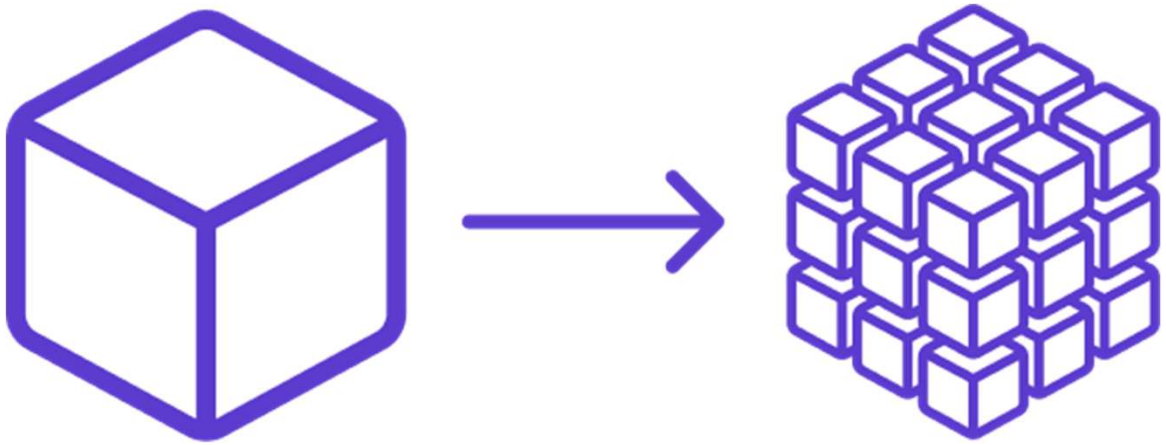
HM Hochschule München University of Applied Sciences

„Do one thing and do it well“

~ Unix Philosophy von Douglas McIlroy

Wandel vom Monolith zur
Microservice-Architektur



Aber wie kann man diese leicht
entwickeln und testen?

Lokale Entwicklung von Microservices mit docker-compose anhand eines Beispiels und Ausblick hinsichtlich Kubernetes Bridge

Sommersemester 2021

Oliver Stempel

08.06.2021

Gliederung

1. Motivation
2. Gliederung
3. Theoretische Grundlagen
    1. Microservices
    2. docker-compose
4. Problematik und Lösungsansatz der lokalen Entwicklung von Microservices
5. Praktischer Teil: lokale Umsetzung eines Microservice-Systems mithilfe docker-compose
    1. Datenbank
    2. Service
    3. API-Gateway
    4. Frontend
    5. Orchestrierung mithilfe docker-compose
6. Ausblick: Bridge to Kubernetes
7. Fazit
8. Quellen

Theoretische Grundlagen: Microservices

ISO/IEC TS 23167:2020

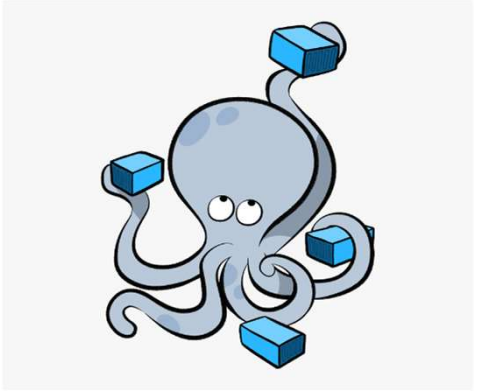
- unabhängig einsetzbares Artefakt, das einen Dienst bereitstellt, der einen bestimmten funktionalen Teil einer Anwendung implementiert
- Designansatz, der eine Anwendung in eine Reihe von Microservices unterteilt

Vorteile	Nachteil
Skalierbarkeit	Komplexe Netzwerkkommunikation
Einfache Bereitstellung	Testbarkeit (Integration/System)
Kompositionsfähigkeit	Leichte Verletzung von Kernkonzepten (Autonomie, gekoppelte Domains, schwache Interfaces, …)
Auswechselbarkeit	
Kleine Codebasis	

Theoretische Grundlagen: docker-compose

- Orchestrierungstool für Docker-Container
  → komplettes Lebenszyklusmanagement von Docker-Container

- 3 Schritte bis zum Starten einer Umgebung:
  1. Dockerimage erstellen, die entweder öffentlich sind oder über ein Dockerfile gebaut werden
  2. docker-compose.yml erstellen, die die zu orchestrierende Umgebung konfiguriert
  3. „docker-compose up“ → starten der Umgebung



