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

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, den 4. Februar 2022

Zusammenfassung

Beim Entwurf von modernen Anwendungen etabliert sich zunehmend das Architekturmuster der Microservices. Microservices teilen große Anwendungen in kleine unabhängige Programme auf, welche verteilt ausgeführt werden und über feste Schnittstellen miteinander kommunizieren. Kubernetes, eine Software zur Containerorchestrierung, hilft dabei, die vielen Microservice bereitzustellen.

In der vorliegenden Bachelorarbeit wird eine moderne verteilte Webanwendung mit einer Microservice-Architektur entworfen und mithilfe von Kubernetes bereitgestellt. Dazu werden im ersten Teil der Arbeit die theoretischen Grundlagen dargelegt. Darauf aufbauend wird ein Fallbeispiel an einem vereinfachten CRM-System durchgeführt. Ziel ist es, ein Verfahren vom Entwurf bis zur Bereitstellung zu implementieren, um Aussagen zur Umsetzung und dem Anwendungsgebiet von containersisierten Microservices mit Kubernetes zu treffen.

Durch den theoretischen Hintergrund und die praktische Anwendung zeigt sich, dass Microservices hauptsächlich bei großen Systemen mit einer hohen fachlichen Breite von Vorteil sind. Vor allem die fachliche Einteilung der Microservices gestaltet sich als herausfordernd. Die vielen Entscheidungen, welche bei der Microservice-Architektur getroffen werden müssen erhöhen das Risiko für Fehler. Die höhere Flexibiltät von Microservices geht mit einer höheren Komplexität des Gesamtsystems einher. Bei der Bereitstellung kann Kubernetes mit wenig Konfigurationsaufwand viele Aufgaben wie die ServiceDiscovery, die Lastverteilung und die Skalierung übernehmen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

CRM Customer-Relationship-Management

CI Continuous Integration

CD Continuous Delivery

UNIX Uniplexed Information and Computing Service

API Application Programming Interface

SOA Serviceorientierte Architektur

DDD Domain-driven Design

UI User Interface

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

URI Uniform Resource Identifier

DNS Domain Name System

HPA Horizontal Pod Autoscaler

CLI Command Line Interface

B2C Business-to-Customer

JSON JavaScript Object Notation

UML Unified Modeling Language

HTML Hypertext Markup Language

CSS Cascading Style Sheets

JAR Java Archive

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 unterstützt bei der Steuerung der riesigen Containermengen. Zusammen bilden all diese Innovationen den Grundstein für moderne Anwendungen aus containerisierten Microservices, welche mit Kubernetes verwaltet werden. In dieser Arbeit wird die Kombination dieser Innovationen genauer betrachtet, um die Chancen 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 also 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 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 ein Fallbeispiel durchzuführen. Das Fallbeispiel wird an einem System zum Customer-Relationship-Management (CRM) durchgeführt. In dem Fallbeispiel soll ein Verfahren vom Entwurf bis zur Bereitstellung implementiert werden. Um Aussagen zur Umsetzung und dem Anwendungsgebiet von containerisierten Microservices mit Kubernetes zu treffen, soll 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 das Fallbeispiel durchgeführt. In Kapitel 3 wird die Problemstellung des Fallbeispieles 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 es 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 diese Ziele zu erreichen, werden etablierte Methoden und Werkzeuge eingesetzt. Die einzelnen Phasen der Entwicklung und des Betriebes bilden gemeinsam einen Prozess, der von jedem Programminkrement durchlaufen wird [vgl. Tremp 2021, S. 63].

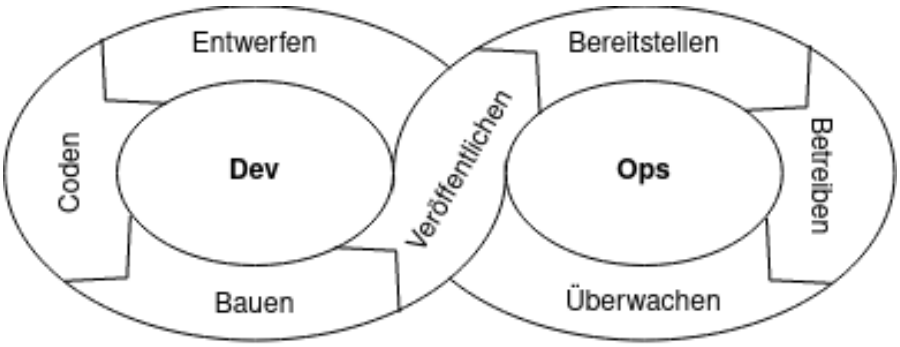


Abbildung 1: Kreislauf und Schritte von DevOps [vgl. Tremp 2021, S. 63]

Durch eine Automatisierung, Continuous Integration (CI) und Continuous Delivery (CD) 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 schwieriger zu trennen ist. DevOps in ein

Unternehmen einzuführen ist ein langwieriger Prozess. Neben der Einführung der neuen Technologien ist vor allem die Transformation der Unternehmenskultur besonders mühevoll. 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.

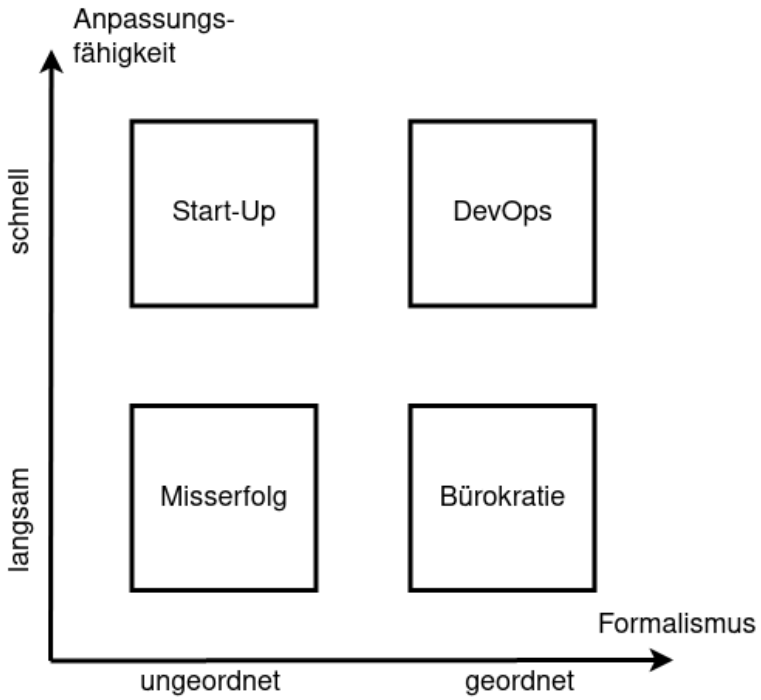


Abbildung 2: Kategorisierung von Unternehmen nach Anpassungsfähigkeit und Formalismus [vgl. Halstenberg u. a. 2020, S. 11]

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 des Monolithen lassen sich diese Probleme abschwächen, können jedoch nicht vollständig behoben werden.

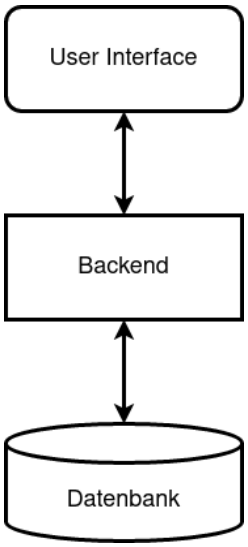


Abbildung 3: Beispielhafter Aufbau einer monolithischen Architektur

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.

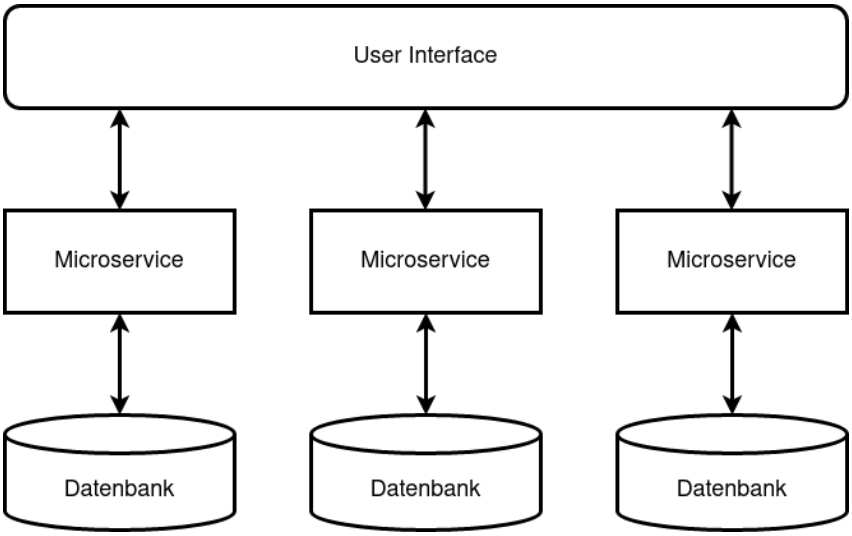


Abbildung 4: Beispielhafter Aufbau einer Microservice-Architektur

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 zudem zustandslos 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 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 werden die Entwicklerteams crossfunktional mit Spezialisten aus verschiedenen Fachbereichen aufgebaut werden. Änderungen betreffen so häufig nur ein Entwicklerteam und der Koordinationsaufwand sinkt.

Microservices werden immer wieder 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 technischen Hürden für unerwünschte Abhängigkeiten sind somit deutlich höher.

Durch die expliziten Schnittstellen ist es auch sehr leicht, 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 hinaus 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. 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 ein einzelner Service. Dadurch, dass nur der veränderte Microservice neu bereitgestellt werden muss, ist die Bereitstellung schneller als bei einem Monolithen. Die Time-to-Market kann so verkürzt werden. 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 die Last kann durch Microservices besser verteilt werden, da sie verstreut 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 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 die 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 ebenso 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, gestaltet sich das Refactoring über mehrere Microservices hinweg als deutlich komplizierter. Bei Monolithen können Teile des Codes bequem von einer Komponente in eine Andere verschoben werden kann. Bei Microservices müssen die Teile in einen anderes eigenständiges Programm verschoben werden, welches vielleicht sogar eine andere Programmiersprache 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 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 Mikro-Architektur und 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.

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 sollte 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 idealerweise nur einen Microservice und ein

Entwicklerteam betreffen. 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 Mikro-Architektur 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 später noch genauer beschrieben.

2.2.5 Integration

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

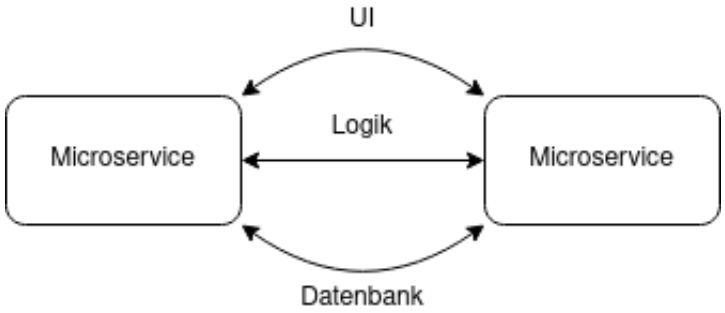


Abbildung 5: Integration von Microservices auf verschiedenen Ebenen [vgl. Wolff 2018, S. 167]

Auf Datenbank-Ebene können Microservices integriert werden, indem sie auf dieselbe Datenbank zugreifen. Diese Methode ist einfach zu implementieren, aber bringt einige 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 beschränkt. Eigene Datenbanken oder zumindest Datenbankschemata erlauben also 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 eigenen UI auszustatten. Somit könnten alle Änderungen, egal ob sie UI, Logik, oder Datenbank betreffen, von einem Entwicklerteam umgesetzt werden. Das Problem ist jedoch, dass UIs 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].

Microservices können auch 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. Ein 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 demselben Kernel ausgeführt werden. Dadurch sind die Container ressourcenschonender als die herkömmliche Virtualisierung mittels Hypervisor, bei dem jede virtuelle Maschine ein vollständiges Betriebssystem ausführt [vgl. Newman 2015, S. 166f].

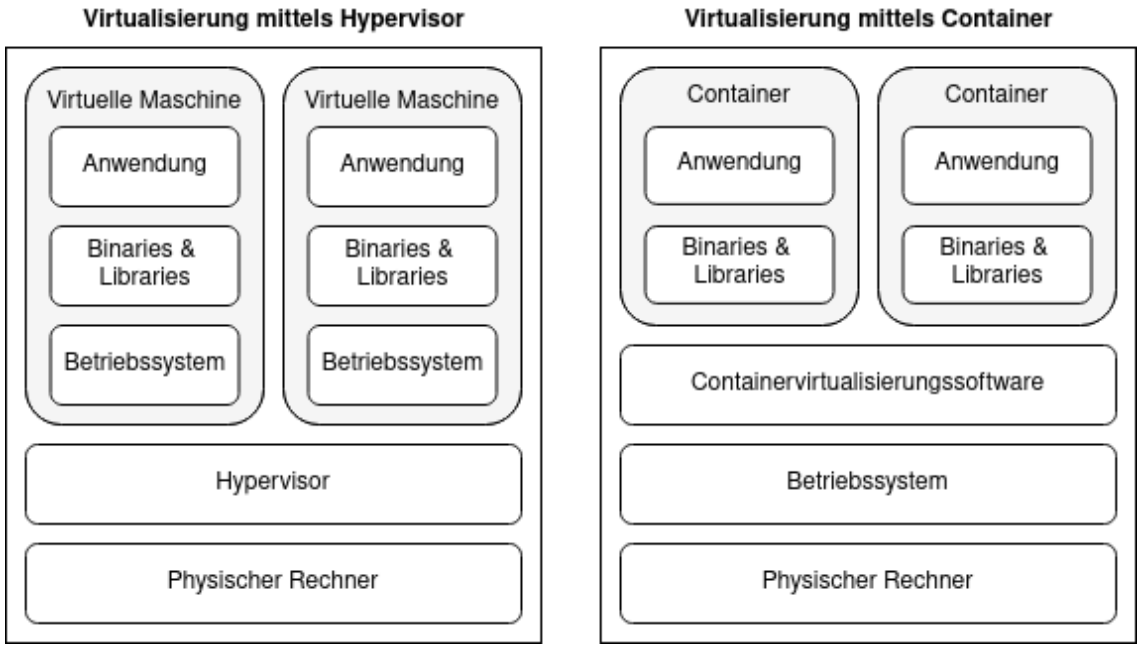


Abbildung 6: Vergleich Virtualisierung mittels Hypervisor und Container

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 einer standardisierten Image-Datei, welche 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 und ergänzt die Containervirtualisierung um benutzerfreundliche Werkzeuge [vgl. Hightower u. a. 2018, S. 20]. 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 benutzen. 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 die einzige Abhängigkeit.

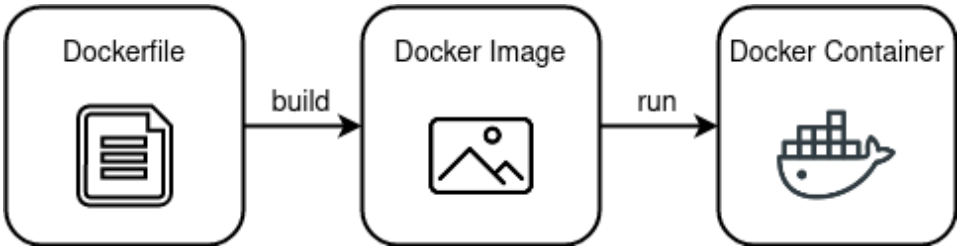


Abbildung 7: Weg vom Dockerfile zum Container

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 gestalten 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 dem 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.

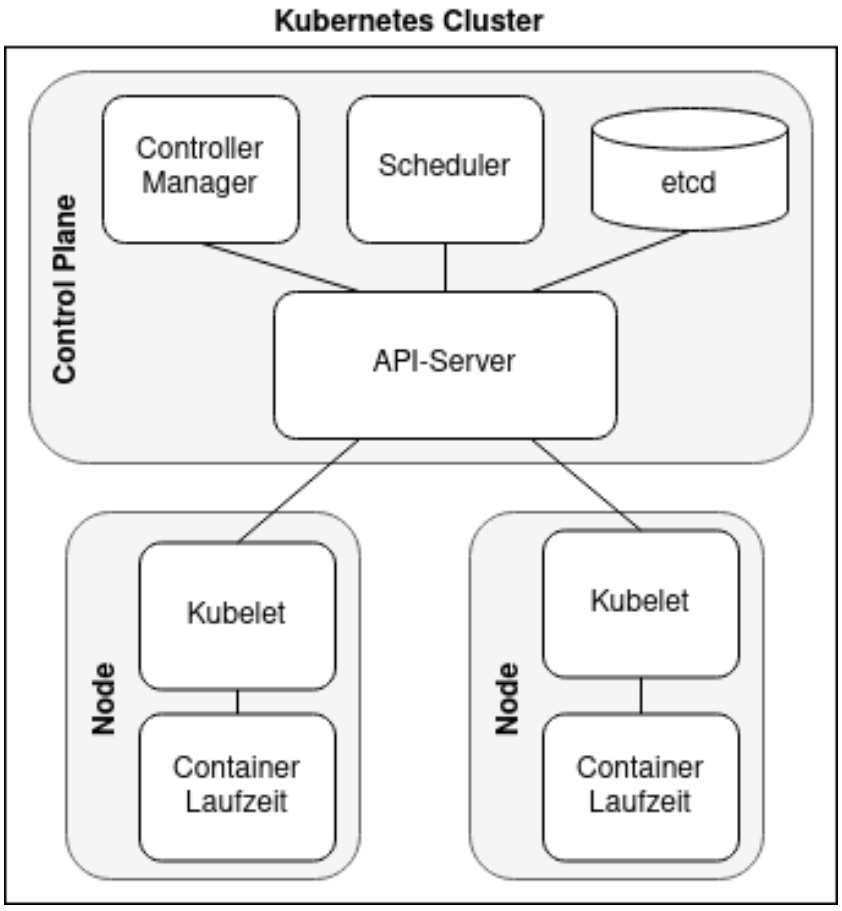


Abbildung 8: Aufbau eines Kubernetes Cluster

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 [vgl. Hightower u. a. 2018, S. 13]. Diese Objekte werden in Manifesten beschrieben. Bei den Manifesten handelt es sich üblicherweise 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 einen einzelnen Container oder eine Gruppe von Containern. Alle Container in einem Pod laufen immer auf dem gleichen Node.
- 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.
- Labels können anderen Objekten zugeordnet werden, um diese zu Gruppieren.
- ReplicaSets sorgen dafür, dass eine bestimmte Anzahl von Kopien eines Pods gleichzeitig laufen.
- Deployments repräsentieren eine zustandslose Anwendung. Deployments verwenden ReplicaSets. Ein Deployment überwacht fortlaufend den Status der Container und kann sie beispielsweise bei Bedarf neu starten.

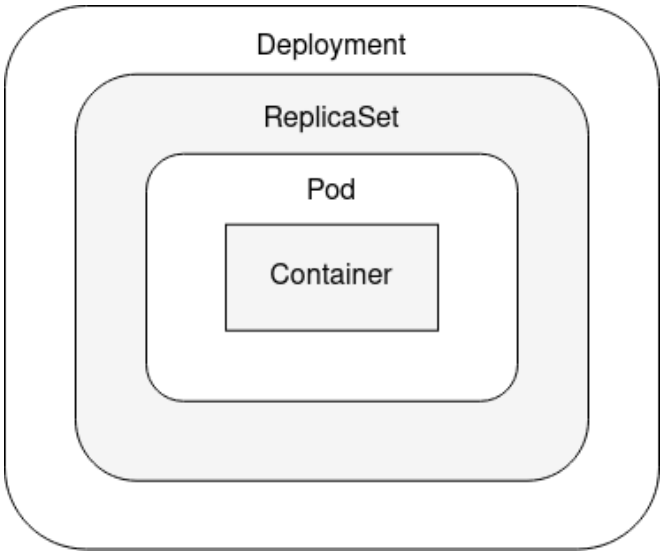


Abbildung 9: Aufbau von einem Deployment

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

2.4.3 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 Endpunkt 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 dem Domain Name System (DNS) aufgelöst wird. Ein Service vom Typ ClusterIP ist von außerhalb des Clusters nicht aufrufbar.

Mit dem Typ NodePort kann der Service von außen zugänglich gemacht werden. Dazu wird ein fester Port auf allen Nodes geöffnet und der Datenverkehr über diesen Port an den entsprechenden Service weitergeleitet.

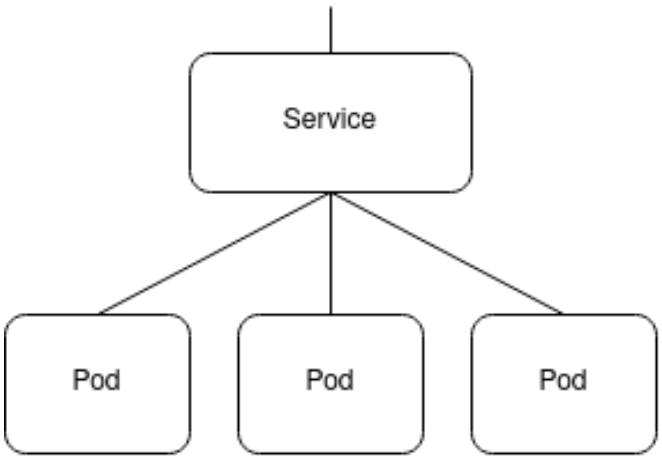


Abbildung 10: Service als Endpunkt für mehrere Pods [vgl. Arundel und Domingus 2019, S. 67]

2.4.4 Skalierung

Über ReplicaSets lässt sich eine feste Anzahl von Kopien eines Pods festlegen. In der Praxis soll die Anzahl häufig nach der Auslastung skaliert werden. Diese Aufgabe kann Kubernetes automatisieren. Dafür wird das das Objekt Horizontal Pod Autoscaler (HPA) angeboten. Mit dem HPA kann beispielsweise eine Skalierung in Abhängigkeit der CPU-Auslastung vorgenommen werden. In dem Objekt wird auch angegeben, wie viele Kopien mindestens vorhanden sein sollten und wie viele maximal erstellt werden dürfen.

