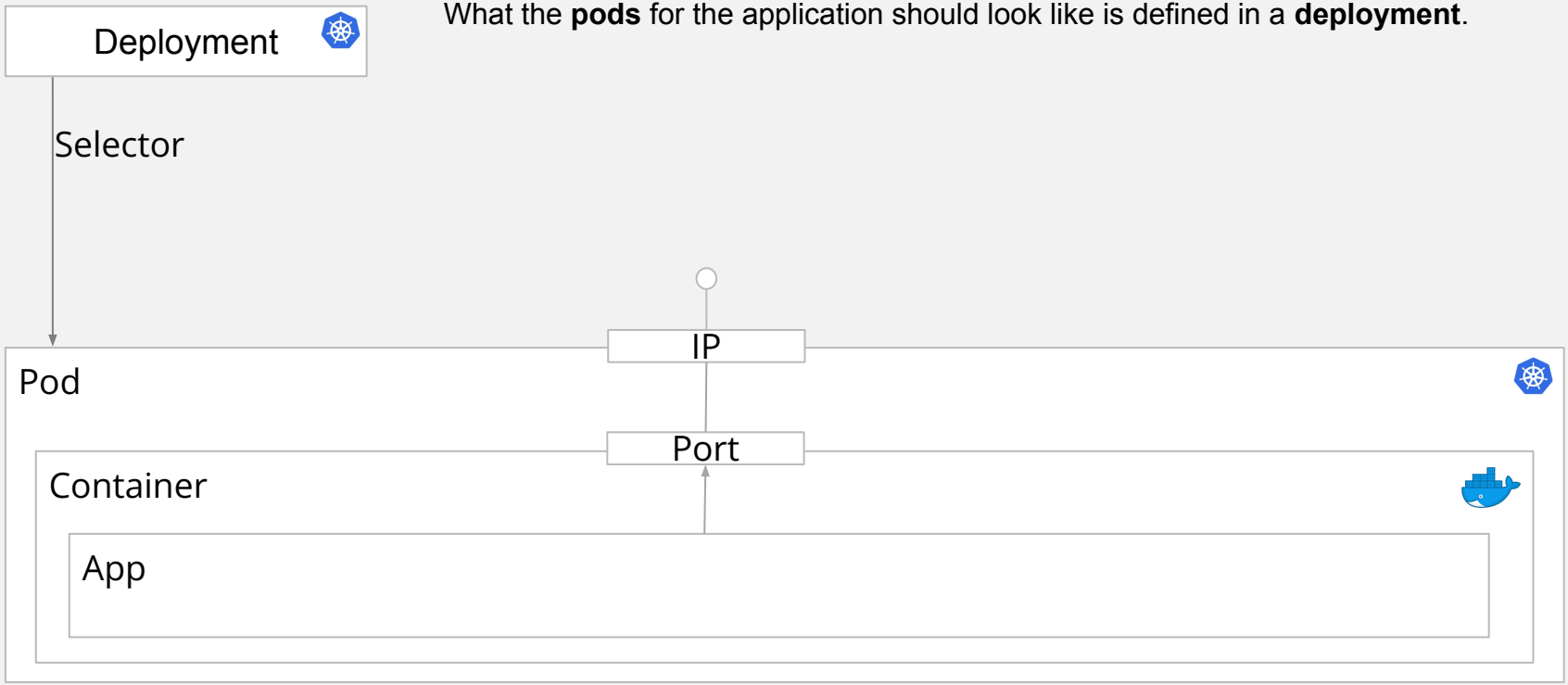


Important Kubernetes Concepts

Multiple **containers** can also run in one **pod**.



