ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Projektarbeit

im Modul Software-Architektur
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Cloud-Native Architekturen

vorgelegt von
Tristan Gläs und Carolin Becker

betreut und begutachtet von
Prof. Dr. Markus Esch

Saarbrücken, 22. März 2021

Zusammenfassung

Diese Ausarbeitung beschäftigt sich mit dem Thema Cloud-Native Architektur. Einleitend wird auf das Cloud Computing eingegangen, welches die Basis für Cloud-Native Architekturen bildet. Anschließend wird Cloud-Native als eine Technologien definiert, die es Unternehmen ermöglicht moderne und dynamischen Umgebungen zu implementieren und zu betreiben.
Als nächstes wird die Microservice-Architektur genauer erläutert. Microservices werden auf Grund des Architekturstils gerne in Cloud-Native Anwendungen verwendet und sind somit ein wichtiger Bestandteil in Cloud-Native Systemen. Des Weiteren erfolgt eine Gegenüberstellung der Microservices mit der monolithischen Architektur, wodurch sich Unterschiede herausfiltern lassen.
Auch die Containerisierung trägt eine bedeutende Rolle in der Cloud-Native Architektur. Durch Container können Anwendungen von ihrer Ausführungsumgebung abstrahiert werden, sodass diese schnell und zuverlässig bereitgestellt werden können. Containerbasierte Anwendungen werden außerdem von virtuellen Maschinen abgegrenzt.
Die Seminararbeit befasst sich zudem mit den wichtigsten Eigenschaften der Cloud-Native Architektur, wodurch sich die Vor- und Nachteile dieser Architektur herauskristallisieren. Dabei wird auch deutlich, dass sich die Eigenschaften der Microservices in Cloud-Native widerspiegeln. Beispiele dafür wäre zum Einen die Skalierbarkeit und Flexibilität und zum Anderen die Änderbarkeit und Wartbarkeit.
Das Cloud-Native Kapitel wird mit typischen Einsatzgebieten abgeschlossen. Diese stellen meist Systeme dar, welche ein hohes Maß an Skalierbarkeit erfordern. Cloud-Native Architekturen werden besonders in den Bereichen Streaming und Big Data verwendet.
Ein weiteres großes Kapitel befasst sich mit einem selbst entworfenem Anwendungsfall einer Cloud-Native Architektur, welcher auch prototypisch umgesetzt wurde. Dabei werden die funktionalen sowie nicht-funktionalen Anforderungen beschrieben. Anhand eines Architekturentwurfs wird der Aufbau der Anwendung deutlich und die einzelnen Bestandteile der Architektur werden genauer erklärt.
Durch das Sequenzdiagramm wird die Funktionsweise der Anwendung bzw. die Beziehungen der einzelnen Services untereinander graphisch dargestellt.
Das letzte Kapitel thematisiert die Diskussion des entworfenen Prototyps. In diesem wird überprüft, ob die Anwendung einer Cloud-Native Architektur entspricht. Dabei werden Schwachstellen und Verbesserungsmöglichkeiten aufgezeigt und abschließend ein Fazit gezogen.

Inhaltsverzeichnis

1 Einleitung

Cloud-Dienste sind heutzutage nicht mehr aus unserem Alltag wegzudenken. Auch im Arbeitsleben bzw. für viele Unternehmen ist das Thema Cloud im Zusammenhang mit der Digitalisierung ein wichtiger Bestandteil geworden. Dadurch wurden neue Geschäftsmodelle ermöglicht und gleichzeitig die Wettbewerbsfähigkeit des Unternehmens gesteigert.

Ein neuer Ansatz ist Cloud-Native. Hierbei werden die Applikationen von Beginn an für den Einsatz in der Cloud entwickelt. Dieser wird als Zukunft der Software-Entwicklung gesehen und basiert auf den bewährten Technologien des Cloud Computings.
Die Weiterentwicklung und der Ausbau des Cloud Computing Bereiches, welche durch die steigenden Zahlen von Benutzern und Daten schnell vorangetrieben werden, benötigen auch entsprechende Softwaresysteme. Diese können mit den bereitgestellten Ressourcen effektiv umgehen und sind auf die Cloud-Umgebung abgestimmt.
Cloud-Native hat sich in den letzten Jahren als eigener Bereich in der Informatik herauskristallisiert und wurde vorallem durch große Unternehmen wie Netflix, Google und Amazon geprägt.
Auch die Verbreitung von Cloud-Native Technologien insbesondere der Container und Container-Orchestrierungs-Tools wie z.B. Kubernetes sind in den letzten Monate deutlich gestiegen.
Durch Cloud-Native ergibt sich eine neue Möglichkeit große komplexe Systeme zu entwickeln. Dies ist besonders für Unternehmen interessant, da somit in kürzester Zeit innovative Anwendungen entwickelt werden können.
Benutzer bzw. Kunden werden zudem immer anspruchsvoller und erwarten, dass schnellstmöglich neue Anwendungen für sie zur Verfügung stehen. Des Weiteren wird eine schnelle Reaktionsfähigkeit, innovative Features und eine möglichst geringe Ausfallzeit erwartet.
Der Cloud-Native Ansatz ist für solche Situationen ausgelegt und bietet die Möglichkeit Applikationen, je nach Bedarf zu skalieren.

2 Cloud-Native

In zweiten Kapitel gehen wir auf die Definition von Cloud-Native Architekturen ein, grenzen sie von anderen Absätzen ab und betrachten wichtige Eigenschaften. Abschließend beschäftigen wir uns mit den Vor-und Nachteilen und den typischen Einsatzgebieten.

2.1 Cloud Computing

Bevor wir uns mit Cloud-Native Architekturen auseinandersetzten können, müssen wir uns zuerst mit dem Cloud Computing beschäftigen, da es die Basis für diese Architekturen bildet und sie maßgeblich beeinflusst. Die NIST Definition von Cloud Computing enthält die wichtigsten Merkmalen.

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Entscheidend für Cloud-Native ist nun die schnelle Bereitstellung von Ressourcen, denn dies eröffnet neue Möglichkeiten hinsichtlich der Skalierbarkeit und hat dadurch einen starken Einfluss auf die Architekturen.

2.2 Definition Cloud Native

Was genau Cloud-Native ist und wie man es definieren kann ist schwierig, da der Begriff noch relativ neu ist. Eine erste Version einer Definition kommt von der Cloud-Native Computing Foundation, einer Organisation, die als Vorreiter in Sachen Cloud-Native gilt.