2.4.5 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.

2.4.6 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. Dadurch kann das Cluster mithilfe von kubectl gesteuert werden kann.

3 Erläuterung des Fallbeispieles

Ziel des Fallbeispieles ist es, ein Verfahren vom Entwurf bis zur Bereitstellung von containerisierten Microservices mit Kubernetes zu implementieren. Auf Basis dieses Verfahrens sollen Aussagen zur Umsetzung und dem Anwendungsgebiet getroffen werden können. Als Beispiel wird ein vereinfachtes CRM-System verwendet. In diesem Kapitel werden die Vorgaben an das Fallbeispiel beschrieben.

3.1 Anwendungseinsatz

Ein CRM-System ist eine Software für das Kundenbeziehungsmanagement. CRM-Systeme sind komplexe betriebliche Anwendungssysteme. Durch ihre Größe und ihre vielen verwobenen Dienste gestaltet sich der Entwurf und die Weiterentwicklung häufig schwierig. CRM-Systeme haben eine große fachliche Breite, was zu einem hohen Koordinationsaufwand bei der Entwicklung und der Bereitstellung führt [vgl. Tremp 2021, S. 62]. Somit eignet sich ein CRM-System als gutes Beispiel für die Umsetzung mit einer Microservice-Architektur, welche die Probleme weitgehend beheben soll.

Das CRM-System soll für das Business-to-Customer (B2C) Umfeld entwickelt werden. Es soll bei der Verwaltung von Kontakten beziehungsweise Kunden helfen. Zu jedem Kontakt sollen Informationen und eine Historie mit allen Interaktionen abgespeichert werden. Darüber hinaus soll es auch möglich sein, Verkaufschancen zu verwalten und einem Kontakt zuzuordnen.

3.2 Anwendungsfunktionen

Die Kernfunktionalität des zu erstellenden CRM-Systems ist das Anlegen, Anzeigen, Bearbeiten und Löschen von Kontakten, Interaktionen und Verkaufschancen. Konkret sollen die folgenden funktionalen Anforderungen von dem System erfüllt werden:

- Kontakte sollen mit Identifikationsnummer, Name, Geburtsdatum, Geschlecht, Telefonnummer, E-Mail-Adresse und Adresse angelegt, angezeigt, geändert und gelöscht werden können.
- Interaktionen mit einem Kontakt sollen mit Identifikationsnummer, Art der Interaktion, Datum, Uhrzeit, Notizen und dem zugehörigen Kontakt angelegt, angezeigt, geändert und gelöscht werden können.
- Mögliche Verkaufschancen sollen mit Identifikationsnummer, 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 einfache grafische Benutzeroberfläche mit dem Webbrowser bedienbar sein. Des Weiteren sollen alle Funktionen auch über APIs angesteuert werden können, um die Integrierbarkeit mit anderen Systemen zu erleichtern. Durch die lose Kopplung einer Microservice-Architektur, soll eine flexible Skalierung und Erweiterbarkeit der Anwendung gewährleistet werden. Authentifizierung, Autorisierung und andere Sicherheitsfunktionen sollen nicht beachtet werden.

4 Entwurf der Microservices

In diesem Kapitel wird die Microservice-Architektur des CRM-Systems entworfen. Dabei wird zuerst die Makro-Architektur des Gesamtsystems ausgearbeitet und anschließend die Mikro-Architektur der einzelnen Microservices festgelegt.

4.1 Makro-Architektur

Die Makro-Architektur muss besonders sorgfältig konzipiert werden, da Veränderungen auf dieser Ebene zu einem späteren Zeitpunkt sehr aufwendig werden können. Das Wichtigste ist eine gute fachliche Aufteilung der Microservices. Für die Aufteilung wird nach dem DDD vorgegangen. Die Anwendung lässt sich in drei Bounded Contexts unterteilen: Kontaktverwaltung, Interaktionsverwaltung und Chancenverwaltung. Jeder dieser drei Bereiche besitzt ein eigenes Datenobjekt für einen Kontakt, eine Interaktion oder eine Chance. Eine Interaktion und eine Chance sollen einem Kontakt zuogeordnet werden können, deshalb benötigen diese Bounded Contexts einen Teil der Kontaktdaten. Die Identifikationsnummer eines Kontaktes reicht aus, um einer Interaktion oder Chance einen eindeutigen Kontakt zuzuordnen. Daraus ergibt sich die folgende Context Map, mit den drei Bounded Contexts und den Daten, an denen jeder Bounded Context interessiert ist. Anhand der identifizierten Kontextgrenzen wird die Anwendung in einen Kontakt-Microservice, einen Interaktions-Microservice und einen Chancen-Microservice aufgeteilt.

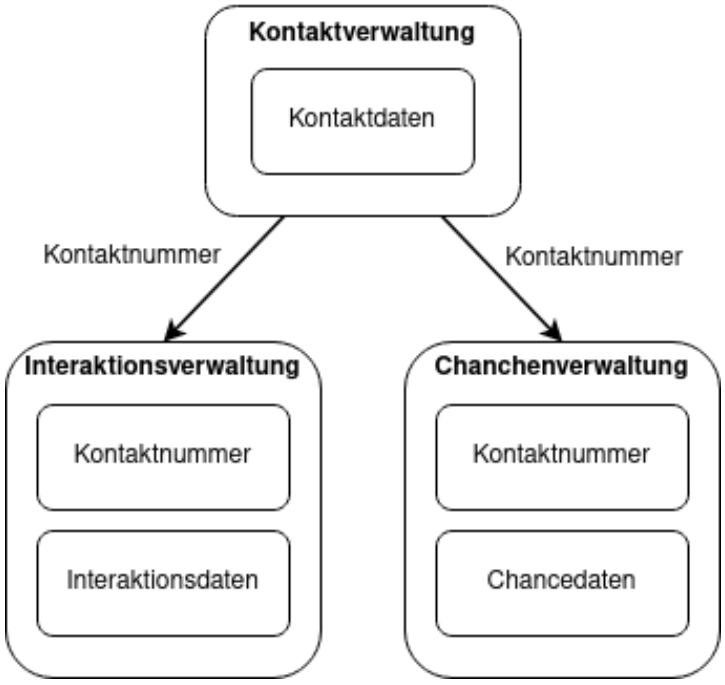


Abbildung 11: Context Map

Nachdem die fachliche Einteilung erledigt ist, kann jetzt die technischen Architektur festgelegt werden. Zur flexiblen Skalierbarkeit müssen die Microservices zustandslos sein. Alle persistenten Daten werden also in einer Datenbank abgelegt. Um eine möglichst große Unabhängigkeit zwischen den Microservices zu haben, wird jedem Microservice eine eigene Datenbank mit einem eigenen Datenbankschema zugeordnet. So können Datenstrukturen geändert werden, ohne unbeabsichtigte Auswirkungen auf andere Microservices zu haben. Der Interaktions-Microservice und der Chancen-Microservice benötigen für die Zuordnung zu einem Kontakt Informationen vom Kontakt-Microservice. Zwischen diesen Microservices ist somit eine Kommunikation nötig. Gibt es zu viele solcher Abhängigkeiten oder zyklische Aufrufe, sollte die Einteilung der Microservices überarbeitet werden. Das ist bei dieser Aufteilung jedoch nicht der Fall. Da alle Funktionalitäten auch über APIs aufrufbar sein sollen, bietet es sich an, die Kommunikation zwischen den Microservices auch über diese APIs abzuwickeln. Um die Microservices klein zuhalten, wird ein zentrales Frontend entwickelt werden, welches alle Funktionalitäten der Microservices bündelt. Um die Funktionalitäten zu integrieren, greift das Frontend auch auf die APIs der Microservices zu. Somit ergibt sich der folgende Architekturentwurf aus drei verschiedenen Microservices mit drei zugehörigen Datenbanken und einem Frontend.

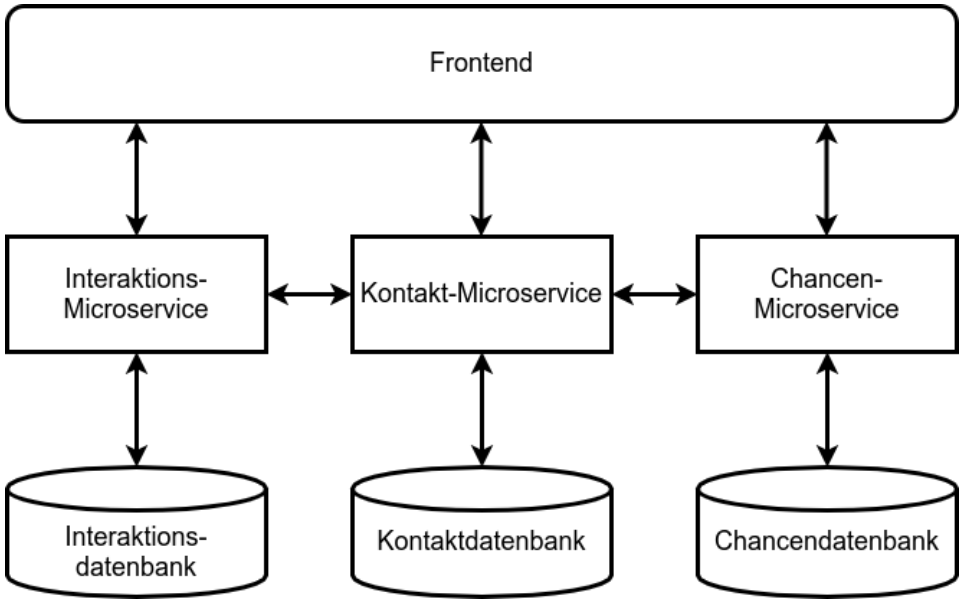


Abbildung 12: Makro-Architektur des CRM-Systems

Als Nächstes müssen die APIs der Microservices spezifiziert werden. Die API wird nach dem REST-Architekturstil entworfen. Die Kommunikation erfolgt dabei über Hypertext Transfer Protocol (HTTP). Die REST-Ressourcen sollen im JavaScript Object Notation (JSON)-Format dargestellt werden. Für jeden Microservice werden die Endpunkte, unter der die API aufrufbar ist und die HTTP-Anfragemethode bestimmt.

Endpunkt	Methode	Beschreibung
Kontakt-Microservice		
/contacts	GET	Gibt alle Kontakte zurück
/contacts	POST	Fügt einen neuen Kontakt hinzu
/contacts/{ID}	GET	Gibt einen Kontakt zurück
/contacts/{ID}	PUT	Ändert einen Kontakt
/contacts/{ID}	DELETE	Löscht einen Kontakt
Interaktions-Microservice		
/interactions	GET	Gibt alle Interaktionen zurück
/interactions	POST	Fügt einen neue Interaktion hinzu
/interactions/{ID}	GET	Gibt eine Interaktion zurück
/interactions/{ID}	PUT	Ändert eine Interaktion
/interactions/{ID}	DELETE	Löscht eine Interaktion
Chancen-Microservice		
/opportunity	GET	Gibt alle Chancen zurück
/opportunity	POST	Fügt einen neue Chance hinzu
/opportunity/{ID}	GET	Gibt eine Chance zurück
/opportunity/{ID}	PUT	Ändert eine Chance
/opportunity/{ID}	DELETE	Löscht eine Chance

Abbildung 13: Entwurf der REST-API

Service Discovery und Lastverteilung muss das System nicht selbst erledigen. Diese Funktionen werden von Kubernetes übernommen und erst bei der Bereitstellung konfiguriert.

