ex|Xcellent solutions

eXXcellent solutions

MICROSERVICES IN KUBERNETES

Mario Akermann, Felix Rieß
Uni Stuttgart, 03.02.2020

the essence for your business.

Geplantes Vorgehen

- „Du“
- Fragen? → fragen!
- Zwei Pausen á 15 Minuten
- Vortrag besteht aus theoretischen & praktischen Teilen

Agenda

- **Vorstellung**
- Microservices
- Docker und Kubernetes
- Microservices in Kubernetes

Vorstellung

Mario Akermann

- **Software Engineer**
- **Ausbildung zum Mechatroniker (LAUFFER GmbH)**
- **Bachelor Medien- und Kommunikationsinformatik**
- **Schwerpunkte:**
  - Kubernetes
  - Cloud Services
  - CI/CD
  - DevOps
- **Neben der Arbeit: Darts spielen**

Vorstellung

Felix Rieß

- **Senior Software Engineer**
- **iSAQB Certified Professional for Software Architecture**
- **Schwerpunkte:**
  - Web-Anwendungen
  - Verteilte Anwendungen
  - Software-Entwicklung mit Java im Backend
- **Neben der Arbeit: Darts & Fußball**

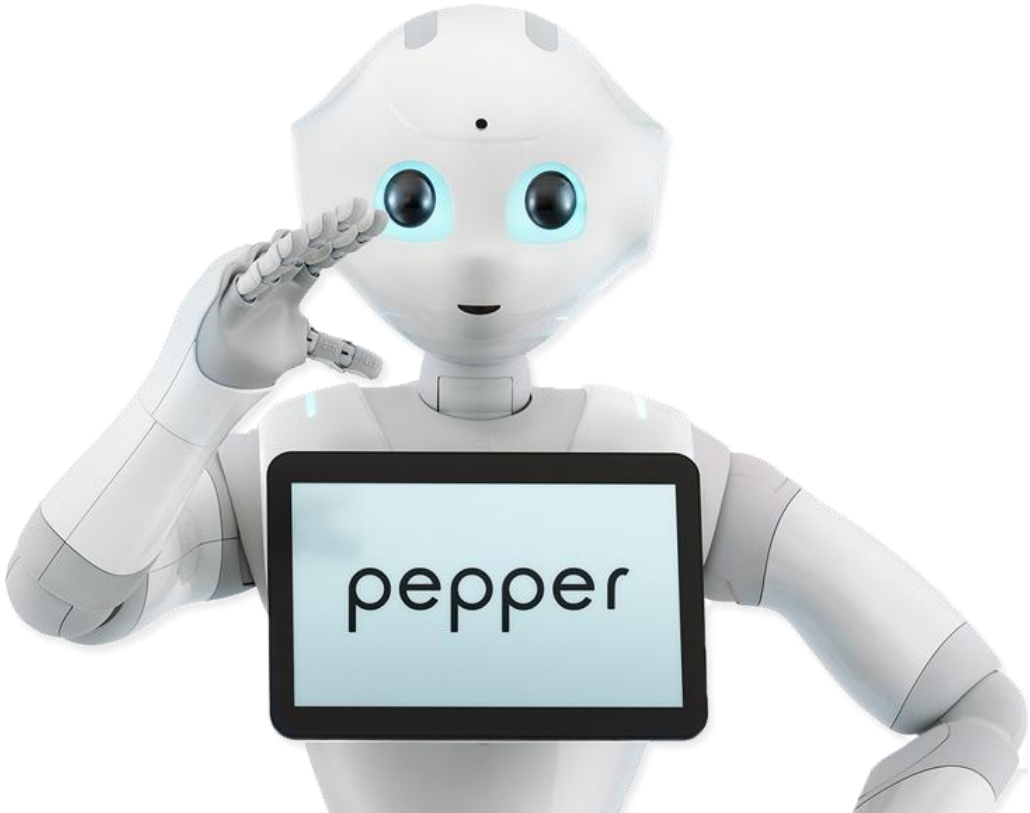
Vorstellung
eXXcellent solutions

Zahlen und Fakten

- 25,5 Mio € Umsatz
- 175 Mitarbeiter
- 5 Standorte & 2 Workspaces

Was tun wir?

- **Business- & Technology-Consulting**
- **Konzeption & Entwicklung individueller Softwarelösungen**
- Agile Vorgehensmodelle
- Microservice-Architekturen
- Cloud
- Full-stack Entwicklung (Frontend & Backend)
- Salesforce & SAP
- KI, Machine Learning, Data-Mining
- Service-Robotik



Hochschulthemen

- **Praktikums- und Werkstudentenstellen**
- **Gastvorträge, Vorlesungen und Workshops**
- Masterclass of Microservices
- Hybride Vorgehensmodelle: Workshop
- **Abschlussarbeiten, wie...**
- Distributed Tracing in Microservices-Architekturen
- Code-basierte Dokumentation evolutionärer Software-Architekturen
- Ein Vergleich von Skalierungsvarianten für Container mit Bezug auf REST-Frameworks, Softwarearchitektur und Kommunikation
- Ausarbeitung eines Konzepts zum ganzheitlichen Monitoring von Microservice-Anwendungen

Weitere Kunden

Auszeichnungen

Great Place to Work

Zum wiederholten Mal

Great Place To Work. 2019 Beste Arbeitgeber ITK

Great Place To Work. 2019 Beste Arbeitgeber Bayern

Platz 1 im Ranking „Beste Chefs“

kununu

Platz 1 & 2 der besten Arbeitgeber IT-Mittelstand D

Focus Business & kununu

Platz 2 & 3 der Familienfreundlichsten Unternehmen D

Freundin & kununu

Agenda

Vorstellung

Microservices

Docker und Kubernetes

Microservices in Kubernetes

Netflix als Pionier

2009

Begin Refactoring

Netflix began moving from a monolithic to AWS cloud-based microservices architecture in 2009, long before the term microservices even existed.

2011

Refactoring Complete

In December 2011, each component was moved to the cloud, breaking up their monolith into hundreds of fine-grained microservices. This process took more than 2 years.

2014

Term Microservices

The term "microservices" gains popularity with software developer and author Martin Fowler.

2018

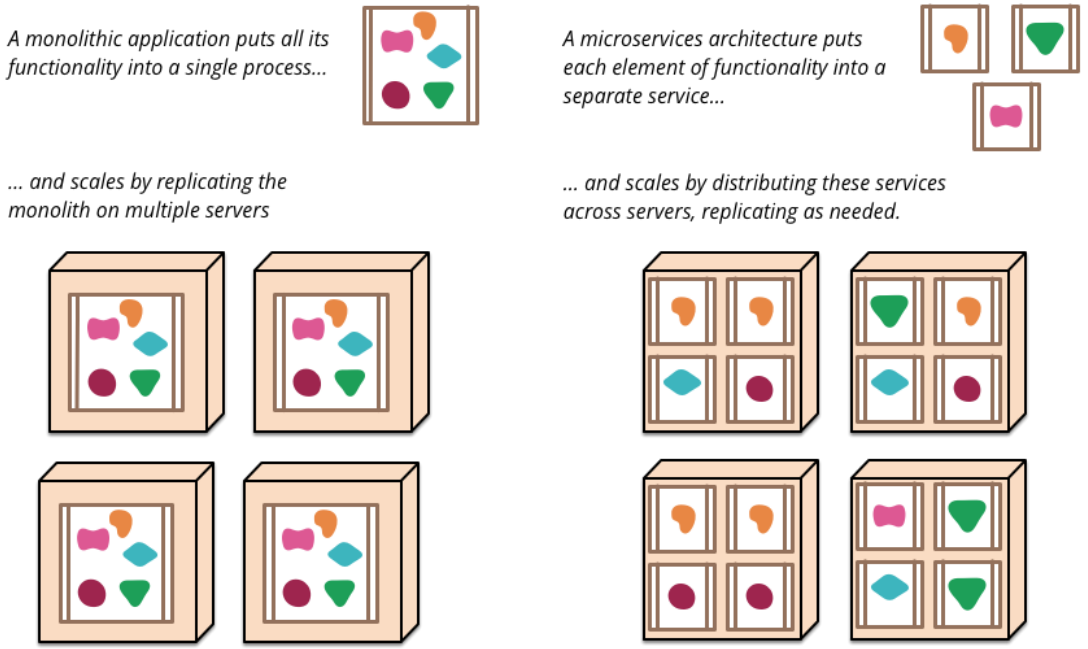
Growth around microservices

Frameworks and systems (e.g. Kubernetes) exist to support the development of the microservice architecture design pattern, the deployment and the management during runtime.

Was ist ein Microservice?

Grobe Faustregeln:

- **Ein Service kann einfach ersetzt werden**
- Umfang ist überschaubar
- Vom zuständigen Team (5-7 Entwickler) mit vertretbarem Zeitaufwand ersetzbar (z.B. innerhalb eines Monats)
- Größe nach unten durch potentiell ressourcenintensive Netzwerkkommunikation und eigene Deployments pro Service begrenzt
- **Ein Service wird von einem Team entwickelt**
- **Ein Team kann mehrere Services entwickeln**
- **Jeder Service kann unabhängig von anderen Services in Produktion gebracht werden**



Was ist eine Microservice Architektur?

Microservices sind ein Architekturmuster, bei dem komplexe Anwendungssoftware aus kleinen, unabhängigen Prozessen komponiert wird, die untereinander mit sprachunabhängigen Programmierschnittstellen kommunizieren.