CNCF Cloud Native Definition v1.0

Cloud-Native Technologien ermöglichen es Unternehmen, skalierbare Anwendungen in modernen, dynamischen Umgebungen zu implementieren und zu betreiben. Dies können öffentliche, private und Hybrid-Clouds sein. Best Practices, wie Container, Service-Meshs, Microservices, immutable Infrastruktur und deklarative Application Programming Interfaces (API), unterstützen diesen Ansatz.
Die zugrundeliegenden Techniken ermöglichen die Umsetzung von entkoppelten Systemen, die belastbar, handhabbar und beobachtbar sind. Kombiniert mit einer robusten Automatisierung können Softwareentwickler mit geringem Aufwand flexibel und schnell auf Änderungen reagieren.
Die Cloud Native Computing Foundation fördert die Akzeptanz dieser Paradigmen durch die Ausgestaltung eines Open Source Ökosystems aus herstellerneutralen Projekten. Wir demokratisieren modernste und innovative Softwareentwicklungs-Patterns, um diese Innovationen für alle zugänglich zu machen.

Diese Definition lässt gewollt viel Spielraum zur Interpretation, jedoch stechen ein paar wichtige Merkmale heraus. Diese sind insbesondere Entkopplung, Belastbarkeit, robuste Automatisierung und Skalierbarkeit. Die Cloud Native Computing Foundation hat diese Definition als Version 1.0 markiert, was darauf schließen lässt, das Änderungen oder Erweiterungen erwartet werden.

2.3 Microservices

Microservices sind in Cloud-Native Systemen zum Erstellen von Anwendungen ein beliebter Architekturstil. Bei dieser Architektur besteht die Software aus kleinen, unabhängigen Services bzw. Modulen, die über definierte API kommunizieren. Jeder Service kann unabhängig von anderen Services entwickelt und bereitgestellt werden, ohne die Funktionalität anderer Services zu beeinträchtigen. Den Aufbau der Microservice-Architektur sowie die Darstellung der einzelnen unabhängigen Services sind in Abbildung 2.1 zu sehen.

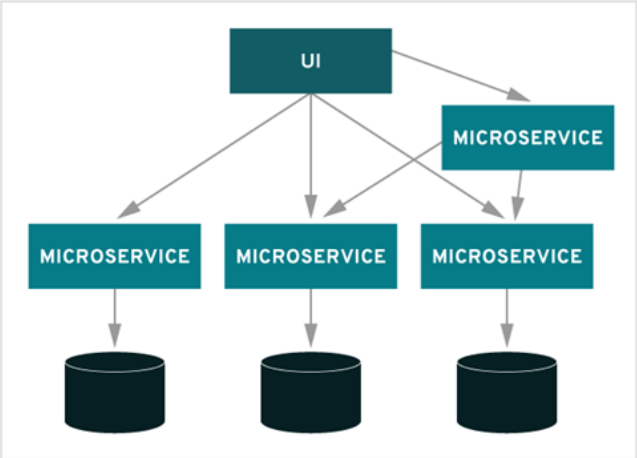

Abbildung 2.1: Aufbau einer Microservice-Architektur [3]

Durch die Eigenständigkeit besitzt jeder Komponentenservice seinen eigenen Code sowie seine eigene Implementierung. Jede einzelne Komponente ist auf eine Reihe von Funktionen spezialisiert, sodass sie sich auf die Lösung eines bestimmten Problems fokussiert. Wenn ein einzelner Service (z.B. hinsichtlich des Codes) zu komplex wird, kann er in kleinere Services unterteilt werden.
In Cloud-Native Systemen sowie in der Microservice-Architektur ist die Skalierung ein wichtiger Bestandteil. Durch die Existenz der einzelnen Services können diese je nach Nachfrage unabhängig voneinander skaliert werden. Dadurch können Subsysteme, die mehr Ressourcen benötigen hochskaliert werden, ohne die gesamte Anwendung hochzuskalieren.
Eine weitere Eigenschaft ist die Flexibilität. Durch die Unabhängigkeit der einzelnen Services wird auch deren Verwaltung vereinfacht. Bei einer Erweiterung sowie Fehlerbehebung der Anwendung muss nur der entsprechende Service verändert werden. Dies führt dazu, dass nicht die gesamte Anwendung erneut bereitgestellt werden muss.
Durch die einfache Bereitstellung können z.B. neue Konzepte ausprobiert und auch schnell wieder rückgängig gemacht werden. Auf Grund der möglichen Experimente ent-

stehen niedrige Ausfallkosten, die Aktualisierung des Codes wird erleichtert und vereinfacht das Hinzufügen neuer Funktionen.[1]

2.4 Container

Auch Container sind ein wesentlicher Bestandteil von Cloud-Native Systemen.
Ein Container ist eine Standard-Software-Einheit, die den Code und seine Abhängigkeiten verpackt, sodass Anwendungen von ihrer Ausführungsumgebung abstrahiert werden. Mit dieser Entkopplung können containerbasierte Anwendungen schnell, zuverlässig bereitgestellt und von einer beliebigen Umgebung, wie z.B. in einer öffentlichen Cloud, ausgeführt werden. In Abbildung 2.2 ist der Aufbau einer containerbasierte Anwendung zu sehen.

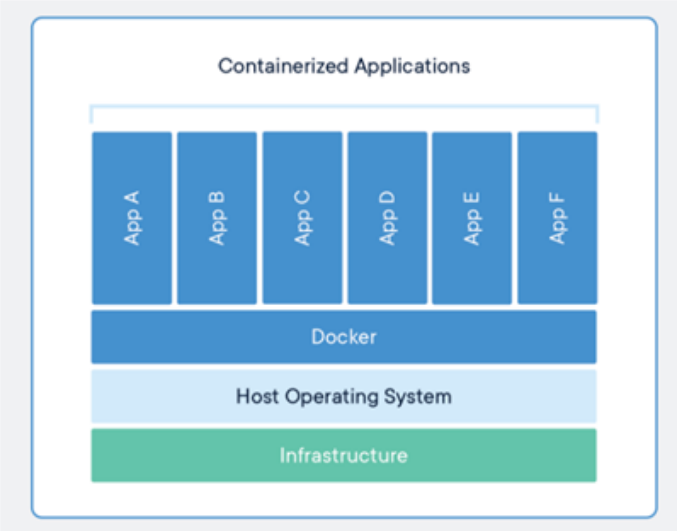

Abbildung 2.2: Aufbau einer containerbasierte Anwendung
[4]

Im Gegensatz zu virtuelle Maschinen wird bei der Containerisierung weniger Speicherplatz benötigt. Hierbei handelt es sich um eine Virtualisierung auf Betriebssystemebene, da die Container direkt auf dem Kernel des Betriebssystems ausgeführt werden. Container teilen sich den Kernel des Betriebssystems, sodass sie schneller starten und im Vergleich zu einem vollständigen Betriebssystem nur einen Bruchteil an Speicher verwenden.
Container haben eine hohe Portabilität, sodass sie überall ausgeführt werden können. Die Softwarepakete enthalten alle Elemente, die zur Ausführung in einer beliebigen Umgebung erforderlich sind.
Auch bei der Containerisierung ist die Skalierbarkeit eine wichtige Eigenschaft. Denn hier besteht die Möglichkeit, dass die einzelnen Container je nach Ressourcenbedarf schnell gestartet sowie angehalten werden können. Zudem laufen sie isoliert von anderen Anwendungen.[4]

Es gibt verschiedene Containerformate, mit deren Hilfe eine Anwendungsbereitstellung durch Containerisierung erleichtert wird. Bekannte Containerformate sind z.B. Docker und Kubernetes.
Docker ist ein beliebtes Open-Source-Containerformat zur Automatisierung der Bereitstellung von Anwendungen, die z.B. in der Cloud ausgeführt werden können. Docker

verpackt die Software mit seinen Abhängigkeiten und alles was zur Ausführung benötigt wird in Container.
Wenn Anwendungen immer größer werden und die Betreibung immer komplexer wird, ist das Orchestierungssystem Kubernetes hilfreich.
Kubernetes ist ein Open-Source-Orchestrierungssystem zur Automatisierung z.B. zur Verwaltung, Platzierung und Skalierung von Container.[2]

2.5 Abgrenzungen

In diesem Abschnitt wird die Monolithische Architektur im Gegensatz zur Microservice Architektur abgegrenzt. Darüber hinaus werden virtuelle Maschinen einer containerbasierten Anwendung gegenübergestellt.