4.2 Mikro-Architektur

Für das Gesamtsystem ist die Architektur eines einzelnen Microservice nicht von Bedeutung. Dadurch besitzt man bei der Gestaltung der Mikro-Architektur und vor allem bei der Auswahl des Technologie-Stacks viele Freiheiten. In diesem Fallbeispiel werden ausschließlich bewährte und beliebte Technologien eingesetzt. Die drei Microservices werden mit demselben Technologie-Stack und mit einer gleichartigen Architektur entworfen, um den Entwicklungsaufwand zu reduzieren. Es wäre aber auch möglich, alle Microservices mit verschiedenen Technologien zu implementieren, solange sie die gewünschten Schnittstellen anbieten können. Es wird Java in Verbindung mit dem Spring Boot Framework verwendet. Spring Boot bietet ein einsatzfertiges Baugerüst für Java-Anwendungen und eine einfache Auswahl von benötigten Abhängigkeiten, wie beispielsweise Datenbanktreibern. Auch REST-APIs werden von Spring Boot unterstützt. Spring Boot ist der De-Facto-Standard für Microservices in Java [vgl. VMware, Inc. 2022]. Für die Datenbanken wird MongoDB verwendet. MongoDB ist ein modernes dokumentorientiertes Datenbankmanagementsystem [vgl. MongoDB, Inc. 2022]. Daten werden in Collections als Dokumente mit einem JSON-ähnlichen Aufbau verwaltet. Die Datenstrukturen sind flexibel und können leichter

umstrukturiert werden als bei klassischen Datenbanken. MongoDB wird auch als NoSQL-Datenbank bezeichnet.

Für alle drei Microservices werden Domänenmodelle erstellt. Die Domänenmodelle enthalten alle fachlichen Entitäten sowie deren Eigenschaften und Beziehungen. Für die Modellierung wird ein Klassendiagramm nach der Unified Modeling Language (UML) verwendet [vgl. *Unified Modeling Language Specification* 2017]. UML ist eine grafische Modellierungssprache, welche im Laufe der Arbeit noch häufiger eingesetzt wird. Die Domänenmodelle enhalten die gewünschten Eigenschaften aus den Anforderungen. Beim Kontakt-Microservice besteht das Datenmodell aus zwei Entitäten. Ein Kontakt besteht aus einer Adresse und mehreren anderen Eigenschaften, wie dem Namen, dem Geburtsdatum und einer Mail-Adresse. Da MongoDB eine NoSQL-Datenbank ist, besitzen die beiden Entitäten keine relationale Beziehung. Die Adresse wird verschachtelt im MongoDB-Dokument vom Kontakt abgespeichert werden. Des Weiteren enthält das Modell zwei Enumerationen für Auswahl des Geschlechts und der Nationalität.

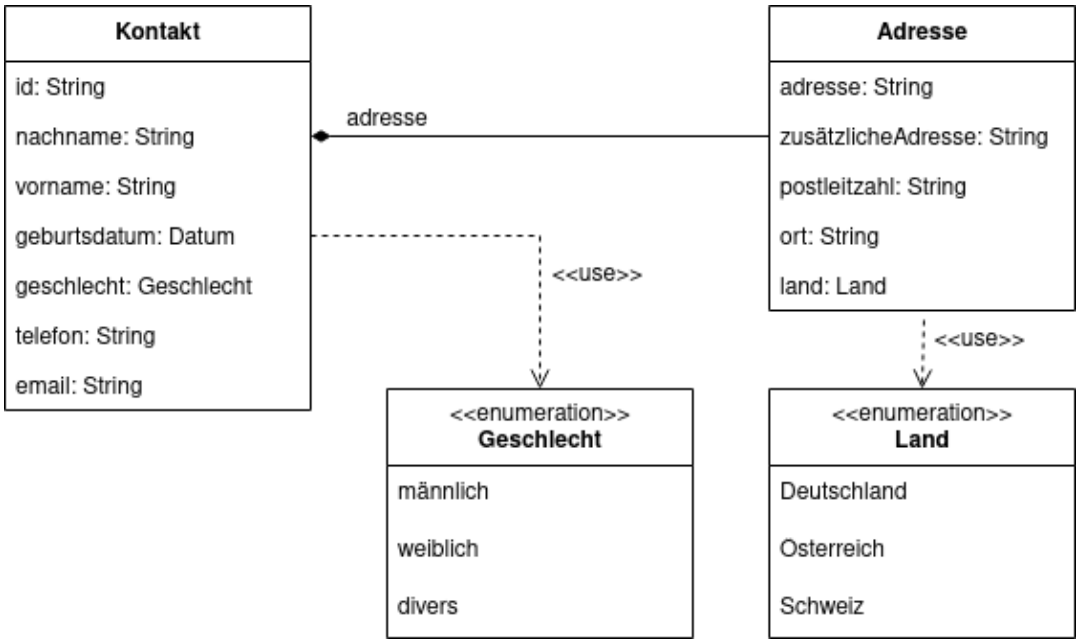


Abbildung 14: Domänenmodell für den Kontakt-Microservice

Das Datenmodell vom Interaktions-Microservice besteht lediglich aus einer einzelnen Entität und einer Enumeration für die Interaktionsart.

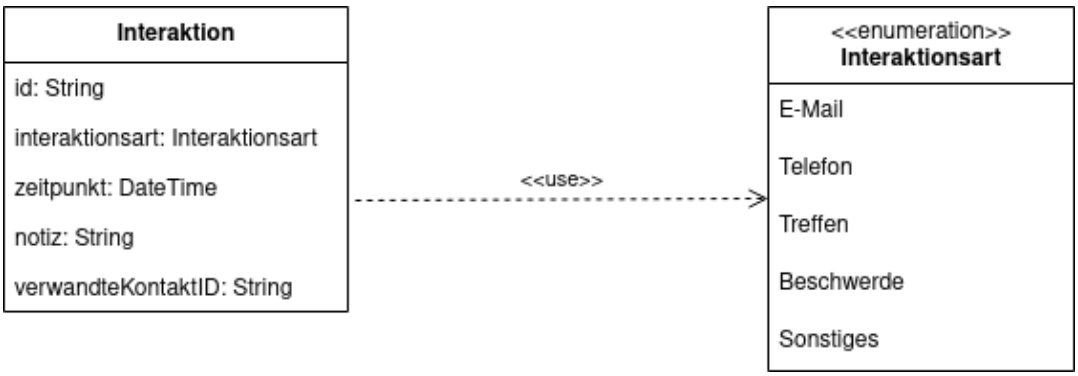


Abbildung 15: Domänenmodell für den Interaktions-Microservice

Der Chancen-Microservice ist ähnlich aufgebaut und besitzt neben einer einzelnen Entität eine Enumeration für den Status einer Verkaufschance.

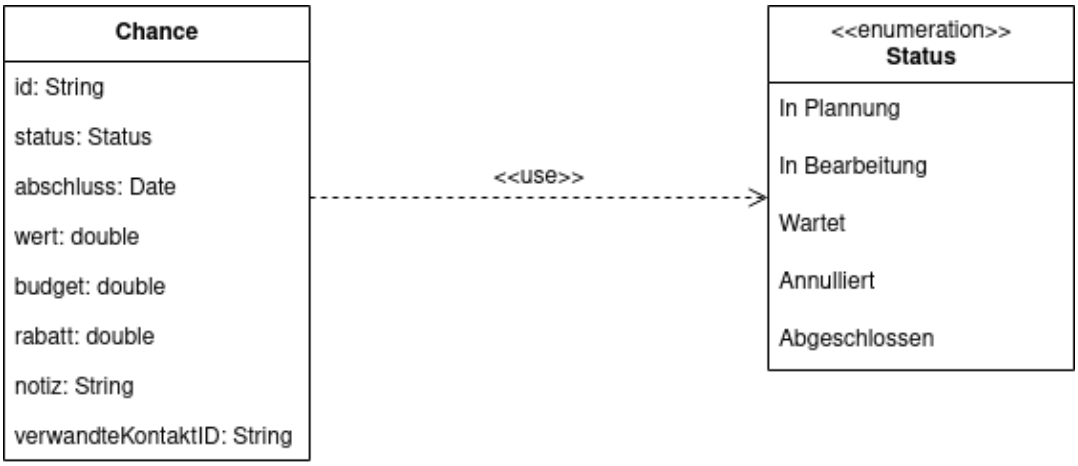


Abbildung 16: Domänenmodell für den Chancen-Microservice

Die einzelnen Microservices werden nach einer hexagonalen Architektur entworfen. Dabei handelt es sich um ein Architekturmuster bei der die Logik im Mittelpunkt steht. Sie hat verschiedene Schnittstellen (Ports), welche mit diversen Adaptern benutzt werden können. Das Architekturmuster wird deshalb auch als Ports and Adapters bezeichnet. Eine hexagonale Architektur ist eine Weiterentwicklung der klassischen Drei-Schichten-Architektur, bei der die Anwendung in eine Präsentationsschicht, eine Logikschicht und eine Datenhaltungsschicht aufgeteilt wird.

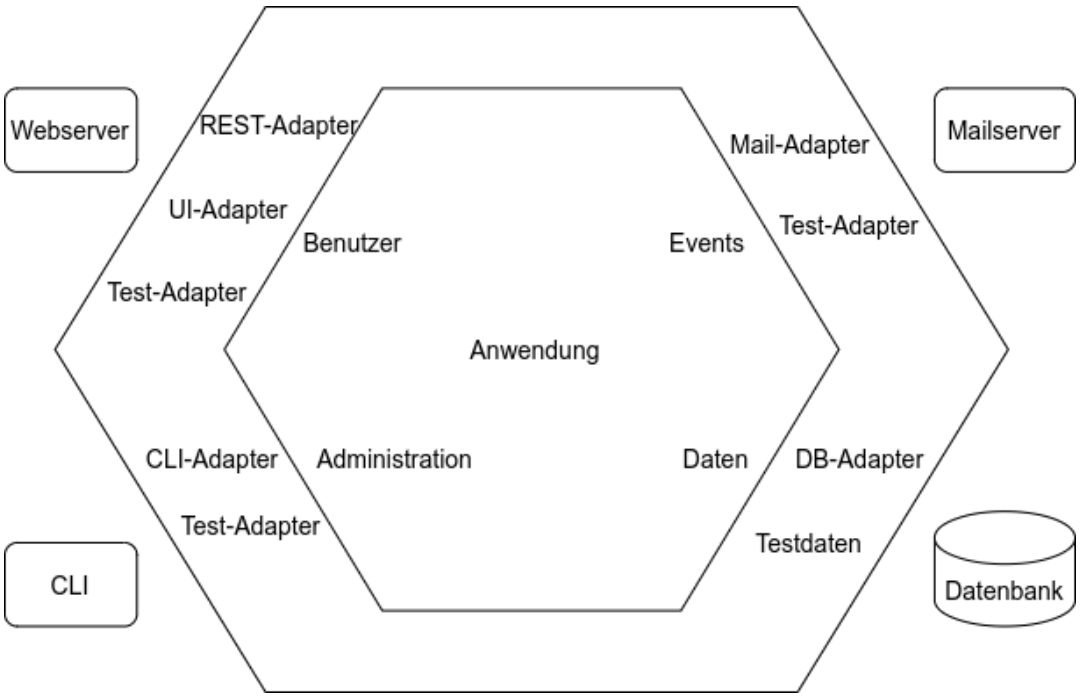


Abbildung 17: Überblick einer hexagonalen Architektur [vgl. Wolff 2018, S. 204]

Jede Facette der Anwendung, wie Benutzer, Daten oder Admin ist ein Port, welcher von den Adaptern auf Technologien wie REST umgesetzt wird [vgl. Wolff 2018, S. 204]. Die Logik der Anwendung wird klar abgetrennt und nur die Adapter ermöglichen eine Kommunikation nach Außen. Dadurch bleibt der fachliche Code unabhängig vom technischen Code. Für das Fallbeispiel wird lediglich ein Port für die Daten und ein Port für die Benutzung benötigt. Zusätzlich wird ein Adapter für die REST-API und ein Adapter für MongoDB benötigt.