Important Kubernetes Concepts

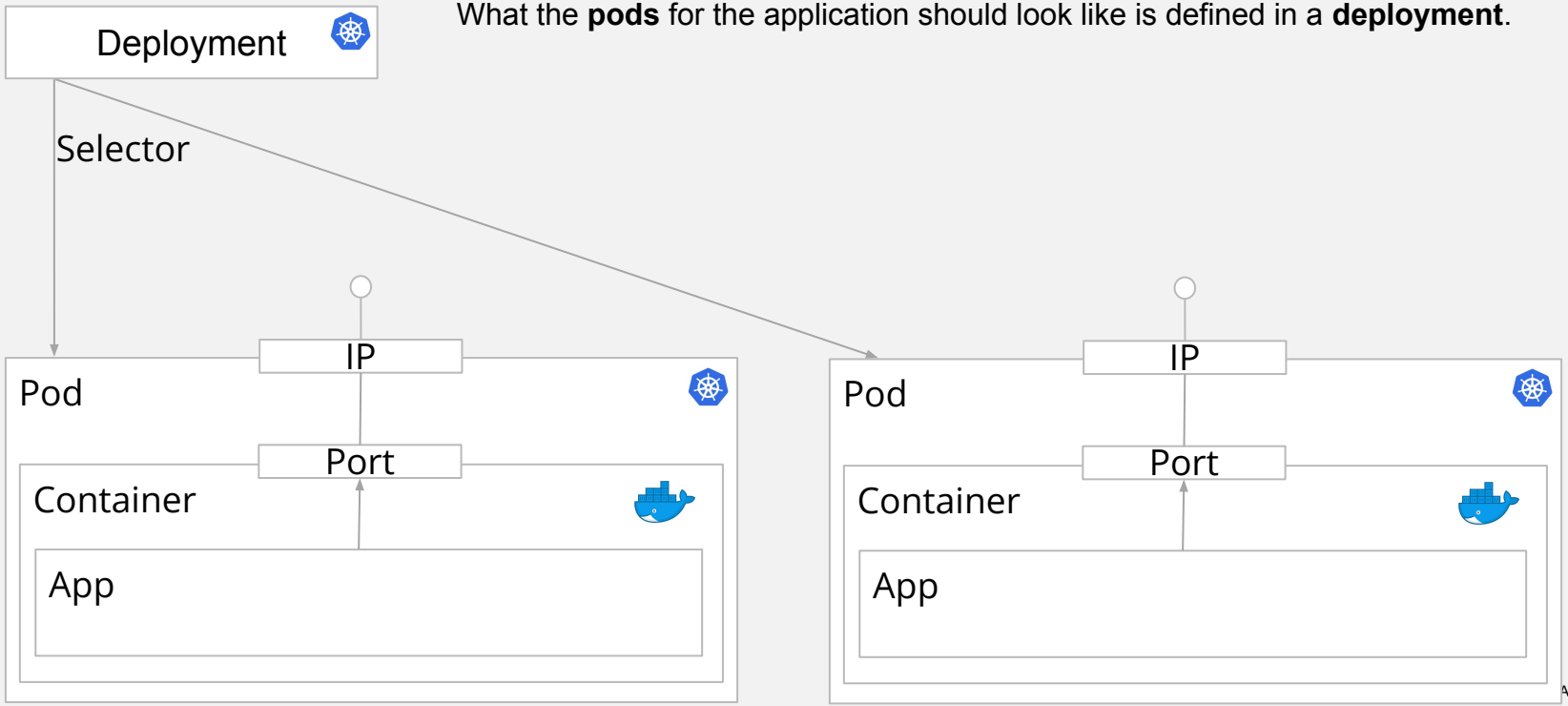


What the **pods** for the application should look like is defined in a **deployment**.

Important Kubernetes Concepts

QA|WARE

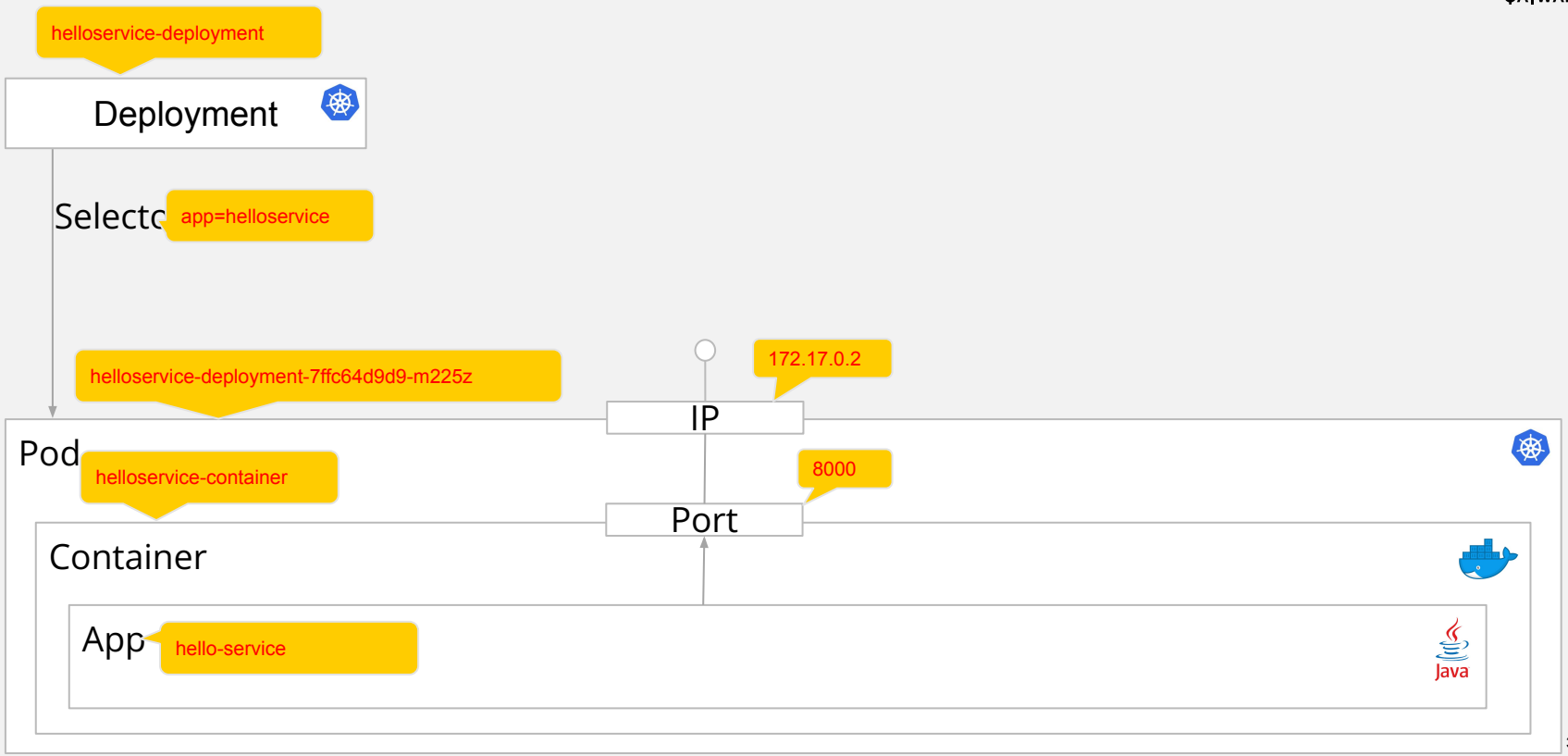
What the **pods** for the application should look like is defined in a **deployment**.



Deployment: Definition


```yaml
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


