Hochschule für angewandte Wissenschaften München
Fakultät für Informatik und Mathematik

Bachelorarbeit zum Thema:

Entwurf und Bereitstellung von Microservices mit Kubernetes am Beispiel eines CRM-Systems

Zur Erlangung des akademischen Grades Bachelor of Science

Vorgelegt von: Simon Hirner

Matrikelnummer: 02607918

Studiengang: Wirtschaftsinformatik

Betreuer: Prof. Dr. Torsten Zimmer

Abgabedatum: 04.02.2022

Zusammenfassung

Bei immer mehr moderne Webanwendungen findet das Architekturmuster der Microservices anwendung. Dieser Trend verändert nicht nur den Entwurf und die Implementierung von Anwendungen, sondern hat auch erhebliche Auswirkungen auf die Bereitstellung und den Betrieb.

In dieser Bachelorarbeit wird der Entwurf und die Bereitstellung von Microservices mithilfe von Kubernetes analysiert.

Abstract

Bei immer mehr moderne Webanwendungen findet das Architekturmuster der Microservices anwendung. Dieser Trend verändert nicht nur den Entwurf und die Implementierung von Anwendungen, sondern hat auch erhebliche Auswirkungen auf die Bereitstellung und den Betrieb.

In dieser Bachelorarbeit wird der Entwurf und die Bereitstellung von Microservices mithilfe von Kubernetes analysiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

USD US-Dollar

CRM Customer-Relationship-Management

UNIX Uniplexed Information and Computing Service

SOA Serviceorientierte Architektur

API Application Programming Interface

CI Continuous Integration

CD Continuous Delivery

DDD Domain-driven Design

UI User Interface

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

URI Uniform Resource Identifier

DNS Domain Name System

CLI Command Line Interface

HTML Hypertext Markup Language

JSON JavaScript Object Notation

B2C Business-to-Customer

1 Einleitung

"Veränderungen begünstigen nur den, der darauf vorbereitet ist."
– Louis Pasteur [Pasteur 1933, S. 348]

Die IT-Branche befindet sich in einem erheblichen Wandel. Eine Reihe von neuen Methoden und Werkzeugen revolutionieren die Software-Welt. Angefangen hat es mit der Verbreitung von Cloud Computing und den damit einhergehenden großen verteilten Systemen. Diese machten es immer schwieriger den Betrieb von der Architektur eines Systems zu trennen. Der DevOps-Ansatz soll dieses Problem durch eine bessere Zusammenarbeit von Softwareentwicklung und IT-Betrieb beheben. Eng verwoben mit den Grundsätzen von DevOps ist das neue Architekturmuster der Microservices, bei dem große Anwendungen in kleine unabhängige Services aufgeteilt werden. Werkzeuge zur Containervirtualisierung wie Docker helfen dabei, die einzelnen Services auf verschiedenen Rechnern bereitzustellen und Kubernetes unterstütze bei der Steuerung der riesigen Containermengen. Zusammen bilden all diese Innovationen den Grundstein für moderne Anwendungen, containerisierten Microservices, welche mit Kubernetes verwaltet werden. In dieser Arbeit wird die Kombination dieser Innovationen genauer betrachtet, um die Möglichkeiten und Herausforderungen vom Entwurf bis zur Bereitstellung kennenzulernen. Zu Beginn wird in diesem Kapitel die Motivation, die Zielsetzung sowie der Aufbau der Arbeit beschrieben.

1.1 Motivation

Container haben sich etabliert und sind in der heutigen IT-Landschaft nicht mehr wegzudenken. Google stellt fast alle seine Dienste in Containern bereit und startet so über zwei Milliarden Container pro Woche [vgl. Liebel 2021, S. 43]. Kubernetes ist mittlerweile der Branchenstandard zur Containerorchestrierung und die Grundlage für moderne Webanwendungen [vgl. Arundel und Domingus 2019, Vorwort]. In IT-Projekten von Unternehmen, in denen Software zur Containerorchestrierung eingesetzt wird, setzen 91% auf Kubernetes [vgl. Cloud Native Computing Foundation 2020, S. 8]. Grundlegende Fähigkeiten in diesen Bereichen sind heutzutage unabdingbar.

Der Übergang zu Microservice-Architekturen ist in vielen Unternehmen in vollem Gange. Immer mehr monolithische Anwendungen werden so in Microservices aufgespalten. Das Marktvolumen für Microservices in der Cloud wurde 2020 auf 831 Millionen US-Dollar (USD) geschätzt. Bis zum Jahre 2026 soll der Markt mit einer durchschnittlichen jährlichen Wachstumsrate von 21.7% auf 2701 Millionen USD anwachsen [vgl. Mordor Intelligence 2020, S. 7].

Während sich Docker und Kubernetes schon feste Größen sind, befinden sich Microservices gerade in einem großen Trend. Ein abflachen dieses Trends ist nicht zu erkennen. Die beiden Softwareentwickler Brendan Burns und David Oppenheimer, welche Kubernetes

mitentwickelten, halten das Konzept von containersierten Microservices sogar für ähnlich revolutionär, wie die Popularisierung der objektorientierten Programmierung [vgl. Burns und Oppenheimer 2016, S. 1]. Der Cloud-Experte John Arundel denkt, dass aufgrund dieser Revolutionen die Zukunft in containersierten verteilten Systemen liegt, die auf der Kubernetes-Plattform laufen [vgl. Arundel und Domingus 2019, S. 1]. Fähgikeiten in diesen Bereichen sind somit sehr gefragt und werden Unternehmen gut entlohnt. Die Kombination dieser Methoden und Werkzeugen ergänzt sich perfekt und ist die Zukunft für große Systeme. Jedoch sind die Technologien diffizil und bringen neben zahlreichen Vorteilen auch viele Herausforderungen mit sich. Es ist von großer Bedeutung die Technologien in ihrer Gesamtheit zu verstehen und anwenden zu können. Diese Arbeit wird sich deshalb dem Entwurf und der Bereitstellung von Microservices mit Kubernetes widmen.

1.2 Zielsetzung

Das Ziel dieser Bachelorarbeit ist es, eine moderne Webanwendung, nach aktuellem Stand der Technik, mit einer Microservice-Architektur zu entwerfen und mithilfe von Kubernetes bereitzustellen. Dazu soll zuerst der theoretische Rahmen erläutert werden, um anschließend eine Fallstudie durchzuführen. Die Fallstudie wird am Beispiel eines Customer-Relationship-Management (CRM)-Systems durchgeführt. In der Fallstudie soll ein Verfahren vom Entwurf bis zur Bereitstellung implementiert werden. Um Aussagen zum Anwendungsgebiet und der Implementierung von containerisierten Microservices mit Kubernetes zu treffen, sollen geklärt werden,

- welche Vorteile und Herausforderungen Microservices bieten,
- wobei Containervirtualisierung sowie Kubernetes den Einsatz von Microservices unterstützt,
- wie eine Microservice-Architektur entworfen werden kann,
- wie Microservices mit Kubernetes bereitgestellt werden können.

1.3 Aufbau der Arbeit

Als Erstes wird in Kapitel 2 der theoretische Rahmen der Arbeit erläutert. Es wird DevOps, das Architekturmuster der Microservices, Docker sowie Kubernetes genauer erklärt. Auf Basis dieser theoretischen Grundlagen wird die Fallstudie durchgeführt. In Kapitel 3 wird zuerst die Problemstellung beschrieben. Danach wird in Kapitel 4 der Entwurf und in Kapitel 5 die Implementierung der Microservices dargelegt. Anschließend wird in Kapitel 6 die Bereitstellung mit Kubernetes erklärt. Zum Schluss wird in Kapitel 7 ein Fazit gezogen und die Ergebnisse diskutiert.

2 Theoretische Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe und Konzepte erläutert, welche wichtig für das Verständnis der Arbeit sind. Es wird zunächst der DevOps-Ansatz und das Architekturmuster der Microservices beschrieben. Im Anschluss werden die Grundlagen von Docker sowie Kubernetes erklärt.

2.1 DevOps

Alle der in diesem Kapitel beschriebenen Architekturmuster, Methoden und Werkzeuge lassen sich dem DevOps-Ansatz zuordnen. Um den Gesamtkontext zu verstehen, ist es deshalb sehr bedeutsam zu wissen, was DevOps bedeutet und warum es so populär ist. Wie das Kofferwort „DevOps“ bereits andeutet, beschreibt er einen Ansatz für eine effektivere und stärkere Zusammenarbeit zwischen Softwareentwicklung (Development) und IT-Betrieb (Operations). Für DevOps gibt es keine einheitliche Definition. Es ist ein Überbegriff für Denkweisen, Kultur, Methoden, Technologien und Werkzeuge. Der Kundennutzen wird dabei immer in den Mittelpunkt gestellt [vgl. Halstenberg u. a. 2020, S. 1]. Das Ziel ist es die Softwarequalität zu verbessern sowie die Geschwindigkeit der Entwicklung und Bereitstellung zu erhöhen [vgl. Arundel und Domingus 2019, S. 6]. Um die Ziele zu erreichen, werden etablierte Methoden und Werkzeuge eingesetzt. Die einzelnen Phasen der Entwicklung und des Betriebs miteinander zu einem Prozess, der von jedem Programminkrement durchlaufen wird [vgl. Tremp 2021, S. 63].

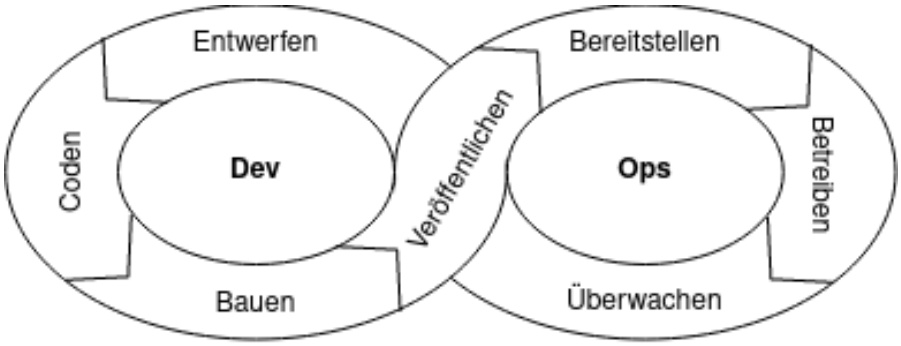

Abbildung 1: Kreislauf und Schritte von DevOps [vgl. Tremp 2021, S. 63]

Durch eine engere Abstimmung, Automatisierung, Continuous-Integration und Continuous-Delivery kann beispielsweise die Time-to-Market reduziert werden. Diese Kennzahl gibt an, wie lange es dauert eine Änderung zum Kunden, also auf die Produktionsumgebung, zu bringen [vgl. Halstenberg u. a. 2020, S. 7]. Auch Microservice-Architekturen und Werkzeuge wie Docker und Kubernetes können dabei unterstützen.