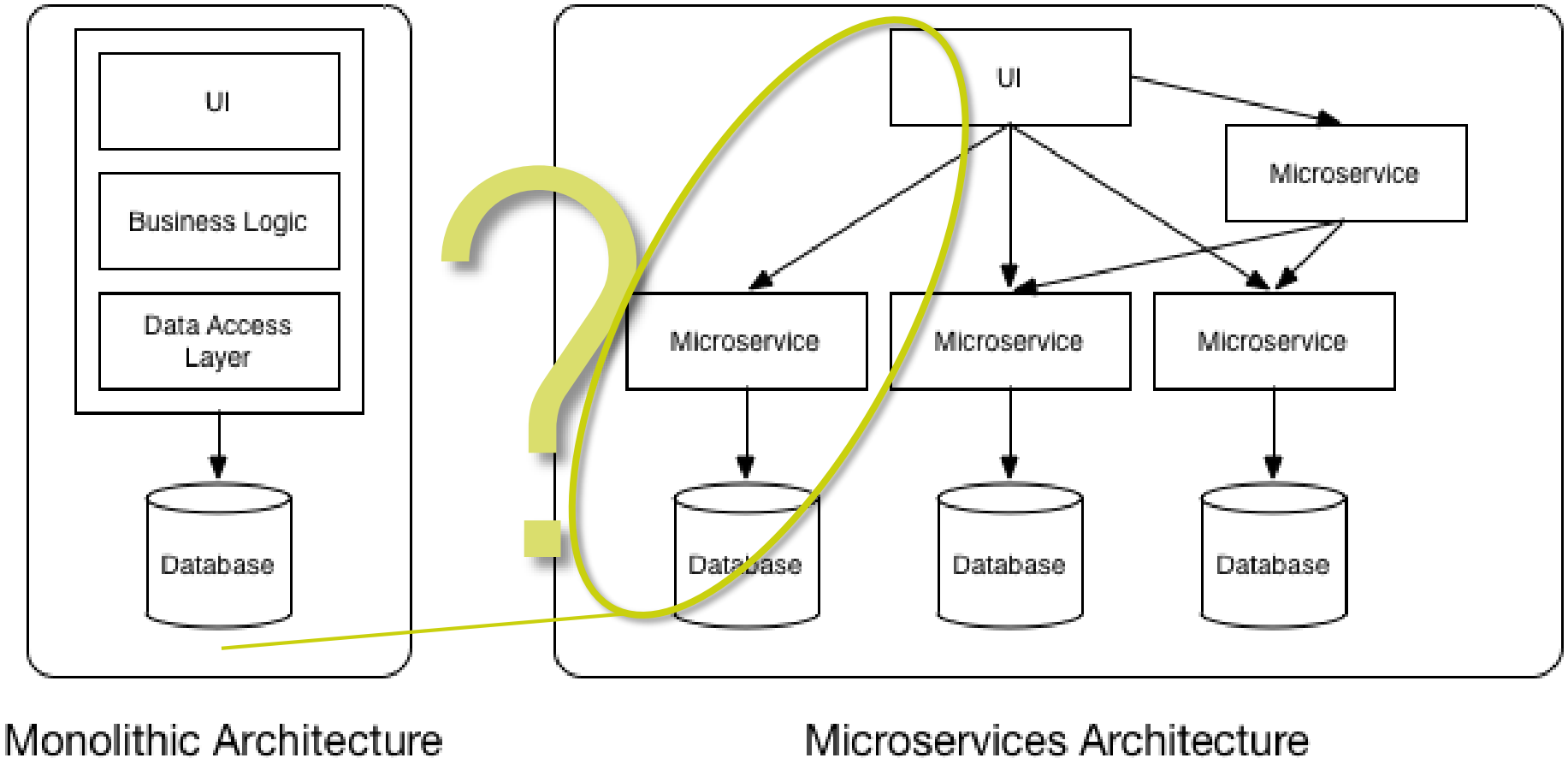
Ziele:

- Erhöhung der Wandelbarkeit und Flexibilität der Software
- Verbesserung der Time-to-market
- Schnellere Entwicklung durch kleinere Einheiten
- Mehr Innovationsfähigkeit sowie Flexibilität bezüglich Technologieauswahl
- Eigene Ablaufumgebung für jeden Microservice

→ Die Dienste sind klein, weitgehend entkoppelt und erledigen eine kleine Aufgabe

→ „Do one thing and do it well“

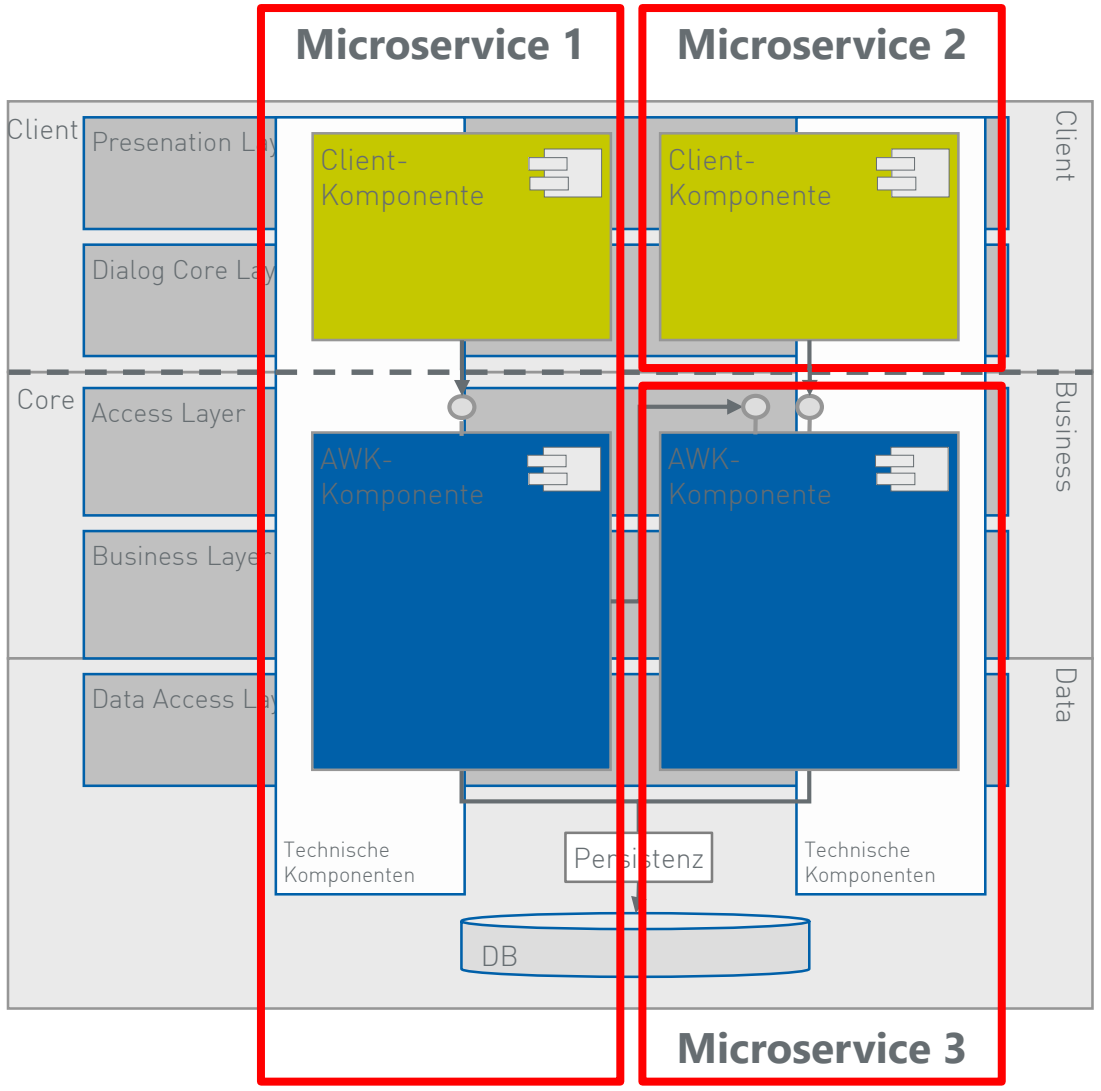
Keine Äpfel mit Birnen vergleichen



Quelle: https://developer.ibm.com/bluemix/wp-content/uploads/sites/20/2015/01/microvsmono.png

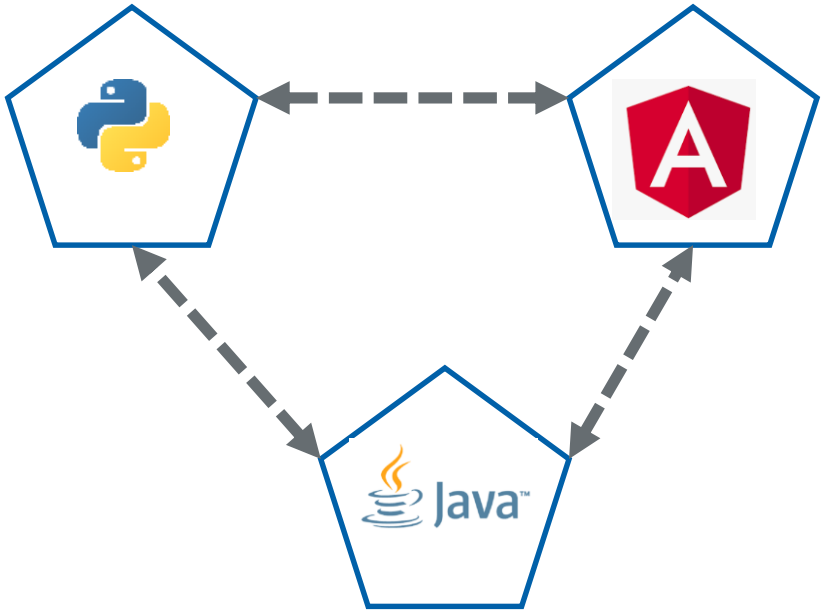
Vergleich zur Komponentenarchitektur

- **Microservices zerlegen ein System auf der Ebene von fachlichen Komponenten**
- Technische Komponenten sind jeweils in eigener Hoheit eines Microservice
- Ein Microservice kann auch nur UI oder Backend enthalten
- **Jeder Microservice ist ein eigenständiger Prozess**
- Kommunikation erfolgt über sprachunabhängige Schnittstellen
- Fachlicher Umfang orientiert sich am Bounded Context aus dem Domain Driven Design



Vorteile

- Sprachunabhängig (Java, Python, Golang,...)
- Können unabhängig voneinander (weiter-) entwickelt werden
- Können unabhängig voneinander getestet werden
- Isolation von Laufzeitfehlern
- Wiederverwendbarkeit in unterschiedlichen Kontexten
- Können separat skaliert werden
- Können bzgl. CPU und RAM optimal konfiguriert werden
- Abhängigkeiten zu Fremdbibliotheken sind „maßgeschneidert“



Nachteile von verteilten Anwendungen

Eine verteilte Architektur erzeugt zusätzliche Komplexität, vor allem für Lösungen im Umgang mit:

- Netzwerk-Latenzen
- Kommunikationsausfall zwischen Services
- Lastverteilung zwischen Services
- Fehlertoleranz der Services
- Datenverteilung und eventually consistence

Zu feingranulare Microservices (Nanoservices) werden als anti-Pattern angesehen:

- Overhead (Kommunikation, Wartung, etc.) ist größer als der Nutzen
- Probleme manifestieren sich in Form von Code-Overhead (Schnittstellen, Umgang mit Verbindungsproblemen), Laufzeit-Overhead (Serialisierung/Deserialisierung, Netzwerkverkehr) und fragmentierter Logik (zusammenhängende Funktionalität nicht an einem Ort, sondern über viele Services kombiniert)

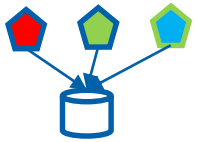
Die komplexen Abhängigkeiten eines Deployment-Monolithen verschwinden nicht, sondern existieren auf der Netzwerk-Ebene weiter.