```yaml
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 throttled (CPU) or restarted (memory)
  # if limits are exceeded
  limits:
    memory: "192M"
    cpu: "0.5"
```

Liveness und Readiness Probes

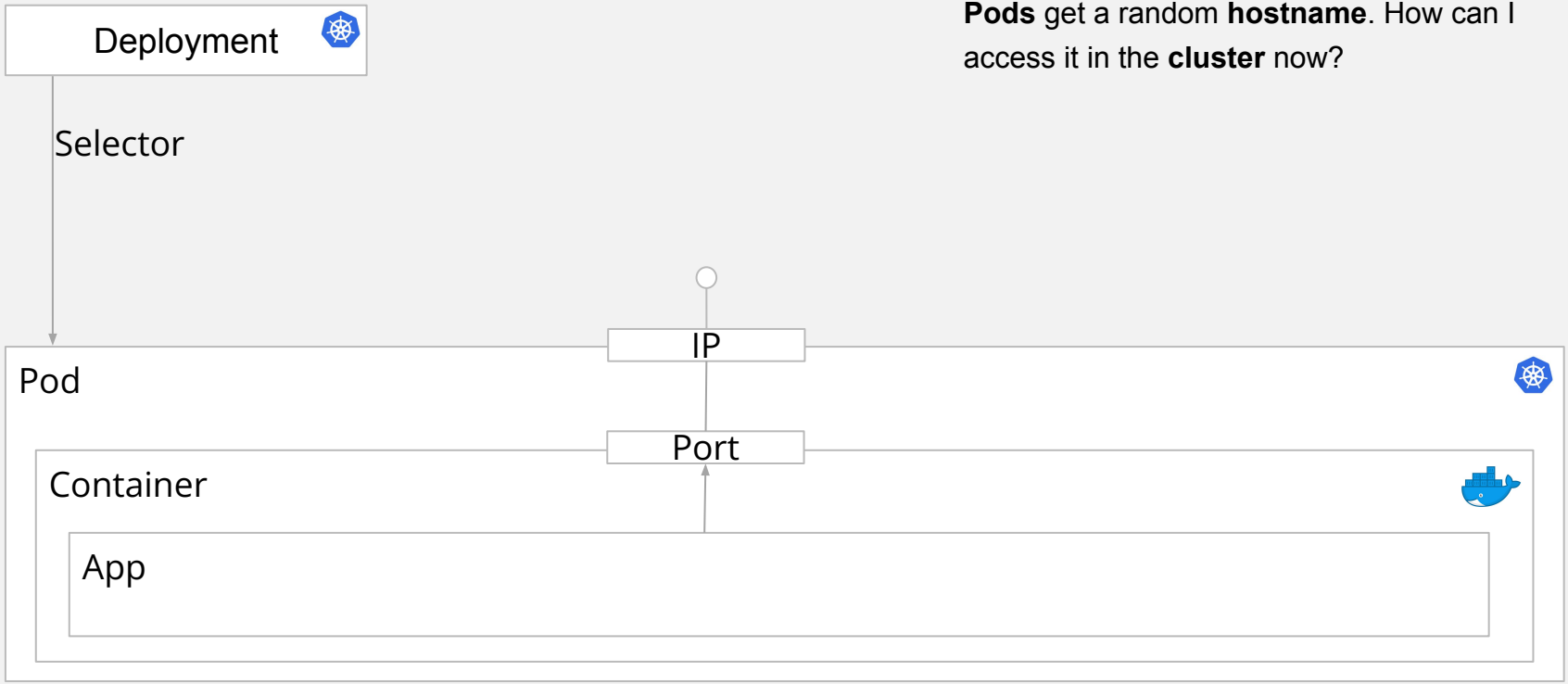

```
# container will receive requests if probe succeeds
readinessProbe:
  httpGet:
    path: /admin/info
    port: 8080
  initialDelaySeconds: 30
  timeoutSeconds: 5

# container will be killed if probe fails
livenessProbe:
  httpGet:
    path: /admin/health
    port: 8080
  initialDelaySeconds: 90
  timeoutSeconds: 10
```

Startup Probe


```yaml
# other probes will start after startup probe succeeds
startupProbe:
  httpGet:
    path: /admin/health
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

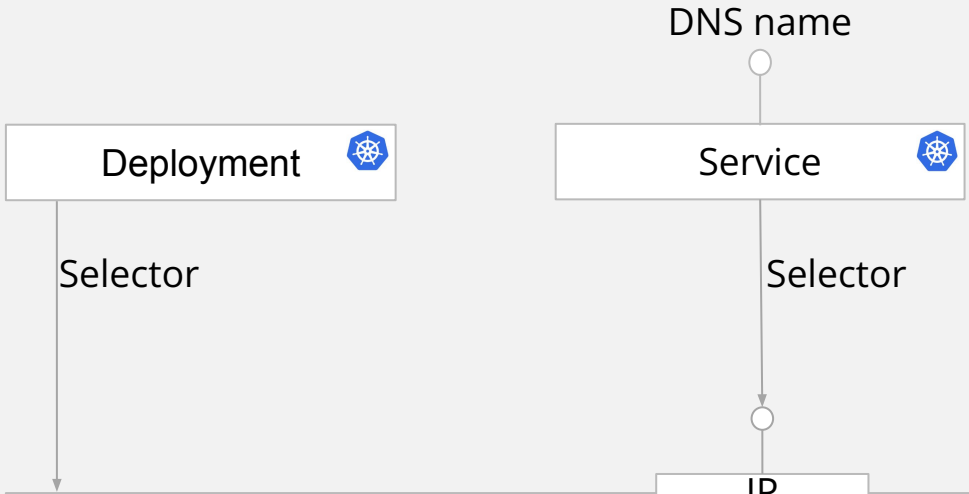
Services

Important Kubernetes Concepts



Pods get a random **hostname**. How can I access it in the **cluster** now?