2.5.1 Monolithische Architektur

Bei einer monolithischen Architektur sind die Prozesse eng miteinander verbunden und werden als einziger Service ausgeführt. Wenn ein Fehler auftritt, muss im Gegensatz zu der Microservice Architektur, die gesamte Anwendung skaliert werden und nicht nur der betreffende Service. Das Hinzufügen und Verbessern von Funktionen sowie das Umsetzen neuer Konzepte kann bei der monolithischen Architektur je nach Aufbau der Implementierung mit zunehmender Codebasis komplexer werden. Die Anwendungsverfügbarkeit ist im Gegensatz zu den Micorservices risikoreicher. Bei der monolithischen Architektur sind die einzelne Prozesse abhängig voneinander und eng miteinander verbunden, sodass die Wahrscheinlichkeit für einen Prozessausfall erhöht wird.
In Abbildung 2.3 sind die beschriebenen Aufbauten der monolithischen Architektur und der Microservice Architektur vergleichend dargestellt.

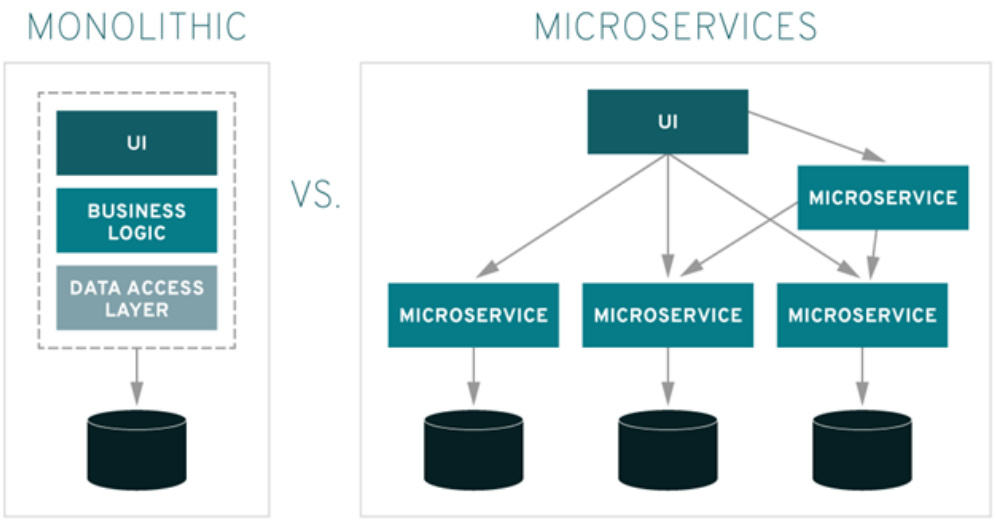

Abbildung 2.3: Vergleich der monolithischen Architektur und der Microservice Architektur [3]

2.5.2 Virtuelle Maschinen

Im Gegensatz zu Container, sind virtuelle Maschinen eine Virtualisierung der gesamten Hardware bzw. Abstraktion der physischen Hardware, die einen Server in viele Server wandelt. In Abbildung 2.4 ist der Aufbau einer containerbasierten Anwendung im Vergleich zu einer virtuellen Maschine dargestellt.

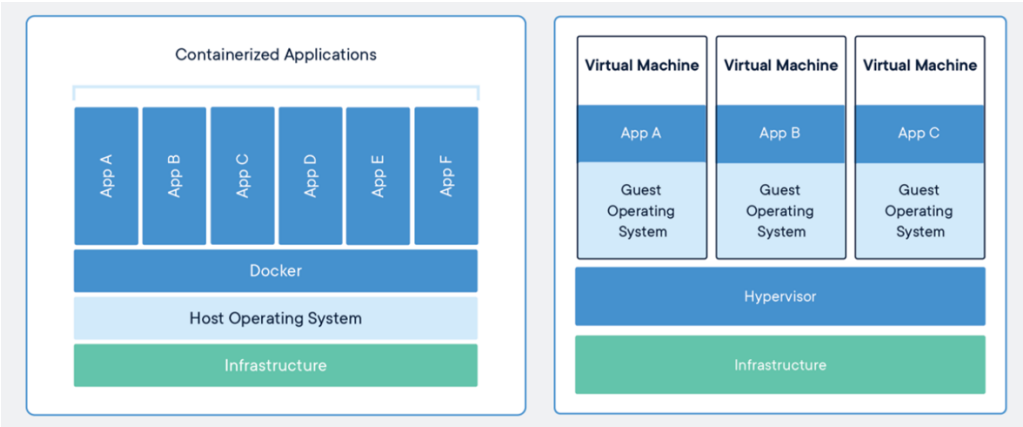

Abbildung 2.4: Vergleich einer containerbasierten Anwendung zu einer virtuellen Maschine [4]

Bei einer virtuellen Maschine ermöglicht der Hypervisor die Ausführung mehrerer virtuellen Maschinen auf einer einzigen Maschine. Jede virtuelle Maschine enthält eine vollständige Kopie eines Betriebssystems, der Anwendung, notwendiger Binärdateien und Bibliotheken. In Abbildung 2.4 sind drei Gastbetriebssysteme zu sehen, die alle von ihrem Hypervisor kontrolliert werden. Jedes System benötigt seine eigenen CPU- und Speicherressourcen sowie seine eigene Kopie verschiedener Binärdateien und Bibliotheken. Dadurch benötigen virtuelle Maschinen im Vergleich zu containerbasierten Anwendungen mehr Speicher.

2.6 Eigenschaften

Als nächstes betrachten wir die häufigst genannten Eigenschaften von Cloud-Native Architekturen.

1. Globale Ebene
Cloud-Native Architekturen sind oft für eine globale Ebene ausgelegt. Das impliziert z.B., dass das System mehrfach installiert werden muss und den Einsatz von verteilten Datenbanken.

2. Skalierbarkeit
Die entstehenden Architekturen sind skalierbar und können eine sehr große Menge von Benutzern unterstützen. Dies ist besonders in Kombination mit der globalen Ebene, wenn man Sychronisation und Konsistenz betrachtet, eine große Herausforderung.