DevOps wird immer wichtiger, da durch das Aufkommen von Cloud Computing die Entwicklung und der Betrieb von Anwendungen nicht mehr trennbar ist. DevOps in ein Unternehmen

einzuführen ist ein langwieriger Prozess. Neben der Einführung der neuen Technologien ist vor allem die Transformation der Unternehmenskultur besonders schwierig. In großen Unternehmen zeigt sich das viele Prozesse schwerfällig geworden sind und nicht mehr dem eigentlichen Kundennutzen dienen. DevOps soll dieses Problem lösen und Unternehmen wieder anpassungsfähiger machen, ohne geordnete Strukturen zu verlieren.

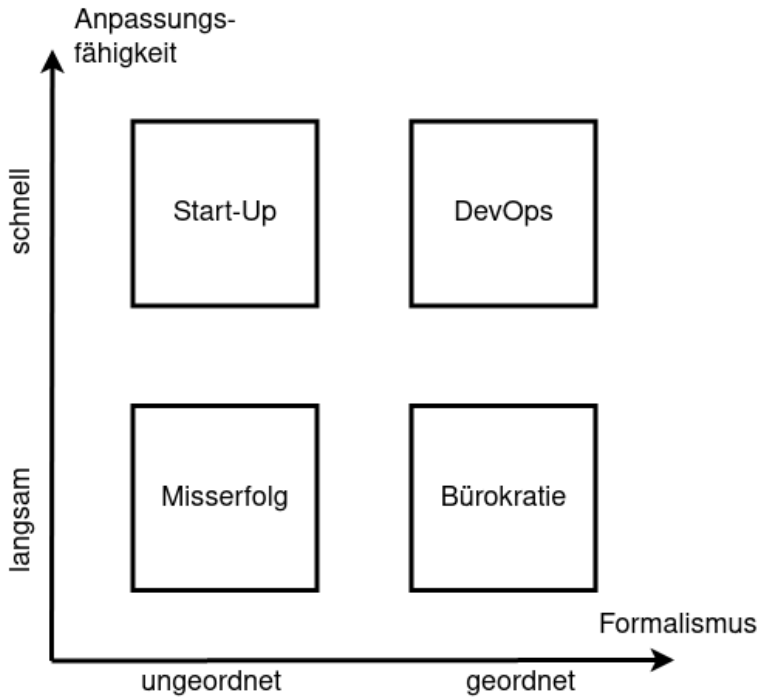

Abbildung 2: Kategorisierung von Unternehmen nach Anpassungsfähigkeit und Formalismus [vgl. Halstenberg u. a. 2020, S. 11]

2.2 Microservices

Im Mittelpunkt dieser Arbeit stehen Microservices. Bei Microservices handelt es sich um ein Architekturmuster zur Modularisierung von Software [vgl. Newman 2015, S. 15]. Eine einheitliche Definition für Microservices gib es nicht [vgl. Wolff 2018, S. 2]. Bei der Beschreibung von Microservices werden Prinzipien und Merkmale einer standardisierten Definition vorgezogen. Im Folgenden werden die wichtigsten Merkmale kurz erläutert.

2.2.1 Merkmale

Microservices sind das Gegenteil von klassischen monolithischen Softwarearchitekturen. Ein Monolith ist eine einzelne, zusammenhängende und untrennbare Einheit. Die Erweiterbarkeit und Wartbarkeit von Monolithen ist häufig komplex, da die Codebasis umfangreich ist und

mit der Zeit immer stärker wächst. Die Arbeit von mehreren Entwicklerteams ist ineffizient, da ein hoher Abstimmungsbedarf nötig ist. Des Weiteren lässt ist die Skalierbarkeit des schwergewichtigen Monolithen sehr beschränkt. Durch Modularisierung einer Anwendung lassen sich diese Probleme abschwächen, können jedoch nicht vollständig behoben werden.

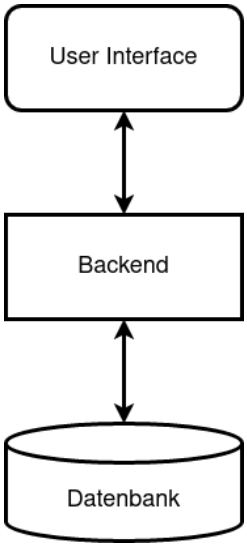

Abbildung 3: Beispielhafter Aufbau einer monolithischen Architektur

Genau hier setzten Microservices an. Obwohl der Begriff Microservices noch realtiv jung ist, sind die dahinterstehenden Konzepte bereits deutlich älter [vgl. Newman 2015, S. 15]. Zur Verständlichkeit und leichteren Weiterentwicklung werden große Systeme schon lange in kleine Module unterteilt. Die Besonderheit von Microservices liegt darin, dass die Module einzelne Programme sind. Das Architekturmuster der Microservices zählt zu den verteilten Systemen. Die einzelnen Microservices laufen zumeist auf vielen unterschiedlichen Rechnern und kommunizieren über das Netzwerk.

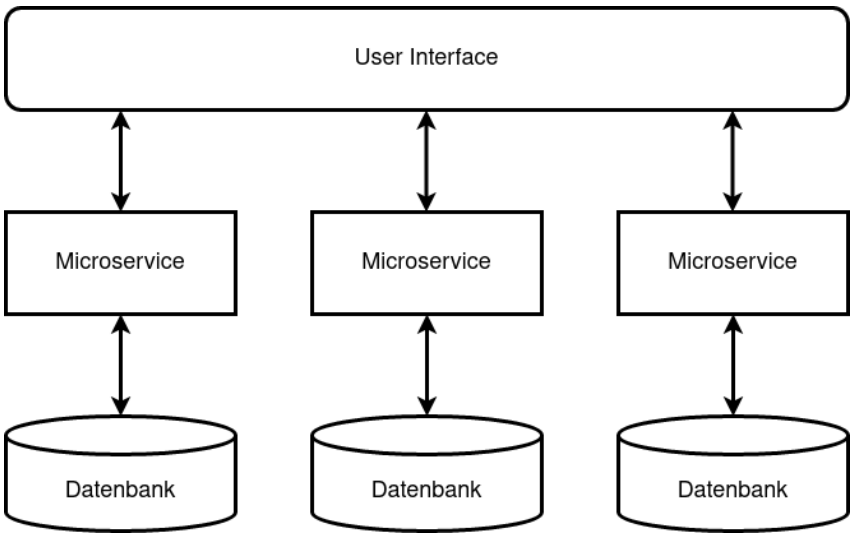

Abbildung 4: Beispielhafter Aufbau einer Microservice-Architektur

Ein einzelner Microservice soll eine Aufgabe bestmöglich erledigen. Dieser Ansatz ist angelehnt an die Philosophie von Uniplexed Information and Computing Service (UNIX): „Mache nur eine Sache und mache sie gut“ [vgl. Salus 1994]. Jeder Microservice bildet so eine klar definierte Funktion des Gesamtsystems ab [vgl. Tremp 2021, S. 64]. Die Microservices müssen dabei eigenständig sein, sodass sie unabhängig voneinander verändert und bereitgestellt werden können. Die Kommunikation zwischen den Microservices erfolgt über das Netzwerk mittels sprachunabhängiger Schnittstellen, sogenannte Application Programming Interfaces (APIs). Um die Microservices unabhängig voneinander skalieren zu können, müssen sie zustandlos sein.

Die Größe eines Microservices ist nicht zwangsläufig entscheidend [vgl. Wolff 2018, S. 2]. Der Name deutet bereits an, dass es sich um kleine Services handelt, jedoch ist eine genaue Festlegung der Größe nicht sinnvoll [vgl. Newman 2015, S. 22]. Eine Messung der Größe durch die Anzahl der Codezeilen wäre zwar denkbar, jedoch hängen derartige Kriterien stark von der verwendeten Programmiersprache ab. Stattdessen sollte sich die Größe an fachliche Gegebenheiten anpassen. Je kleiner die Services gestaltet werden, umso stärker kommen die in den nachfolgenden Abschnitten beschriebenen Vor- und Nachteile zur Geltung. Eine Obergrenze für die Größe eines Microservices stellt die Teamgröße dar. An einem Microservice darf immer nur ein Entwicklerteam arbeiten [vgl. Newman 2015, S. 23]. Kann der Microservice nicht mehr von einem Team alleine entwickelt und gewartet werden, so ist er zu groß. Ein Microservice sollte auch nur so groß sein, dass er von einem Entwickler allumfassend verstanden werden kann. Jedoch sollten Microservices auch nicht zu klein gewählt werden, da ansonsten der die Bereitstellung der vielen Microservices zu aufwendig wird.

Microservices wirken sich auf die Organisation und den Entwicklungsprozess aus [vgl. Wolff 2018, S. 2]. Um von Microservices zu profitieren müssen Strukturen in Unternehmen

überarbeitet werden. Das Gesetz von Conway besagt, dass durch die Kommunikationsstrukturen einer Organisation auch die Struktur der Systeme, welche die Organisation entwirft, vorgegeben wird [Conway 1968, vgl.]. Bei monolithischen Anwendungen werden die Entwicklerteams häufig nach ihrem Fachbereich aufgeteilt. Es bilden sich so beispielsweise Teams spezialisiert auf das Frontend, das Backend und die Datenbank. Die entwickelte Anwendung wird, nach dem Gesetz von Conway, auch aus diesen drei Bereichen bestehen. Wenn nun ein neues Feature umgesetzt werden soll, müssen sich alle drei Teams miteinander absprechen. Bei einer Microservice-Architektur müssen die Entwicklerteams crossfunktional mit Spezialisten aus verschiedenen Fachbereichen aufgebaut werden. Änderungen betreffen so häufig nur ein Entwicklerteam und der Koordinationsaufwand sinkt.