Als Frontend wird eine clientseitige Webanwendung eingesetzt. Diese wird mithilfe der JavaScript-Bibliothek React implementiert. React ist die beliebteste Bibliothek zum Erstellen von Web-UIs [vgl. Stack Overflow 2021]. Anwendungen werden dabei aus wiederverwendbaren Komponenten zusammengesetzt werden, welche effizient gerendert und aktualisiert werden. Die Komponenten können sowohl aus JavaScript, Hypertext Markup Language (HTML) und Cascading Style Sheets (CSS) bestehen.

5 Implementierung

In diesem Kapitel wird der erstellte Entwurf implementiert. Zuerst werden die Microservices angefertigt und anschließend das Frontend. Es sei auch erwähnt, dass der vollständige Quellcode unter dem folgenden GitHub Repository einsehbar ist: github.com/SimonHirner/bachelor-thesis.

5.1 Microservices

Die Umsetzung aller drei Microservices ist sehr ähnlich, deshalb wird die Implementierung hauptsächlich am Beispiel des Kontakt-Microservices erläutert. Als Erstes wird das erstellte Domänenmodelle für den Kontakt-Microservice implementiert. Jede Entität wird mit seinen Eigenschaften wird in Java als eine Klasse mit Attributen dargestellt. Die Endpunkte der spezifizierten REST-API werden in der Klasse ContactController erstellt. Dieser fungiert als ein Adapter im Sinne der hexagonalen Architektur. Er benötigt einen Port, mit dem er auf Funktionen der Geschäftslogik zugreifen kann. Dafür wird das Interface ContactService erstellt. Dieses definiert die Funktionalitäten der Geschäftslogik, implementiert sie aber noch nicht. Erst in der Klasse ContactServiceImpl wird das Interface implementiert und die eigentliche Geschäftslogik niedergeschrieben. Dadurch ist die Logik vom Controller, der sich um die Anfragen auf die REST-Schnittstelle kümmert, unabhängig. Einen Adapter für die Anbindung an MongoDB bringt Spring bereits mit. Es muss lediglich noch das Interface ContactRepository erstellt werden, welches als Port den Adapter mit der Geschäftslogik verbindet. Die Klassen ContactController und ContactServiceImpl werden vollständig im Anhang aufgelistet.

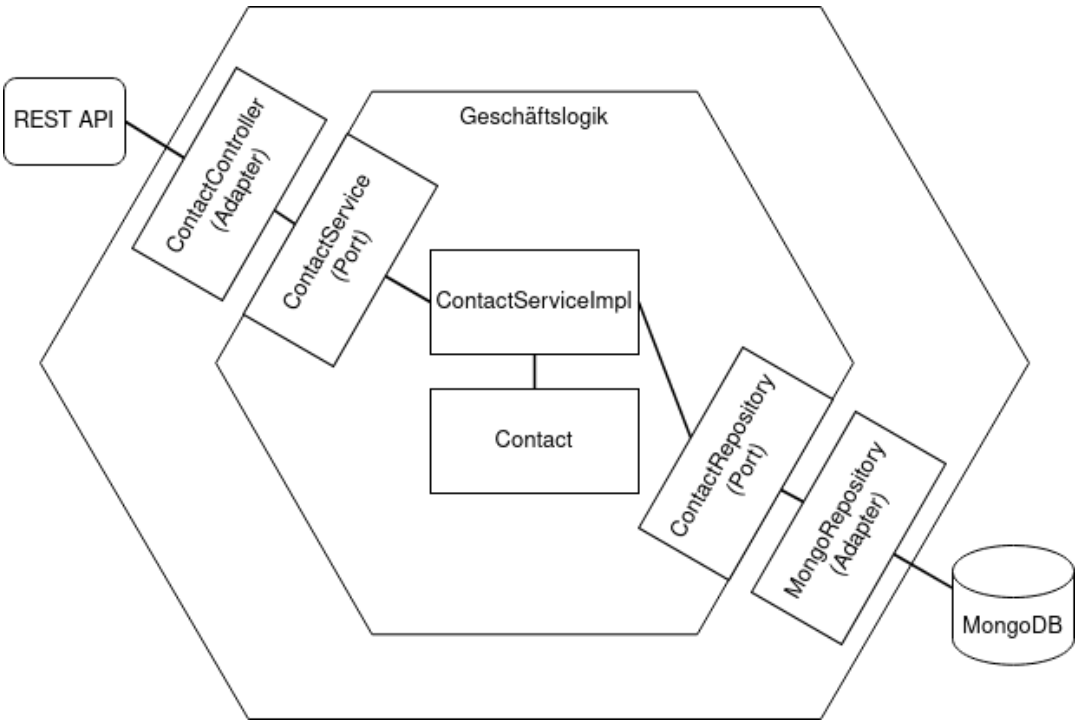


Abbildung 18: Implementierung der hexagonalen Architektur beim Kontakt-Microservice

Den Mittelpunkt des Microservices bildet die Klasse ContactServiceImpl. Sie enthählt den Code, der die Geschäftslogik beschreibt. Die Klasse enthält verschiedene Methoden, um Kontakte zurückzugeben, zu löschen, zu speichern und zu ersetzen.

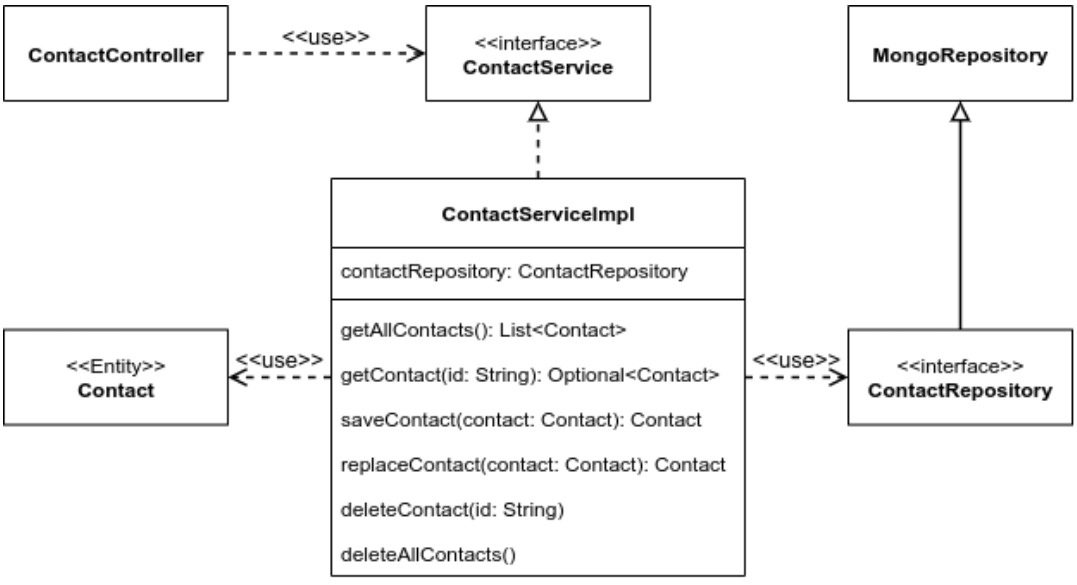


Abbildung 19: Struktur vom Kontakt-Microservice

Der Interaktions-Microservice und der Chancen-Microservices unterscheiden sich bei der Geschäftslogik vom Kontakt-Microservice. Der Interaktions-Microservice muss beim Speichern einer Interaktion überprüfen, ob die Kontaktidentifikationsnummer der Interaktion gültig ist. Dafür ruft der Interaktions-Microservice den Kontakt-Microservice über seine REST-API auf und testet, ob zu der angegebenen Kontaktidentifikationsnummer ein Kontakt vorhanden ist. Das folgende UML-Sequenzdiagramm zeigt den zeitlichen Verlauf der API-Anfragen und die involvierten API-Endpunkte. Da bei einer Chance auch eine Kontaktidentifikationsnummer angegeben werden kann, wird beim Chancen-Microservice dieselbe Logik implementiert.

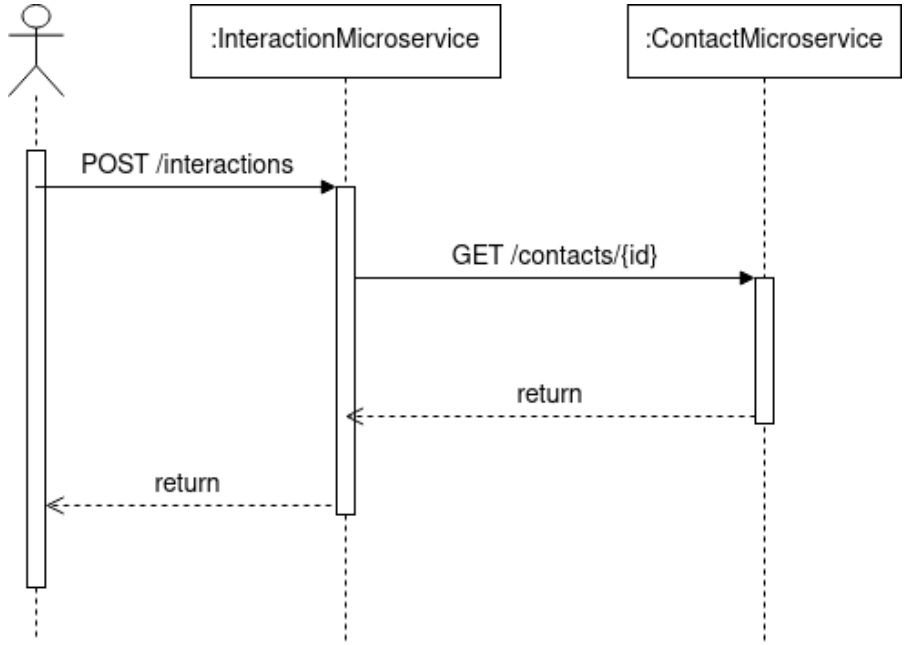


Abbildung 20: Kommunikationsablauf zwischen Interaktions-Microservice und Kontakt-Microservice

Des Weiteren wird Swagger in die Microservices integriert. Bei Swagger handelt es sich ein Werkzeug zur sprachunabhängigen Spezifikation von APIs [vgl. SmartBear Software 2022]. Swagger kann durch eine Abhängigkeit unkompliziert zu Spring Boot hinzu gefügt werden. Es erstellt automatisch eine Webseite mit einer Dokumentation der REST-API.

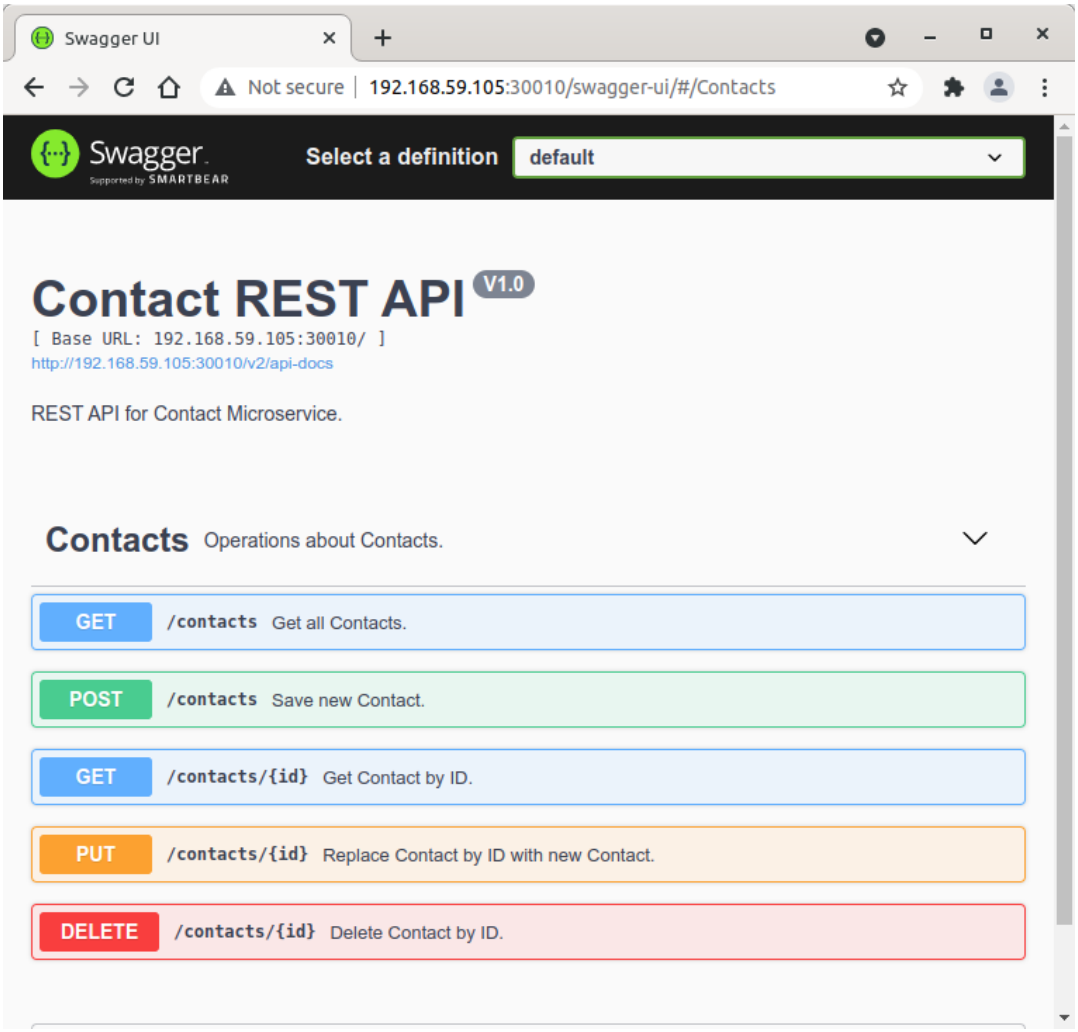


Abbildung 21: Swagger Dokumentation der Kontakt-API

Alle drei Microservices benötigen die Adresse 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 können so die Umgebungsvariablen der Container mit den richtigen Adressen besetzt werden.

5.2 Frontend

Das Frontend besteht aus insgesamt zehn verschiedenen Webseiten. Für jeden Kontakt, jede Interaktion und jede Verkaufschance gibt es jeweils eine Seite zum Einsehen aller Objekte, eine Seite zum Einsehen sowie Bearbeiten eines Objektes und eine Seite zum Anlegen eines neuen Objektes. Dazu hat das Frontend noch eine Startseite. Die Seiten setzen sich auch

verschiedenen wiederverwendbaren Komponenten, wie beispielsweise der Navigationsleiste, Formularen und Tabellen zusammen. Die API-Aufrufe werden über Funktionen in Service-Klassen implementiert, die von den Komponenten aufgerufen werden. Daraus ergibt sich der folgenden Aufbau für das Frontend.

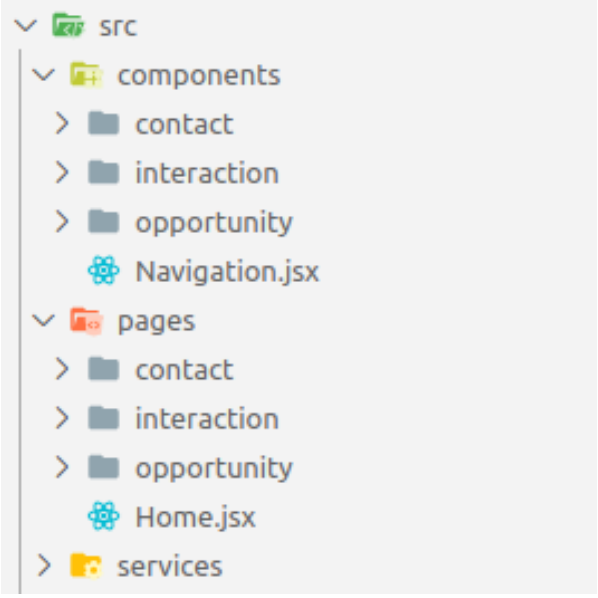


Abbildung 22: Aufbau des Frontends

Zum Styling der Anwendung wird das CSS-Framework Bootstrap verwendet. Bei der Erstellung oder Bearbeitung einer Chance oder Interaktion kann über eine Dropdown-Liste ein zugehöriger Kontakte ausgewählt werden. Bei der Detailansicht eines Kontaktes wird neben den Attributen auch die zugehörigen Interaktionen und Verkaufschancen angezeigt. Im Anhang finden sich Bildschirmfotos der beschriebenen Seiten. Auch ist dort eine der Service-Klassen zu finden, welche die Anfragen auf den Kontakt-Microservice regelt. Das Frontend benötigt die Adresse aller Services. Auch hier werden die Verbindungsinformationen über eine Umgebungsvariable übergeben. Bei React werden Umgebungsvariablen in die Datei .env.production geschrieben, da der Code des Frontends clientseitig ausgeführt wird.

6 Bereitstellung mit Kubernetes

Im letzten Teil des Fallbeispieles wird das fertige CRM-System mit Kubernetes bereitgestellt. Dafür wird Docker, Minikube und Kubectl benötigt.

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 ähnelt einer Anleitung, welche Schritt für Schritt abgearbeitet wird. Die Dockerfiles der Microservices haben alle einen analogen Aufbau. Der erste Befehl in den Dockerfiles bestimmt, auf welchem Docker Image das neue Image basieren soll. Für die Microservices 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"]
```

Quelltext 1: Dockerfile für den Kontakt-Microservice

Das Frontend benötigt ein eigenes Dockerfile. Dieses hat einen mehrstufigen Aufbau. In der ersten Stufe, der Build-Stage, wird die React-Anwendung gebaut. Um die fertig gebaute Webanwendung an einen Browser auszuliefern wird ein Webserver benötigt. Die zweite 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
```

```
RUN rm -rf ./*
COPY --from=build /app/build .
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

Quelltext 2: Dockerfile für das Frontend

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

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

Quelltext 3: Docker-Befehl für das Bauen eines Images

6.2 Bereitstellung

Die fertigen Images können jetzt eingesetzt werden. Dazu wird zuerst das Kubernetes Cluster mit Minikube gestartet. Anschließend werden YAML-Dateien erstellt, in denen die gewünschten Kubernetes Objekte beschrieben werden. Für alle drei Microservices, alle drei Datenbanken und das Frontend muss jeweils ein Service und ein Deployment erstellt werden. In den Deployment-Objekten wird der Name der zuvor gebauten Images angegeben. Mit dem Attribut Replicas kann zudem die Anzahl der Pods, welche von einem Microservice gleichzeitig laufen sollen verändert werden. Bei den Microservices wird zudem die Adresse, unter der die Datenbank erreichbar ist, als Umgebungsvariable übergeben. Als Adresse wird der Name vom Service-Objekt der entsprechenden Datenbank verwendet werden.

```
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
```

```
20                value: contact-service
21                valueFrom:
22                  configMapKeyRef:
23                    name: contact-db-config
24                    key: host
```

Quelltext 4: Deployment-Objekt vom Kontakt-Microservice

Die Service-Objekte der Datenbanken sind vom Typ ClusterIP, da sie nur innerhalb des Clusters aufgerufen werden. Die Service-Objekte der Microservices sind dagegen vom Typ NodePort. Somit kann ein fester Port angegeben werden, unter dem der Service auch von außerhalb des Clusters erreichbar ist. Anfragen auf einen Service werden automatisch von Kubernetes auf die Pods, welche dem Service zugeordnet sind, verteilt.

```
1  kind: Service
2  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
```

Quelltext 5: Service-Objekt vom Kontakt-Microservice

Sämtliche erstellte YAML-Dateien müssen über einen Kubectl-Befehl auf das Cluster angewendet werden. Kubernetes arbeitet dann automatisch die nötigen Schritte, um die beschriebenen Objekte zu erstellen.

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

Quelltext 6: Kubectl-Befehl für das Anwenden einer YAML-Datei

6.3 Skalierung

Um die Vorteile von der Microservices auszunutzen, soll die Anzahl der gleichzeitig laufenden Pods eines Microservices flexibel verwaltet werden. Abhängig von der Auslastung soll so eine horizontale Skalierung vorgenommen werden. Auch dafür wird eine YAML-Datei erstellt, in der ein HPA-Objekt für jeden Microservice beschrieben wird. In der Datei wird

angegeben, welches Deployment skaliert werden soll. Als Metrik für die Skalierung, wird die CPU-Auslastung des Pods verwendet. Darüber hinaus wird angegeben wie viele Pods vom entsprechenden Microservice minimal und maximal ausgeführt werden sollen.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
    name: contact-service
spec:
    scaleTargetRef:
        apiVersion: apps/v1
        kind: Deployment
        name: contact-service
    minReplicas: 2
    maxReplicas: 4
    targetCPUUtilizationPercentage: 80
```

Quelltext 7: HPA-Objekt vom Kontakt-Microservice

Nun ist die Bereitstellung abgeschlossen. Minikube bietet ein browserbasiertes Dashboard an, mit dem der Status des Clusters auch grafisch überprüft werden kann. Ein Bildschirmfoto des Dashboards kann im Anhang gefunden werden. Das folgende Verteilungsdiagramm zeigt das Ergebnis der durchgeführten Bereitstellung.

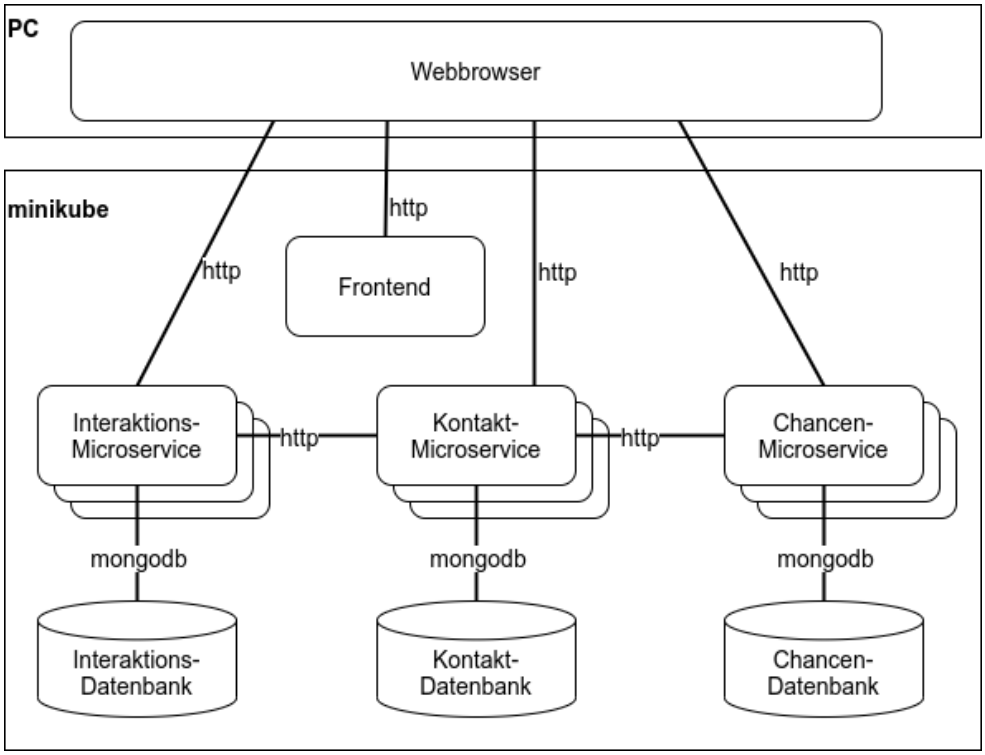


Abbildung 23: Verteilungsdiagramm

7 Schlussbetrachtung

Durch den theoretischen Hintergrund und die praktische Anwendung mithilfe des Fallbeispiels kann ein umfassendes Fazit gezogen werden. Im Anschluss an das Fazit wird ein Ausblick in die Zukunft gegeben.

7.1 Fazit

Die Fallstudie bestätigt, dass der Entwurf einer Microservice-Architektur diffizil ist. Microservices bieten viel Flexibilität, es müssen jedoch auch erheblich mehr Entscheidungen getroffen werden. Die Architektur der einzelnen Microservices kann dabei sehr frei gewählt werden und die Auswahl ist mit einem geringen Risiko verbunden. Die fachliche Aufteilung der Microservices hingegen ist besonders herausfordernd. Mit einer schlechten Aufteilung ist das gesamte System zum Scheitern verurteilt. Durch die vielen Freiheiten bei der Umsetzung der einzelnen Microservices kann das System aber auch schnell in einem technologischen Pluralismus enden. Vor allem bei großen betrieblichen Anwendungen an denen viele Entwicklerteams arbeiten und welche eine große fachliche Breite besitzen, können Microservices enorme Vorteile bringen. Für kleinere und auf einen Fachbereich spezialisierte Anwendungen ist der erhöhte Aufwand von Microservices beim Entwurf und der Bereitstellung nicht zu rechtfertigen. Das Anwendungsgebiet von Microservices ist somit sehr große, aber doch eingeschränkt.

Container ermöglichen eine einfache Bereitstellung auf verschiedenen Rechnern in einem großem verteilten System und sind für Microservices so eine große Hilfe. Bei der Bereitstellung zeig sich außerdem, wieso Kubernetes so populär ist. Es löst viele Problematiken der Microservice-Architektur mit wenig Aufwand. Service Discovery und Lastverteilung kann leicht über ein entsprechendes Service-Objekt von Kubernetes übernommen werden. Die automatische Skalierung kann auch mit wenig Zeilen konfiguriert werden. Die abstrakten Objekte von Kubernetes sind so eine enorme Erleichterung. Es muss fast nichts mehr manuell eingerichtet werden. Der Großteil der Konfiguration läuft über deklarative Dateien. Doch die Abstrahierung versteckt auch die unmittelbaren Auswirkungen und erschwert so die Verständnis. Aufgrund dessen hat Kubernetes eine durchaus steile Lernkurve.

Containerisierte Microservices mit Kubernetes sind eine mächtige Kombination für moderne verteilte Anwendungen. Sie sind aber auch keine perfekt Lösung für jeden Zweck. Mit dem Fallbeispiel konnte ein Verfahren vom Entwurf bis zur Bereitstellung implementiert werden, welches das nötige praktische Verständnis über den richtigen Einsatz von Microservices und Kubernetes liefert.

7.2 Ausblick

Microservices sind noch jung und ihr Potenzial mit Sicherheit noch nicht voll ausgeschöpft. In Zukunft werden sie vor allem im geschäftlichen Umfeld eine immer wichtigere Rolle einnehmen. Microservices sind aber auch nicht für jedes Anwendungsgebiet vorteilhaft und werden Monolithen so nicht verdrängen, sondern nur in bestimmten Bereichen ablösen. Microservices passen perfekt in das moderne DevOps-Umfeld. Für verteilte Cloud-Infrastrukturen sind Microservices unumgänglich. Agile Softwareentwicklung und crossfunktionale Teams ergänzen sich bestens mit ihnen. Bei Microservices handelt es sich nicht nur um einen kurzzeitigen Trend, sondern um die logische Reaktion auf ein dynamisches IT-Umfeld.

Kubernetes hat sich etabliert und wird so schnell nicht mehr verschwinden. Dafür bietet es zu viele Vorzüge, gerade in der Verwendung mit Microservices. Es ist bereits ein fester Bestandteil vieler großer Cloud-Anbieter. Mit der Zeit wird Kubernetes somit immer weiter aus dem Fokus geraten und zu einem normalen Teil der Infrastruktur werden.

Ein logischer nächster Schritt wäre es, das Fallbeispiel auf einer Cloud-Plattform durchzuführen. Durch den Einsatz von CI/CD-Pipelines könnte zudem, die Bereitstellung weiter automatisiert werden.

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*. 3. Auflage. Bonn: Rheinwerk Computing. ISBN: 978-3-8362-7774-7.

Arundel, John und Justin Domingus (2019). *Cloud Native DevOps Mit Kubernetes*. 1. Auflage. Heidelberg: dpunkt.verlag. ISBN: 978-3-86490-698-5.

Halstenberg, Jürgen, Bernd Pfitzinger und Thomas Jestädt (2020). *DevOps: Ein Überblick*. Wiesbaden: Springer Vieweg. ISBN: 978-3-658-31404-0.

Tremp, Hansruedi (2021). *Architekturen Verteilter Softwaresysteme*. Wiesbaden: Springer Vieweg. ISBN: 978-3-658-33179-5 978-3-658-33178-8.

Newman, Sam (2015). *Microservices: Konzeption Und Design*. 1. Auflage. Frechen: mitp-Verl. ISBN: 978-3-95845-081-3.

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

Salus, Peter (1994). *A Quarter Century of UNIX*. Reading: Addison-Wesley Pub. Co. 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. ISBN: 978-3-86490-542-1.

Artikel

Cloud Native Computing Foundation (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).

Burns, Brendan und David Oppenheimer (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.

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

Evans, Eric (2015). *Domain-Driven Design Reference*. URL: https://www.domainlanguage.com/ddd/reference/ (besucht am 01. 02. 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. URL: https://www.sciencedirect.com/science/article/pii/S0950584921000549.

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).

Unified Modeling Language Specification (2017). URL: https://www.omg.org/spec/UML/2.5.1/.

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).

VMware, Inc. (2022). *Spring Can Help You Write Microservices*. URL: https://spring.io/microservices (besucht am 02. 02. 2022).

MongoDB, Inc. (2022). *MongoDB Documentation*. URL: https://docs.mongodb.com/ (besucht am 03. 02. 2022).

Stack Overflow (2021). *Most Used Web Frameworks among Developers Worldwide*. URL: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/ (besucht am 03. 02. 2022).

SmartBear Software (2022). *Swagger API Documentation*. URL: https://swagger.io/ (besucht am 03. 02. 2022).

Anhang

ContactController vom Kontakt-Microservice

```
@RestController
@RequestMapping("/contacts")
@CrossOrigin(origins = "*")
@Api(tags = {SwaggerConfig.CONTACT_TAG})
public class ContactController {

    private static final Logger logger = LoggerFactory.getLogger(
        ContactApplication.class);

    private final ContactService contactService;

    public ContactController(ContactService contactService) {
        this.contactService = contactService;
    }

    @GetMapping
    @ApiOperation(value = "Get all Contacts.")
    List<Contact> getAllContacts() {
        logger.debug("GET /contacts");
        return contactService.getAllContacts();
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    @ApiOperation(value = "Save new Contact.")
    Contact saveContact(@RequestBody Contact newContact) {
        logger.debug("POST /contacts");
        try {
            return contactService.saveContact(newContact);
        } catch (ConstraintViolationException exception) {
            throw new BadRequestException(exception.getMessage());
        }
    }

    @GetMapping("/{id}")
    @ApiOperation(value = "Get Contact by ID.")
    Contact getContact(@PathVariable String id) {
        logger.debug("GET /contacts/{}", id);
        return contactService.getContact(id)
                .orElseThrow(() -> new ResourceNotFoundException(id));
    }

    @PutMapping("/{id}")
    @ApiOperation(value = "Replace Contact by ID with new Contact.")
    Contact replaceContact(@PathVariable String id, @RequestBody
        Contact newContact) {
```

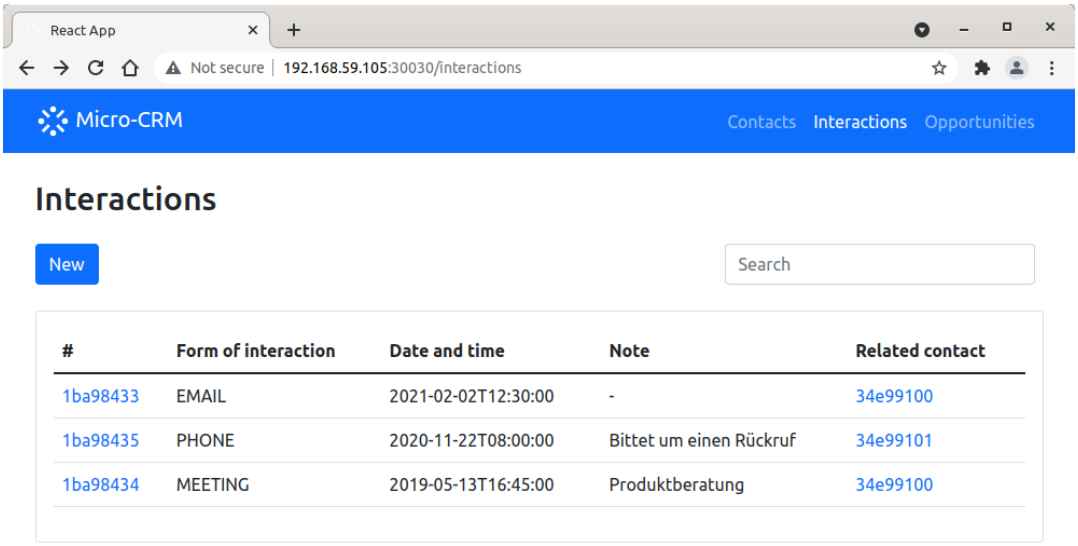
```
45          logger.debug("PUT /contacts/{}", id);
46          return contactService.replaceContact(id, newContact);
47      }
48
49      @DeleteMapping("/{id}")
50      @ApiOperation(value = "Delete Contact by ID.")
51      @ResponseStatus(HttpStatus.NO_CONTENT)
52      void deleteContact(@PathVariable String id) {
53          logger.debug("DELETE /contacts/{}", id);
54          contactService.deleteContact(id);
55      }
56
57  }
```

ContactServiceImpl vom Kontakt-Microservice

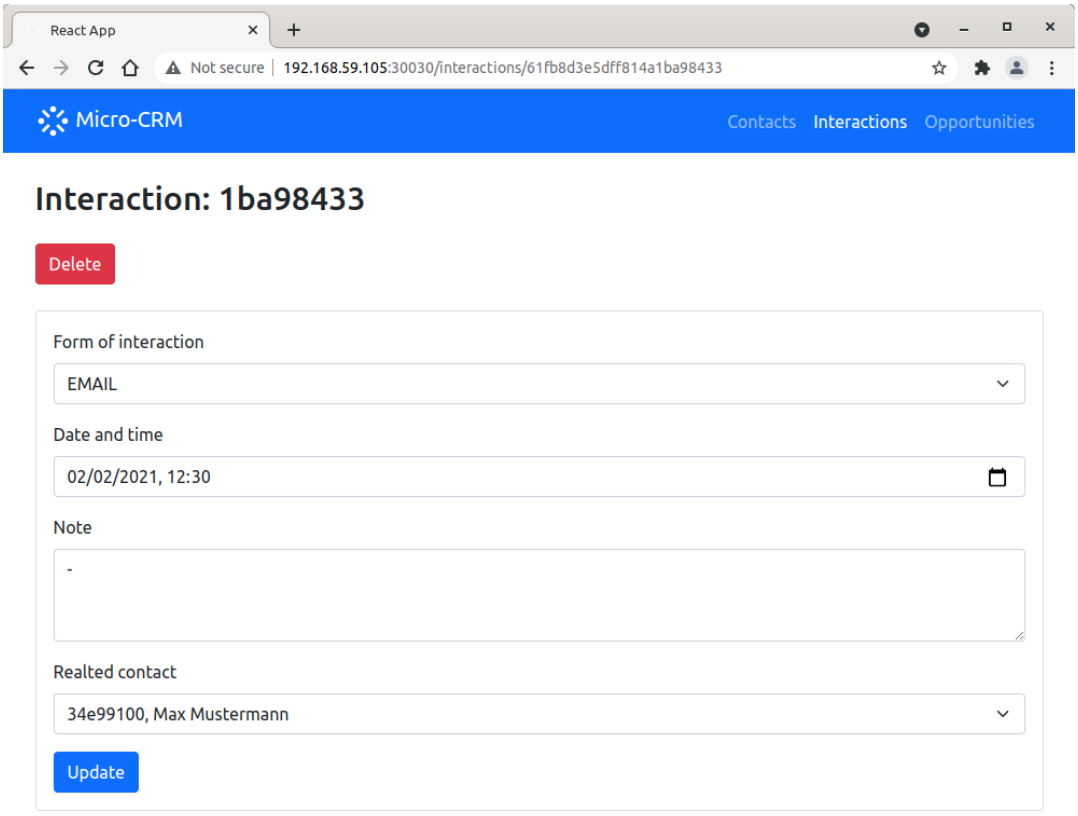
```
@Service
public class ContactServiceImpl implements ContactService {

    private static final Logger logger = LoggerFactory.getLogger(
       ContactApplication.class);

    private final ContactRepository contactRepository;

    public ContactServiceImpl(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }

    @Override
    public List<Contact> getAllContacts() {
        logger.debug("Find all contacts");
        return contactRepository.findAllByOrderByLastNameAscIdAsc();
    }

    @Override
    public Optional<Contact> getContact(String id) {
        logger.debug("Find contact {}", id);
        return contactRepository.findById(id);
    }

    @Override
    public Contact saveContact(Contact newContact) {
        logger.debug("Save new contact");
        return contactRepository.save(newContact);
    }

    @Override
    public Contact replaceContact(String id, Contact newContact) {
        logger.debug("Replace contact {}", id);
        return getContact(id)
                .map(contact -> {
                    contact.setFirstName(newContact.getFirstName());
                    contact.setLastName(newContact.getLastName());
                    contact.setGender(newContact.getGender());
                    contact.setEmail(newContact.getEmail());
                    contact.setDateOfBirth(newContact.getDateOfBirth())
                       ;
                    contact.setPhoneNumber(newContact.getPhoneNumber())
                       ;
                    contact.setAddress(newContact.getAddress());
                    return saveContact(contact);
                })
                .orElseThrow(() -> new ResourceNotFoundException(id));
    }

```

```
47      @Override
48      public void deleteContact(String id) {
49          logger.debug("Delete contact {}", id);
50          contactRepository.deleteById(getContact(id).orElseThrow(() ->
                new ResourceNotFoundException(id)).getId());
51      }
52
53      @Override
54      public void deleteAllContacts() {
55          logger.debug("Delete all contacts");
56          contactRepository.deleteAll();
57      }
58
59  }
```

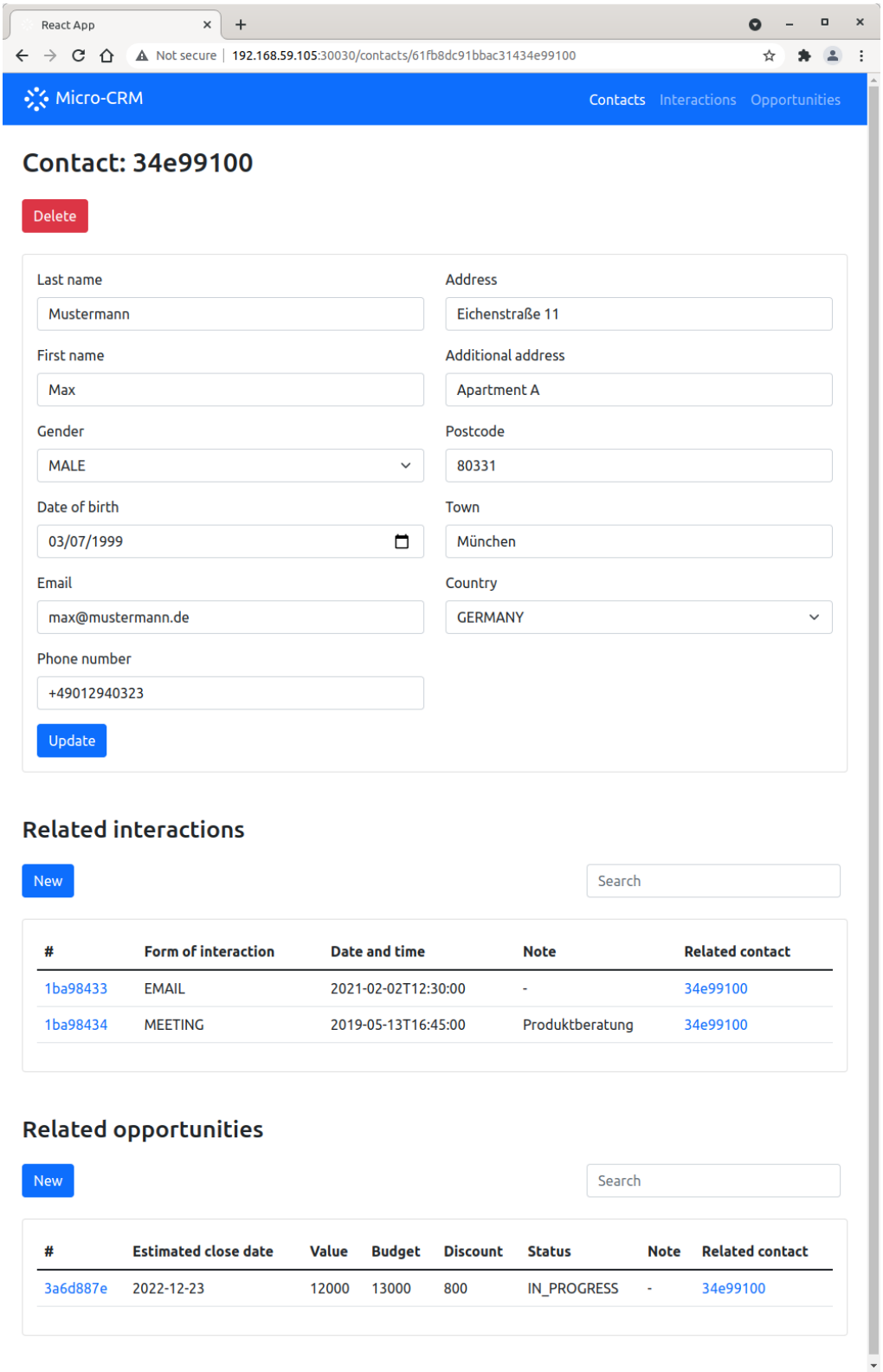
Ansicht der Interaktionen vom Frontend



Ansicht einer Interaktion vom Frontend



Ansicht eines Kontaktes vom Frontend



Kontakt-Service-Klasse vom Frontend

```
1  import axios from "axios";
2
3  const BACKEND = process.env.REACT_APP_BACKEND;
4
5  const CONTACT_API = 'http://' + BACKEND + (BACKEND === 'localhost' ? '
       :8080' : ':30010') + '/contacts';
6
7  class ContactService {
8
9      getAllContacts() {
10         return axios.get(CONTACT_API);
11     }
12
13     saveContact(contact) {
14         return axios.post(CONTACT_API, contact);
15     }
16
17     getContact(id) {
18         return axios.get(CONTACT_API + "/" + id);
19     }
20
21     replaceContact(id, contact) {
22         return axios.put(CONTACT_API + "/" + id, contact);
23     }
24
25     deleteContact(id) {
26         return axios.delete(CONTACT_API + "/" + id)
27     }
28
29 }
30
31 export default new ContactService()
```

Minikube Dashboard nach der Bereitstellung