3. Annahme über Infrastruktur
Die Annahme „infrastructure is fluid“, im Deutschen etwa Infrastruktur ist ständig änderbar, bedeutet, dass die unterliegenden Strukturen nicht konstant sind, wie sie es vergleichsweise bei einem Server sind, der eine bestimmte Anzahl von CPUs hat. So können z.B. Recheneinheiten (CPUs) hinzukommen oder wegfallen. Diese Annahme resultiert

aus der Verwendung von Cloud-Technologien und bildet die Basis für das Entwerfen von skalierbaren Architekturen.
Die zweite Annahme, dass Fehler konstant auftreten, ergibt sich ebenfalls aus der Verwendung von Cloud-Technologien, denn wenn eine große Anzahl von Hard- und Softwarekomponenten verwendet werden steigt die Wahrscheinlichkeit von einem unerwarteten Ausfall. Die Architektur muss also die Möglichkeit von solchen Fehlern miteinbeziehen, denn anders ist sie nicht widerstandsfähig genug, um effektiv in einer Cloud-Umgebung zu bestehen.

4 Updates und Tests verlaufen unscheinbar
Die Architekturen sind so entworfen, dass Systeme, ohne Verlust von Verfügbarkeit, upgedatet und getestet werden können. Techniken, die hier zum Einsatz kommen sind unter Anderem CI/CD Pipelines, die den Prozess von Entwicklung bis Installation automatisieren und Immutable Infrastructures ein Ansatz, bei dem bestehende Systeme nicht verändert werden und stattdessen eine neuere Version auf einem neuem System neu installiert wird.

5. Sicherheit
Sicherheit spielt eine wichtige Rolle in Cloud-Native Architekturen. Die meisten Systeme bestehen aus vielen Komponenten, was eine breite Angriffsfläche bietet. Außerdem ist Autorisierung und Authentifizierung keine einfache Sache in einem entkoppelten verteilten System. Deshalb sollte Sicherheit schon beim Entwurf der Architektur eine Rolle spielen.

2.7 Vor- und Nachteile

In diesem Abschnitt nennen und erklären wir einige Vor- und Nachteile von Cloud-Native Architekturen. Wir beginnen mit den Vorteilen.

1. Skalierbarkeit/Elastizität
Aus der Kombination von entkoppelten Komponenten, Orchestrierung und Cloud-Infrastruktur entstehen effektiv skalierbare (auch elastische) Architekturen. Orchestrierung spielt dabei eine große Rolle, da ohne sie in größeren Systemen Skalierbarkeit kaum realisierbar ist.

2. Zuverlässigkeit/Widerstandsfähigkeit
Cloud-Native Architekturen sind robust gegen Ausfälle der unterliegenden Strukturen. Hierzu werden Komponenten überwacht und bei Fehlern neu gestartet. Komponenten sind dabei oft, auch Zwecks Skalierbarkeit, redundant vorhanden. Auch hat die Orchestrierung eine wichtige Rolle, da sie für die Überwachung verantwortlich ist.

3. Änderbarkeit/Wartbarkeit
Da Komponenten (wie z.B. Microservices) weitestgehend voneinander unabhängig sind, können leichter Änderungen gemacht werden, ohne andere Komponenten auch ändern zu müssen. Auch können Teile der Architektur unabhängig vom Rest installiert oder ausgetauscht werden.

4. Übertragbarkeit
Durch die Verwendung von Containern und CI/CD Pipelines ist es möglich die Systeme in andere Umgebungen einfach zu installieren. Container ermöglichen dabei eine Uniforme Umgebung für die Komponenten und ist damit unabdingbar für Cloud-Native

Architekturen.

5. Cloud-Native Computing Foundation
Die CNCF ist eine Organisation, die die Entwicklung im Cloud-Native Bereich unterstützt. Sie ist ein zentraler Punkt für Informationen zu Best Practices, Standards, Open-Source Projekte und Neuigkeiten im Bereich Cloud-Native.

Die Liste der Nachteile ist zwar kurz, jedoch sind die Punkte hoch zu gewichten.

1. Komplexität
Wie in der Vorlesung gelernt gilt für das Entwerfen von Software-Architekturen „There is no silver bullet“. Dies trifft wahrscheinlich am meisten auf Cloud-Native Architekturen zu. Zwar existieren Tools und Herangehensweisen, die in fast allen Cloud-Native Architekturen zum Einsatz kommen, jedoch ist die schiere Auswahl an Möglichkeiten, die getroffen werden müssen, bereits eine Herausforderung. Jede Cloud-Native Applikation ist ein verteiltes System, welche an sich schon sehr komplex werden können. Addiert man hierzu noch Container/Container-Orchestrierung, Sicherheitsaspekte, Datenhaltung und Skalierung/Lastenverteilung kann sehr schnell der Überblick verloren gehen. Des Weiteren ist eine solche Architektur nicht einfach zu testen, installieren oder upzudaten und es müssen weiter Tools verwendet werden um dies zu bewerkstelligen. Abschließend ist zu sagen, dass die hohe Komplexität aus den hohen Anforderungen (Skalierbarkeit, Erreichbarkeit, Sicherheit, etc.) stammt, die an Cloud-Native Architekturen gestellt werden und daher unvermeidbar ist. Die Komplexität ist ein Tradeoff, der bei Cloud-Native Architekturen eingegangen wird, da diese in einem Zielkonflikt mit anderen Anforderungen steht, d.h. je höher die Ansprüche desto komplexer gestaltet sich die Architektur.

2. Neuer Ansatz/Technologie
Die Cloud Native Computing Foundation treibt den Bereich Cloud-Native stark voran, dennoch ist es ein relativ neuer Ansatz. Ständig werden neue Innovationen gemacht und bestehende Tools erweitert. Dies bedeutet, dass z.B. Fachliteratur und Erfahrene Entwickler kaum vorhanden sind. Jedoch wird wegen der hohe Komplexität eine hohe Expertise benötigt, was den Cloud-Native Bereich so schwierig macht.

2.8 Einsatzgebiete

Cloud-Native Architekturen werden derzeit meistens für Systeme benutzt, die entweder mit vielen Daten und/oder mit einer großen Anzahl von Benutzern umgehen müssen. Also generell Systeme, die ein hohes Maß an Skalierbarkeit fordern. Besonders in den Bereichen Streaming und Big Data werden häufig Cloud-Native Architekturen verwendet. Beispiele sind der Streaming-Dienst von Netflix sowie Produktivitätsprodukte von Google. Mittlerweile werden auch Cloud-Plattformen Angeboten (Google Cloud Platform und AWS), die es wiederum ermöglichen Cloud-Native Applikation zu entwickeln. Anzumerken ist, dass viele Unternehmen eine Migration ihrer Dienste in die Cloud vorgenommen haben, da die Möglichkeiten für Cloud basierte Systeme erst im letzten Jahrzehnt wirklich zu einer Option wurde.