Häufig werden Microservices mit Serviceorientierte Architektur (SOA) in Verbindung gebracht. Microservices übernehmen viele Prinzipien von SOA. SOA ist ein Ansatz mit dem Ziel Funktionalitäten von betrieblichen Anwendungen durch Services von außerhalb zugreifbar zu machen [vgl. Wolff 2018, S. 2]. Ein Service bildet in diesem Kontext einen Geschäftsprozess ab. Dadurch soll Flexibilität und Wiederverwendbarkeit in der IT von Unternehmen erhöht werden. Es gibt also durchaus viele Parallelen zu Microservices, jedoch setzen sie an verschiedenen Ebenen an. Während Microservices ein konkretes Architekturmuster für ein einzelnes System ist, beschreibt SOA wie viele Systeme in einem Unternehmens miteinander interagieren können.

2.2.2 Vorteile

Bei monolithischen Anwendungen, entstehen schnell unerwünschte Abhängigkeiten zwischen verschiedenen Komponenten. Die vielen Abhängigkeiten sind schwer zu überblicken und die Änderung von einer Komponente wird erschwert, da es zu unerwünschten Nebeneffekten kommen kann. In der Praxis wird so die Architektur von Monolithen mit der Zeit zunehmend schlechter [vgl. Wolff 2018, S. 3]. Die Microservices besitzen dagegen nur eine lose Kopplung über explizite Schnittstellen. Die Hindernisse für unerwünschte Abhängigkeiten sind hier deutlich höher.

Durch die expliziten Schnittstellen ist es auch einfach einen gesamten Microservice zu ersetzten. Der neue Microservice muss lediglich die selbe Schnittstelle anbieten wie der Alte. Auch die vollständige Neuentwicklung eines Microservices ist durch die begrenzte Größe in der Regel nicht schwer. Somit können Microservices schneller an neue Technologien angepasst werden. Die Ablösung von großen Monolithen gestaltet sich dagegen häufig als eine fast unmögliche Aufgabe [vgl. Newman 2015, S. 29]. Darüber können Microservices auch komplett unterschiedliche Technologie-Stacks verwenden [vgl. Wolff 2018, S. 5]. Die eingesetzten Technologien müssen dabei lediglich die entsprechende Schnittstelle anbieten können. Durch diese Freiheit kann für jeden Microservice kompromisslos die am besten geeignete Technologie ausgewählt werden.

Ein weiterer wesentlicher Grund für Microservices ist Continuous Delivery (CD). Die Microservices können unabhängig voneinander bereitgestellt werden. Tritt bei einer Bereitstellung

ein Fehler auf, sind die verbundenen Risiken deutlich geringer. Es ist nicht das gesamte System davon betroffen, sondern nur der entsprechende Service. Dadurch, dass nur der veränderte Microservice neu bereitgestellt werden muss, ist die Bereitstellung schneller als bei einem Monolithen. Dadurch ermöglichen Microservices eine bessere Time-to-Market. Außerdem ist die Absicherung einer Bereitstellung, durch das parallele betreiben einer älteren Version, deutlich ressourcenschonender. Bei Monolithen wäre in so einem Fall der Ressourcenverbrauch doppelt so hoch, wie die gesamte Anwendung eigentlich benötigt.

Microservices können unabhängig voneinander skaliert werden. So kann eine einzelne Funktionalität, welcher stärker genutzt wird, einzeln hoch skaliert werden, ohne das gesamte System zu skalieren [vgl. Wolff 2018, S. 5]]. Flaschenhälse welche eine Anwendung ausbremsen, können somit besser vermieden werden. Auch kann die Last kann durch Microservices besser verteilt werden, da sie verteilt auf unterschiedlichen Rechnern laufen können. Durch eine bestmögliche Ressourcenausnutzung können so Kosten eingespart werden [vgl. Newman 2015, S. 27].

2.2.3 Herausforderungen

Microservices bringen neben den vielen Vorteilen auch einige erhebliche Herausforderungen mit sich. Die Aufteilung eines Systems in viele Microservices erhöht die Komplexität. Die Struktur des Systems kann mit einem schlechten Architekturmanagement schnell unübersichtlich werden [vgl. Wolff 2018, S. 77]. Welcher Microservice eine Schnittstelle eines anderen Microservices aufruft kann von außen nicht direkt eingesehen werden. Auch auf Code-Ebene können sich ungewollte Abhängigkeiten einschleichen. Wenn mehrere Microservices beispielsweise die selbe Bibliothek verwenden, geht die lose Kopplung verloren und die Microservices können unter Umständen nicht mehr unabhängig voneinander bereitgestellt werden.

Die technologische Freiheit der Microservices kann schnell zu einer Herausforderung werden. Zu viele unterschiedliche Technologien in den Microservices erhöhen die Komplexität und den Erhalt von Fachwissen [vgl. Tremp 2021, S.65]. Darüber hinaus wird der Wechsel von Mitarbeitern in andere Entwicklerteams erschwert.

Während Änderungen in einem Microservice sehr einfach sind gestalten sich Refactoring über mehrere Microservices hinweg als deutlich komplizierter. Bei Monolithen können Teile des Codes leicht von einer Komponente in eine Andere verschoben werden kann. Bei Microservices müssen die Teile in einen anderes Programm verschoben werden, welches vielleicht sogar einen anderen Technologie-Stack verwendet. Die Auswirkung von Fehlentscheidungen bei der Einteilung der Microservices sind somit sehr hoch [vgl. Wolff 2018, S. 6].

Microservices sind verteilte Systeme und bringen auch die damit verbundenen Nachteile mit sich. Da die Kommunikation zwischen den Microservices über das Netzwerk läuft, ist die Geschwindigkeit der Microservices von der Netzwerklatenz abhängig. In der Zeit, die ein Microservice benötigt, um einen anderen Microservice aufzurufen, könnte ein moderner Prozessor Millionen von Instruktionen abarbeiten [vgl. Wolff 2018, S. 73]. Außerdem ist

Kommunikation über ein Netzwerk unzuverlässig [vgl. Wolff 2018, S. 76]. Ausfälle von einzelnen Microservices können die Funktionalität des Gesamtsystems einschränken.

2.2.4 Architektur

Die Architektur von Microservices lässt sich in die Micro-Architektur und die Makro-Architektur untergliedern. Die Micro-Architektur bezieht sich auf die Architektur eines einzelnen Microservices. Sie besitzt keine Relevanz für das Gesamtsystem und ist von außen nicht einsehbar. Lediglich die Schnittstellen sind von Bedeutung. Das Entwicklerteam eines Microservices sollte bei der Micro-Architektur größtmögliche Freiheit haben.

Bei der Makro-Architektur liegt die zentrale Herausforderung in der Aufteilung der Microservices [vgl. Wolff 2018, S. 102]. Die Architekturentscheidungen sollten hierbei gut überlegt sein, da das Refactoring zwischen Microservices aufwendig ist. Jeder Microservice soll eine abgeschlossene Funktion in einem fachlichen Kontext darstellen. Die Aufteilung nach Fachlichkeit ist wichtig, damit die Microservices ihre Vorteile ausspielen können. Eine Änderung an einer Fachlichkeit sollte so idealerweise nur einen Microservice und ein Entwicklerteam betreffen. Der Abstimmungsaufwand in Unternehmen wird somit minimiert. Eine frühe Aufteilung in zu viele Services erhöht die Gefahr einer falschen Aufteilung. Es ist ratsam, mit wenigen Microservices zu beginnen und diese, ab einer gewissen Größe, weiter aufzuteilen.

Zur Einteilung der Microservices wird häufig Domain-driven Design (DDD) eingesetzt. DDD ist eine Vorgehensweise für die Modellierung komplexer Systeme [vgl. Evans 2015, S. 66]. Dabei werden Kontextgrenzen (Bounded Context) als Trennung zwischen unabhängigen Problembereichen, sogenannten Domänen, identifiziert. Innerhalb eines Bounded Context wird eine einheitliche fachlich orientierte Sprache verwendet (Ubiquitous Language). Eine Context Map gibt einen Überblick über alle Bounded Contexts und deren Interaktionen.

Bei der Implementierung der einzelnen Microservices sollte größtenteils technologische Freiheit bestehen. Die gemeinsamen Schnittstellen und Kommunikationsprotokolle sollten jedoch von der Makro-Architektur definiert werden. Außerdem sollte geklärt werden, wie Service Discovery, Lastverteilung und Skalierung implementiert wird. Alle diese Funktionen kann Kubernetes übernehmen und werden im entsprechenden Abschnitt genauer beschrieben.

2.2.5 Integration

Die Integration und Kommunikation der einzelnen Microservices ist einer der wichtigsten Aspekte. Die Integration und Kommunikation der Microservices ist auf drei verschiedenen Ebenen denkbar.

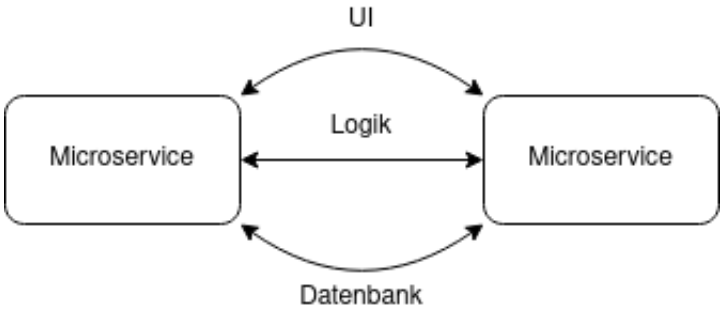

Abbildung 5: Integration von Microservices auf verschiedenen Ebenen [vgl. Wolff 2018, S. 167]

Auf Datenbank-Ebene können Microservices integriert werden, indem sie auf dieselbe Datenbank zugreifen. Diese Methode ist einfach zu implementieren, aber bringt jedoch Nachteile mit sich [vgl. Newman 2015, S. 69]. Wenn durch Änderungen in einem Microservice die Datenstruktur angepasst wird, wirkt sich das auf die anderen Microservices aus und die Unabhängigkeit wird reduziert. Auch die Technologiefreiheit wird dadurch beschränkt. Eigene Datenbanken oder zumindest Datenbankschemata erlauben so mehr Freiheiten und eine schnellere Änderung von Datenstrukturen.

Die Benutzerschnittstelle oder User Interface (UI) ist die Ebene auf der alle Funktionalitäten der Microservices zusammengeführt werden. Um auch hier ein hohe Unabhängigkeit zu gewährleisten, ist es ratsam jeden Microservice mit einer UI auszustatten. Sommit könnten alle Änderungen, egal ob sie UI, Logik, oder Datenbank betreffen, von einem Entwicklerteam umgesetzt werden. Das Problem ist jedoch, dass !s (!s) schnell sehr umfangreich werden und so einen Microservice unnötig groß machen. Außerdem benötigen moderne Anwendungen häufig unterschiedliche Frontends für verschiedene Gerätetypen. Deshalb werden Frontends häufig weiterhin monolithisch Aufgebaut. Mittlerweile gibt es jedoch auch immer mehr Micro-Frontends, welche den Microservice-Ansatz auf Frontends übertragen [vgl. Peltonen u. a. 2021, S. 1].

Zuletzt können Microservices auf Logik-Ebene miteinander verbunden werden. Microservices müssen untereinander kommunizieren, wenn sie die Funktionalität eines anderen Microservices benötigen. Wichtig dabei ist, dass die Microservices trotzdem eine größtmögliche Unabhängigkeit bewahren. Zu viel Kommunikation zwischen zwei Microservices sind ein Hinweis auf eine schlechte Aufteilung [vgl. Wolff 2018, S. 104]. Zyklische Abhängigkeiten sollten unter allen Umständen vermieden werden. Die Kommunikation läuft über APIs. Eine beliebter Architekturstil für APIs ist Representational State Transfer (REST). REST vereinheitlicht die Struktur und das Verhalten von Schnittstellen [vgl. Fielding 2000, S. 76]. Bei einer REST-APIs gibt es eine Vielzahl von Ressourcen, die über eindeutige Uniform Resource Identifiers (URIs), identifiziert werden. Die Ressourcen können über HTTP-Anfragen mit verschiedenen HTTP-Anfragemethoden manipuliert werden.

2.3 Docker

Anwendungen mit Microservice-Architektur verwenden häufig Containervirtualisierung zur Bereitstellung. Durch die leichtgewichtige Containervirtualisierung können mehrere isolierte Instanzen eines Betriebssystems auf dem selben Kernel ausgeführt werden. Dadurch sind die Container ressourcenschonender als die herkömmliche Virtualisierung mittels Hypervisor, bei dem jede virtuelle Maschine ein eigenes vollständiges Betriebssystem ausführt [vgl. Newman 2015, S. 166f].

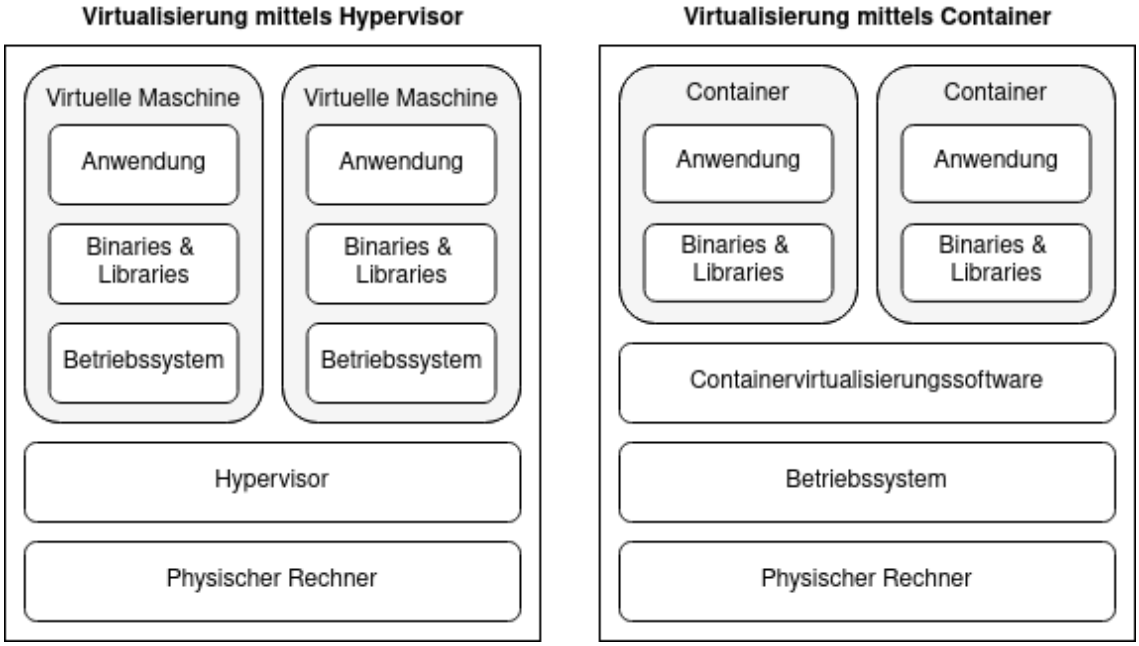

Abbildung 6: Vergleich Virtualisierung mittels Hypervisor und Container

Für die Ausführung einer Anwendung werden Abhängigkeiten wie zum Beispiel Bibliotheken, Compiler und Interpreter benötigt. Des Weiteren muss die Anwendung richtig konfiguriert werden. Vor allem bei einer Microservice-Architektur kann das ein Problem werden, da die Microservices über große Netzwerke verteilt auf verschiedenartigen Rechnern bereitgestellt werden sollen. Containervirtualisierung löst dieses Problem mit einem standardisierten Image-Datei, das die Anwendung mitsamt aller Abhängigkeiten und Konfigurationen beinhaltet [vgl. Arundel und Domingus 2019, S. 9]. Diese Image-Datei läuft unabhängig von der Plattform auf jedem Rechner, sofern die zugehörige Containervirtualisierungssoftware installiert ist.

Eine freie Software zur Containervirtualisierung ist Docker [vgl. Docker Inc. 2022]. Sie ist die beliebteste und verbreitetste Software für diesen Zweck [vgl. Hightower u. a. 2018, S. 20] und ergänzt die Containervirtualisierung um benutzerfreundliche Werkzeuge. Docker basiert auf der Virtualisierung mit Linux-Containern. Durch herkömmliche Virtualisierung

kann Docker jedoch auch auf anderen Betriebssystemen betrieben werden. Im Folgenden werden die wichtigsten Begriffe und Funktionen von Docker näher beschrieben.

2.3.1 Docker Image

Ein Docker Image ist das Speicherabbild eines Containers. Das Image beinhaltet alle Informationen, die zum Starten eines Containers notwendig sind. Bei Docker besteht das Image aus mehreren Schichten. Jede Schicht repräsentiert eine Abhängigkeit oder Konfiguration, welche für die Anwendung benötigt wird. Docker optimiert automatisch den verwendeten Speicherplatz durch Wiederverwendung, wenn zwei Images eine gleiche Schicht verwenden. Die Docker Images sind portabel. Über zentrale Registrys können die Images verwaltet, gespeichert und verteilt werden. Docker Hub ist die größte öffentliche Registry mit einer Vielzahl an Images, die von anderen Benutzern bereitgestellt werden. Beim Ausführen eines Images wird auf Basis des Images ein Container gestartet. Das Image ist wiederverwendbar und es können beliebig viele Container aus einem Image erzeugt werden.

2.3.2 Dockerfile

Ein Dockerfile ist eine Textdatei mit mehreren Befehlen, die ein Docker Image beschreiben. Aus einem Dockerfile kann das entsprechende Image gebaut werden. Dazu werden die einzelnen Befehle abgearbeitet und für jeden Befehl eine neue Schicht in dem zugehörigen Image angelegt. Begonnen wird meistens mit einem Basis-Image, welches bereits vorhanden ist. Danach folgen spezifische Änderungen, damit die gewünschte Anwendung ausgeführt werden kann.

2.3.3 Container

Ein Container ist die aktive Instanz eines Images. Er besitzt eine begrenzte Lebensdauer und wird, nachdem der in ihm laufende Prozess abgeschlossen ist, beendet. Container sind in der Regel unveränderlich. Soll ein Container geändert werden, so wird der alte Container gegen einen neuen ausgetauscht. Jeder Container besitzt sein eigenes Dateisystem, Anteil an CPU, Speicher und Prozessraum. Er besitzt auch seine eigenen Bibliotheken, Compiler und Interpreter und ist so unabhängig von allen Softwareversionen, die auf dem eigentlichen Betriebssystem installiert sind. Lediglich der Kernel wird geteilt und bildet somit die einzige Abhängigkeit.

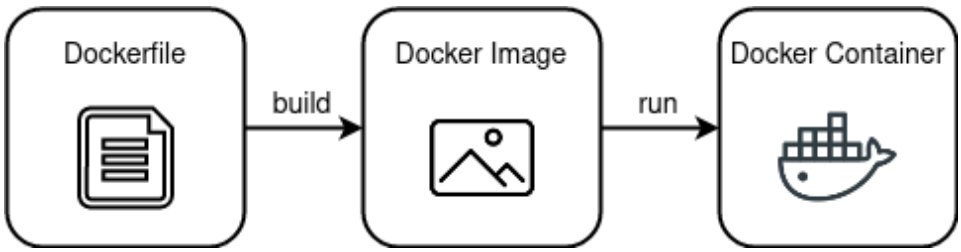

Abbildung 7: Weg vom Dockerfile zum Container

2.4 Kubernetes

Wenn Microservices in Containern bereitgestellt werden, wird es schnell nicht mehr möglich, die Container manuell zu verwalten. Service Discovery, Skalierung und die Lastverteilung gestaltet sich als aufwendige Probleme. Die Open-Source-Plattform Kubernetes versucht diese Probleme zu lösen [vgl. Linux Foundation 2022]. Der Name „Kubernetes“ stammt aus dem griechischen und bedeutet soviel wie Steuermann. Kubernetes hilft bei der Koordination und Sequenzierung verschiedener Aktivitäten sowie bei der Verwaltung der verfügbaren Ressourcen und bei einer effizienten Lastverteilung [vgl. Arundel und Domingus 2019, S. 11]. Kubernetes bietet somit viele Funktionen, die die Bereitstellung von Microservices erleichtern. Häufig wird Kubernetes mit Docker verwendet, es unterstützt aber auch andere Anwendungen zur Containervirtualisierung.

2.4.1 Aufbau

Die größte Organisationseinheit der Plattform ist ein Kubernetes Cluster. Ein Cluster besteht aus mindestens einem Control Plane und einem Node. Der Control Plane verwaltet sämtliche Nodes. Um Ausfallsicherheit zu gewährleisten können auch mehrere Control Planes in einem Cluster betrieben werden. Ein Control Plane enthählt eine Key-Value-Datenbank namens etcd. In ihr wird die gesamte Konfiguration des Clusters gespeichert. Des Weiteren enthält der Control Plane einen API-Server, mit der die Nodes kommunizieren. Auch externe Komponenten können mit acAPI-Server kommunizieren und so Informationen abfragen oder das Cluster konfigurieren. Der Controller Manager steuert über den acAPI-Server die einzelnen Nodes. Des Weiteren besitzt der Control Plane einen Scheduler, der die Last verteilt und überwacht.

In der Regel besteht ein Cluster aus vielen Nodes. Dabei kann es sich um physische Rechner aber auch um virtuelle Maschinen handeln. Auf den Nodes laufen die Container mit den eigentlichen Anwendungen. Außerdem besitzt jeder Node einen Kubelet. Dieser Kubelet kommuniziert mit dem Controller Manager und verwaltet den Status der Container auf dem jeweiligen Node.

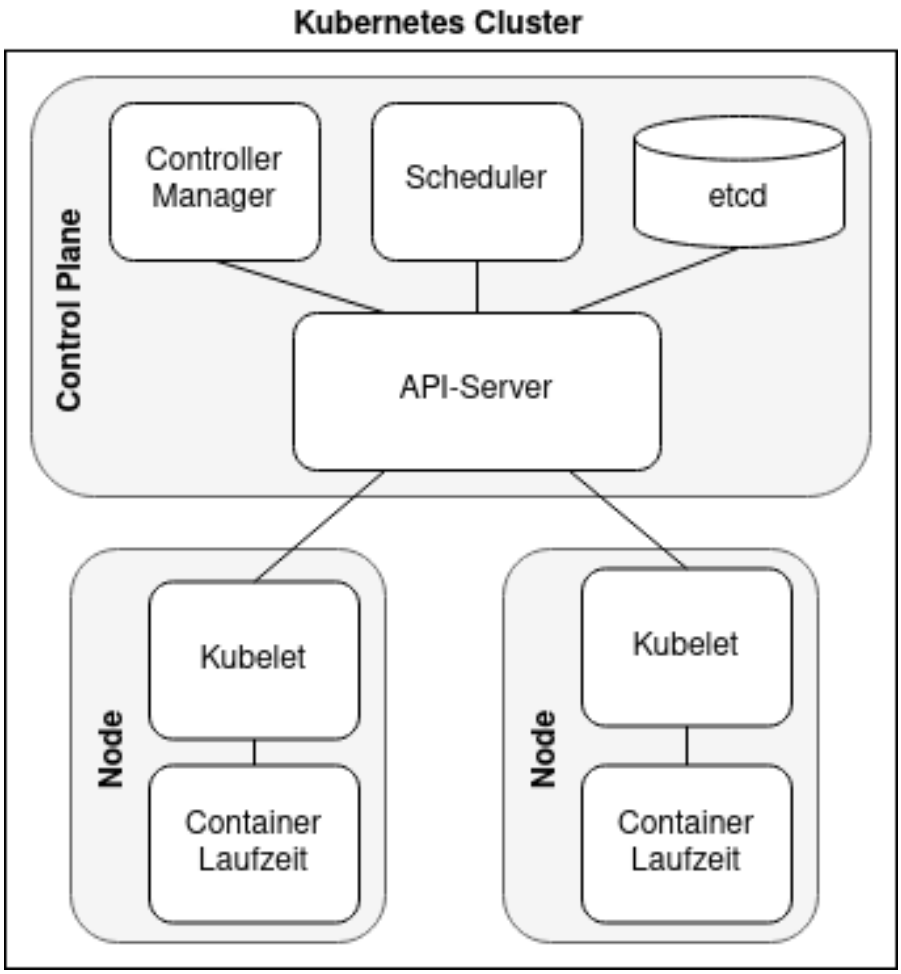

Abbildung 8: Aufbau eines Kubernetes Cluster

2.4.2 Objekte

Kubernetes stellt eine Reihe von abstrakten Objekten zur Verfügung, mit denen der Status des Systems dargestellt wird und welche es erleichtern eine Microservice-Architektur zu bauen [Hightower u. a. 2018, S. 13]. Diese Objekte werden in Manifesten beschrieben. Bei den Manifesten handelt es sich um YAML-Dateien. Die Manifeste sind deklarativ aufgebaut, das bedeutet der gewünschte Ausgangszustand wird beschrieben. Nachdem das Manifest übergeben wurde, führt Kubernetes die entsprechenden Aktionen aus, um den beschriebenen Zustand zu erreichen.

Zu den grundlegenden Objekten gehören die Folgenden:

- Pods sind die kleinste einsetzbare Einheit. Ein Pod repräsentiert eine einen einzelnen Container oder eine Gruppe von Containern. Alle Container in einem Pod laufen immer auf dem gleichen Node.

- Deployments
- Namespaces werden zur logischen Unterteilung des Clusters verwendet. Sie können zur Isolation verschiedener Entwicklerteams oder Anwendungsmodule genutzt werden.
- Volumes bieten persistenten Speicher, der auch nach der Lebenszeit eines Pods bestehen bleibt.
- ConfigMaps ermöglichen das Speichern von Konfigurationsdaten als Key-Value-Paare. Sie können beispielsweise von Pods als Umgebungsvariablen konsumiert werden.
- Labels können anderen Objekten zugeordnet werden, um diese zu Gruppieren.

In den nächsten Abschnitten werden noch ein paar weitere Objekte beschrieben, welche für die Bereitstellung, Service Discovery und Skalierung nützlich sind.

2.4.3 Workloads

2.4.4 Service Discovery

Kubernetes ist ein dynamisches System. Ein Microservice, der in einem Pod läuft, kann schnell gestoppt, gestartet oder repliziert werden. Da auch mehrere Instanzen eines Microservices gleichzeitig laufen können, wird ein fester Enpunkt benötigt, über den gleichartige Pods erreichbar sind. In Kubernetes kann dafür ein Service-Objekt erstellt werden. Ein Service stellt eine unveränderliche virtuelle IP-Adresse bereit, welche Kubernetes per Lastverteilung auf die passenden Pods verteilt. Innerhalb des Clusters kann ein Service auch über den Service-Namen aufgerufen werden, welcher mit Domain Name System (DNS) aufgelöst wird. Ein Service vom Typ ClusterIP ist von außerhalb nicht aufrufbar. Mit dem Typ NodePort wird die Kommunikation auf einen Port aller Nodes weitergeleitet und der Service wird somit von außen zugänglich.

2.4.5 Skalierung

2.4.6 Kubectl

kubectl ist der offizielle Kubernetes-Client und dient der Steuerung von Kubernetes. Bei kubectl handelt es sich um ein Command Line Interface (CLI) für die Interaktion mit dem API-Server des Control Planes. Mit kubectl können beispielsweise Objekte verwaltet werden und der Status des gesamten Clusters untersucht werden.

2.4.7 Minikube

Minikube ist ein Werkzeug, um ein lokales Kubernetes Cluster zu betreiben. Minikube erstellt ein Cluster bestehend aus nur einem Node in einer virtuellen Maschine. Der Node fungiert dabei sowohl als Control Plane sowie auch als Node. Minikube unterstützt mittlerweile auch den Betrieb mit mehreren Nodes. Des Weiteren kann das gesamte Cluster selbst auch in einem Docker Container anstatt in einer virtuellen Maschine betrieben werden. Minikube erstellt beim Start automatisch eine kubectl-Konfiguration, welche auf das Cluster zeigt und somit über kubectl-Befehle gesteuert werden kann.

3 Erläuterung Fallstudie

Nachdem die theoretischen Grundlagen nun erläutert worden sind, wird mit der Fallstudie begonnen. Für die Fallstudie soll ein vereinfachtes CRM-System entwickelt und bereitgestellt werden. Ein CRM-System ist eine Software für das Kundenbeziehungsmanagement. In diesem Kapitel werden die Anforderungen an das CRM-System beschrieben.

3.1 Anwendungseinsatz

Das CRM-System soll für das Business-to-Customer (B2C) Umfeld entwickelt werden. Die Kernfunktionalität des zu erstellenden CRM-Systems ist das Anlegen, Anzeigen, Bearbeiten und Löschen von Kontakten, Interaktionen und Verkaufschancen. Die Software soll mit einer Microservice-Architektur umgesetzt werden und containerisiert mit Kubernetes bereitgestellt werden. Das System soll über eine grafische Benutzeroberfläche mit einem Webbrowser bedienbar sein.

3.2 Anwendungsfunktionen

Das CRM-System soll die folgenden primären Funktionen implementieren:

- Kontakte sollen mit Namen, Geburtsdatum, Geschlecht, Telefonnummer, E-Mail-Adresse und Adresse angelegt, angezeigt, geändert und gelöscht werden können
- Interaktionen mit einem Kontakt sollen mit Art der Interaktion, Datum, Uhrzeit, Notizen und dem zugehörigen Kontakt angelegt, angezeigt, geändert und gelöscht werden können
- Mögliche Verkaufschancen sollen mit Status, voraussichtlichem Abschlussdatum, Verkaufswert, Rabatt, Budget des Kunden, Notizen und dem zugehörigen Kontakt angelegt, angezeigt, geändert und gelöscht werden können

Alle Funktionen sollen über eine grafische Benutzeroberfläche mit dem Webbrowser bedienbar sein. Darüber hinaus sollen alle Funktionen auch über eine REST-API aufrufbar sein. Die Anwendung sollte flexibel skaliert werden können.

Die Zugriffskontrolle und Benutzersicherheit wird bei der Anwendung nicht beachtet.

4 Entwurf der Microservices

An dieser Stelle sei auch erwähnt, dass der gesamte Quellcode im folgenden GitHub Repository einsehbar ist: github.com/SimonHirner/bachelor-thesis.

Als Erstes wird die Architektur der Microservices festgelegt. Bei der Architektur kann zwischen Makro-Architektur und Mikro-Architektur unterschieden werden. Die Makro-Architektur befasst sich mit dem Gesamtsystem. In der Mikro-Architektur geht es um den Aufbau der einzelnen Microservices.

4.1 Makro-Architektur

Für die Makro-Architektur ist die genaue Umsetzung der einzelnen Microservices nicht relevant. Die Makro-Architektur ist besonders wichtig, da Veränderungen hier zu einem späteren Zeitpunkt sehr aufwendig werden können. Das Wichtigste ist die Aufteilung in Microservices nach der Fachlichkeit. Das CRM-System lässt sich in drei fachliche Bereiche unterteilen: Kontaktverwaltung, Interaktionsverwaltung und Chancenverwaltung. Die Microservices werden nach den Bereichen aufgeteilt. Jeder dieser drei bereiche Besitzt ein eigenes Datenobjekt. Das ist ein weiterer Hinweis für eine gute Aufteilung. Die Microervices machen ihre Dienste über REST-Schnittstellen verfügbar. Für die Skalierbarkeit ist es wichtig, dass die Microservices selbst keinen zustandlos sind. Alle persistenten Daten werden in Datenbanken gehalten.

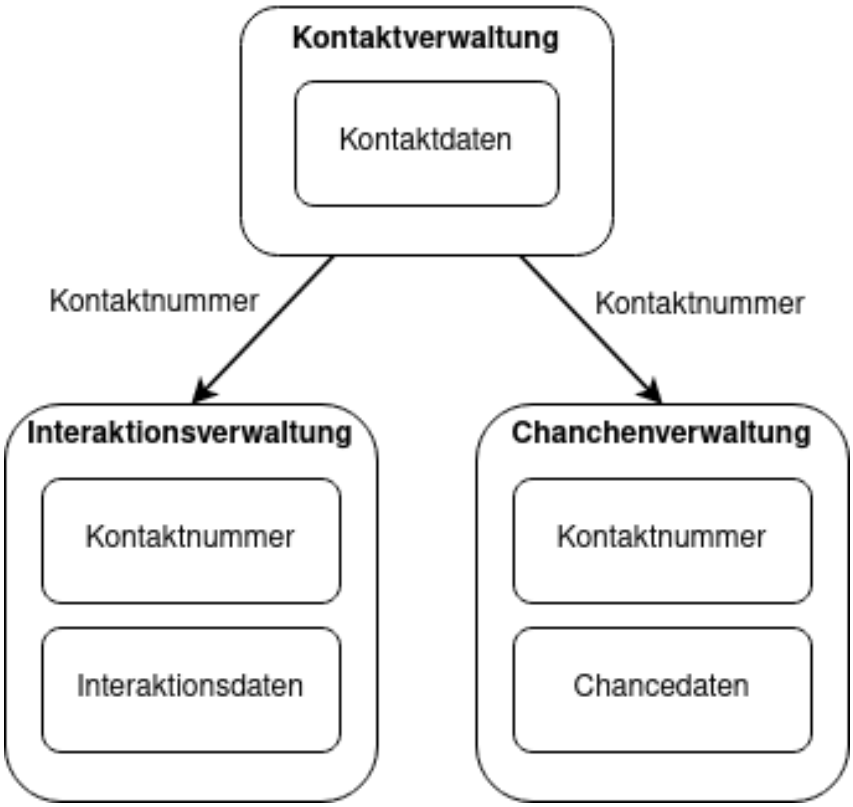

Abbildung 9: Context Map

Alle der drei eingeteilten Microservices sollen eine eigene Datenbank erhalten. Dadurch sind die Datenschemata übersichtlich und Änderungen haben nur unmittelbare Auswirkungen auf einen Microservice.

Das Frontend wird monolithisch Aufgebaut und integriert alle Funktionalitäten der Microservices. Das Frontend ist dadurch einheitlich. Das System könnte auch um andere Frontends erweitert werden. Das Frontend wird dabei eine clientseitige Webanwendung, welche die REST-Schnittstellen der Microservices konsumiert.

Da sich Interaktionen und Verkaufschancen einem Kontakt bzw. Kunden zuordnen lassen sollen, ist eine Abhängigkeit zwischen diesen Microservices zu erwarten. Auch die Kommunikation zwischen Microservices soll über die gleichen REST-Schnittstellen laufen. Daraus ergibt sich der folgende Entwurf.

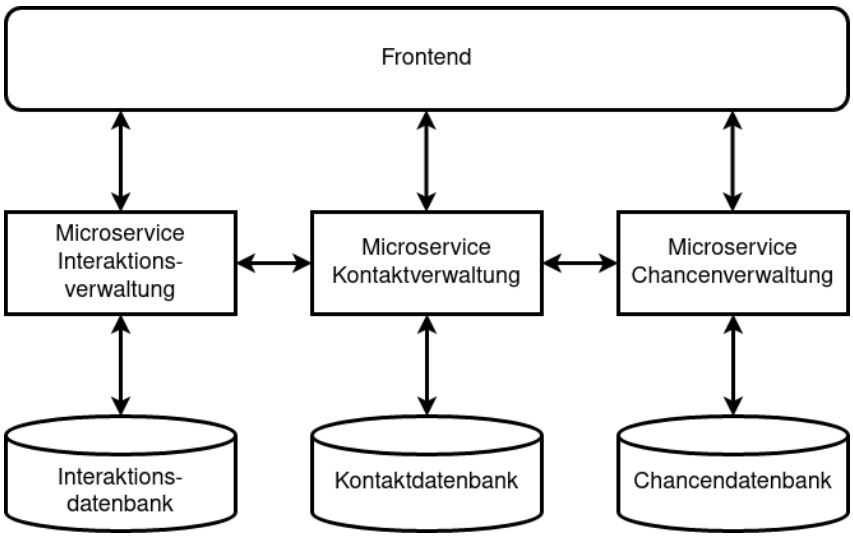

Abbildung 10: Entwurf des **CRM-System!**s

Die REST-Schnittstellen sollten einheitlich über alle Microservices hinweg sein. Die folgende Tabelle zeigt unter welchen Endpunkten, mit welchen HTTP-Methoden und welchen Parametern die REST-APIs aufrufbar sein sollen.

Endpunkt	HTTP-Methode	Parameter	Beschreibung
/contacts	GET	-	Gibt alle Kontakte zurück
/contacts	POST	neuer Kontakt	Fügt einen neuen Kontakt hinzu
/contacts/{ID}	GET	-	Gibt einen Kontakt zurück
/contacts/{ID}	PUT	veränderter Kontakt	Ändert einen Kontakt
/contacts/{ID}	DELETE	-	Löscht einen Kontakt

Tabelle 1: Prozessoren

Für das Frontend wird React verwendet. [React Erklärung] Für alle drei Microservices wird derselbe Technologie-Stack verwendet, um den Entwicklungsaufwand geringer zu halten. Es wird Java mit dem Framework Spring Boot verwendet. Als Datenbank wird MongoDB eingesetzt. [MongoDB Erklärung] Darüber hinaus wird Swagger in die Microservices integriert. Bei Swagger handelt es sich ein Werkzeug zur sprachunabhängigen Spezifikation von APIs. Swagger kann durch eine Abhängigkeit zu Spring Boot hinzu gefügt werden. Es erstellt automatisch eine Webseite mit der Dokumentation unserer API. Um das Monitoring, Service Discovery, Load Balancing wird mit Kubernetes gelöst. Unser System muss diese Aufgaben also nicht selbst bewältigen und diese Bestandteile werden erst in der Bereitstellung konfiguriert.

4.2 Micro-Architektur

Die Mikro-Architektur befasst sich mit der Architektur eines einzelnen Microservice. Für das Gesamtsystem ist die Architektur eines einzelnen Microservice nicht von Bedeutung. Aus diesem Grund besitzt man eine große Freiheit bei der Auswahl. Es sollte die Architektur, welche am Simpelsten alle Anforderungen bietet. Der ausgewählte Technologie-Stack schränkt natürlich die möglichen Architekturen auch weiter ein.

Unser System besteht aus drei Microservices und einem Frontend. Das Frontend dient lediglich zur Visualisierung und Verbindung aller Funktionen der drei Microservices. Die folgenden Domänenmodelle zeigen, welche Daten von den entsprechenden Microservices verwaltet werden müssen.

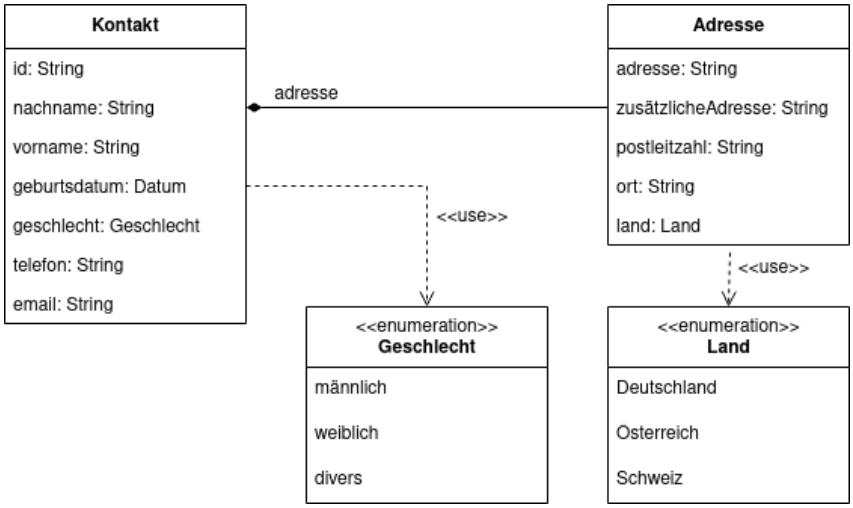

Abbildung 11: Context Map

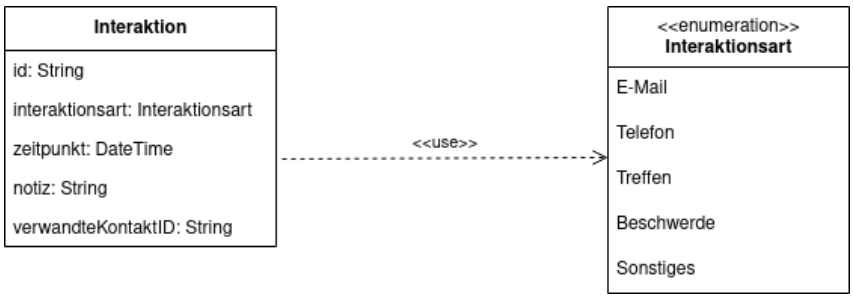

Abbildung 12: Context Map

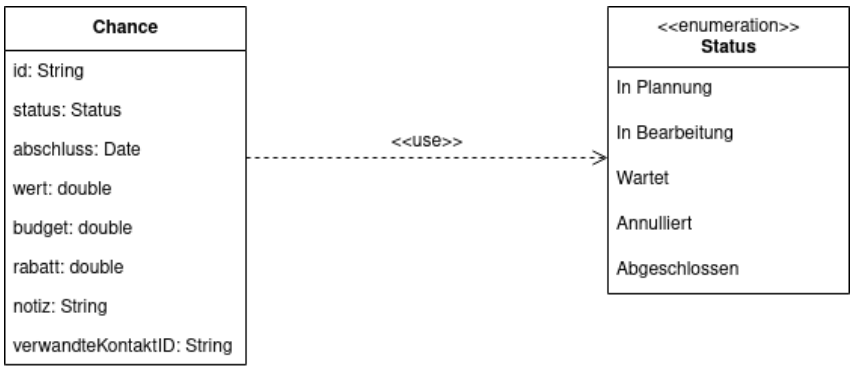

Abbildung 13: Context Map

Für die Architektur der Microservices wird eine hexagonale Architektur (Ports und Adapter) verwendet. Eine Hexagonale Architektur bietet sich

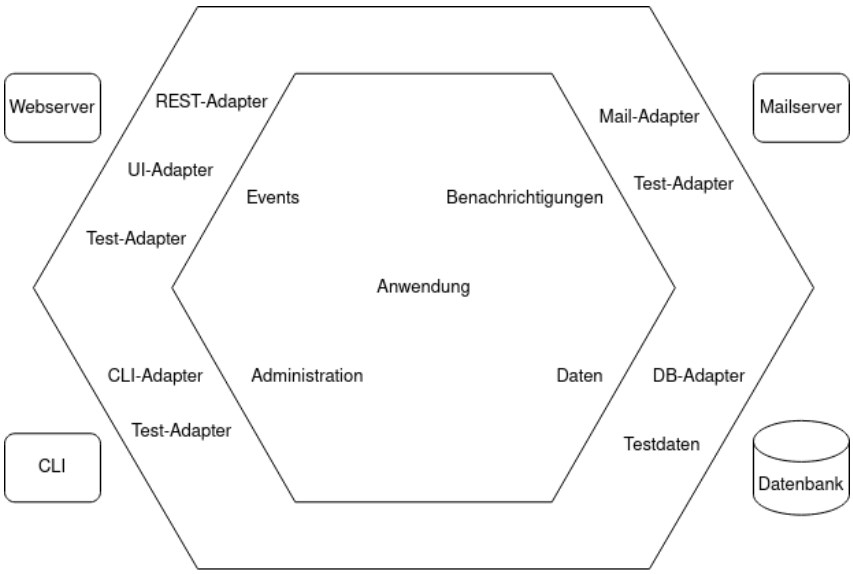

Abbildung 14: Entwurf des **Hexagonale Architektur!**s

5 Implementierung

In diesem Kapitel wird das CRM-System dem Entwurf nach implementiert. Als Erstes wird die Implementierung der Microservices erklärt, da alle drei durch den gleichen Technologie-Stack sehr ähnlich sind. Anschließend wird das Frontend, welches alle Microservices integriert, implementiert.

5.1 Microservices

Alle drei Microservices benötigen die URI ihrer Datenbank, um mit ihr eine Verbindung aufzubauen. Der Interaktions-Microservice und der Verkaufschancen-Microservice brauchen darüber hinaus die Adresse des Kontakt-Microservices, um mit diesem zu Kommunizieren. Diese Verbindungsinformationen, werden den Anwendungen über Umgebungsvariablen übergeben. Später bei der Bereitstellung, kann mit Kubernetes den Containern die passenden Umgebungsvariablen übergeben werden.

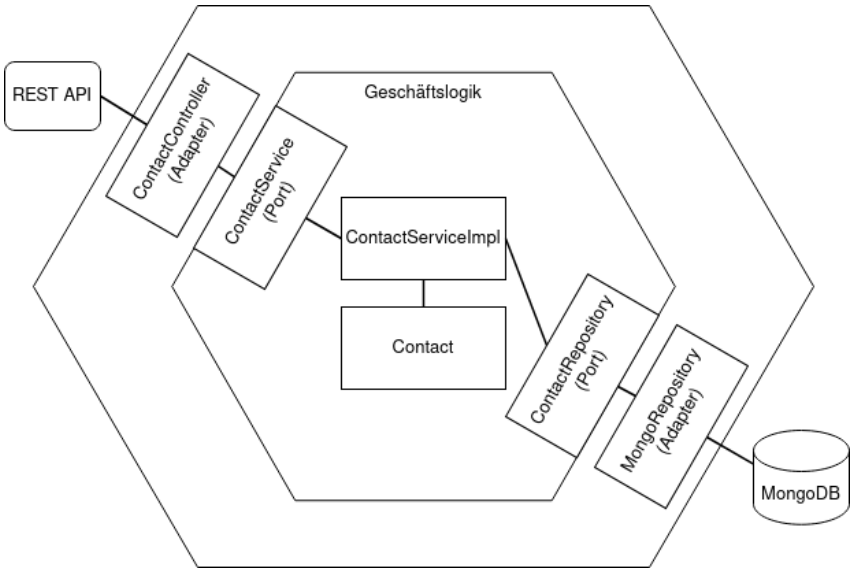

Abbildung 15: Entwurf des **Hexagonale Architektur!**s

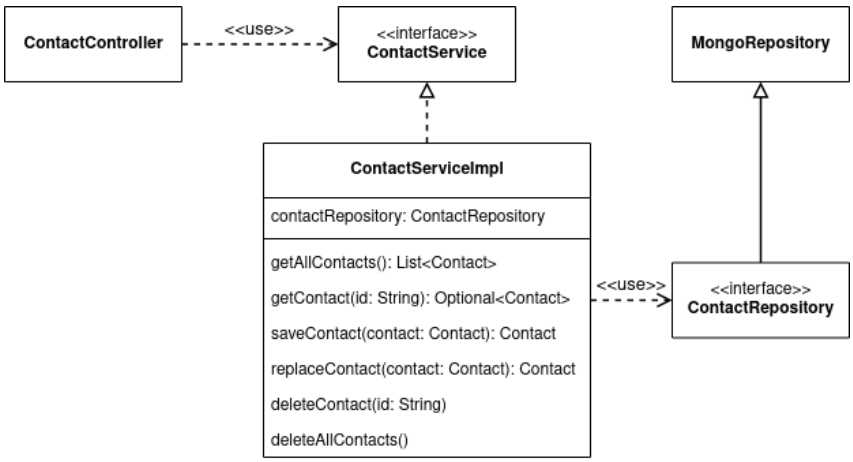

Abbildung 16: Entwurf des **Hexagonale Architektur!**s

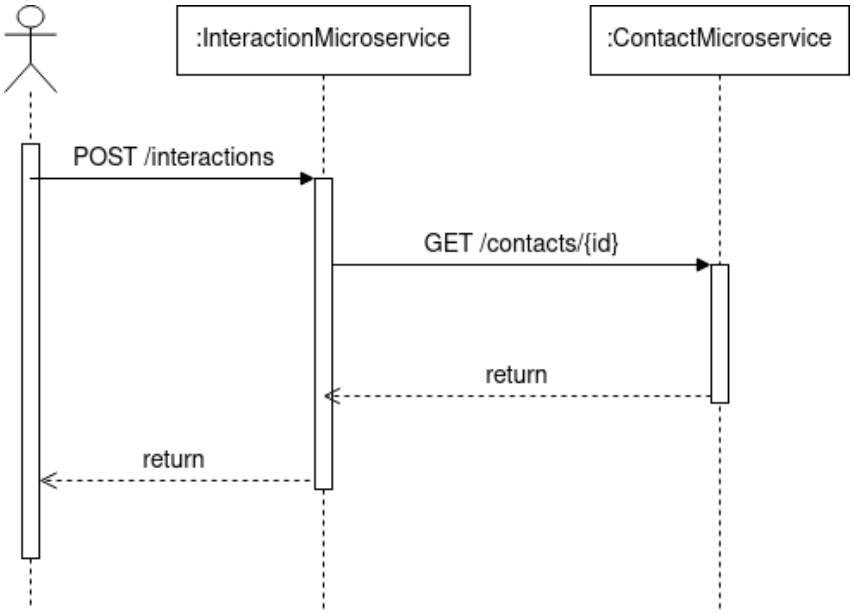

Abbildung 17: Entwurf des **Hexagonale Architektur!**s

5.2 Frontend

Das Frontend. Das Frontend soll später auch mit Kubernetes bereitgestellt werden, es wird aber nicht als Microservices angesehen. Das Frontend benötigt die URI aller Services. Auch hier werden die Verbindungsinformationen über eine Umgebungsvariable übergeben.

6 Bereitstellung mit Kubernetes

Im letzten Teil der Fallstudie wird das fertige CRM-System nun mit Kubernetes bereitgestellt.

6.1 Containerisierung

Um die Microservices und das Frontend in Pods in einem Kubernetes Cluster laufen zu lassen, müssen sie erst mit Docker containerisiert werden. Dazu wird als Erstes ein Dockerfile für jeden Microservice erstellt. Anschließend kann aus dem Dockerfile ein Docker Image gebaut werden, mit dem dann ein entsprechender Container gestartet werden kann.

Dockerfiles besitzen eine eigene Syntax. Ein großgeschriebener Befehl wird gefolgt von einem oder mehreren Parametern. Es ist aufgebaut wie eine Anleitung, welche Schritt für Schritt abgearbeitet wird. Die Dockerfiles der Microservices haben alle den selben Aufbau, da alle drei Projekte auch die selben Aufbau und die selben Technologien verwenden. Der erste Befehl in den Dockerfiles bestimmt, auf welchem Docker Image unser eigenes Image basieren soll. Hier wird ein Image mit einer Java-Plattform verwendet, welches automatisch aus dem öffentlichen DockerHub heruntergeladen wird. Anschließend wird die JAR-Datei der Anwendung in das Image kopiert. Als letzter Befehl wird festgelegt, dass die JAR-Datei beim Start des Containers ausgeführt werden soll.

```
1 FROM openjdk:11-jdk-slim
2 ARG JAR_FILE=target/*.jar
3 COPY ${JAR_FILE} app.jar
4 EXPOSE 8080
5 ENTRYPOINT ["java","-jar","/app.jar"]
```

Quellcode 1: Dockerfile für Kontakt-Microservice

Das Frontend benötigt ein eigenes Dockerfile. Dieses hat einen mehrstufigen Aufbau. Im ersten Stufe, der Build-Stage, wird unsere React-Anwendung gebaut. Um die fertig gebaute Webanwendung an einen Browser auszuliefern wird ein Webserver benötigt. In der zweiten Stufe des Dockerfiles basiert auf einem Image mit dem Webserver Nginx. Die React-Anwendung aus der Build-Stage wird nun in das finale Image kopiert.

```
1 FROM node:alpine as build
2 WORKDIR /app
3 COPY package*.json ./
4 RUN npm install --silent
5 COPY . .
```

```
6  RUN npm run build
7
8  FROM nginx:alpine
9  WORKDIR /usr/share/nginx/html
10 RUN rm -rf ./*
11 COPY --from=build /app/build .
12 ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

Quellcode 2: Dockerfile für Frontend

Für die Datenbanken müssen keine eigenen Dockerfiles erstellt werden. Hier reichen unveränderte Images, welche aus dem DockerHub heruntergeladen werden können, aus. Mit dem folgenden Befehl kann aus dem Dockerfile nun ein Docker Image gebaut werden.

```
1 docker build -t contact-microservice:latest .
```

Quellcode 3: Befehl

6.2 Bereitstellung

Für jede Datenbank wird eine Anwendung

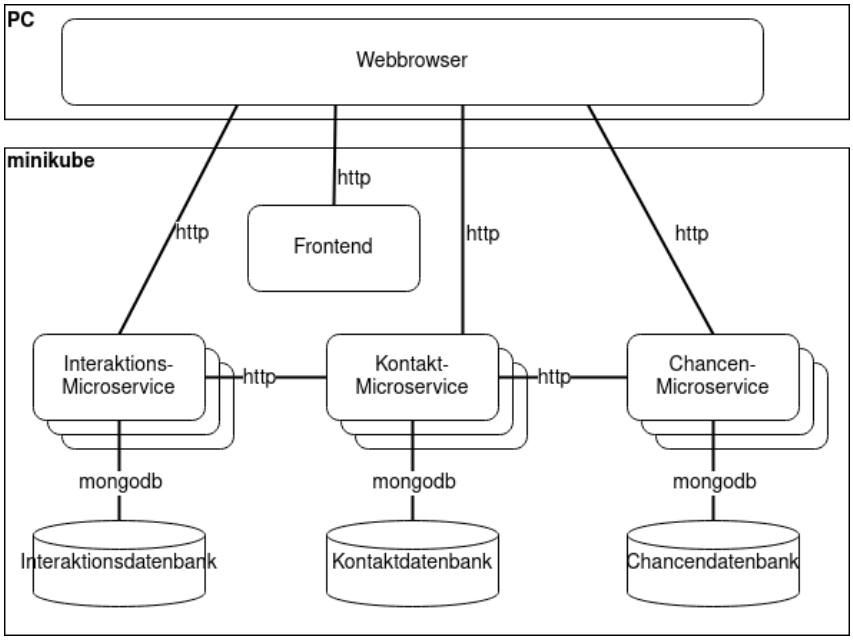

Abbildung 18: Entwurf des **Hexagonale Architektur!**s

Jetzt müssen die fertigen Images in unserem Kubernetes Cluster bereitgestellt werden. Dafür werden YAML-Dateien erstellt, in denen die gewünschten Kubernetes Objekte beschrieben werden. Für jeden der drei Microservices und das Frontend muss ein Service und ein Deployment erstellt werden. Das Deployment repräsentiert die Anwendung. Mit Parametern kann festgelegt werden wie viele Pods mit der Anwendung gestartet werden sollen.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: contact-service
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: contact-service
    spec:
      containers:
        - name: contact-service
          image: contact-microservice:latest
          imagePullPolicy: IfNotPresent
          ports:
          - containerPort: 8080
          env:
            - name: MONGODB_HOST
              valueFrom:
                configMapKeyRef:
                  name: contact-db-config
                  key: host
```

Quellcode 4: Befehl

```
apiVersion: v1
kind: ConfigMap
metadata:
   name: contact-service-config
data:
 host: contact-service
```

Quellcode 5: Befehl

Nach der Erstellung der Services wird die Service Discovery und Lasterverteilung von Kubernetes übernommen.

```
kind: Service
apiVersion: v1
```

```
3  metadata:
4    name: contact-service
5  spec:
6    selector:
7      app: contact-service
8    ports:
9    - protocol: TCP
10     port: 8080
11     nodePort: 30010
12   type: NodePort
```

Quellcode 6: Befehl

Nun müssen nur noch alle YAML-Dateien über den folgenden kubectl-Befehl angewendet werden. Kubernetes sorg nun dafür, dass die beschriebenen Objekte, erstellt werden.

```
1  kubectl apply -f contact-microservice.yaml
```

Quellcode 7: Befehl

6.3 Skalierung

Um die Vorteile von unseren Microservices auszunutzen, sollen die Microservices nun horizontal skaliert werden. Auch dafür wird eine YAML-Datei erstellt, in der ein HorizontalPodAutoscaler-Objekt für jeden Microservice beschrieben wird. In der Datei wird angegeben, welches Deployment skaliert werden soll. Als Metrik, wann hochskaliert werden soll, kann die CPU-Auslastung des Pods verwendet werden. Darüber hinaus wird angegeben wie viele Pods von dem entsprechenden Microservice minimal und maximal ausgeführt werden sollen.

```
1  apiVersion: autoscaling/v1
2  kind: HorizontalPodAutoscaler
3  metadata:
4      name: contact-service
5  spec:
6      scaleTargetRef:
7          apiVersion: apps/v1
8          kind: Deployment
9          name: contact-service
10     minReplicas: 2
11     maxReplicas: 4
12     targetCPUUtilizationPercentage: 80
```

Quellcode 8: Befehl

7 Schlussbetrachtung

7.1 Diskussion

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

7.2 Ausblick

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Literatur

Bücher

Pasteur, Louis (1933). *Oeuvres de Pasteur*. Paris: Médiathèque scientifique de l'Institut Pasteur. URL: https://gallica.bnf.fr/ark:/12148/bpt6k6211139g (besucht am 26. 01. 2022).

Liebel, Oliver (2021). *Skalierbare Container-Infrastrukturen: Das Handbuch für Administratoren*. ISBN: 9783836277747 9783836277730.

Arundel, John und Justin Domingus (2019). *Cloud Native DevOps mit Kubernetes: Bauen, Deployen und Skalieren moderner Anwendungen in der Cloud*. 1. Auflage. Heidelberg: dpunkt.verlag. 342 S. ISBN: 978-3-86490-698-5.

Halstenberg, Jürgen, Bernd Pfitzinger und Thomas Jestädt (2020). *DevOps: ein Überblick*. essentials. Wiesbaden [Heidelberg]: Springer Vieweg. 52 S. ISBN: 978-3-658-31404-0.

Tremp, Hansruedi (2021). *Architekturen verteilter Softwaresysteme: SOA & Microservices - Mehrschichtenarchitekturen - Anwendungsintegration*. Erfolgreich studieren. Wiesbaden [Heidelberg]: Springer Vieweg. 180 S. ISBN: 978-3-658-33179-5 978-3-658-33178-8.

Newman, Sam (2015). *Microservices: Konzeption und Design*. 1. Aufl. Frechen: mitp-Verl. 318 S. ISBN: 978-3-95845-081-3.

Wolff, Eberhard (2018). *Microservices: Grundlagen flexibler Softwarearchitekturen*. 2., aktualisierte Auflage. Heidelberg: dpunkt.verlag. 374 S. ISBN: 978-3-86490-555-1.

Salus, Peter H. (1994). *A Quarter Century of UNIX*. Reading, Mass: Addison-Wesley Pub. Co. 256 S. ISBN: 978-0-201-54777-1.

Hightower, Kelsey, Brendan Burns und Joe Beda (2018). *Kubernetes: eine kompakte Einführung*. Übers. von Thomas Demmig. 1. Auflage. Heidelberg: dpunkt.verlag. 191 S. ISBN: 978-3-86490-542-1.

Artikel

Conway, Melvin (Apr. 1968). „How Do Committees Invent?“ In: *Datamation*. URL: http://www.melconway.com/Home/Committees_Paper.html (besucht am 27. 01. 2022).

Peltonen, Severi, Luca Mezzalira und Davide Taibi (2021). „Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review“. In: *Information and Software Technology* 136, S. 106571. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106571. URL: https://www.sciencedirect.com/science/article/pii/S0950584921000549.

Berichte

Cloud Native Computing Foundation (17. Nov. 2020). *Cloud Native Survey*. San Francisco. URL: https://www.cncf.io/blog/2020/11/17/cloud-native-survey-2020-containers-in-production-jump-300-from-our-first-survey/ (besucht am 26. 01. 2022).

Mordor Intelligence (2020). *Global Cloud Microservices Market*. Hyderabad. URL: https://www.mordorintelligence.com/industry-reports/cloud-microservices-market (besucht am 26. 01. 2022).

Konferenzpapier

Burns, Brendan und David Oppenheimer (Juni 2016). „Design Patterns for Container-based Distributed Systems“. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). USENIX Association. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns.

Online

Docker Inc. (2022). *Docker Documentation*. Docker Documentation. URL: https://docs.docker.com/ (besucht am 01. 02. 2022).

Linux Foundation (2022). *Kubernetes Documentation*. Kubernetes. URL: https://kubernetes.io/docs/home/ (besucht am 01. 02. 2022).

Sonstige

Evans, Eric (2015). *Domain-Driven Design Reference*. URL: https://www.domainlanguage.com/ddd/reference/ (besucht am 01. 02. 2022).

Fielding, Roy (2000). „Architectural Styles and the Design of Network-based Software Architectures“. University of California. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (besucht am 28. 01. 2022).

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als die angegebenen verwendet habe.

München, den 1. Februar 2022