Problematik und Lösungsansatz der lokalen Entwicklung von Microservices

Problematik:

- Schwierigkeit einen isolierten Kontext für das System zu schaffen
- Hoher manueller Mehraufwand
- Komplexes Management von Volumes, Umgebungsvariablen, Ports, etc…

Lösungsansatz:

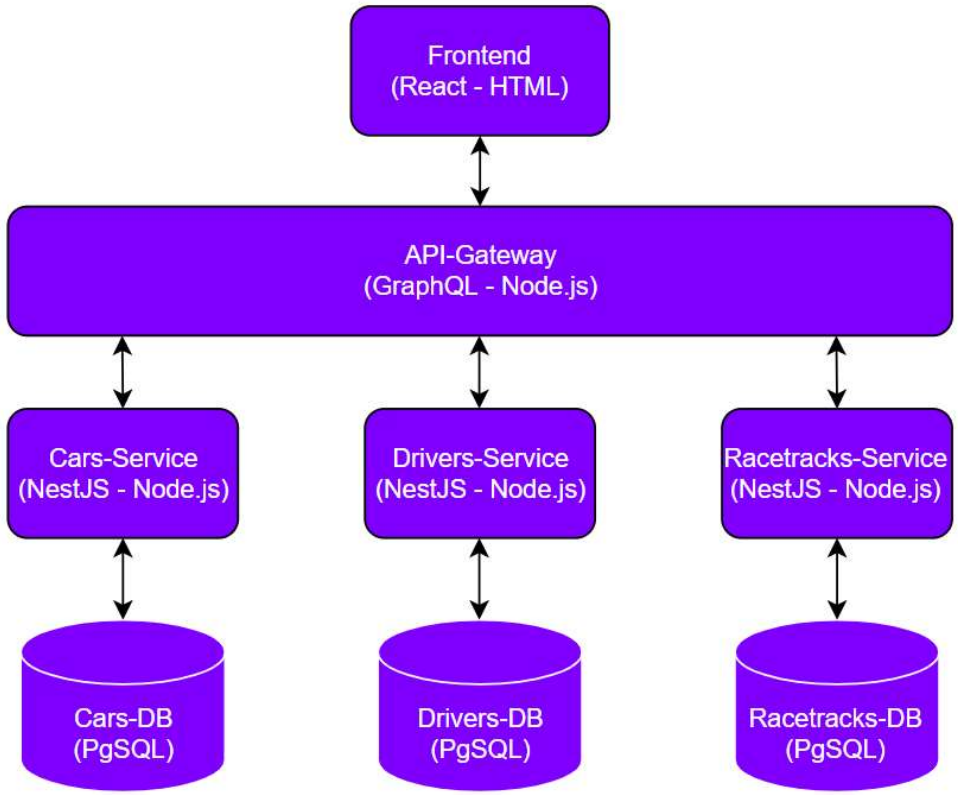
- Für jede Komponente ein eigenes Docker-Image bereitstellen
- Diese über docker-compose orchestrieren

Praktischer Teil: lokale Umsetzung eines Microservice-Systems mithilfe docker-compose

Aufbau:

- Frontend
  - WebApp
- Backend
  - API-Gateway
  - Microservices
  - Datenbanken

Ziel: Diese Komponenten über Docker orchestrieren



Datenbank

- PostgreSQL: relationale Datenbank
- Jeder Service hat seine eigene Datenbank

Dockerfile:

- Postgres-Image basierend auf Debian
- SQL-Ordner und Init-Script kopieren
- Init-Skript ausführen

```
FROM postgres:13

RUN set -x && \
    apt-get update && \
    apt-get install -y --no-install-recommends dos2unix

COPY ./db /db
COPY init.sh /docker-entrypoint-initdb.d/
RUN dos2unix /docker-entrypoint-initdb.d/init.sh
WORKDIR /db
```

Services

- Verwalten Entitäten → Verbindung zur jeweiligen Datenbank
- Bieten CRUDL-Operationen über REST-Schnittstellen an
- Sind auf ihre Domain begrenzt

Dockerfile:

- Node.js auf alpine als Basis-Image
- Multistage Build
- Build-Stage:
  - Baut Anwendung
- Run-Stage:
  - Kopiert nur gebaute Anwendung und führt diese aus