3 Fallbeispiel

Wir haben uns einen Anwendungsfall selbst ausgedacht und dafür eine Architektur entworfen und prototypisch implementiert. In diesem Kapitel stellen wir Anwendungsfall, Architektur und Implementierung vor.

3.1 Beschreibung

Im Rahmen des Beispiels, haben wir eine Cloud-Native Architektur für ein Chatprogramm (ChatApp) entworfen. Die Applikation soll drei Funktionen besitzen. Benutzer sollen sich registrieren, ein-und ausloggen und Nachrichten an andere Benutzer schreiben können.

3.2 Anforderungen

Die Anforderungen sind in nicht-funktionale und funktionale Anforderungen gegliedert, die im folgenden aufgeführt sind.

3.2.1 Nichtfunktionale Anforderungen

1. Skalierbarkeit
Das System muss mit einer großen Zahl von Benutzern, die das System gleichzeitig verwenden, umgehen können.

2. Verfügbarkeit
Das System soll eine hohe Verfügbarkeit haben.

3. Sicherheit
Das System muss Berechtigungen überprüfen können (z.B. darf ein Benutzer die Nachrichten, die er abrufen will, lesen) und das System sollte sich gegen übliche Cyberangriffe (z.B. DDoS) schützen können.

4. Änderbarkeit/Erweiterbarkeit
Das System muss es zulassen, dass weitere Komponenten (z.B. Hochladen von Bildern und Videos) einfach hinzugefügt werden können.

3.2.2 Funktionale Anforderungen

1. Registrierung
Ein Benutzer kann sich mit seiner E-Mail Adresse und einem Passwort im System registrieren.

2. Ein- und Ausloggen
Ein Benutzer kann sich mit seiner E-Mail Adresse und seinem Passwort im System anmelden und danach die Funkionen nutzen bis er sich abmeldet.

3. Nachrichten schreiben/lesen
Ein Benutzer kann Nachrichten an andere Benutzer senden und Nachrichten, die an ihn gesendet worden sind, abrufen.

In Abbildung 3.1 sind die funktionalen Anforderungen in Form eines Use-Case-Diagramms dargestellt.

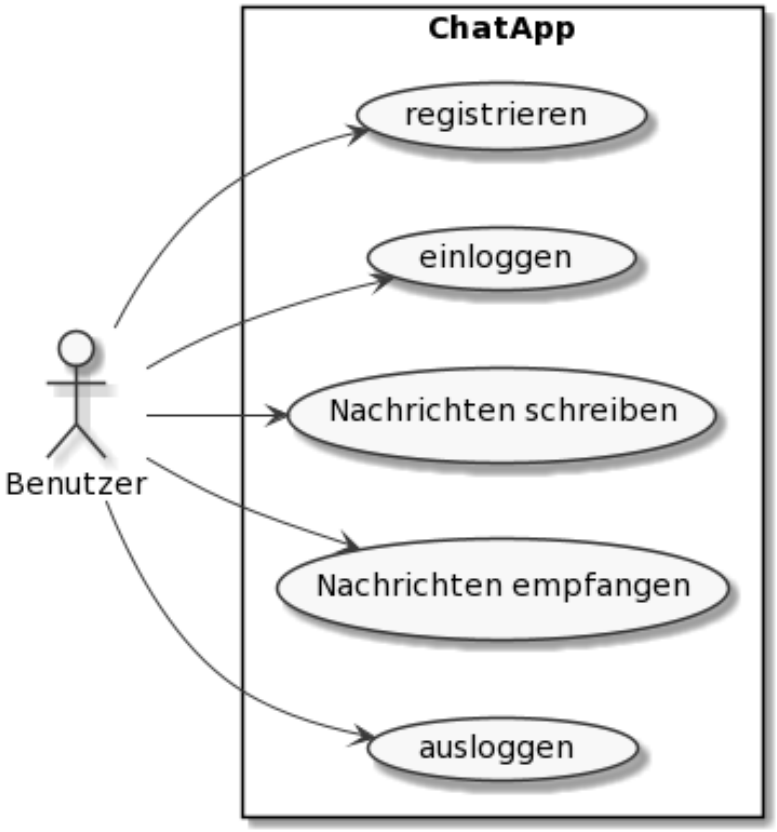

Abbildung 3.1: Use-Case-Diagramm der ChatApp

3.3 Architekturentwurf

Im diesem Abschnitt stellen wir die von uns entworfene Architektur vor und erläutern die wichtigsten Merkmale.

3.3.1 Übersicht

Wie in Abbildung 3.2 zu sehen ist besteht die Architektur aus einem Client mit graphischer Oberfläche (UI) einem API-Gateway, das alle Anfragen von Clienten empfängt und drei Microservices mit je einer eigenen Datenbank, die die Hauptfunktionalitäten des Systems implementieren.

3.3.2 Microservices

Jede Funktion aus den Anforderungen korrespondiert mit einem Microservice. Die Aufteilung der Microservices ergibt sich aus der Annahme, dass die Funktionalitäten, die sie

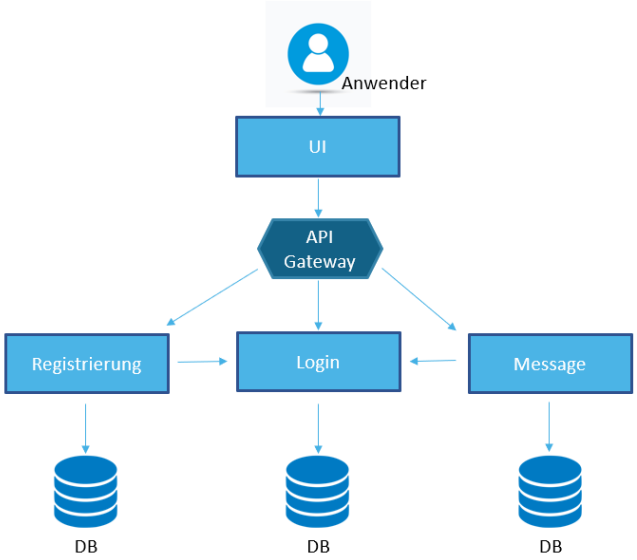

Abbildung 3.2: Architekturentwurf der ChatApp

bereitstellen unterschiedlich oft genutzt werden: Anzahl der Registrierungen < Anzahl Login/Logout < Anzahl der geschriebenen Nachrichten/Abruf von Nachrichten). Daraus gewinnen wir eine individuelle Skalierbarkeit der Funktionen, ohne Redundanz.

3.3.3 Registration-Microservice

Über den Registration-Microservice kann sich ein Benutzer registrieren. Dazu stellt er eine Schnittstelle zur Verfügung an die man eine Nachricht mit E-Mail, Passwort und Land senden kann. Beim Eintreffen einer neuen Nachricht wird ein neuer Eintrag in seiner Datenbank generiert. Des Weiteren wird eine Nachricht an den Login-Service gesendet, mit der Anfrage, auch hier den neuen Benutzer anzulegen.