Nachteile einer großen Anzahl an Microservices

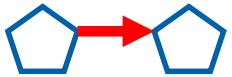
- **Vielzahl an Services macht Softwareverteilung/Deployments und das Testen komplexer**
- Logische Deploy-Abhängigkeit zwischen Services
- Anzahl an Build/Deploy Pipelines muss verwaltet und überwacht werden
- **Verschieben von fachlicher Funktionalität zwischen Microservices ist schwieriger**
- Definition neuer Schnittstellen
- Neue Vereinbarungen zwischen Teams
- Änderungen beim Transaktionsverhalten
- **Wildwuchs und Verlust der Homogenität**
- Unterschiedliches „internes Design“ und Wildwuchs an Sprachen, Bibliotheken, etc. erschweren die Wartbarkeit
- „Der verteilte Monolith“: Wiederverwendung von Code durch eine Common-Lib führt zur engen Kopplung

Es müssen mehr Aufwände in die Themen Automatisierung, Homogenität und Refactoring investiert werden.

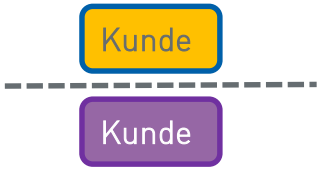
Herausforderungen



Data Ownership



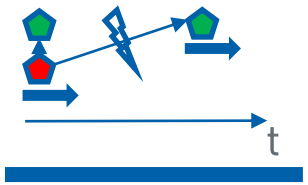
Netzwerk
Infrastruktur



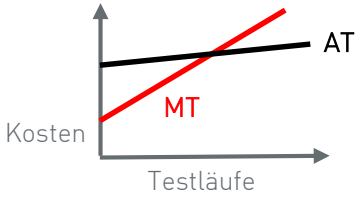
Bounded
Context

V1 vs. V1.1

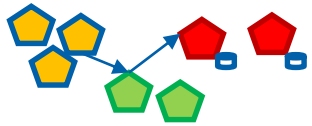
Verlässlichkeit



Unabhängige
Entwicklung



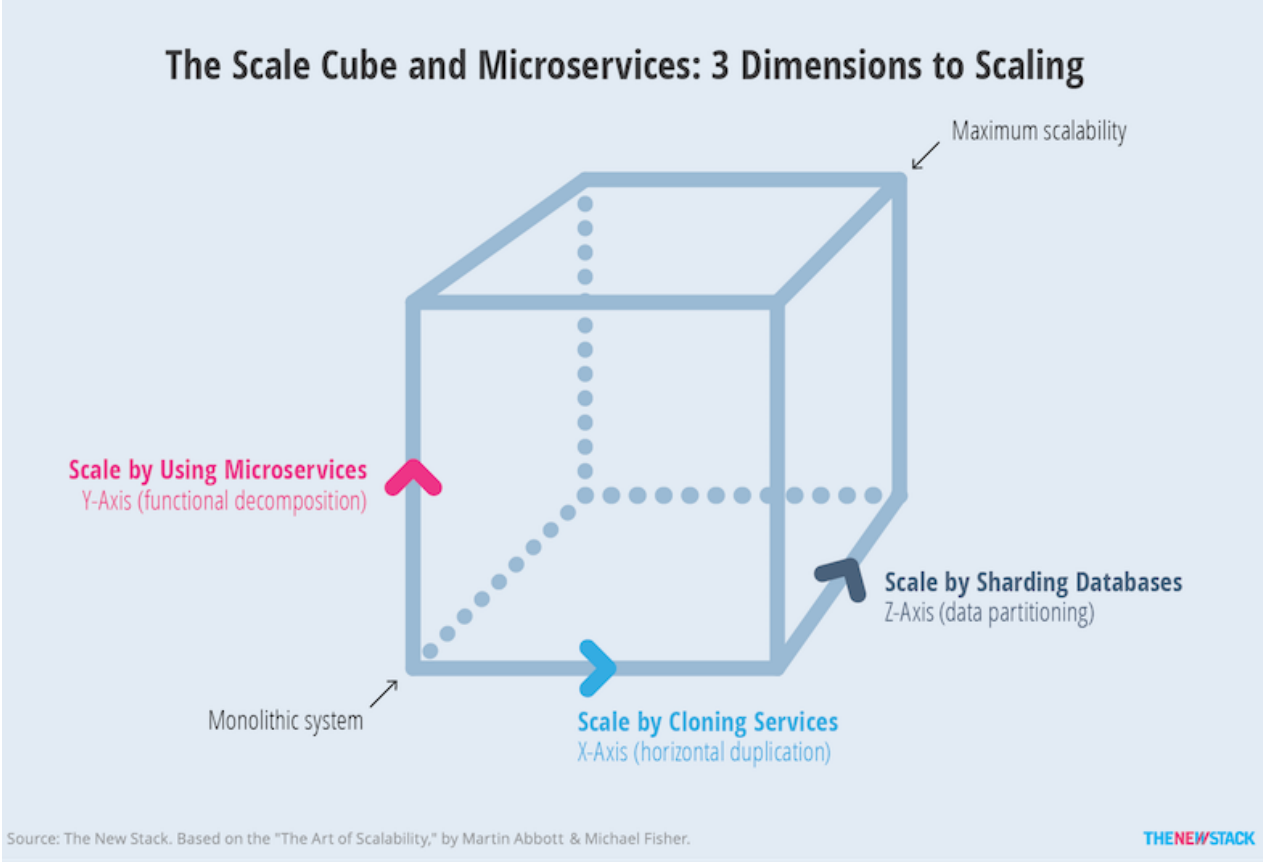
Test & Depl.
Automatisierung



Logging

Wie können Microservices skaliert werden?

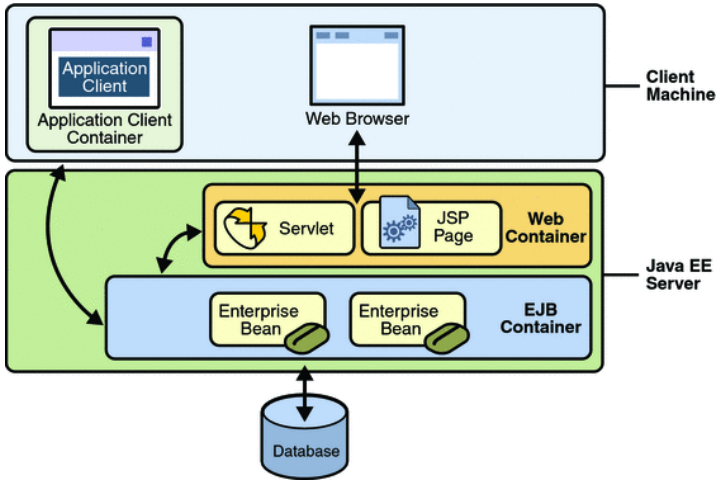
- X-Achse: klassisches load balancing
- Y-Achse: funktionale Dekomposition
- Z-Achse: z.B. Zerlegung nach Regionen



Ein Microservice ist noch kein Container

Ein Microservice ist ausführbarer Code, definiert aber nicht, wie und wo er laufen wird

- Klassische JEE Architekturen packen Code in JAR, WAR oder EAR
- JEE Laufzeitumgebung erzeugt Container (Web- oder auch EJB-Container), in diesen wird der Code ausgeführt
- JEE Container stellen Ablaufkontext bereit und definieren Lebenszyklus

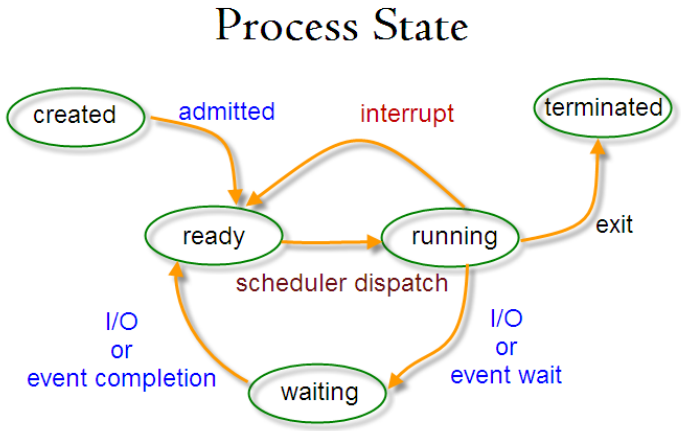


Im einfachsten Fall kann ein Microservice direkt auf einem OS per Kommandozeile gestartet werden

- Eigener Prozessraum durch OS sichergestellt

ABER:

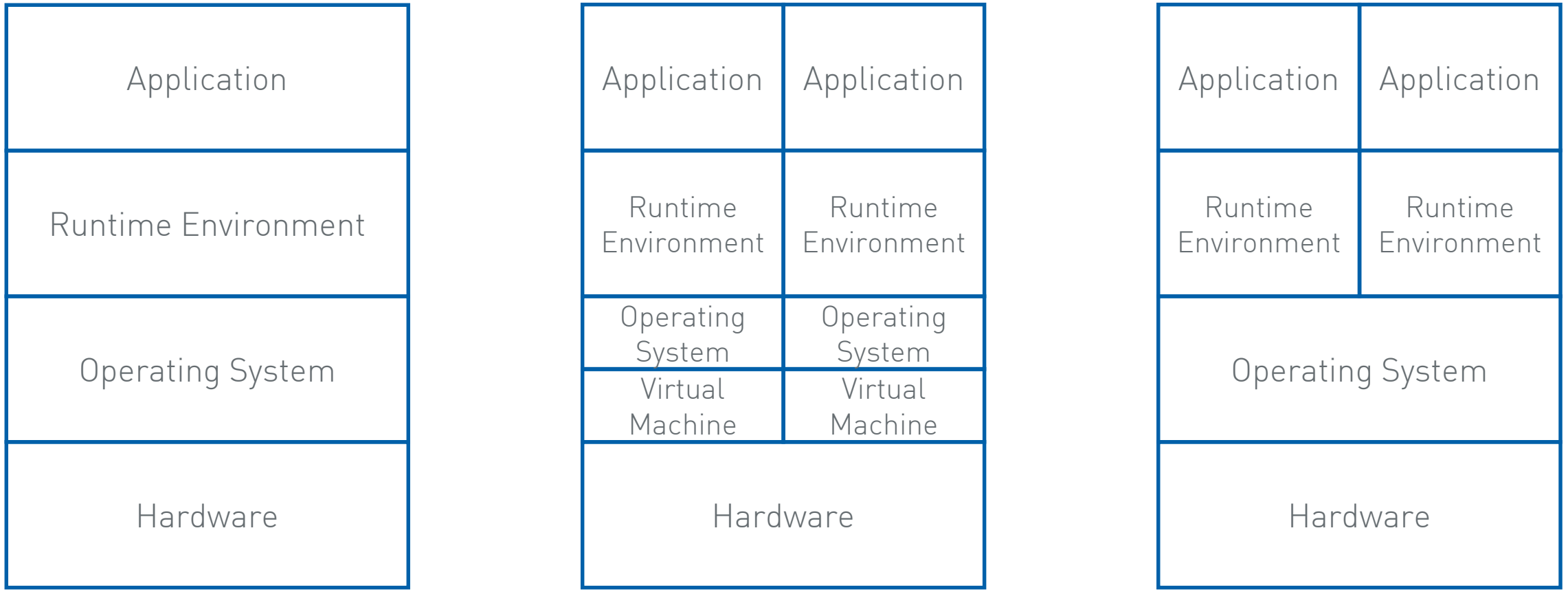
- Steht in Konkurrenz zu anderen Prozessen für RAM/CPU/IO
- Kontext kann nicht leicht unabhängig konfiguriert werden



Anforderungen für Microservices

- **Jeder Microservice läuft unabhängig von anderen Services**
- RAM- und CPU-Verbrauch
- Netzwerkkommunikation und I/O Nutzung
- Weitere Ressourcen (persistenter Speicher, Zertifikate, Bibliotheken, usw.)
- **Jeder Microservice ist eigenständig deploy- und betreibbar**
- Im einfachsten Fall als Systemprozess
- Konfiguration (z.B. Environment, Netzwerk) unabhängig von anderen Services
- Fehlerverhalten (Abstürze) bleiben begrenzt auf einen Service
- **Jeder Microservice ist separat skalierbar**
- Schnelles starten/stoppen von Services
- Keine Seiteneffekte auf andere Services

Container als Ablaufumgebung



Warum passen Docker und Microservices so gut zusammen?

State Of The Art

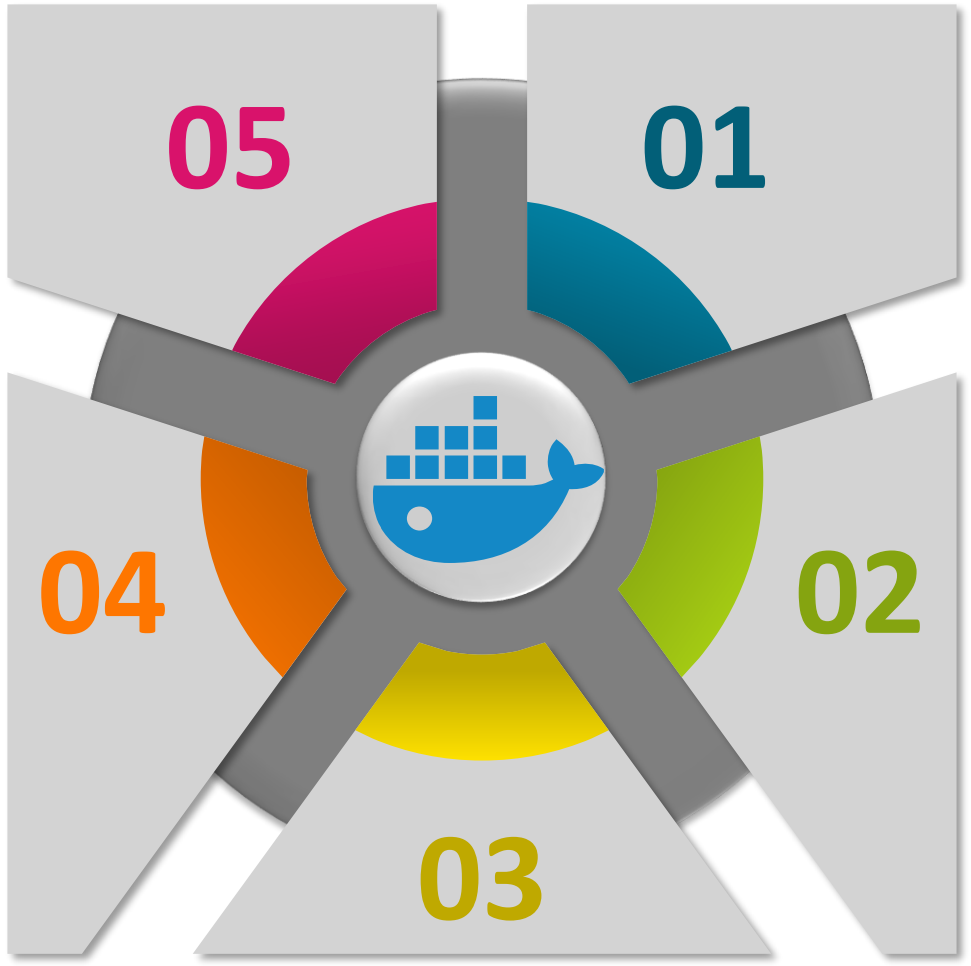
More than 2 million dockerized application in the hub and more than 50 billion container downloads. The docker user community has grown huge and so the evolution of this technology is saved.

Security

Applications are safer in containers and Docker provides great isolation capabilities. Containers can provide user environments whose resource requirements can be strictly controlled.

Resource Utilization

Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs.



Deployment

Microservices can be deployed independendly by Docker. A Deployment is a simple replacement of one docker container by another. If correctly managed, there is no downtime.

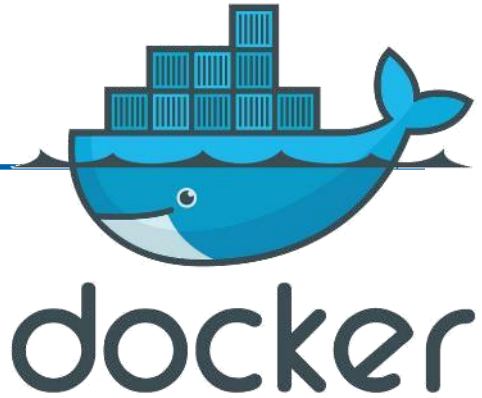
Application Portability

Docker puts application and all of its dependencies into a container which is portable among different platforms, Linux distributions and clouds. Each container is self contained.

Agenda

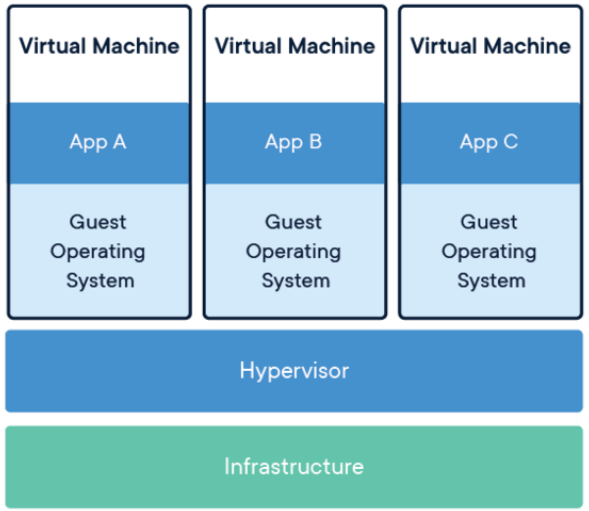

Vorstellung
Microservices
Docker und Kubernetes
Microservices in Kubernetes

Docker

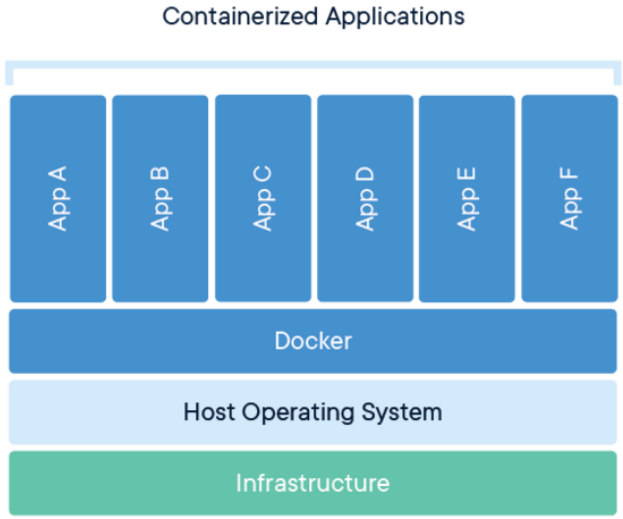


Docker is a computer program that performs **operating-system-level virtualization**, also known as "containerization".

Operating-system-level virtualization, also known as containerization, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances.

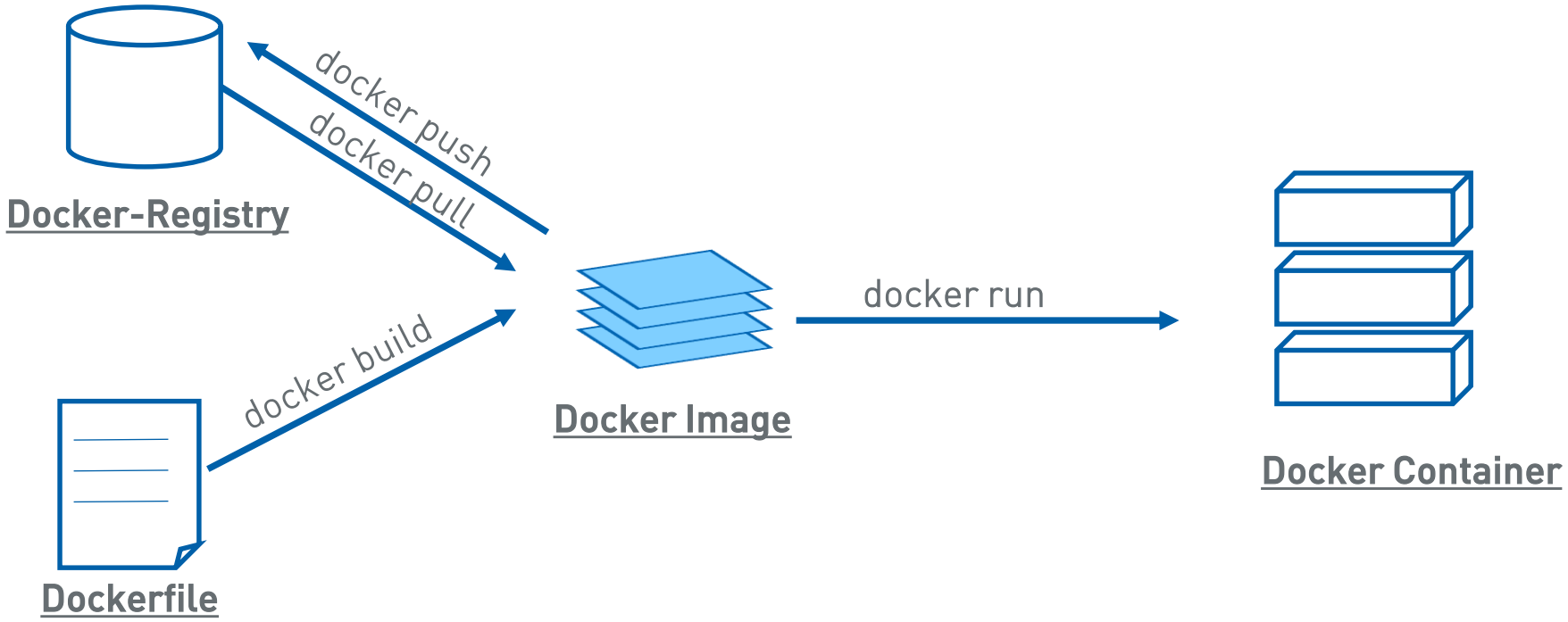


Bisher

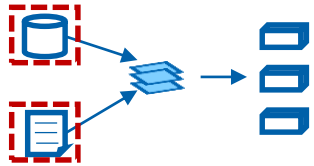


Heute

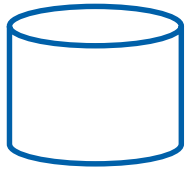
Überblick von Docker Befehlen



Wichtige Begriffe

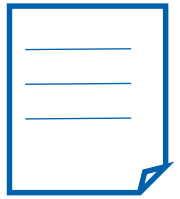


Docker Registry



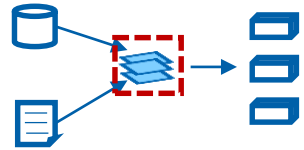
- Dient dem Speichern und Verteilen von Docker Images.
- Ermöglicht die Umsetzung eines „docker from the shelf“ Paradigmas.
- Die bekannteste öffentliche Docker-Registry ist https://hub.docker.com.

Dockerfile

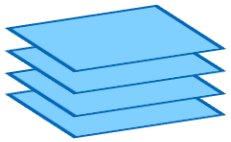


- Einfache Textdatei mit dem Namen: „*Dockerfile*“
- Ausgehend vom Basisimage, beschreibt das Dockerfile die notwendigen Schritte zum Aufsetzen der Laufzeitumgebung.
- Kopiert generell das statische Applikations-Binary in das Image.
- Definiert Volumes (Persistenter Speicher), Ports und Umgebungsvariables der Laufzeitumgebung.

Wichtige Begriffe



Docker Image



- Entsteht als Resultat von der Durchführung der Instruktionen im Dockerfile.
- Jede Anweisung innerhalb des Dockerfile erstellt einen eigenen Layer.
- Bei Änderung des Dockerfile werden alle darüber liegenden Layer neu gebaut.
- Via „*docker image history*“ kann man sich die Layer eines Images anzeigen
- Besitzt eine Versionstag, andernfalls wird ‚latest‘ hinzugefügt.

Dockerfile

Java without Docker

(RUN: java -jar app.jar)

Jar to Docker

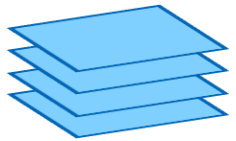
```
1   FROM openjdk:8
2
3   RUN mkdir -p /opt/app
4
5   COPY ./target/app.jar /opt/app
6
7   COPY ./dropwizard.yml /opt/app
8
9   WORKDIR /opt/app
10
11  EXPOSE 3004
12
13  ENV NOTIFY_MAIL true
14  ENV NOTIFY_AT_START true
15
16  ENTRYPOINT ["java", "-jar", "app.jar", "server", "dropwizard.yml"]
```

Dockerfile

Jar to Docker

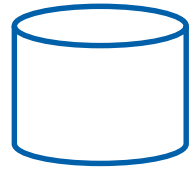
```
1   FROM openjdk:8
2
3   RUN mkdir -p /opt/app
4
5   COPY ./target/app.jar /opt/app
6
7   COPY ./dropwizard.yml /opt/app
8
9   WORKDIR /opt/app
10
11  EXPOSE 3004
12
13  ENV NOTIFY_MAIL true
14  ENV NOTIFY_AT_START true
15
16  ENTRYPOINT ["java", "-jar", "app.jar", "server", "dropwizard.yml"]
```

$ Docker build →



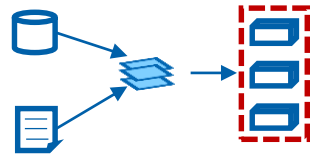
Docker Image

↑ $ Docker push



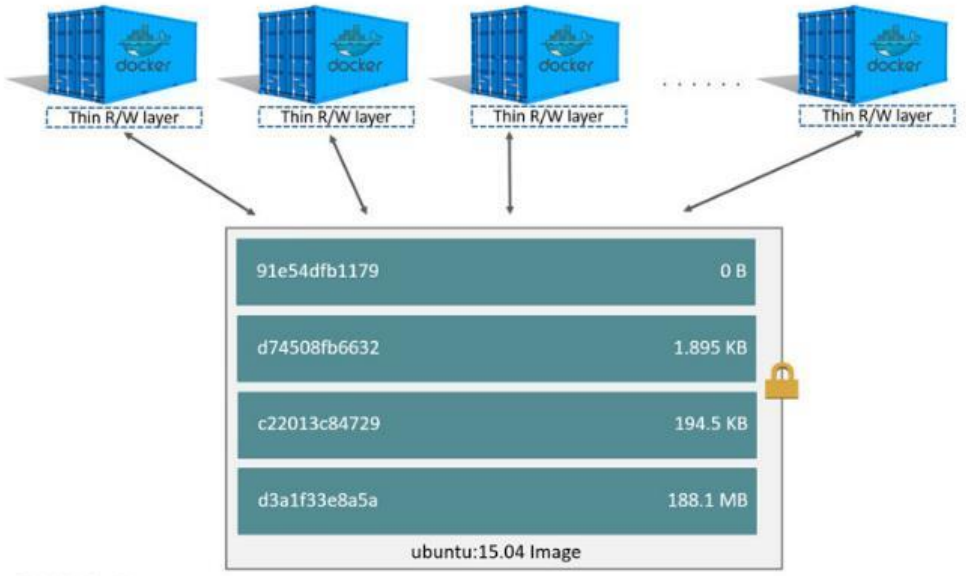
Docker Registry

Wichtige Begriffe

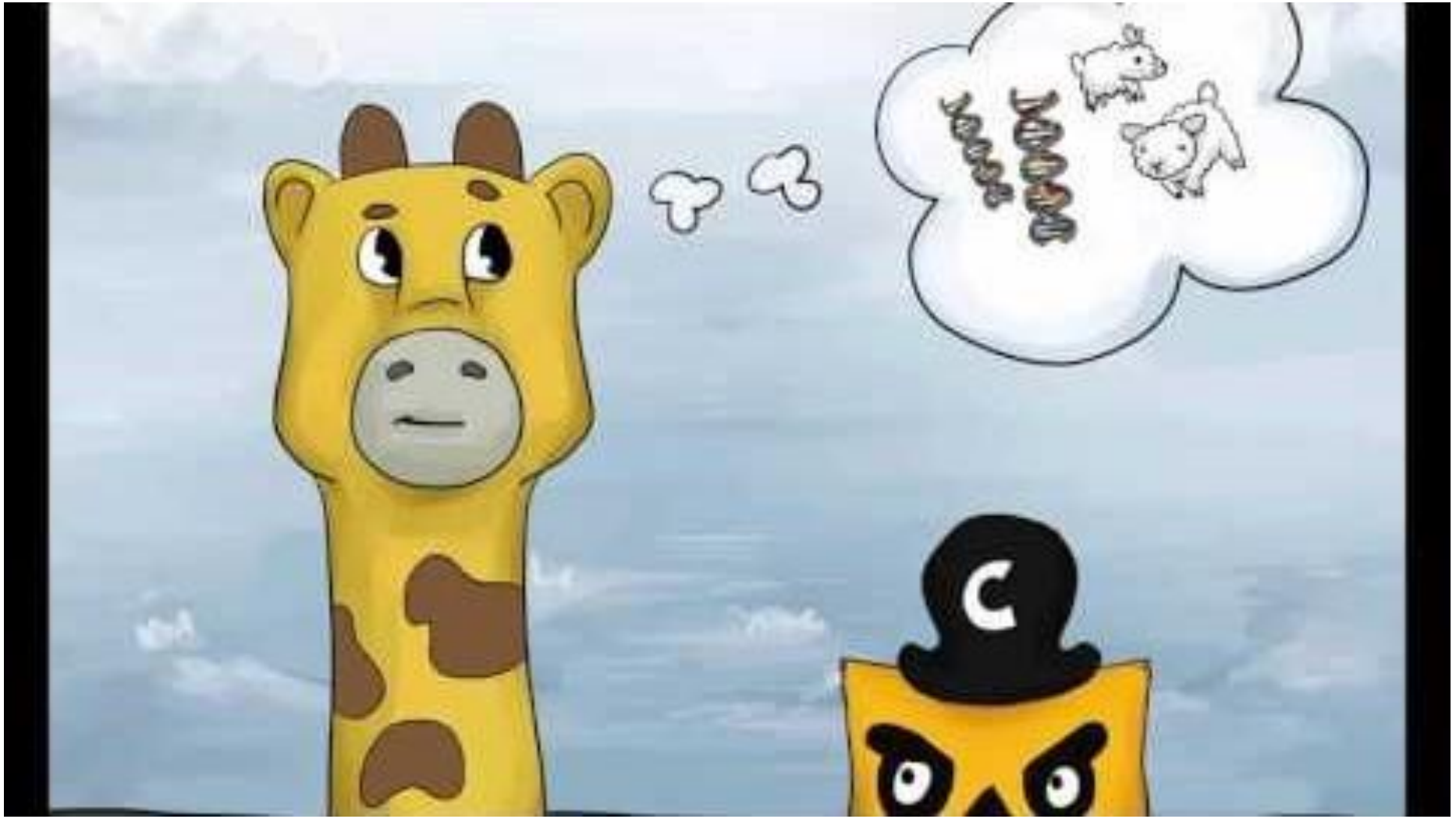


Docker Container

- Entspricht der „Instanziierung“ eines Dockerimages.
- Zur Laufzeit wird für jeden Container ein ‚writable layer‘ hinzugefügt in den Änderungen gemäß des „Copy-On-Write“ Prinzip abgebildet werden.
- Alle Änderungen im writable Layer gehen nach dem Beenden des Container verloren.



The illustrated children‘s guide to Kubernetes



Quelle: https://www.youtube.com/watch?v=4ht22ReBjno

Kubernetes

Was ist Kubernetes

“Kubernetes (K8s) is an open-source **container orchestration** platform. Container orchestration means that Kubernetes takes care of the **deployment**, **scaling** and **management** of containerized applications.”

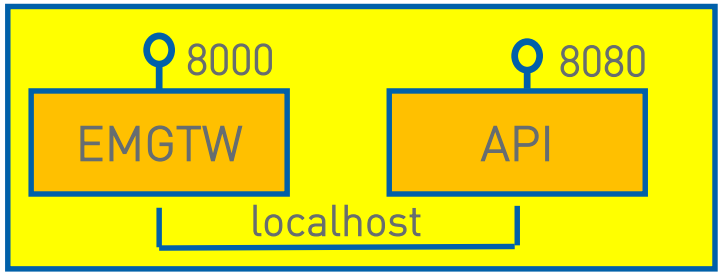
Wobei hilft Kubernetes

- Automatisches Starten, Neustarten und Stoppen von Containern
- Rolling-Update (Deployen eines neuen Containers ohne Downtime)
- Transparente Replikation
- Abstraktion der Bare-Metal Hardware zum Docker Deamon

Grundlagen K8s

Pod

- Atomare Arbeitseinheit von Kubernetes
- Besteht aus einem oder mehreren Containern
- Container in einem Pod teilen sich das Netzwerkinterface und das Filesystem



Pod

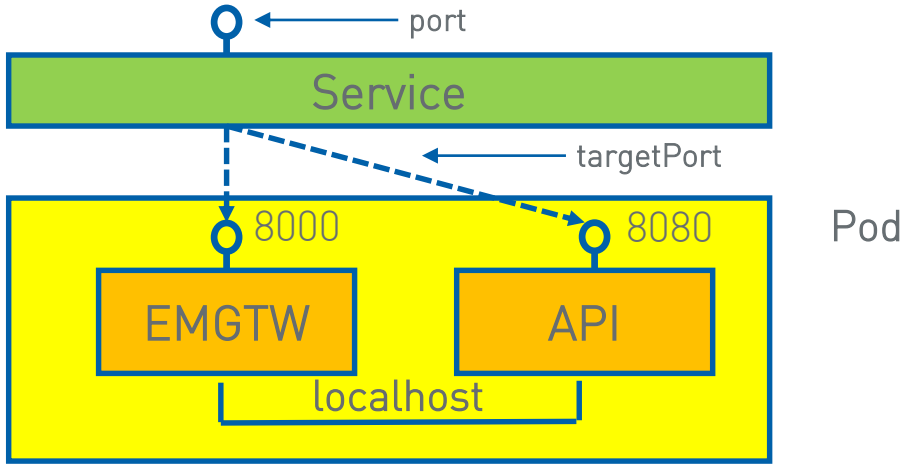
Deployment

- Deklarative Beschreibung (Blaupause) eines Pod
- Spezifiziert die Zusammensetzung des Pods, das Replicaset sowie weitere Metadaten
- Kubernetes sorgt selbstständig für die Einhaltung des Zielzustandes

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: MyApp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
        ports:
        - containerPort: 80
```

Grundlagen k8s

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  # Name des Pods
  name: country-app-svc-deployment
  # Name des Namespaces
  namespace: atum
  # Label zur Identifikation des Pods via Key-Value Paar
  labels:
    service: country-app_api_service
spec:
  # Anzahl wie oft der Pod repliziert wird
  replicas: 1
  selector:
    matchLabels:
      name: country-app-api-service-selector
  template:
    metadata:
      labels:
        name: country-app-api-service-selector
    spec:
      containers:
        - name: country-app-api
          # image welches für den Container geladen wird
          image: exxcellent/cps-country-app-service:atum
          # wann soll das Image gepullt werden
          imagePullPolicy: Always
          ports:
          # geöffneter Port des Containers
          - containerPort: 80
          securityContext:
          # environment variablen aus Configmap laden
          envFrom:
          - configMapRef:
              name: country-app-svc-configmap
```



Service

- Nach außen sichtbare Abstraktion eines oder mehrerer Pods
- Fokus auf OSI-Layer 4 Routing
- Load-Balancing für eingehende Aufrufe
- Wird über Label Selector mit Pod verknüpft
- Ein Service hat folgende Formen der Sichtbarkeit: ClusterIP, NodePort, LoadBalancer

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

Grundlagen k8s

```yaml
apiVersion: v1
kind: Service
metadata:
  name: country-app-svc-service
  namespace: atum
spec:
  type: ClusterIP
  ports:
    - port: 8484
      targetPort: 80
      name: api
      protocol: TCP
  selector:
    name: country-app-api-service-selector
```

ConfigMap

- ConfigMaps erlauben es Dateien oder Umgebungsvariablen zur Laufzeit bereitzustellen.
- Um eine ConfigMap in einem Pod zu verwenden muss es als Volume und VolumeMount definiert werden.

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
data:
  # Configuration values can be set as key-value properties
  database: mongodb
  database_uri: mongodb://localhost:27017

  # Or set as complete file contents (even JSON!)
  keys: |
    image.public.key=771
    rsa.public.key=42
```

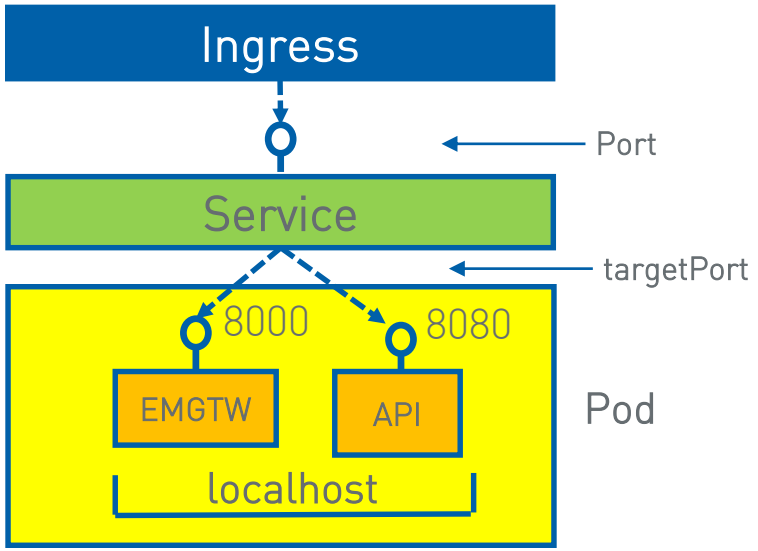
```yaml
kind: Pod
apiVersion: v1
metadata:
  name: pod-using-configmap
spec:
  # Add the ConfigMap as a volume to the Pod
  volumes:
    # `name` here must match the name
    # specified in the volume mount
    - name: example-configmap-volume
      # Populate the volume with config map data
      configMap:
        # `name` here must match the name
        # specified in the ConfigMap's YAML
        name: example-configmap
  containers:
    - name: container-configmap
      image: nginx:1.7.9
      # Mount the volume that contains the configuration data
      # into your container filesystem
      volumeMounts:
        # `name` here must match the name
        # from the volumes section of this pod
        - name: example-configmap-volume
          mountPath: /etc/config
```

Grundlagen

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  # Name der ConfigMap
  name: country-app-svc-configmap
  namespace: atum
data:
  # Environment Variable
  COUNTRY_SERVICE_URL: "http://ae8333ccd462511eaba7d0af1187e991-581492011.eu-central-1.elb.amazonaws.com:8081/atum-cs"
  CURRENCY_SERVICE_URL: "http://ae8333ccd462511eaba7d0af1187e991-581492011.eu-central-1.elb.amazonaws.com:8081/atum-cu"
  LANGUAGE_SERVICE_URL: "http://ae8333ccd462511eaba7d0af1187e991-581492011.eu-central-1.elb.amazonaws.com:8081/atum-ls"
```

Ingress

- Zuordnung eines Service auf einen Hostnamen und einen Pfad
- Fokus auf OSI-Layer 7 Routing
- Konkreter Pfad wird mit einem Service verknüpft
- Neben http wird tls unterstützt
- Ingresses werden von Ingress-Controllern ausgeführt, die unterschiedliche Umfänge haben (bspw. traefik, nginx, Contour)

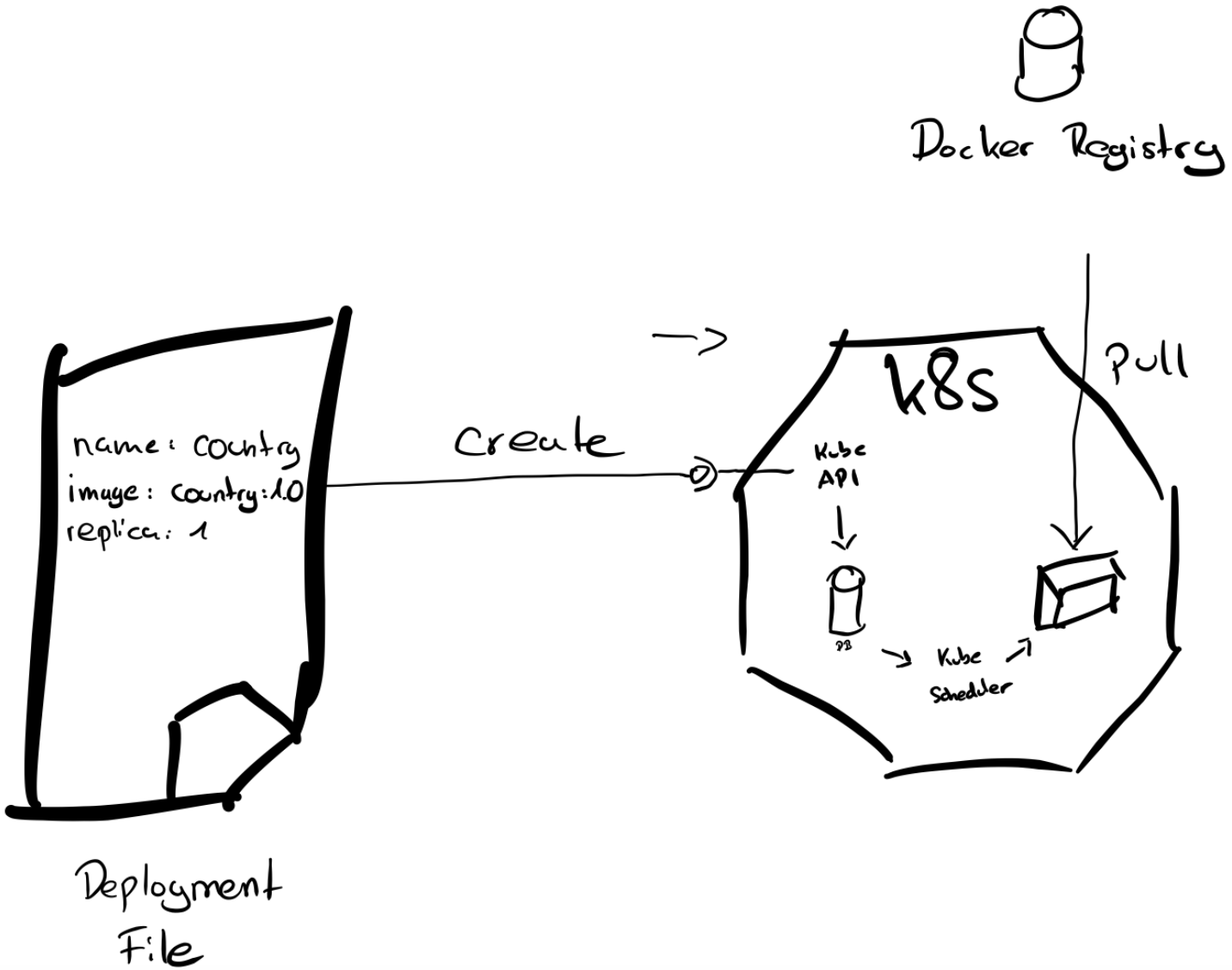



```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
 spec:
   rules:
   - host: foo.bar.com
     http:
       paths:
       - path: /foo
         backend:
           serviceName: my-service
           servicePort: 80
```

Grundlagen k8s

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  # Name der Ingress-Route
  name: country-app-svc-ingress
  # Namespace der Ingress-Route
  namespace: atum
  annotations:
    kubernetes.io/ingress.class: traefik
    # Pfad weiterleiten oder nicht
    #traefik.ingress.kubernetes.io/rule-type: PathPrefixStrip
spec:
  rules:
  - host:
    http:
      paths:
      # Pfad welcher zum angegebenen Service weiterleitet
      - path: /atum/
        backend:
          serviceName: country-app-svc-service
          servicePort: 8484
```

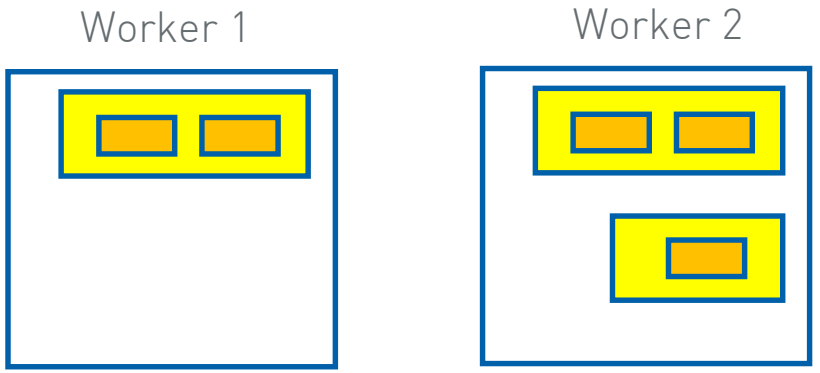
Deployment


Docker Registry
name: country
image: country:1.0
replica: 1
Create
k8s
Pull
Kube API
Kube Scheduler
Deployment File

Grundlagen K8s

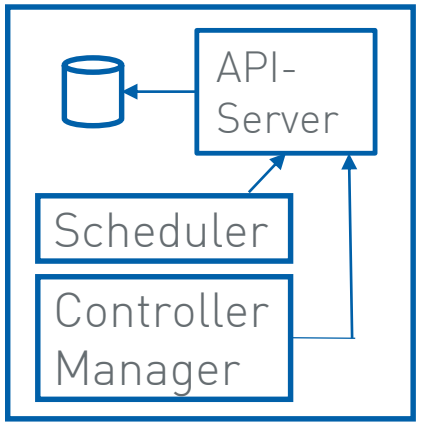
Worker Node

- Abstraktion von bare-metal Hardware
- In einem k8s Cluster befinden sich 1:n Worker Nodes
- Kubernetes entscheidet basierend auf der aktuellen Auslastung auf welcher Worker Node ein Pod ausgeführt wird



Master Node

- Übernimmt das Scheduling und Monitoring der Worker Nodes
- kubectl ist ein Interface, das mit der API der Master Node kommuniziert



Agenda


Vorstellung
Microservices
Docker und Kubernetes
Microservices in Kubernetes

Macht mal Microservices & Kubernetes!

Praktisches Beispiel: Country-Provider-Service

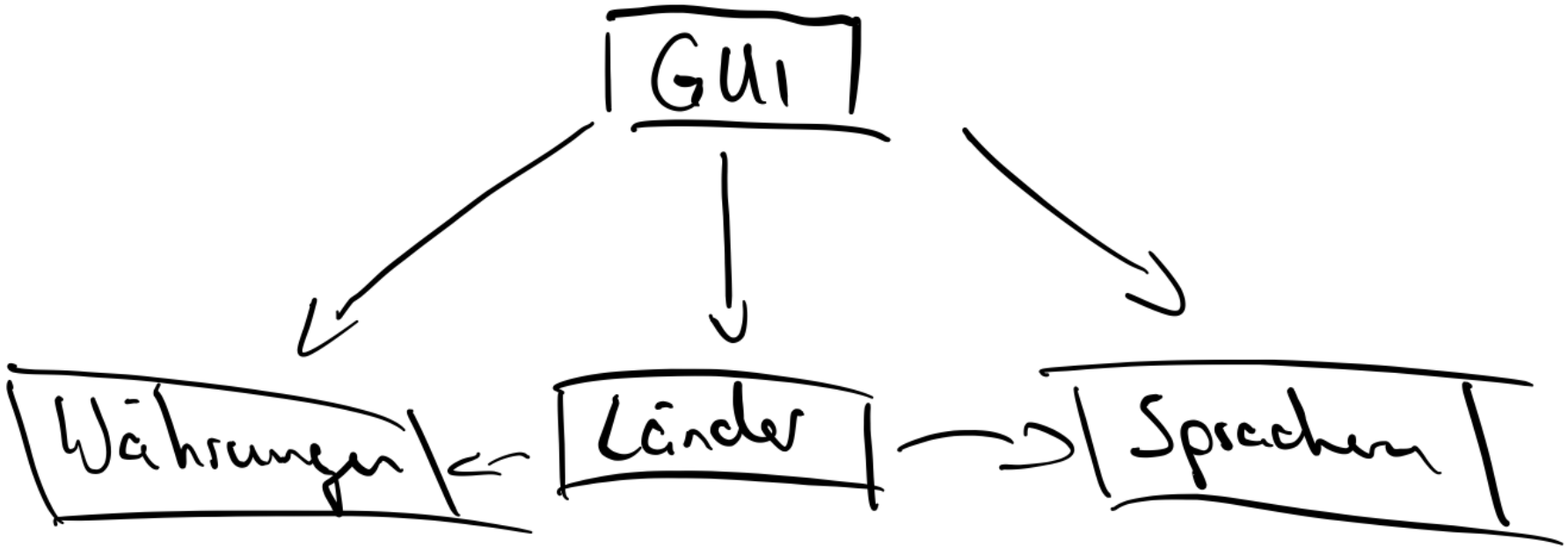
Ein Kunde wünscht sich eine Anwendung mit den fachlichen Anforderungen:

- Die Länderabteilung soll Länder (bestehend aus einem Namen und einem eindeutigen Kürzel) anlegen und einsehen können
- Die Währungsabteilung soll Währungen (bestehend aus einem Namen und einem eindeutigen Kürzel) anlegen und einsehen können
- Die Sprachenabteilung soll Sprachen (bestehend aus einem Namen und einem eindeutigen Kürzel) anlegen und einsehen können
- Ferner soll eine Verknüpfung eines Landes mit einer Währung und einer Sprache möglich sein
- Vollständig verknüpfte Länder (Länder mit definierter Währung und Sprache) sollen einsehbar sein

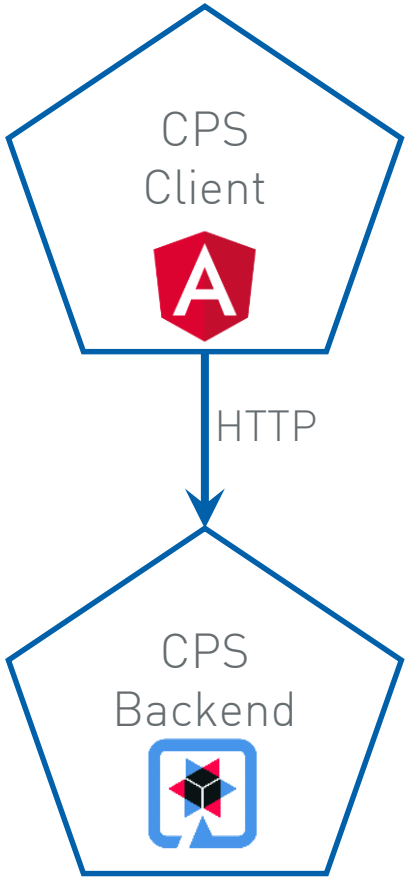
Country-Provider-Service: was bisher geschah...

Anforderungen:

- Länderverwaltung
- Währungsverwaltung
- Sprachenverwaltung
- GUI zur Ansicht



Country-Provider-Service: was bisher geschah...

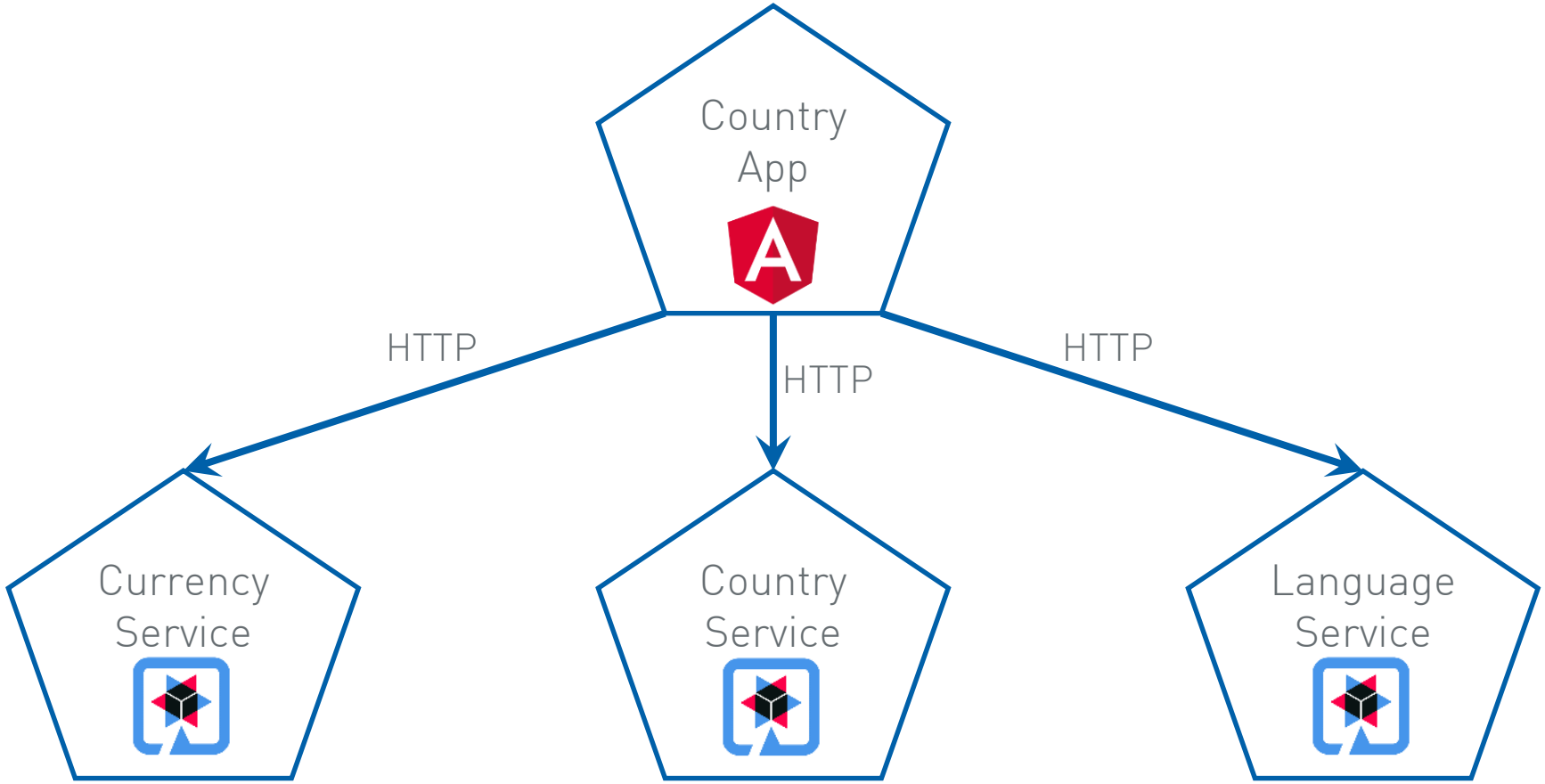


Country-Provider-Service: was bisher geschah...

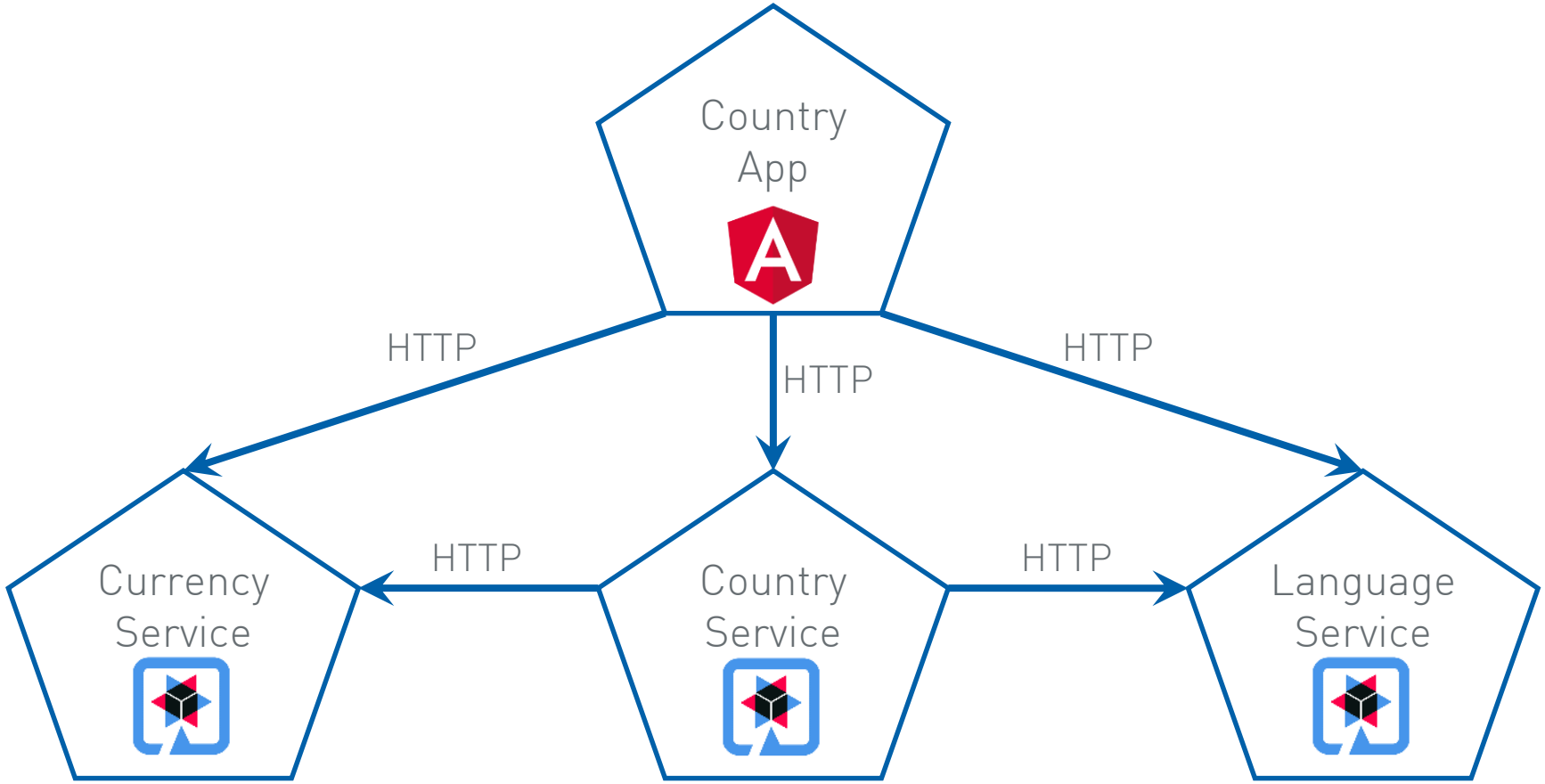
Modules:
- Country
- Language
- Currency

Components:
- Country
- Language
- Currency

Country-Provider-Service: was bisher geschah...



Country-Provider-Service: was bisher geschah...



Country-Provider-Serivce: was bisher geschah...

Die Services sind fertig implementiert und liegen auf Github:

- Country App Service (Frontend): https://github.com/exxcellent/microservice-country-app
- Country Service (Backend-Service): https://github.com/exxcellent/microservice-country-service
- Language Service (Backend-Service): https://github.com/exxcellent/microservice-language-service
- Currency Service (Backend-Service): https://github.com/exxcellent/microservice-currency-service
- **Die Services liegen als Docker-Image fertig auf Dockerhub**

Country-Provider-Service: Eure Aufgabe

Stellt die Anwendung für
den Kunden auf AWS bereit!

Country-Provider-Service: Eure Aufgabe

Im Detail:

- Wir haben für Euch ein Kubernetes-Cluster auf AWS aufgesetzt
- Bildet 4er bzw. 3er Gruppen (5x 4er, 2x 3er)
- Jede Gruppe soll die komplette Anwendung (ein Frontend Service, 3 Backend Services) deployen
- Anleitung/Hilfestellung: https://github.com/exxcellent/microservices-kubernetes-docs

Ergebnispräsentation

Wo gabs Schwierigkeiten?

Wir freuen uns über Euer Feedback.