→ Bereinigtes Image

```
FROM node:lts-alpine as builder
ARG NODE_ENV
WORKDIR /usr/src/app
COPY package.json ./
COPY yarn.lock ./
RUN yarn install --production=false
COPY . .
RUN yarn run build

FROM node:lts-alpine as runner
EXPOSE 3001
WORKDIR /usr/src/app
COPY . .
COPY --from=builder /usr/src/app/build build
RUN yarn install --production=true
RUN yarn cache clean --all
CMD ["node", "--inspect=9229", "build/src/main"]
```

API-Gateway

- „Tor zu den Microservices“ → zentrale Komponente, die als Ansprechpartner des Frontends dient
- Leitet alle Anfragen und Operationen an die entsprechenden Microservices weiter

Bestandteile von GraphQL:

- Schema: Beschreibt verfügbare Typen und Operationen
- Resolver: Bearbeiten Anfragen

```
FROM node:lts-alpine as builder
ARG NODE_ENV
WORKDIR /usr/src/app
COPY package.json ./
COPY yarn.lock ./
RUN yarn install --production=false
COPY . .
RUN yarn run build

FROM node:lts-alpine as runner
EXPOSE 3001
WORKDIR /usr/src/app
COPY . .
COPY --from=builder /usr/src/app/build build
RUN yarn install --production=true
RUN yarn cache clean --all
CMD ["node", "--inspect=9229", "build/server.js"]
```

Frontend

- WebApp
- Allgemeine Startseite und für jeden Service eine weitere Verwaltungsseite
- Nutzt das Gateway um mit Services zu kommunizieren

Dockerfile:

- Multistage
- Run-Stage:
  - NGINX-Webserver
  - Hostet die Webapp

```
FROM node:lts-alpine as builder
ARG NODE_ENV
WORKDIR /usr/src/app
COPY package.json ./
COPY yarn.lock ./
RUN yarn install --production=false
COPY . .
RUN yarn run build

FROM nginx:alpine as runner
COPY --from=builder /usr/src/app/build /usr/share/nginx/html
CMD ["nginx", "-g", "daemon off;"]
```

Orchestrierung durch docker-compose

Schritte:

1. docker-compose.yml im root-Verzeichnis der Systemarchitektur
2. Netzwerk konfigurieren, damit Docker-Container miteinander kommunizieren können
3. diverse Services definieren

Wichtigsten Commands:

- docker-compose up --build [SERVICES]
- docker-compose up [SERVICES]
- docker-compose down
- docker-compose restart [SERVICES]
- docker-compose stop [SERVICES]
- docker-compose rm [SERVICES]

```yaml
version: "3.7"

networks:
  project-network:

services:
  service-cars:
    build:
      context: "./service-cars"
      target: builder
    command: "yarn start:debug"
    ports:
      - "3001:3001"
      - "9230:9229"
    volumes:
      - ./service-cars:/usr/src/app
    networks:
      - project-network
    depends_on:
      - database-cars
    environment:
      NODE_ENV: docker-dev
    container_name: service-cars
```

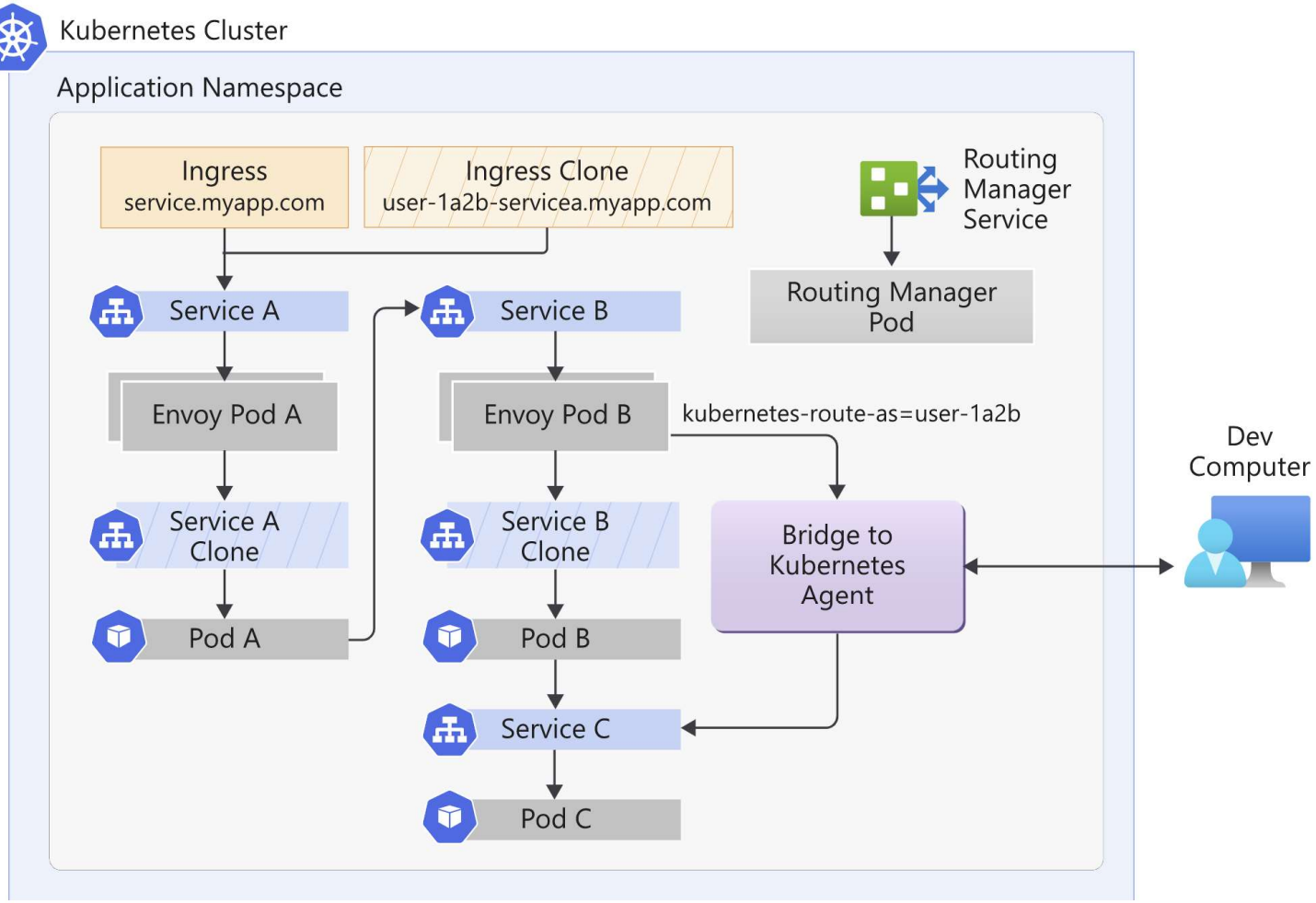
Ausblick: Bridge to Kubernetes

Was ist Kubernetes Bridge?

- Hybride Entwicklung
  - Lokales Ausführen und Debuggen von Code
  - Rest des System läuft im Cluster
- Verbindung des Entwicklungscomputer mit dem Cluster
- Umleitung des Datenverkehrs

→ Ermöglicht es als wären lokale Dienste ebenfalls im Cluster

Ausblick: Bridge to Kubernetes



Fazit

- docker-compose als kurz- bis mittelfristiger Ansatz zur Entwicklung geeignet
  - Schnelles konfigurieren
  - Lokales Debuggen
  - Keine Infrastruktur benötigt

- Bei wachsender Komplexität und Größe können Hybrid- und Remote Entwicklungsumgebungen bessere Wahl sein
  - Bessere Live-Umgebung Bedingungen wiederspiegeln
  - Auslagerung der Last
  - Zentraler Cluser → nicht jeder Entwickler hat seine eigenen lokalen Konfigurationen

Quellen

Bildverzeichnis:

https://www.blocshop.io/wp-content/uploads/2020/10/monolith-to-microservices.png

http://www.willhoeft-it.com/images/posts/2016-04-compose.png

https://static.thenounproject.com/png/2139239-200.png

https://kubernetes.io/images/favicon.png

https://docs.microsoft.com/de-de/visualstudio/containers/media/bridge-to-kubernetes/kubr-cluster-devcomputer.svg?view=vs-2019

Literatur:

Sam, N. (2015). Building Microservices. Designing fine-grained systems (5. Aufl.). O'Reilly.

Öggl, B. & Kofer, M. (2020). Docker. Das Praxisbuch für Entwickler und DevOps-Teams (2. Aufl.). Rheinwerk Computing.

GitHub-Repository:

https://github.com/ostempel/DevOps-CloudComputing

Hochschule
München
University of
Applied Sciences

Gibt es noch Fragen?

Vielen Dank für Ihre Aufmerksamkeit!