3.3.4 Login-Microservice

Der Login-Microservice stellt drei verschiedene Funktionen über eine Schnittstelle zur Verfügung: Anlegen eines neuen Benutzers, Einloggen und Ausloggen. Soll ein neuer Benutzer angelegt werden (dies wird vom Registration-Microservice angestoßen), so erzeugt der Microservice einen neuen Eintrag in der Login-Datenbank, in dem E-Mail (auch als ID verwendet) und Passwort des Benutzers gespeichert werden. Will sich ein Benutzer anmelden, gleicht der Microservice die Daten mit der Datenbank ab und stellt dem Benutzer, falls die Kredentialen korrekt waren, einen Token aus, welcher auch in der Datenbank gespeichert wird. Mit diesem Token, kann er sich dann bei anderen Microservices dann autorisieren. Meldet sich ein Benutzer ab, wird der Token einfach aus der Datenbank gelöscht.

3.3.5 Message-Microservice

Die Schnittstelle des Message-Microservices umfasst das Senden einer Nachricht und das Abrufen von empfangenen Nachrichten. Im Fall des Sendens müssen die folgenden Daten angegeben werden: eine Nachricht, eine Empfänger E-Mail Adresse und einen Token. Mithilfe des Tokens stellt der Message-Microservice sicher, dass der Sender eine gültige Session hat. Dazu sendet er eine entsprechende Nachricht an den Login-Microservice,

der dann den Token bestätigt oder ablehnt. Im positiven Fall wird die Nachricht in der Datenbank des Microservices gespeichert. Will ein Benutzer seine Nachrichten abrufen, so muss er nur einen gültigen Token vorzeigen, den der Message-Microservice, wie beim Senden einer Nachricht, über den Login-Microservice abgleicht.

3.3.6 API-Gateway

Das API-Gateway schottet die Microservices ab. Das bedeutet, dass alle Anfragen vom Clienten nur an das API-Gateway gesendet werden und niemals direkt an die Microservices. Das Gateway leitet dann die Anfrage an den entsprechenden Mircoservice weiter und sendet dessen Antwort wieder an den Clienten. Die Abschottung hat mehrere Vorteile. Sicherheitsrelevante Aspekte wie z.B. DDOS-Protection oder der Schutz vor unbefugtem Zugriff (von Adressen, die nicht zum Client gehören) können auf das API-Gateway verlagert werden und müssen daher nicht in jedem Microservice implementiert werden. Des Weiteren erfüllt das API-Gateway die Rolle eines Load-Balancers, der die Anfragen auf verschiedene Instanzen von Microservices verteilt.

3.3.7 Client (UI)

Mithilfe des Clients können Anfragen an das System, genauer gesagt das API-Gateway, gesendet werden. Er enthält keine Funktionalitäten und dient lediglich zum Abrufen und Darstellen von Inhalten.

3.3.8 Verbindungen zwischen Komponenten

In Abbildung xx sind die drei wichtigsten Abläufe in einem Sequenzdiagramm festgehalten. Dargestellt sind die Registrierung, die Anmeldung und das Senden einer Nachricht. Es dient zum Verständis der Zusammenhänge der einzelnen Komponenten.

3.4 Implementierung des Prototyps

In diesem Kapitel stellen wir unsere prototypische Implementierung vor. Das gesamte System ist in fünf Teile aufgeteilt (UI, API-Gateway, Registration-Microservice, Login-Microservice und Message-Microservice), wobei jeder Teil einem unabhängigem Projekt entspricht, aus welchem jeweils ein eigenständig ausführbares Programm entsteht. Sämtliche Kommunikation wird über das HTTP Protokoll abgewickelt. Die Inhalte der Nachrichten sind dabei im JSON Format.

3.4.1 UI

Die Benutzeroberfläche ist eine eigenständige React App. ReactJS ist eine Open-Source-JavaScript Bibliothek mit der Webanwendungen erstellt werden können.
Des Weiteren wurde zu dem HTML und JavaScript verwendet. HTML ist eine Auszeichnungssprache, die zur Strukturierung und Erstellung von Webseiten verwendet wird. JavaScript ist für das Verhalten und Aussehen einer Webseite zuständig.
Der Benutzer interagiert mit der ChatApp, indem er sich z.B. in der ChatApp registriert. Die Eingaben werden anhand von HTTP Requests an das API-Gateway geschickt, das die Daten zur Weiterverarbeitung an den entsprechenden Microservices weiterleitet. Empfange Daten werden ebenfalls von dem API-Gateway an die Benutzeroberfläche gesendet, sodass die ChatApp ausschließlich mit dem API-Gateway kommuniziert.

3.4.2 API-Gateway

Das API-Gateway ist mit Java und Maven als Build-Management-Tool implementiert. Es stellt eine HTTP Schnittstelle zur Verfügung, an die Nachrichten gesendet werden können, die es dann an die Microservices weiterleitet. Für das Gateway sind die Microservices ebenfalls über HTTP Schnittstellen erreichbar. Das Loadbalancing ist für den Message-Microservice implementiert. Dabei kennt das API-Gateway zwei verschiedene Endpunkte unter denen jeweils eine Instanz des Message-Microservices auf Anfragen hört. Bekommt das Gateway eine Anfrage für den Message-Microservice wird abwechselnd eine der beiden Instanzen ausgewählt, an welche die Anfrage weitergeleitet wird. Die Wahl von zwei Instanzen ist hier zwecks Einfachheit gewählt. Es ist möglich mehr als nur zwei Instanzen des Message-Microservices oder eines anderen Microservices zu verwenden. Eine Möglichkeit um die Lastenverteilung zu Regeln wäre das Round-Robin-Verfahren, bei dem die Anfragen der Reihe nach auf die Microservices verteilt werden.

3.4.3 Microservices

Alle Microservices sind mit Java und Maven als Build-Management-Tool implementiert. Die Wahl der Programmiersprache und anderen Tools ist könnte dabei beliebig gewählt sein. Entscheidend ist nur die Schnittstelle die nach außen hin zur Verfügung stellt wird. Des Weiteren sind die Microservices Zustandslos. Für den Registration-Microservice und den Message-Microservice ist dies trivial, da sie sowieso keine temporären Daten speichern müssen. Der Login-Microservice muss sich jedoch den Token merken, den jeder Benutzer zur Autorisierung verwendet. Dies tut er indem er den Token in der Datenbank speichert. Der Einfach halber wurden die Datenbanken für den Registration-Microservice und den Login-Microservice mit einfachen Java Klassen implementiert. Dies war aber für den Message-Microservice nicht möglich, da für ihn immer mindestens zwei Instanzen existieren, welche dann keine gemeinsame Datenquelle hätten. Um zumindest die Lösungsidee zu demonstrieren, haben wir eine einfache Datei als Datenbank verwenden, die für beide Instanzen immer zugänglich ist.