Important Kubernetes Concepts



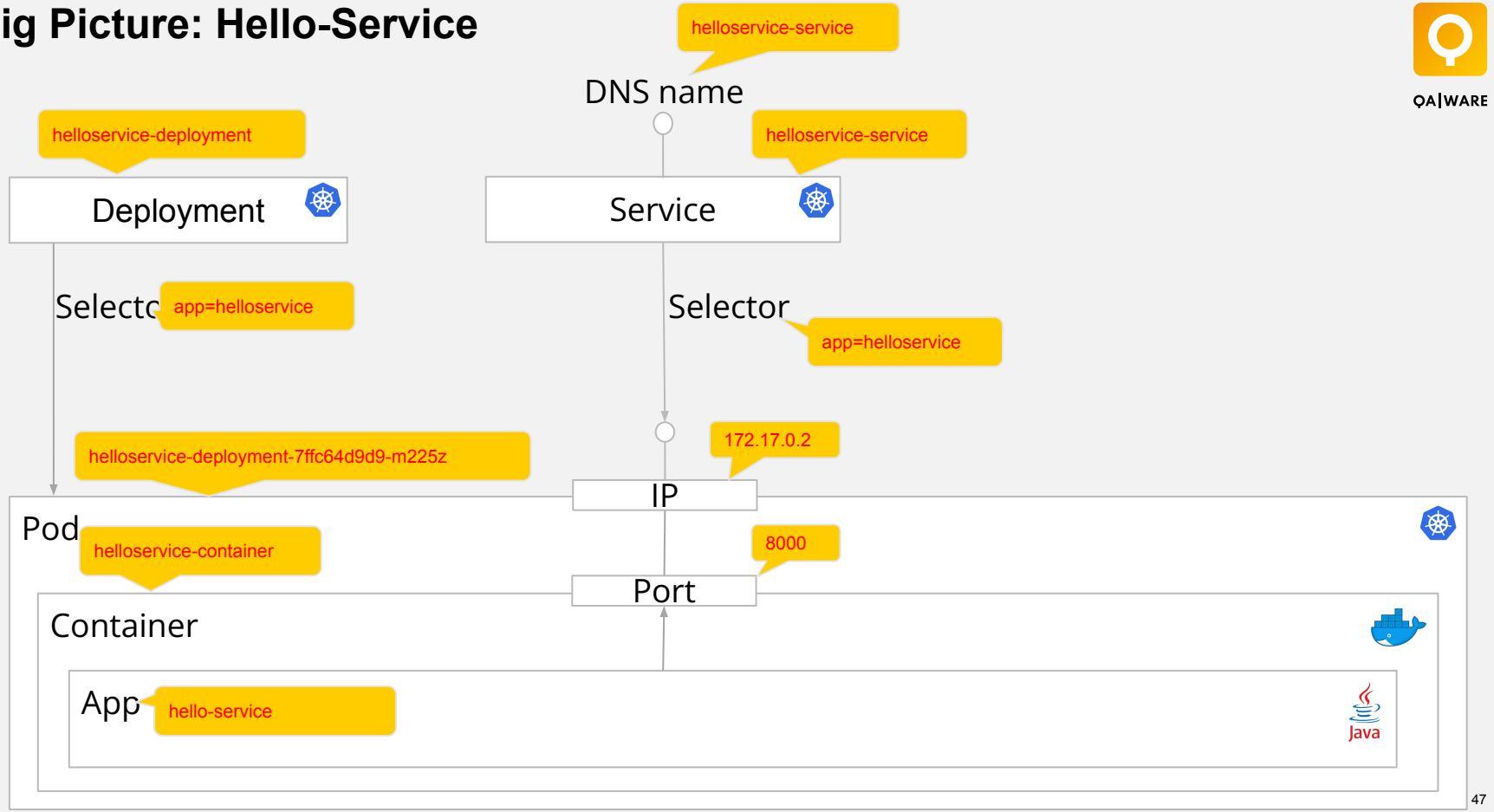
Pods get a random **hostname**. How can I access it in the **cluster** now?

A **service** assigns a **DNS name** for a set of **pods**.

Service: Definition

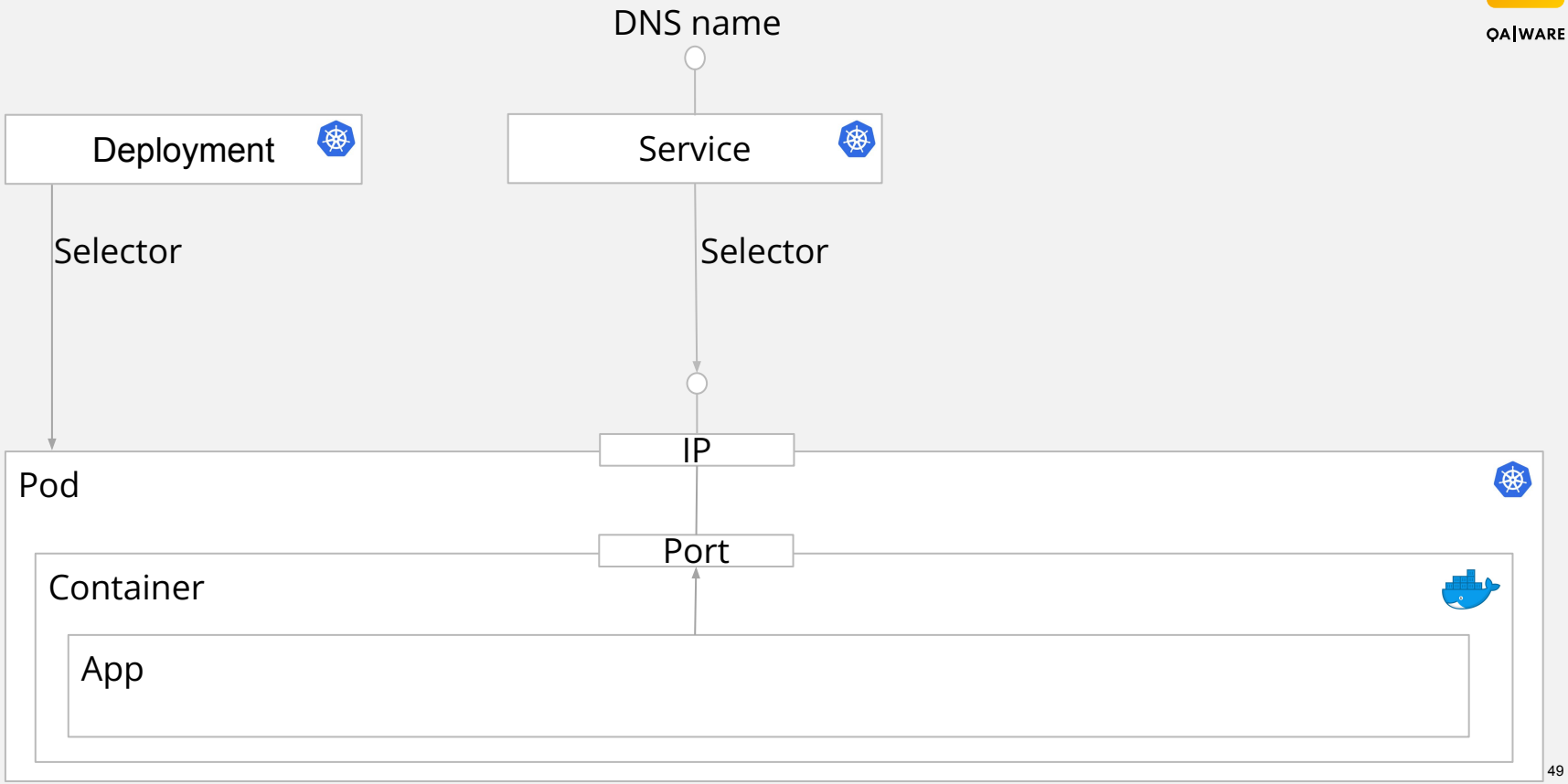

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-service
  labels:
    app: helloservice
spec:
  # use NodePort here to be able to access the port on each node
  # use LoadBalancer for external load-balanced IP if supported
  type: NodePort
  ports:
  - port: 8080
  selector:
    app: helloservice
```

Big Picture: Hello-Service

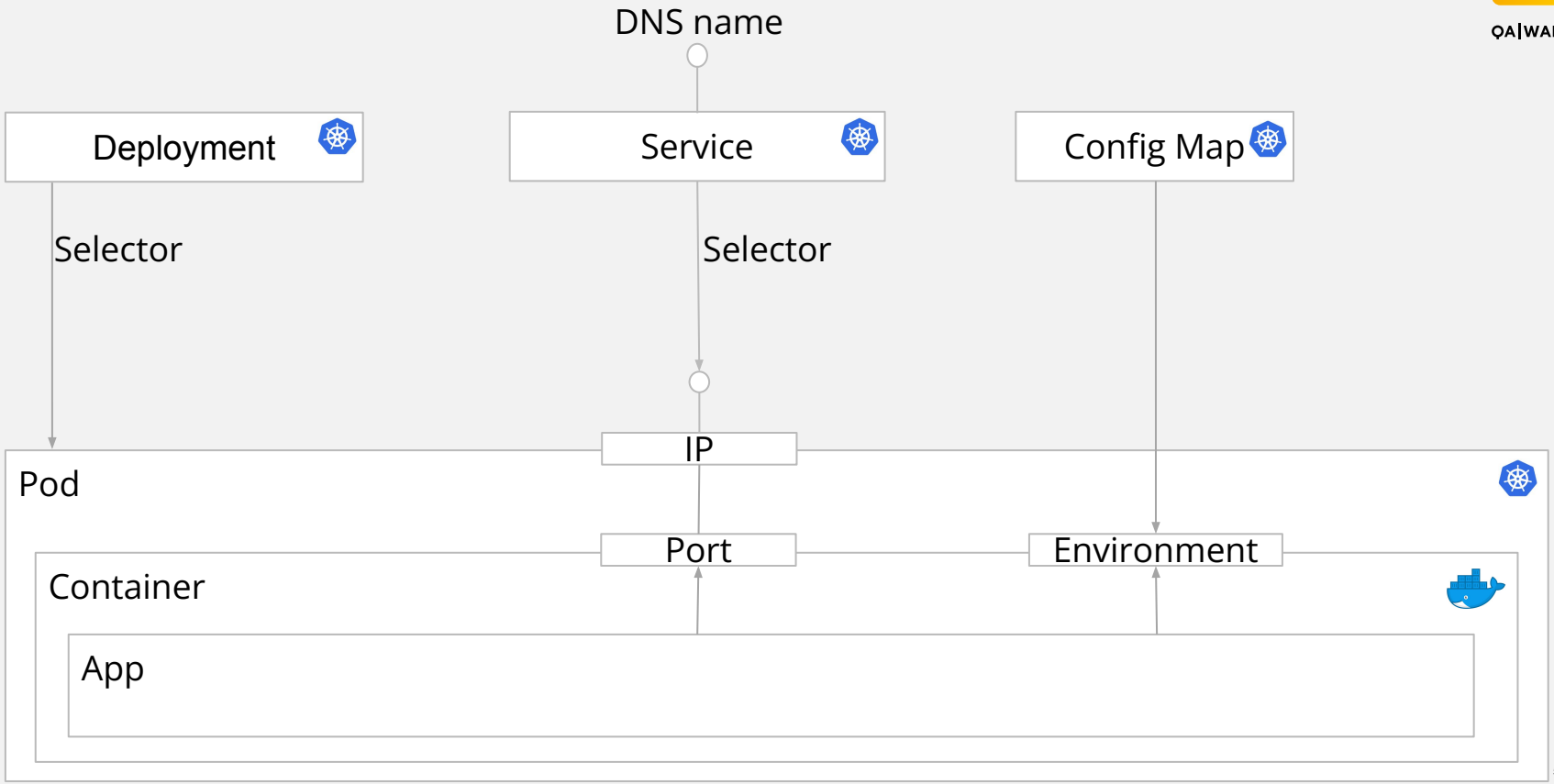


Config Maps

Important Kubernetes Concepts



Important Kubernetes Concepts



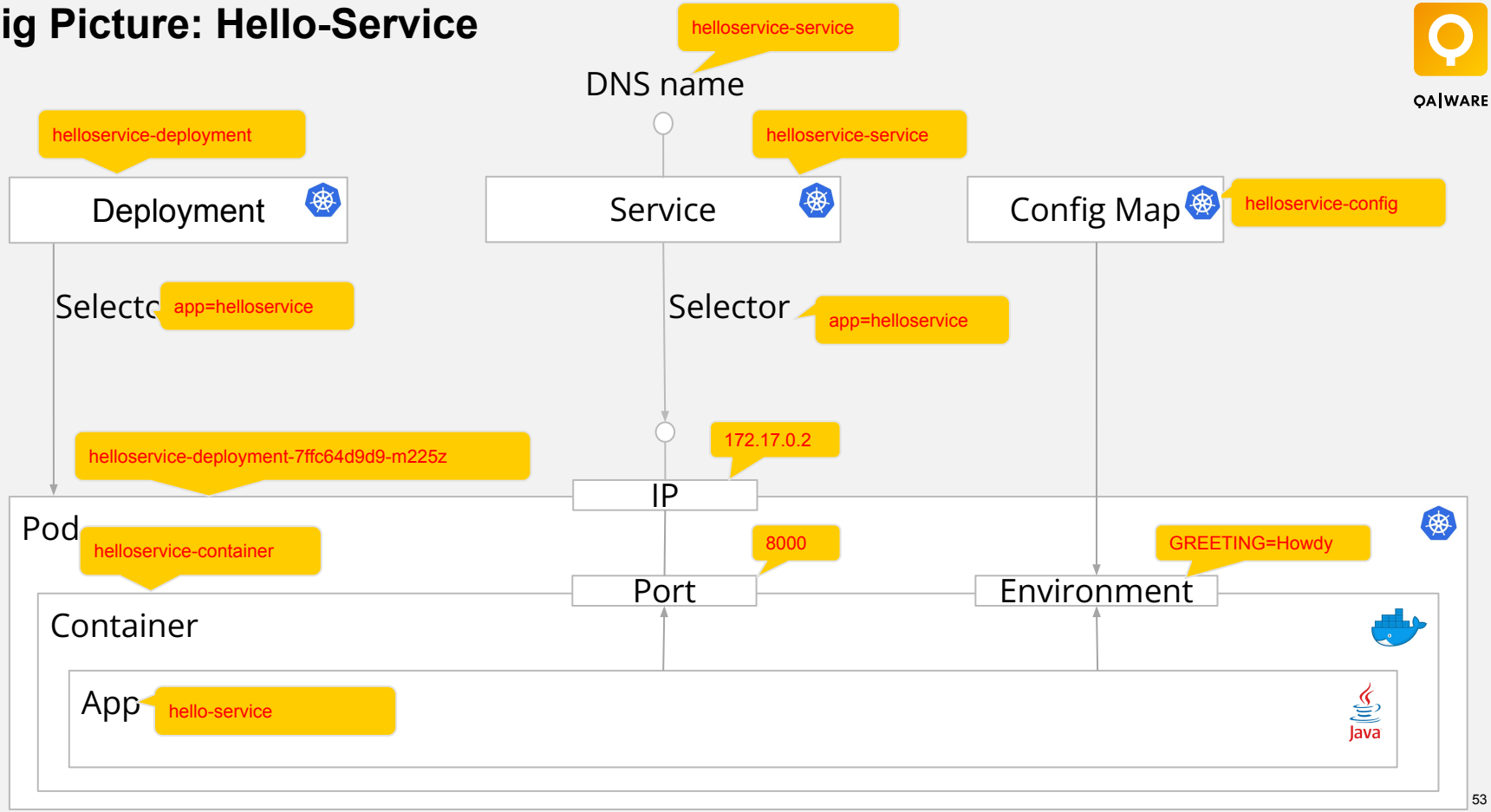
Configuration: Config Maps (1)

```yaml
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"
```

Configuration: Config Maps (2)

```yaml
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



Declarative approach with Kustomize

- Normal procedure:

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

- Not suitable for a collection of many files
- Therefore: Group the yaml files with **kustomize**!

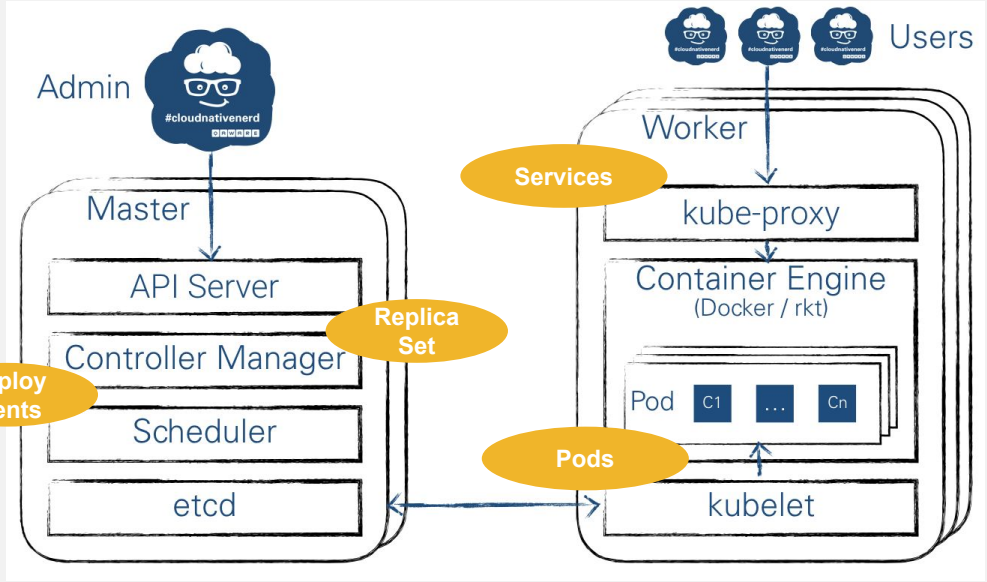
```
$ kubectl apply -k kustomization.yaml
```

https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/

Kustomize
If you don't want to push each file individually into the cluster


```yaml
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.

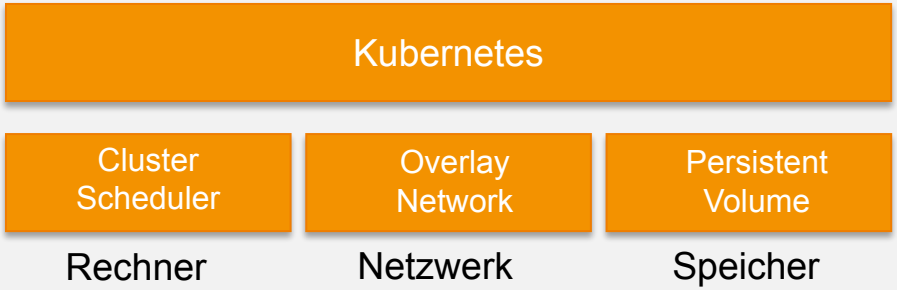
kube-proxy: Stellt einen Service nach Außen zur Verfügung.

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

QA|WARE

Netzwerk-Virtualisierung (Overlay Network)

- OpenVSwitch
- Flannel
- Weave
- Calico

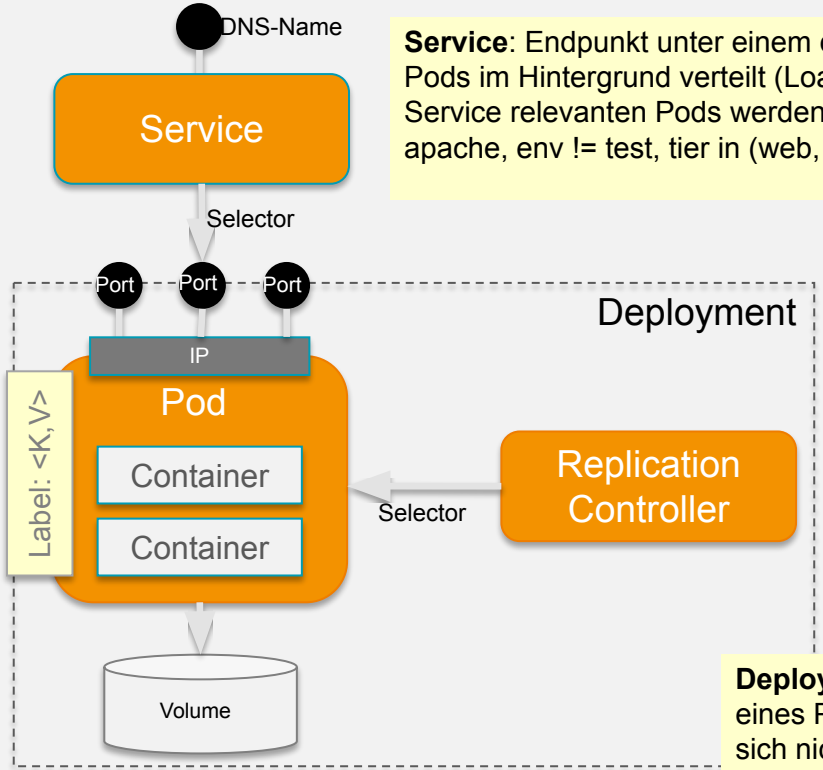


https://kubernetes.io/docs/concepts/cluster-administration/networking/

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.

Further concepts of Kubernetes

Applications:

- **StatefulSet**: When an application needs a state
- **Persistent Volumes**: Access to persistent storage

Global:

- **DaemonSet**: When a service has to run on every node in the cluster
- **Job**: When a task has to be executed regularly

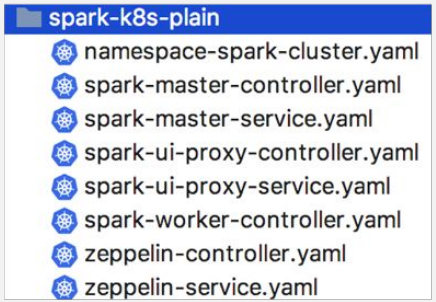
Security:

- **Network Policies**: Who can communicate with whom?

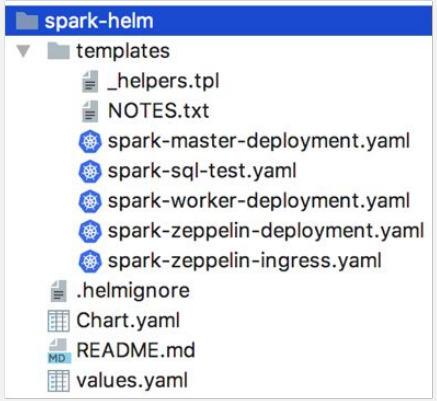
Helm: Application package management for Kubernetes.

QA|WARE

HELM https://helm.sh



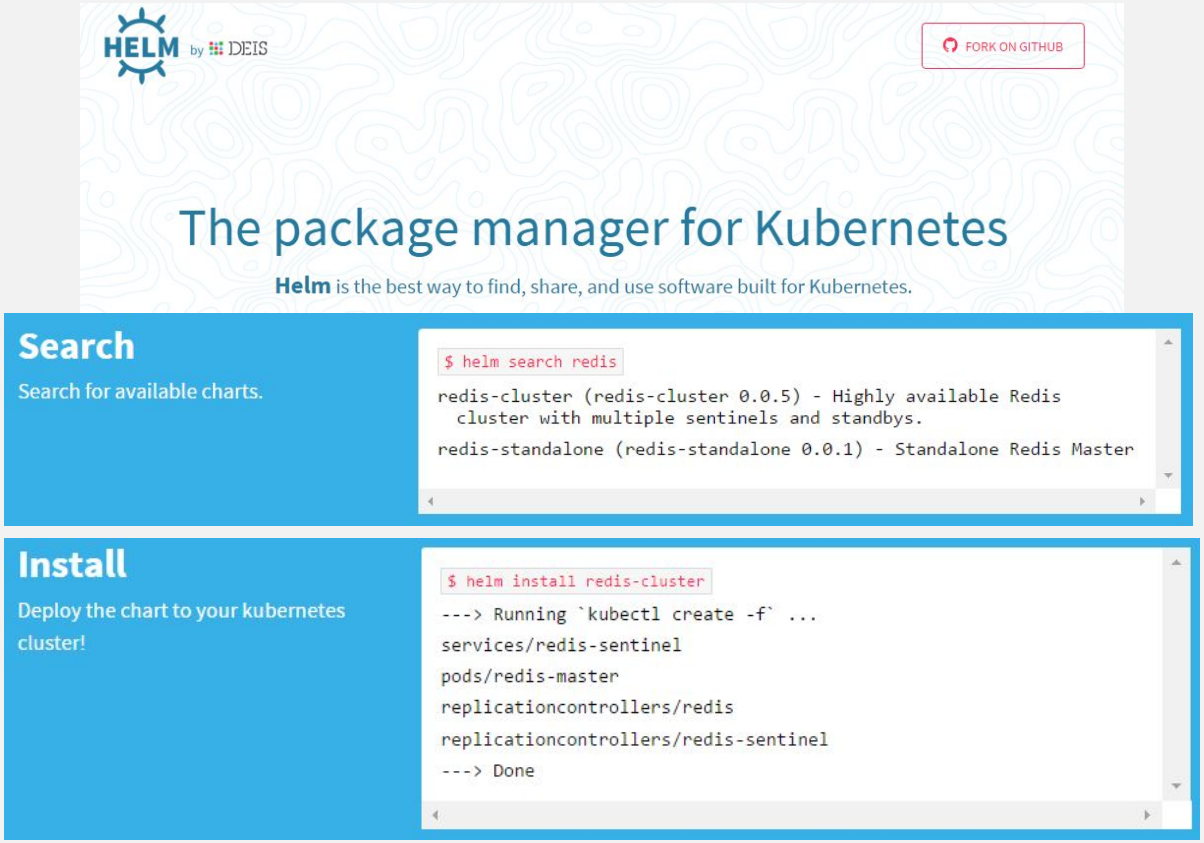
- `kubectl, kubectl, kubectl, ...`
- Konfiguration?
- Endpunkte?



- Chart suchen auf https://hub.kubeapps.com
- 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
```

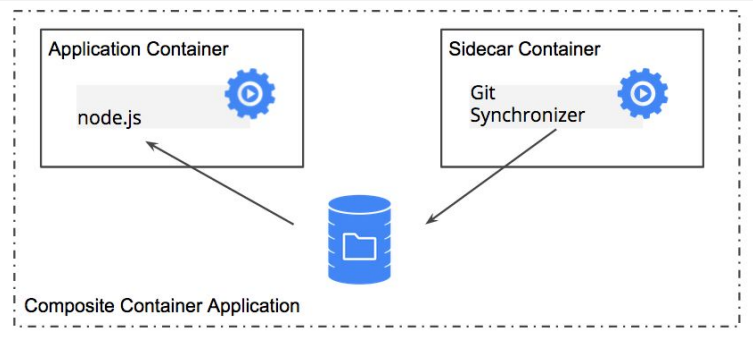
Helm: Application package management for Kubernetes.



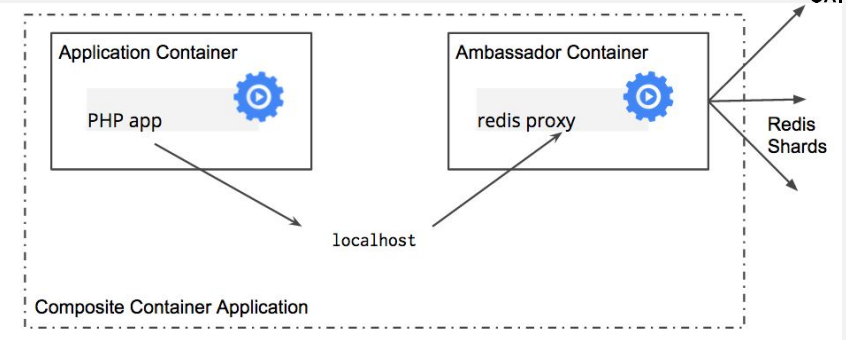
Source: https://github.com/helm/helm

Orchestration patterns

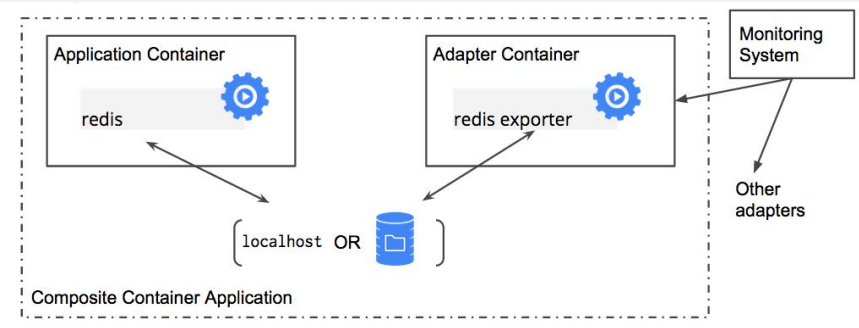
Orchestration patterns
Separation of Concerns mit modularen Containern



Sidecar Container



Ambassador Container



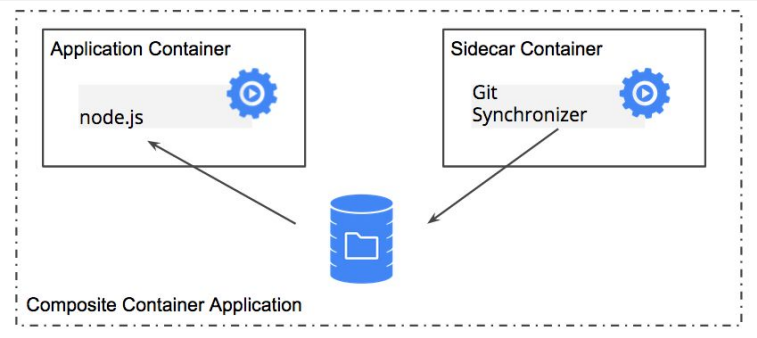
Adapter Container

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

Orchestration patterns
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.

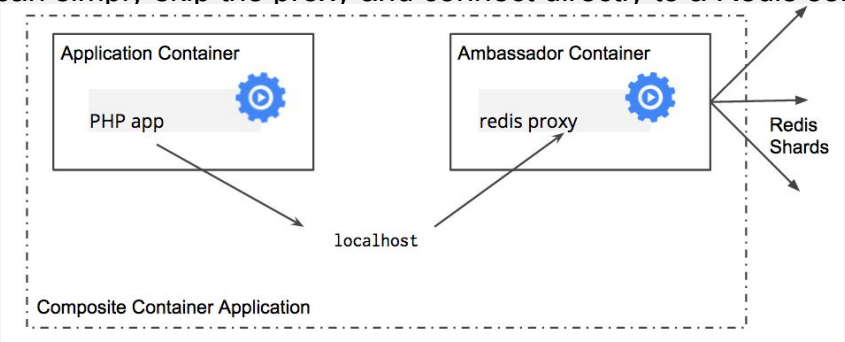


Quelle: https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/

Orchestration patterns
Ambassador containers

QA|WARE

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.

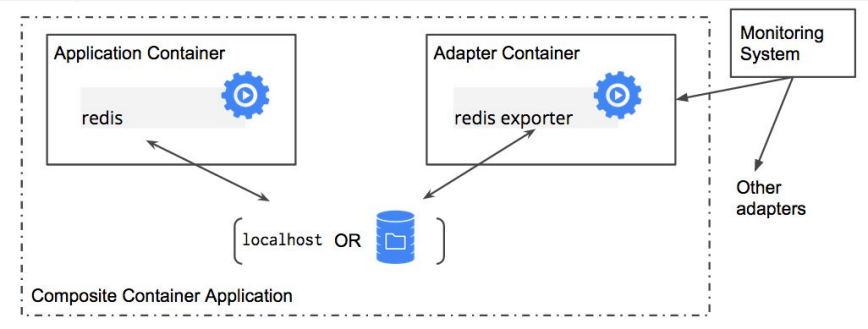


Quelle: https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/

Orchestration patterns
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/