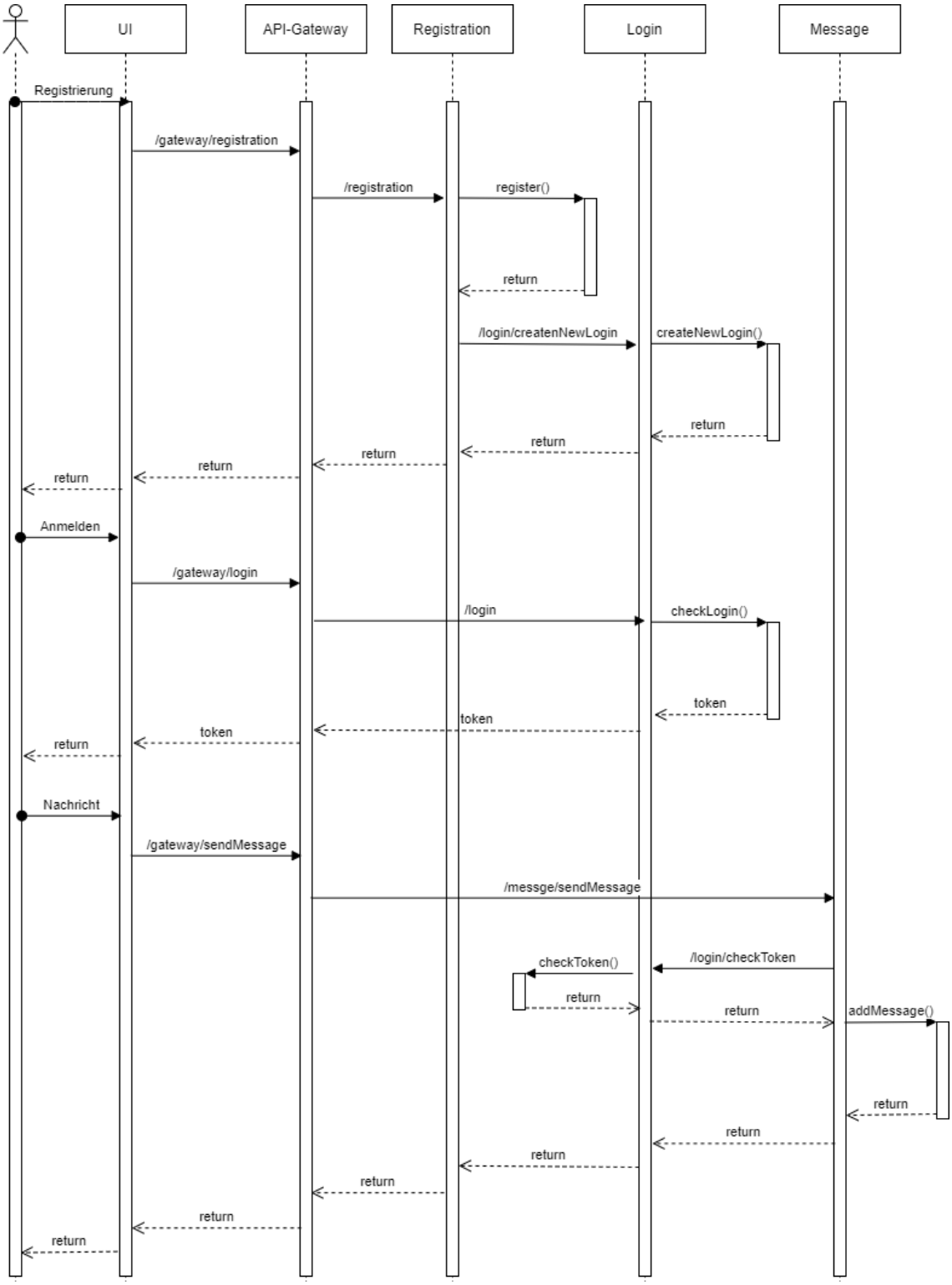

Abbildung 3.3: Sequenzdiagramm

4 Diskussion

In diesem Kapitel diskutieren wird die entworfene Architektur und den dazugehörigen Prototypen.

4.1 Vergleich mit Eigenschaften Cloud-Nativ

In diesem Abschnitt sehen wir uns an, welche Cloud-Native Eigenschaften unsere Architektur erfüllt und welche nicht.

1. Skalierbarkeit
Die Kombination von Microservices und eines Load-Balancers (API-Gateway) erfüllt auf den ersten Blick die Eigenschaft, denn es können Dank des Designs des Message-Microservices mehrere Instanzen gleichzeitig verwendet werden, auf die dann der Load-Balancer die Anfragen gleichmäßig verteilt. Da jedoch der Message-Microservice bei jeder Anfrage überprüfen muss, ob der vom Benutzer bereitgestellte Token gültig ist, wird für jede Anfrage auch eine weitere Anfrage an den Login-Microservice generiert. Diese Abhängigkeit wirkt sich stark negativ auf die Skalierbarkeit aus, da der Effekt von mehreren Message-Microservice Instanzen gleich null ist, wenn nur eine Instanz des Login-Microservices vorhanden ist. Dieses Problem ist lösbar, indem man vom Login-Microservice ebenfalls mehrere Instanzen startet und die Anfragen des Message-Microservices an das API-Gateway sendet, welches dann die Anfragen an die Login-Microservice Instanzen verteilt.

2. Aufgebaut auf der Annahme „infrastructure is fluid and failure is constant“
Hier ist leicht festzustellen, dass die Architektur diese Eigenschaft nicht erfüllt. Es wird nicht berücksichtigt, dass Infrastruktur sich ändern kann oder das Fehler auftreten können. Beide dieser Eigenschaften sind essentiell für Cloud-Native Architekturen, da ohne diese keine Anwendung in einer Cloud-Umgebung bestehen kann. Die Architektur hat jedoch das Potenzial diese Eigenschaften zu erfüllen, indem man z.B. Docker in Kombination mit Kubernetes verwendet.

3. Updates und Tests verlaufen unscheinbar
Wir können eine neue Version des Systems (API-Gateway und Mircoservices) auf neuen Strukturen installieren und testen, während die alte Version weiter verfügbar ist. Sind Installation und Test abgeschlossen kann der Verkehr von den Clients auf die neue Version geleitet werden. Dieser Ansatz nennt sich Immutable Infrastructure und wäre eine Möglichkeit diese Eigenschaft umzusetzen. Durch den modularen Aufbau der Architektur, können auch einzelne Microservices auf gleiche Weise upgedatet werden. Insgesamt erfüllt die Architektur diese Eigenschaft in der Theorie wurde aber nicht in der Praxis ausgiebig getestet.

4. Sicherheit ist ein Teil der Architektur
Diese Eigenschaft ist erfüllt. Einerseits ist ein Authorisierungsmechanismus mithilfe von Tokens vorhanden, andererseits können im API-Gateway z.B. Logging und Schutz vor

DDoS-Attacken implementiert werden. Es sei gesagt, dass dadurch das System noch weit entfernt ist von einem in der Praxis sicherem System.

5. Globale Ebene
Die Architektur ist nicht für eine globale Ebene geeignet. Es wäre zwar möglich die Anwendung mehrfach zu installieren, jedoch würden diese Installationen keine Daten teilen. Um dies zu bewerkstelligen müsste man auf verteilte Datenbanken bauen.

4.2 Microservices

Eine Möglichkeit den Prototyp zu erweitern, ist der Einsatz des Open-Source-Containerformats Docker. Dadurch wird die Anwendung bzw. die Microservices mit seinen Abhängigkeiten und alles was zur Ausführung benötigt wird, mit Hilfe von Docker in Container verpackt. Somit bietet sich dadurch die Möglichkeit, dass die Anwendung in jeder Umgebung schnell bereitgestellt und skaliert werden könnte.
Auch der Einsatz von Kubernetes wäre mit zunehmender Größe und Komplexität der Betreibung der Anwendung sinnvoll. Denn Kubernetes würde z.B. die Verwaltung und Platzierung der Container automatisieren.
Durch den Einsatz der Containerformate steigt dann auch die Widerstandsfähigkeit und Robustheit der Anwendung. Denn diese würde durch die Orchestrierungssysteme überwacht werden und bei Ausfällen oder Fehlern neu gestartet werden.

4.3 TODO

Will sich ein Benutzer registrieren so wird eine Anfrage an den Registration-Microservice generiert, welcher den neuen Benutzer dann anlegt und danach eine Anfrage an den Login-Microservice sendet, damit dort die Kredentialien gespeichert werden. Tritt nun ein Fehler auf nachdem der Benutzer in der Registration-Microservice Datenbank gespeichert wurde, wird kein Eintrag in der Login-Microservice Datenbank erzeugt, sodass der Benutzer registriert ist sich aber nicht einloggen kann. Solch eine Situation ist natürlich zu vermeiden. Es soll an diesem Beispiel deutlich werden, dass nicht nur Orchestrierung für Robustheit und Widerstandsfähigkeit verantwortlich sind, sondern auch jeder Microservice muss dafür Sorge tragen. Eine Lösung in unserem Fall wäre nach dem Ausfall des Login-Microservices einen Rollback zu machen und einen Fehler auszugeben. Ein weiterer Lösungsansatz ist, eine andere Instanz des Login-Microservices zu verwenden.

4.4 Tradeoff

Schon beim Entwurf der Architektur fällt auf, dass der Message-Microservice abhängig vom Login-Microservice ist, da er jede Anfrage autorisieren muss. Skaliert man nun den Message-Mircoservice muss man auch den Login-Microservice hochskalieren, denn ansonsten wird der Login-Microservice zu einem Flaschenhals und man hat keine skalierbares System. Generell wird an der Architektur der Zielkonflikt zwischen Sicherheit und Skalierbarkeit deutlich, da die eingesetzten Sicherheitsmaßnahmen (API-Gateway und Login-Microservice) das System verlangsamen. Jede Anfrage durchläuft eine Sicherheitsüberprüfungen im API-Gateway und muss danach noch von dem jeweiligen Microservice beim Login-Microservice authentifiziert werden.

4.5 Erweiterbarkeit

Auf Grund der Microservice-Architektur lässt sich die Anwendung gut erweitern. Dies ist der Fall, da die Anwendung aus verschiedenen Microservices besteht, die größtenteils unabhängig mit einander fungieren.
Wie in Abschnitt 4.4 beschrieben, muss bei einer Skalierung des Message-Services auch der Login-Service berücksichtigt werden. Was die Erweiterbarkeit in diesem Fall einschränkt. Die restlichen Microservices wie z.B. das API-Gateway und die Benutzeroberfläche laufen unabhängig von einander. Diese können erweitert werden, ohne die Funktionalität der anderen Services zu beeinträchtigen.

5 Fazit

Das Ziel der Seminararbeit war es, sich mit dem Thema Cloud-Native Architekturen auseinander zu setzten und prototypisch eine solche Architektur zu implementieren.
In den Kapiteln eins und zwei wurde das Thema Cloud-Native Architekturen genauer betrachtet und unter anderem die Definition, Eigenschaften und Vor-und Nachteile dargestellt. Bereits in diesen einleitenden Kapiteln wird die Komplexität des Cloud-Native Bereiches deutlich. Auf Grund der Neuheit und der aktuellen Präsenz des gesamten Cloud-Bereichs, gibt es viele unseriöse Quellen, die auf nicht wissenschaftlichem Halbwissen basieren. Dies erschwert die Suche nach zuverlässigen Informationen, welche herausgefiltert werden müssen.
In den Kapiteln drei und vier wird ein eigens formuliertes Fallbeispiel beschrieben und die entworfene Architektur sowie die prototypische Implementierung vorgestellt und abschließend diskutiert. Als Ergebnis resultiert eine Microservice Architektur, die mit einem API-Gateway und einer beispielhaften graphischen Oberfläche ergänzt wurde. Beim Entwurf der Architektur und bei der anschließenden Implementierung waren schon Schwachstellen zu erkennen, die in Kapitel vier nochmal aufgegriffen wurden. Ein Beispiel für eine solche Schwachstelle ist die Skalierbarkeit. Der Message-Service und der Login-Service können in dem entwickelten Prototypen nicht unabhängig von einander skaliert werden. Für dieses Problem wurde auch eine Lösung aufgezeigt, die jedoch nicht mehr im Rahmen dieser Ausarbeitung umgesetzt werden konnte.
Dennoch wurde ein funktionsfähiger Prototyp nach einer Microservice-Architektur implementiert, welcher die Eigenschaften des Cloud-Native Bereichs widerspiegelt.

Abschließend lässt sich sagen, dass Cloud-Native Architekturen ein sehr interessantes aber auch mächtiges Werkzeug sind, um mit sehr großen Menge von Daten und Benutzern umzugehen. Jedoch sind sie sehr komplex und benötigen eine gewisse Expertise und Erfahrung um sie effektiv einzusetzen. Daher werden sie auch hauptsächlich von Unternehmen wie Google, Facebook und Netflix verwendet.
Cloud-Native befindet sich noch in einer jungen Entwicklungsphase, spielt aber bereits jetzt schon eine wichtige Rolle im Cloud-Bereich. Cloud-Native wird sich im Laufe der Jahren immer weiter entwickeln. Dadurch könnte sich die Komplexität immer mehr verringern (z.B. durch Frameworks und andere Tools), sodass Cloud-Native Architekturen auch für kleinere Vorhaben eine Rolle spielen.

Literatur

[1] *Microservices*. URL: https://aws.amazon.com/de/microservices/ (besucht am 08. 01. 2021).

[2] *Was ist Kubernetes?* URL: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/ (besucht am 19. 01. 2021).

[3] *Was sind Microservices?* URL: https://www.redhat.com/de/topics/microservices/what-are-microservices (besucht am 08. 01. 2021).

[4] *What is a Container?* URL: https://www.docker.com/resources/what-container (besucht am 09. 01. 2021).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

API Application Programming Interfaces

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.1). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt