Bachelor Thesis

zur Erlangung des akademischen Grades
Bachelor

Technische Hochschule Wildau
Fachbereich Wirtschaft, Informatik, Recht
Studiengang Wirtschaftsinformatik (B. Sc.)

Thema (Deutsch)
Die Umstellung einer monolithischen in eine Microservices Architektur
am Beispiel von PluraPolit

Thema (Englisch)
The conversion of a monolithic to a microservices architecture using the example of PluraPolit.

Autor Edgar Muss

Matrikelnummer 50033021

Seminargruppe I1/16

Betreuer Prof. Dr. Christian Müller

Zweitgutachter Prof. Dr. Mike Steglich

Eingereicht am 14.07.2020

Abstract

Inhaltsverzeichnis

1 Einleitung

Für ein junges Unternehmen, im Bereich der digitalen Produktentwicklung, ist es wichtig, schnell Ideen umzusetzen und erstes Feedback zu erhalten. Dies helft bei der Bestätigung von Annahmen und bei der Beschaffung von Investoren.

Diese schnelle Softwareentwicklung führt jedoch zu einigen Abstrichen hinsichtlich der Qualität. So wird zu Beginn oftmals auf einen automatischen Prozess zur Bereitstellung der Applikation verzichtet und leicht umsetzbare Lösungen vor langfristigen bevorzugt. Auch hinsichtlich der Auswahl der Architektur wird anfängliche Performance als oberstes Auswahlkriterium bestimmt. Endet der Weg für ein Start-up nicht nach wenigen Monaten, dann müssen die ursprünglichen Entscheidungen hinterfragt werden.

Genau an diesem Punkt ist das junge Unternehmen PluraPolit, welches sich erst Mitte letzten Jahres gegründet hat und innerhalb von wenigen Monaten ein fertiges Produkt entwickelte. PluraPolit hat sich zur Aufgabe gestellt eine Bildungsplattform für Jung- und Erstwähler zu Entwicklern und bei der Meinungsbildung zu unterstützen. Gefördert wird das Projekt von der Zentrale für politische Bildung und ist politisch unabhängig. Des Weiteren handelt es sich bei PluraPolit um ein gemeinnütziges Unternehmen, welches keine Absicht verfolgt, Gewinne auszuzahlen. Ich bin seit Anfang Januar an diesem Projekt beteiligt und begleite es seitdem als Frontendentwickler. Gemeinsam mit einem der drei Gründer sind wir zu zweit die technische Abteilung des Unternehmens und kümmern uns um die Weiterentwicklung der Plattform.

Die Inhalte der Plattform werden von den zwei anderen Gründern gepflegt und eingeholt. Es handelt sich dabei um neun verschiedene Tonaufnahmen zu einer Frage. Die Audioaufzeichnung kommen ausschließlich von Politikern und beziehen sich auf eine aktuelle politische Fragestellung. So gibt es zum Beispiel das Thema: Sollte der öffentlich-rechtliche Rundfunk abgeschafft werden? Die jeweiligen Politikerinnen und Politiker werden direkt zu einem Statement angefragt, welches anschließend ohne inhaltliche Veränderung auf die Webseite geladen wird. Angefragt werden immer alle Parteien, die im Bundestag vertreten sind. Neben Fragen, die ausschließlich von Politikern diskutiert werden, kommen gleichermaßen Themen auf die Plattform, die von den jeweiligen Expertinnen und Experten beantwortet werden. So gibt es bei der oben genannten Fragestellung auch eine Äußerung des Vertreters der Landesrundfunkanstalten ARD. Im Gegensatz zu anderen Anbieter für Nachrichten rund um Politik, stellt PluraPolit ausschließlich Sprachnachrichten auf die Plattform, die von jeweiligen Expertinnen und Experten kommen. Es wurde sich exklusiv für das Medium Tonaufnahme entscheiden, um eine junge Zielgruppe anzusprechen und die einzelnen Beiträge wie einen Podcast hören zu können.

1.1 Problemstellung

Umgesetzt wurde die Plattform in Ruby on Rails im Backend und React.js[1]im Frontend. Dabei liefert das Backend auf Anfrage Inhalte an das Frontend und kümmert sich um die Speicherung von Daten. Das Frontend im Gegensatz fordert beim Laden der Webseite alle genötigten Informationen an und stellt sie anschließend dar. Trotz dieser Einteilung handelt es sich um eine Applikation mit gemeinsamer Codebase und einem Bereitstellungsprozess.

Gehostet werden die Applikationen über den Cloud-Computing-Anbieter Amazon Web Services[2](AWS). Es wurde sich für diesen Dienst entschieden, um möglichst geringe Fixkosten zu haben und bei belieben die Kapazität ändern zu können. Die Anwendung wird in einem Docker-Container[3] gespeichert und per Github Actions[4] an AWS geliefert. Dort wird die Applikation in das Elastic Container Service[5] (ECS) geladen und von Fargate[6] verwaltet. Die Daten werden in einer PostgreSQL[7] Datenbank abgespeichert, die auf einer Relational Database Service[8] (RDS) Instanz hinterlegt ist. Bilder und Tonaufnahmen werden in einem Simple Storage Service-Bucket[9] (S3) gespeichert und stehen der Webseite per URL zur Verfügung. Um automatisch zu jedem Beitrag Intros zu generieren, wurde eine AWS Lambda[10] Funktion geschrieben, die aus der Basis von Beschriftungstexten für jede Audioaufzeichnung eine weitere Aufnahme für die Einleitung erstellt.

[1]React.js ist eine JavaScript Bibliothek zum Erstellen von Benutzeroberflächen. Diese verwaltet die Darstellung im HTML-DOM und ermöglicht dem Entwickler Informationen zwischen Funktionen zu administrieren (*React* 2020).

[2]AWS ist ein Tochterunternehmen des Online-Versandhändlers Amazon mit einer Vielzahl an Diensten im Bereich Cloud-Computing (*AWS Homepage* 2020).

[3]Docker-Container sind isolierte virtuelle Umgebungen, in der eine Anwendung separat vom System des Rechners betrieben wird. Dadurch können Applikationen leicht von einem Computer zu einem Hosting Dienst geladen werden (*What is a Container?* 2020).

[4]Github Actions ist ein Software Dienst von Github, welches hilft Prozesse zu automatisieren. Es kann zum Beispiel zum automatischen Bereitstellen einer Webseite verwendet werden (*Github Actions* 2020).

[5]Elastic Container Service ist ein Container-Orchestrierungs-Service von Amazon Web Services, mit dessen Hilfe Container skalierbar verwaltet werden können (*AWS ECS* 2020).

[6]Fargate ist eine Serverless-Datenverarbeitungs-Engine, welche Container im Rahmen der vordefinierten Parameter verwaltet. So werden zum Beispiel durch diesen Dienst bei erhöhtem Bedarf neue Instanzen bereitgestellt und bei Verlust von Last Container-Instanzen eliminiert (*AWS Fargate* 2020).

[7]PostgreSQL ist eine objektrelationale Datenbank, welche sowohl Elemente einer relationalen als auch einer Objektdatenbank besitzt (*PostgreSQL* 2020).

[8]RDS ist ein Service von Amazon Web Services, mit dessen Hilfe relationale Datenbanken verwaltet werden. Der Dienst ermöglicht das Aufsetzen, Managen und Skalieren von Datenbanken, wie zum Beispiel MySQL, MariaDB und PostgreSQL (vlg. Baron, Baz und Bixler 2016, S.161 f.).

[9]Der Speicherdienst von AWS S3 ist einer der ersten Dienste des Cloud-Computing-Anbieters. Er erleichtert die Speicherung von Objekten in der Cloud jeglichen Formats und lässt sich einfach verwalten. Die verwendete Speichermenge ist dynamisch und richtet sich automatisch nach der Größe der Dateien (vlg. Baron, Baz und Bixler 2016, S. 23).

[10]Amazon Lambda ist ein Service von AWS, über welchen Funktionalität innerhalb der Cloud ausgeführt wird. Es kann sich dabei um ein Service der Serverless ist, was bedeutet, dass sich nicht um den Server gekümmert werden muss. Somit können kleine Programme mit wenig Aufwand ausgeführt werden (vlg. Wolff 2018, Kap. 15.3; *AWS Lambda* 2020).

Mit wachsender Codebase erhöht sich der Aufwand, der notwendig ist, um neue Funktionen zu entwickeln und zu implementieren. Dies liegt besonders daran, dass sich im Laufe der Entwicklung viele Abhängigkeiten zwischen Klassen und Methoden hervorgetan haben. Hierdurch steigt der Aufwand, der nötig ist, um sich in den Quellcode einzuarbeiten. Verursacht wird diese Korrespektivität, indem im Frontend die Funktionen und Klassen in logisch getrennte Bausteine geteilt und an mehreren stellen verwendet werden. Dies ermöglicht zwar eine schnelle Entwicklung, führt jedoch dazu, dass eine Veränderung einer Komponente Änderungen an mehreren Stellen auslöst. Diese Abhängigkeiten machen es mit steigender Menge an Sourcecode, immer komplexer weitere Funktionen umzusetzen, ohne bestehende Logik zu verändern. Hinzukommt, dass neben dem eigenen Quellcode auch externe Funktionalitäten genutzt werden, welche durch den Paketverwaltungsdienst von Node.js (Node.js 2020) npm installiert werden.

Diese werden jedoch nur in Teilen der Anwendung verwendet, werden allerdings zum gesamten Frontend hinzugefügt. Ingesamt verlangsamt es die Bereitstellung der Applikation, da sie während des Prozesses installiert werden müssen. Für eine schnelle Entwicklung ist es somit wichtig, einen rapiden Bereitstellungsprozess zu entwickeln und die Zahl der externen Pakete auf das Nötigste zu begrenzen.

Um in Zukunft eine schnelle Weiterentwicklung der Applikation sicherzustellen, hat PluraPolit beschlossen, den aktuellen Aufbau in eine Microservice-Architektur zu ändern und die gesamte Plattform in inhaltlich getrennte Module zu teilen.

1.2 Zielsetzung

Schon im Jahr 2005 hat Peter Rodgers auf der Web Services Edge Conference über Micro-Web Services referiert. Er kombinierte die Konzepte der Service-Orientierten-Architektur (SOA) mit den der Unix-Philosophie und sprach von verbundenen REST-Services. Er versprach sich dadurch eine Verbesserung der Flexibilität der Service-Orientierten-Architektur (vlg. Rodgers 2018). Erstmalig 2011 wurde dieser Ansatz als Microservice-Architektur bezeichnet (vlg. Dragoni u. a. 2017). Ab 2013 entwickelte sich rund um das Thema eine immer größer werdendes Interesse, welches dazu führte, dass mehr Blogposts, Bücher, sowie wissenschaftliche Arbeiten geschrieben wurden. Somit sind die Definition und die Charakteristiken bis ins Detail beleuchtet. Des Weiteren gibt es einige Beispiele von bekannten Unternehmen, wie Netflix und Amazon, die die Herausforderungen der Überführung ihres Systems zu einer Microservice-Architektur beschreiben.

Trotz der Informationslage ist jedoch noch relativ unbekannt, ob auch Start-ups Microservices umsetzen können und welche Bedingungen dafür erforderlich wären. Es gibt kaum Erfahrungen, die es PluraPolit ermöglicht abzuschätzen, ob sich eine Umstellung zum aktuellen Zeitpunkt lohnt und welche Eigenschaften ein Unternehmen erfüllen müsste.

Aus diesem Grund ist das Ziel dieser Arbeit für PluraPolit die Bedingungen zu ermitteln, die für eine mögliche Umstellung erforderlich wären und eine klare Bewertung für die Sinnhaftigkeit des Vorhabens abzugeben. Insbesondere das Ausarbeiten der notwendigen Anforderungen an ein Unternehmen, welches sein System von einer monolithen Architektur zu einer Microservices-Architektur umstellen möchte, soll PluraPolit und anderen Start-ups helfen bewerten zu können, ob sich eine solche Überführung lohnt.

1.3 Vorgehen

Die Arbeit teilt sich in drei Bereiche auf: Den theoretischen Rahmen, die Methodik und die Auswertung. So wird im ersten Abschnitt die theoretische Grundlage für Microservices gebildet. Es werden einzelne wichtige Merkmale beleuchtet und beschrieben. Außerdem wird ein Vergleich zwischen der aktuellen Software-Architektur des Unternehmens und Microservices erstellt. Anschließend werden aus den Merkmalen und der Gegenüberstellung wichtige Bedingungen für die Umstellung zu einer Microservice-Architektur abgeleitet, welche Grundlage für die Einschätzung sind.

Im nächsten Abschnitt werden diese Bedingungen im Rahmen einer qualitativen Befragung von Experten im Bereich Microservices eingeschätzt und bewertet. Hierfür werden Interviews durchgeführt. Es wird beschrieben, welche Experten ausgewählt werden und welche Expertise sie mitbringen. Des Weiteren werden die einzelnen Interviewfragen vorgestellt und deren Zusammenhang zur Zielsetzung erklärt. Dadurch wird deutlich, welchen Einfluss die Expertenaussagen auf die Einschätzung für PluraPolit haben.

Abschließend werden die Aussagen aus den Befragungen mit der theoretischen Ausarbeitung verglichen und auf PluraPolit bezogen. Hierfür wird die Umsetzbarkeit für das junge Unternehmen hinterfragt und die Auswertung diskutiert. Neben Microservices wird auch SOA als alternative Lösung vorgestellt. Beendet wird die These mit einer Einschätzung für PluraPolit, in der eine klare Beurteilung für oder gegen eine Überführung abgegeben wird.

2 Theoretischer Rahmen

2.1 Software Architektur

Seit dem die ersten Großrechner gebaut wurden und ein Projekt nicht mehr von einem Team allein entwickelt werden konnte, entstand der Bedarf komplexe Systeme aufzuteilen und zu strukturieren. So war es schon in den 60er Jahren notwendig, die Entwicklung des Betriebssystems OS/360 von IBM in mehrere Team aufzuteilen und klare Schnittstellen zwischen den Teilen zu bestimmen (Brooks 1995). Es entwickelte sich daraus eine der ersten Anwendungen und Umsetzungen von Softwarearchitektur, welche erstmalig 1969 bei einer Softwaretechnik Konferenz in Rom auch als solche bezeichnet wurde (vgl. Buxton und Randell 1970, S. 12). In den darauf folgenden Jahren wuchs das Interesse an der Thematik und die Anwendungen der Teilung und Strukturierung von Softwaresystemen. Hieraus entstand die im Jahr 2000 veröffentlichte Norm IEEE1471:2000, welche am 15. Juli 2007 als ISO/IEC 42010 übertragen wurde. In diesem Standard werden Anforderungen an die Beschreibung von System-, Software- und Unternehmensarchitekturen definiert (Hilliard 2020).

2.1.1 Definition

Nach der Norm IEEE1471:2000 handelt es sich bei dem Begriff Softwarearchitektur, um eine *„grundlegende Organisation eines Systems, die in einzelne Komponenten und ihren Beziehungen untereinander unterteilt ist“*, sowie der technischen Umsetzung und weiterführenden Prinzipien hinsichtlich des Programmierparadigma (Clements 2005, S. 12).

Helmut Balzert, einer der führenden Pioniere im Bereich Softwarearchitektur und Autor der Bücherreihe *Lehrbuch der Softwaretechnik*, beschreibt diese als *„eine strukturierte oder hierarchische Anordnung der Systemkomponenten, sowie Beschreibung ihrer Beziehungen“* (Balzert 2011, S. 580). Ihm nach lässt sich somit jedes System in mehrere einzelne Komponenten teilen, welche untereinander in Verbindung stehen und gemeinsam das Gesamtsystem formen.

Paul Clements, Autor der IEEE Norm, schließt sich Balzert an und beschreibt Softwarearchitektur als *„Strukturen eines Softwaresystems: Softwareteile, die Beziehungen zwischen diesen und die Eigenschaften der Softwareteile und ihrer Beziehungen“* (Clements u. a. 2010, S. 23).

Neben diesen rein technischen Betrachtungen des Begriffes, gibt es auch eine von Martin Fowler, die mehr auf den soziale Austausch beim Entwicklen abzielt und Softwarearchitektur als *„geteiltes Verständnis von hart zu änderten Entscheidungen sieht“* (Fowler 2003, S. 3). Dabei gibt Fowler keinerlei Vorgaben hinsichtlich jeglicher Untergliederung, sondern beschreibt mehr den Austausch, durch welchen Softwarearchitektur im Unternehmen aus-

geführt wird. Ihm nach handelt es sich dabei um ein offenen Disput in dem Jeder die Rolle des Softwarearchitekten einnehmen kann (Fowler 2003, S. 3 f.).

Somit wird, abgesehen von der Betrachtung von Fowler, Softwarearchitektur als Strukturierung von einzelnen Komponenten, die untereinander in Beziehung stehen definiert. Dabei können sowohl die Komponenten als auch die Beziehungen Eigenschaften besitzen. Die einzelnen Komponenten zusammen ergeben das Gesamtsystem, welches in einer bestimmten Struktur vorliegt und beschrieben wird. Folglich beinhaltet die Softwarearchitektur alle nötigen Informationen über die Struktur der einzelnen Systemkomponenten und deren Kommunikationen untereinander.

Wird ein Softwaresystem in Komponenten geteilt, welches jedoch selbst eine Funktionalität besitzt, bedeutet dies, dass auch diese Logik in Teile geteilt wird und jede einzelne Komponente einen Teil erfüllt. Um gleichermaßen den selben Funktionsumfang, wie das Gesamtsystem bewältigen zu können, müssen die einzelnen Komponenten zusammenarbeiten. Demnach ist es wichtig, die Zuständigkeit jeder einzelnen Komponente genau zu klären und die Abhängigkeiten zu bestimmen. Auf Grund der Abhängigkeiten werden im Anschluss Schnittstellen definiert, über welche die einzelnen Komponenten Informationen austauschen können.

Bei der Teilung in einzelne Komponenten können bekannte Architekturstile helfen, da sich diese im Laufe der Zeit entwickelt haben und gut dokumentiert sind. Architekturstile sich dabei einige Regeln zur Strukturierung eines Systems und fassen Merkmale eines IT-Systems, sowohl hinsichtlich der Komponenten, als auch ihrer Kommunikation zusammen (vlg. Starke 2015, S. 102). Seit Beginn der Softwarearchitektur hat sich eine Vielzahl an solchen Stilen entwickelt, die aus unterschiedlichen Intention entstanden sind. Jedes System hat dabei unterschiedliche Herangehensweisen, sowie Vor- als auch Nachteile. Da die Lösung eines Problem von der eigentlichem Umsetzung unabhängig ist, können auch unterschiedliche Architekturstile zur ein und der selben Lösung verwendet werden. Hinsichtlich der Auswahl des Stiles gibt es demnach keine richtige oder falsche Antwort. Es ist viel mehr das Abwiegen von Pro und Kontra, sowie eine persönliche Präferenz der Entwickler.

Da die unterschiedlichen Architekturstile nicht im Fokus dieser Bachelor Thesis sind, werden nur folgende Stile betrachtet:

1. Verteilte Systeme,
2. Interaktionsorientierte Systeme,
3. REST-Architektur,
4. Monolithe Architektur

2.1.2 Verteilte Systeme

Nach Andrew Tanenbaum werden verteilte Systeme als eine Menge unabhängiger Computer bezeichnet, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen (Tanenbaum und Steen 2007). Weiterführend beschreibt Gernot Starke die einzelnen Komponenten als entweder Verarbeitungs-, oder Speicherbausteine, die über definierte Schnittstellen innerhalb eines Kommunikationsnetz zusammenarbeiten (vlg. Starke 2015, S. 116). Im Gegensatz zur Definition von Softwarearchitektur grenzt sich das verteile System dadurch ab, dass von unabhängigen Computer geredet wird. Dadurch entstehen einige Vorteile. So lassen sich auf Grund der Unabhängigkeit, diese einzelnen Rechner unterschiedlich skalieren. Es entsteht dadurch ein Netzwerk an heterogenen Computern. Somit kann je nach Anforderung der einzelnen Komponenten, eine entsprechende Rechenleistung genutzt werden. Des Weiteren gibt es auf Grund der Verteilung eine gewisse Ausfallsicherheit. Diese entsteht, da das Gesamtsystem nicht durch eine, sondern durch mehrere Maschinen getragen wird. Dadurch können einzelne Computer ausfallen ohne, dass das Anwendungssystem ausfällt. Jedoch kann dadurch ein Teil der gesamten Funktionalität wegfallen, der ggf. für das System notwendig ist. Um dies zu verhindern, sollten geeignete Maßnahmen getroffen werden, indem zum Beispiel nicht allein ein Computer für die Funktionalität verantwortlich ist, sondern mehrere.

Jedoch entsteht mit der Verteilung auch ein Anstieg der Komplexität, sowohl bei dem Konzipieren des Systems, als auch bei der Wartung und Managen dessen. Außerdem muss nicht nur ein Rechner abgesichert werden, sondern ein ganzes Netzwerk. Dadurch entsteht ein höherer Aufwand zur Absicherung.

Die einzelnen Computer können über verschiedene Mechaniken miteinander kommunizieren: Einerseits durch direkten Aufruf entfernter Funktionalität und andererseits durch indirekten Austausch von Informationen (vlg. Starke 2015, S. 116). Dabei kann der Transfer synchron oder asynchron ablaufen. Bei einem synchronen Aufruf, der nur direkt ausgelöst wird, führt ein Computer über das Netzwerk die Funktionalität eines anderen aus und wartet auf dessen Antwort (*Synchrone Kommunikation* 2018). Bei einem asynchronen Aufruf wird entweder direkt oder indirekt Logik eines anderen Computer aufgerufen, während der aufrufende Rechner, ohne auf die Antwort zu warten, weiter verarbeitet. Ist die aufgerufene Rechner fertig gibt er das Resultat zurück, welches vom ersten Computer aufgenommen und verarbeitet wird (*Asynchrone Kommunikation* 2019).

Der Austausch von Informationen und das Aufrufen von externer Funktionalität kommt in einem System dauerhaft vor. Dadurch besteht das Risiko, dass einzelne Informationen verloren gehen können und das System sicherstellen muss, dass das Anwendungssystem nicht darunter leidet.

2.1.3 Interaktionsorientierte Systeme

Interaktionsorientierte Systeme zeichnen sich dadurch aus, dass sie den Fokus auf die Interaktion zwischen Mensch und Maschine legen (vgl. Starke 2015, S. 124). Ein viel verwendeter Vertreter hiervon ist der Model-View-Controller Ansatz. Hierbei werden die einzelnen Komponenten in drei unterschiedliche Kategorien eingeteilt, von dem jeweils ein Repräsentant vorhanden sein muss. Eingeteilt wird in die drei Kategorien: Model, View und Controller, unterdessen jede Gattung eine eigene Funktion besitzt.

So kümmert sich das Model um die Datenspeicherung, den Datenabruf und Verarbeitung von Informationen und ist eine Verbindung zwischen dem Speichermedium und der Anwendung (siehe Abb. 1 und Abb. 2 Punkt 4 und 5 bzw. Punkt 5 und 6). Da jegliche Speicherung ausschließlich über das Model abläuft, wird dadurch das Speichern, sowie das Abrufen einheitlich geregelt.

Davon getrennt sind die graphischen Darstellungen, welche durch Views definiert werden. Sie erhalten ihre Informationen vom Model. Dabei können die Daten entweder direkt vom View angefordert (siehe Abb. 2 Punkt 4) oder indirekt vom Controller übergeben werden (siehe Abb. 1 Punkt 3 und 7). Unabhängig des Informationsflusses hilft es die Darstellung von der Restlichen Logik zu trennen, um eine einheitliche Verantwortung zu waren. Dies ist insbesondere empfehlenswert, da Dateien, zum beschreiben von Views, schnell unübersichtlich werden.

Der Controller hingegen kümmert sich, um die Verwaltung der Benutzereingabe und um das Aufrufen der Views (siehe Abb. 1 und Abb. 2 Punkt 2 und 7 bzw. Punkt 2 und 3). Er sorgt dafür, dass die Events oder Aktionen, die vom Benutzer über die Views auslöst werden, verarbeitet werden und führt entsprechende Datenverarbeitungen im Model aus. Falls notwendig lädt er auch Informationen aus der Datenbank und übergibt sie der entsprechenden View, welche er abschließend rendert.

Beim Model-View-Controller Ansatz handelt es sich um ein Muster, welches oft in der Softwarearchitektur verwendet wird, da es die Verantworlichkeiten trennt und das System somit verständlicher wird. Im Unterschied zu verteilten Systemen stellt das Muster jedoch keine Anforderungen bezügliche der Hardware. Viel mehr beschreibt es eine Art den Quellcode hinsichtlich seiner Funktion zu teilen. Demnach kann dieser Stil auch für Codeteilung innerhalb einer Komponente verwendet werden und beschreibt nicht zwingend ein Architekturstil, der sich auf das Gesamtsystem besieht.

2.1.4 REST-Architektur

Neben den bislang genannten Architekturstilen gibt es eine Vielzahl von weiteren Strukturierungen, die sich erst in den letzten 20 Jahre entwickelt haben. Einer dieser Architektur-

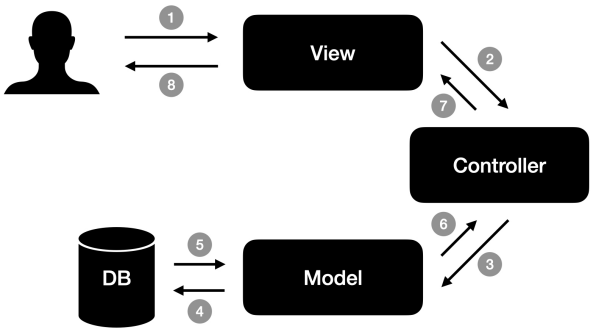

Abbildung 1: Kommunikation zwischen Controller und Model,
(1) Der Benutzer sieht die graphische Darstellung und führt eine Aktion aus (2). Der Controller verarbeitet die Aktion und ruft über das Model Daten ab (3). Das Model greift auf die Datenbank zu und lädt die entsprechenden Informationen (4 und 5). Anschließend gibt es die Inhalte an den Controller zurück (6). Dieser gibt die Daten an die View weiter (7), welche Abschließend die neuen Inhalte dem Benutzer anzeigt (8).

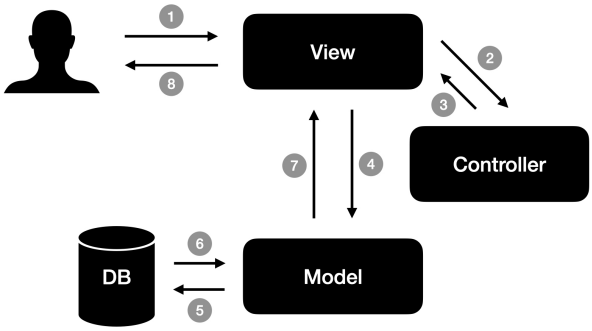

Abbildung 2: Kommunikation zwischen View und Model,
(1) Der Benutzer sieht die graphische Darstellung und führt eine Aktion aus (2). Der Controller verarbeitet die Aktion und rendert umgehend die graphische Darstellung (3). Der View lädt über das Model die Daten (4). Das Model greift auf die Datenbank zu und lädt die angeforderten Informationen (5 und 6). Anschließend gibt es die Inhalte an den View zurück (7). Dieses Abschließend die neuen Inhalte dem Benutzer anzeigt (8).

stile ist die REST-Architektur, welche vom Miterfinder des HTTP-Standards Roy Fielding definiert wurde (Starke 2015, S. 128). Er beschrieb diesen Stil in seiner Dissertation an der Universität von Kalifornien im Jahr 2000 und charakterisiert ihn als Architekturstil für das Web.

Dabei steht REST für *Representational State Transfer*, welches einen Architekturstil für verteilte Systeme beschreibt und auf der Server-Client Architektur aufbaut (Fielding 2000, S. 76). Server-Client Architektur beschreibt eine Ausprägung eines verteilten Systems, bei dem die Anwendung in Server und Clients geteilt werden (Starke 2015, S. 117).[11]

Ein Server ist dabei eine Software Komponente im Netzwerk, welche Services anbietet. Ein Service könnte beispielsweise zuständig sein, alle Informationen der hinterlegten Kunden auszugeben. Der Client hingegen konsumiert lediglich diese Informationen und dient dem Benutzer als Bedienungsoberfläche. Dies bedeutet, dass der Server nur passiv auf Anfragen vom Client wartet, während der Client selbst keine Informationen verarbeitet, sondern ausschließlich anzeigt. Nichtsdestoweniger handelt es sich um ein Programm, welches auf dem Endgerät des Benutzers (Computer, Mobiltelefon) ausgeführt wird.

Die REST-Architektur verwendet diese Aufteilung, um eine feste Trennung der Zuständigkeit zu integrieren (vgl. Fielding 2000, S. 78).

Die zweite Bedingung, die Fielding an den Architekturstil stellt, ist dass die Kommunikation zustandslos, zu Englisch (stateless), abläuft (Fielding 2000, S. 78). Dies bedeutet, dass die Nachrichten, die zwischen Server und Client ausgetauscht werden, alle nötigen Informationen beinhalten (Starke 2015, S. 128). Somit gibt der Server auf Anfrage des Clients stets die gleiche Antwort zurück, egal ob dieser zum Ersten, oder wiederholten Mal angefragt wurde. Des Weiteren hängt die Antwort nicht vom Client ab. Diese Entkopplung zwischen den Komponenten ermöglicht, dass die Aufgabe des Servers, sowie des Clients durch mehrere Computer verrichtet werden kann und somit das System skalierbar ist (Fielding 2000, S. 79).

Der Hauptunterschied zwischen der REST-Architektur und anderen Stilen, liegt jedoch in der genauen Bestimmung der zu verwendenden Kommunikationsschnittstellen. So bestimmt die REST-Architektur sehr explizit, welche From zur Kommunikation verwendet werden darf. Anders als andere Stile beruht der Aufruf von Methodiken nicht auf individueller Funktionalität, sondern auf dem HTTP-Standard. Konkret bedeutet dies, dass die einzelnen Dienste des Servers sich an die HTTP-Optionen (GET, PUT, POST und DELETE) richten und keinen eigenen verwenden (vlg. Starke 2015, S. 128). Somit baut die REST-Architektur auf einem Kommunikationsstandard auf, der sich im Internet etabliert

[11] Dabei beziehen sich die Begriffe *„Server“* und *„Client“* auf Software Komponenten und auf den physischen Server und das Endgerät des Nutzers (Client). Des Weitern grenzt sich der Architekturstil von der Mainframe Architektur ab, bei der über Terminals Anweisungen an ein Großrechner gestellt werden.

hat.

Auf Grundlade der standardisierten Kommunikation können zwischen Server und Client intelligente Zwischenstationen geschaltet werden, die dafür zuständig sind häufig vorkommende Anfragen abzuspeichern (vlg. Fielding 2000, S. 79 f. Starke 2015, S. 128). Somit lässt sich eine Vielzahl von Serveranfragen im Voraus beantworten.

Die Antwort des Servers erfolgt durch Repräsentationen der Daten, wovon es für jede Ressource mehr Formate gibt. So kann eine Schnittstelle abhängig des angeforderten Mediums, sowohl JSON, als auch XML oder HTML zurückgeben (vgl. Starke 2015, S. 128).

Verwendet wird die REST-Architektur ausschließlich für Anwendungen im Internet, da es auf die Anwendung des Hypertext Transfer Protokoll[12] (HTTP) angewiesen ist. Dabei findet der Architekturstil sowohl Anwendung für ganze Systeme, als auch in komplexen Anwendungen mit einer Vielzahl einzelner Services.

2.1.5 Monolithe Architektur

Der Begriff *„Monolith“* leitet sich vom altgriechischen *„monólithos“* ab und bedeutet *„aus einem Stein“* (vlg. *Duden Monolith* 2020; vgl. *DWDS – Digitales Wörterbuch der deutschen Sprache* 2020). In der Gesteinskunde wird damit ein natürlich entstandener Gesteinsblock bezeichnet, der komplett aus einer Gesteinsart besteht (vgl. *DWDS – Digitales Wörterbuch der deutschen Sprache* 2020).

Nach Rod Stephens liegt eine monolithische Softwarearchitektur vor, wenn jegliche Funktionalität des Systems miteinander verbunden ist. Dabei spricht er über die Verbindung von Dateneingabe, Datenausgabe, Datenverarbeitung, sowie Fehlerhandhabung und Benutzeroberflächen (vgl. Stephens 2015, S. 94).

Anders sieht es Sam Newman. Ihm nach liegt ein monolites System schon dann vor, wenn die gesamte Funktionalität eines Systems gemeinsam über einne Deployment-Prozess bereitgestellt wird (vgl. Newman 2019, Kap. 2.2). Folglich muss nicht zwingend jegliche Logik miteinander verbunden sein. Er unterteilt monolithe Systeme in drei Kategorien: Einzelprozess Monolithe, Modulare Monolithe und verteilte Monolithe (vgl. Newman 2019, Kap. 2.2).

Der Einzelprozess Monolith ist die gängigste Form und deckt sich mit der Definition von Rod Stephens. Somit handelt es sich dabei, um ein System bei dem das gesamte System einen Prozess abbildet. Dies bedeutet, dass jegliche Funktionalität aufeinander aufbauend ist und nur eine Datenspeicherung für die gesamte Anwendung verwendet wird (vgl. Newman 2019, Kap. 2.2.1).

[12] Weitere Informationen zum Hypertext Transfer Protokoll kann unter folgender Literatur gefunden werden (Leach u. a. 2020).

Anders ist dies beim Modularen System. Dieses zeichnet sich darin aus, dass die Funktionalität in einzelne Module geteilt wird und sogar einzelne Module eine separate Datenspeicherung besitzen können (vgl. Newman 2019, Kap. 2.2.2). Diese Form des monolithem System ist jedoch seltener und wird nur vereinzelt von Unternehmen eingesetzt.

Im Gegensatz zu verteilten Systemen sind die einzelnen Komponenten nicht auf separaten Computern verteilt und werden durch einen Deployment-Prozess online gestellt. Des Weiteren sind die einzelnen Module nur leicht entkoppelt, so kann es immer noch Abhängigkeiten geben, anders als bei verteilten Monolithen (vgl. Newman 2019, Kap. 2.2.2). Diese sind komplett entkoppelt und kommunizieren nur noch über definierte Schnittstellen (vlg. Starke 2015, S. 116). Sie erfüllen somit jegliche Anforderungen an ein verteiltes System, sind jedoch in einem einzigen Bereitstellungsprozess gebündelt. Diese From wird jedoch kaum verwendet.

Weder Stephens noch Newman geben Vorgaben hinsichtlich der Gliederung innerhalb eines monolithischen Systems. Demnach kann ein Model-View-Controller Ansatz als monolithisches System gelten, solange es einheitlich bereitgestellt wird. Anders ist es mit einem verteilten System, da nach Definition ein Monolith kein verteiltes System sein kann.

Im Rahmen dieser Arbeit wird bei jeglichen weiteren Referieren auf den Begriff Monolith stets von einem Einzelprozess Monolithen ausgegangen, außer es wird expliziert von einem Modularen, oder verteilten Monolithen geschrieben.

Im Vergleich zu einem verteilten System gibt es einige Vor- als auch Nachteile (vgl. Newman 2019, Kap. 2.2.4 und Kap. 2.2.5). So ist das Bereitstellen eines Monolith einfacher, da es einen Bereitstellungsprozess für die gesamte Anwendung gibt. Wiederum führt dies dazu, dass der Prozess deutlich länger dauert. Diese Tatsache ist insbesondere dann gravierend, wenn vermehrt kleine Änderungen vorgenommen werden. Anderseits vereinfacht eine Anwendung, die als ein Prozess zu sehen ist, die Fehlersuche und ermöglicht es Funktionen mehrfach zu verwenden. So lassen sich Funktionen und Klassen in einem Einzelprozess Monolithen mehrfach verwenden und schneller neue Funktionen umsetzen, jedoch verursacht dies, dass schnell Abhängigkeiten entstehen können und Änderungen ungewollte Fehler verursachen. Dadurch wird die Umsetzung von neuen Funktionen mit steigender Codemenge verlangsamt und der Einstieg von neuen Teammitgliedern erschwert.

Bei größeren Unternehmen mit mehreren Team kommt hinzu, dass es leicht zu Konflikten kommen kann, da alle auf die gleiche Codebase zugreifen. So führt ein Monolith dazu, dass bei einer großen Anzahl an Entwickler viele Absprachen nötig sind und es zu Problemen bei der Zusammenführung von Funktionen kommen kann (vgl. Newman 2019, Kap. 2.2.4). Anders ist es beim Erstellen von System übergreifenden Test: Diese werden durch ein monolithisches System begünstigt und können im Vergleich zu einem verteilten System einfacher umgesetzt werden (vgl. Newman 2019, Kap. 2.2.5).

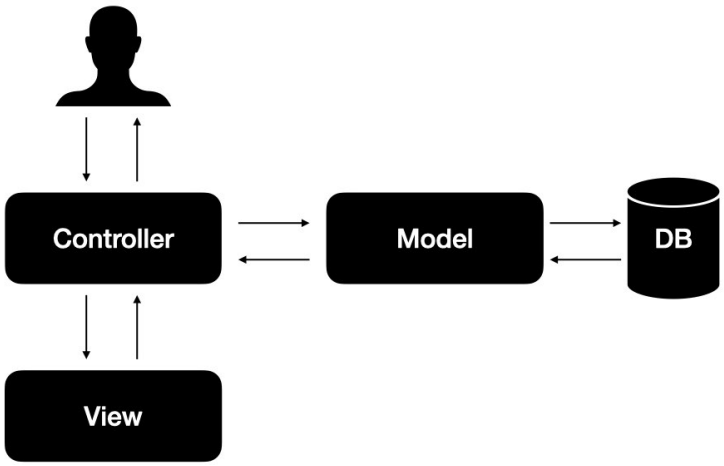

Abbildung 3: Darstellung des Model-View-Controller-Ansatz

2.1.6 Einordnung des aktuellen Systems von PluraPolit

Wie bereits in der Einleitung erwähnt, bietet PluraPolit eine Plattform an, die Jung- und Erstwählern bei der politischen Bildung hilft. Dafür wurde ein Softwaresystem entwickelt, in welches sich zum einen leicht neuer Content einpflegen lässt und zum anderen die hinterlegten Tonaufnahmen den Benutzern darstellt. Um dies zu erreichen, wurde neben der eigentlichen Plattform eine Content Management System (CMS) konzipiert. Umgesetzt wurde dies in einer Ruby on Rails Anwendung mit Zugriffsbeschränkung durch Passwortabfrage.

Ruby on Rails, kurz auch Rails bezeichnet, ist ein quellenoffenes Webframework, welches für die Programmiersprache Ruby entwickelt wurde (vgl. Hartl 2016, S.4; *Ruby on Rails* 2020; vgl. Tate 2011, S. 24). Es ist komplett Open Source und wird von einer aktiven Community entwickelt. Es gibt eine Vielzahl von kostenfreien Paketen, sogenannte Gems, die nach Belieben zu einem Projekt hinzugefügt werden können. Im Vergleich zu anderen Frameworks zeichnet sich Rails besonders durch seine Implementierung der REST-Architektur aus (vgl. Hartl 2016, S. 5). Diese Implementierung führt jedoch dazu, dass eine Vielzahl an Bedingungen an die Entwicklung und Implementierung von Rails Anwendungen gestellt werden. Getreu dem Motto *„Konvention vor Konfiguration“* nutzt Rails die Bestimmungen als Vorteil und integriert ein System, indem externe Pakete ohne Konfigurationsaufwand hinzugefügt werden können (*Ruby on Rails Doctrine* 2020).

Dabei orientiert sich Rails bei der Teilung der Dateien an dem Model-View-Controller-Ansatz (vgl. Hartl 2016, S. 66 ff.).

Für das Einpflegen von neuen Tonaufnahmen bedeutet dies konkret, dass anhand der Eingabe der Url der für das verwalten von Tonaufnahmen zuständige Controller die gewünschte Darstellung anzeigt. Darauf hin kann die Aufnahme als Datei hochgeladen werden. Dies geschieht, indem per Klick im CMS ein HTTP-Anfrage mit der geladenen Datei

an den Controller geschickt wird. Dieser kümmert sich anschließend darum, dass die Datei über das Model in der Datenbank gespeichert wird und gibt die Bestätigung der Aktion im View wieder (siehe Abb. 3).

Dabei wird die Tondatei selbst nicht in der Datenbank gespeichert. Vielmehr wird der Speicherservice von Amazon Web Services (S3) genutzt. Somit sendet, nachdem die Datei im View hinterlegt wurde, der Controller die Datei automatisch an S3 und erhält anschließend eine Url zurück. Diese wird anschließend abgespeichert und dient als Referenz zur eigentlichen Tonaufnahme (siehe Abbildung).

Neben dem Hochladen von Tonaufnahmen dient jedoch das CMS auch dazu bestehende Informationen zu bearbeiten und ggf. anzupassen. Im Detail bedeutet dies, dass die jeweilige Webseite per Url Aufruf angefragt wird, der Controller die angeforderten Daten über das Model aus der Datenbank lädt und die Entsprechende graphische Darstellung rendert. Dabei übergibt der Controller die Informationen an den View, der anschließend die Daten über Embedded RuBy in HTML integriert und anzeigt (siehe Abbildung).

Auch die Plattform ist in Ruby on Rails implementiert. Genauer beschrieben handelt es sich bei der Plattform um die selbe Applikation wie das CMS nur das die graphische Darstellung durch das JavaScript Framework React übernommen wird. Im Detail bedeutet dies, dass beim Aufruf der Plattform eine Anfrage an den Controller geschickt wird und dieser anschließend die kompilierte React Anwendung ausgibt. Diese lädt eigenständig jegliche Informationen über HTTP Anfragen und kümmert sich um interne Seitenaufrufe. Die Anfragen werden dabei asynchron an die Rails Applikation geschickt und vom Controller beantwortet. Somit erhält die React Anwendung über die Kommunikation von Controller und Modul, den Content aus der Datenbank, der vorher eingepflegt wurde (Siehe Abbildung).

Demnach gibt es eine Aufteilung hinsichtlich der graphischen Darstellung in Content Management System und Plattform. Die Datenspeicherung und Verwaltung ist jedoch gleich. Auch wird die gesamte Codebase in einem Bereitstellungsprozesses dem Hosting Service bereit gestellt. Somit handelt es sich beim System von PluraPolit um ein Monolith, welches teilweise Modular ist, die REST-Standards erfüllt und nach dem Model-View-Controller-Ansatz aufgeteilt ist. Des Weiteren nutz es vereinzelt externe Services von AWS, um Tondateien zu speichern und Transcripierungen vorzunehmen.

2.2 Microservicesarchitektur

Nachdem nun die einzelne Architekturstile vorgestellt wurden und das System von PluraPolit eingeordnet wurde, wird nun Microservicesarchitektur vorgestellt.

Dabei handelt es sich um eine Architekturstil, der ein Vertreter der verteilten Systeme

ist und sich historisch aus der Service Orientierten Architektur abgeleitet hat (siehe Abschnitt 1.2).

Eberhard Wolff beschreibt Microserivces als Ansatz Software in einzelne Module zu teilen und definiert es als Modularisierungskonzept, welches Einfluss auf die Unternehmesorganisation und Software-Entwicklungsprozess hat (vgl. Wolff 2018, Kap. 1.1). Dabei ist jedes Module ein eigenes Programm.

Sam Newman stützt die Aussage von Wolff und beschreibt Microservices als voneinander unabhängig einsetzbare Dienste, die um eine Geschäftsdomäne herum modelliert sind (vgl. Newman 2019, Kap. 2.1).

Somit beschreiben Beide Microservices als ein System aus einzelnen, unabhängigen Services, die sich an ein Geschäftsdomäne richten. Insbesondere die Abhängigkeit zum Geschäftsprozess wird im Abschnitt 2.2.3 näher beschrieben.

2.2.1 Kohärenz und Kopplung

Wenn es um das aufteilen von Software geht, ist es wichtig zu verstehen wie die einzelnen Funktionen und Klassen zusammenhängen und welche Verknüpfungen es zwischen ihnen gibt.

Dabei bezieht sich Kohärenz auf die funktionale Abhängigkeit zweier Funktionen oder Klassen (vgl. Newman 2019, Kap. 2.3.1). Somit liegt ein hoher Kohärenz vor, wenn der Quellcode anhand seiner logischen Zugehörigkeit geordnet ist. Eine konkrete Umsetzung dieses Bestreben ist im Abschnitt zum Model-View-Controller-Ansatz zu finden.

Kopplung hingegen beschreibt in welchem Maß, Funktionen und Klassen verbunden sind, ohne dass sie logisch zusammen gehören (vgl. Newman 2019, Kap. 2.3.2). Somit bezieht sich Kopplung ausschließlich auf eine technisch vorliegende Verknüpfung. Ein typisches Beispiel hier für ist, wenn ein Datenabruf an mehreren Stellen direkt über das Datenbankschema abläuft. Dadurch entsteht eine Abhängigkeit auf das Schema, sodass falls sich das Schema ändert auch die jeweiligen Aufrufe geändert werden müssen.

Insbesondere in monolithischen Systemen können viele Kopplungen entstehen, da keine festen Abgrenzungen zwischen einzelnen logischen Bereiche definiert sind. Somit kann es sein, dass ein Monolith, welches historisch wächst, keine klare Struktur aufzeigt. Es entsteht hieraus ein System, welches bei kleinen Veränderung viele ungewollte Fehler erzeugt, die ggf. weitere Anpassungen erfordern.

Für ein Microservicesarchtektur ist jedoch das Bestreben klare Abgrenzung zu erlangen, und einzelne stabile Services zu etablieren. Diese sollen soweit es geht von einander entkoppelt funktionieren. Somit ist das Ziel für eine Microservicearchitektur einen hohen

Kohärenz bei geringer Kopplung zu besitzen. Dabei ist dies nicht nur die Zielsetzung für Microservices allgemein, sondern ein generelles Bestreben für stabile Systeme.[13] Für Microservices bedeutet dies Konkret, dass ein Services ausschließlich aus funktional abhängigem Quellcode besteht und technische Implementierungen, wie zum Beispiel Datenbankstrukturen und Funktionsaufrufe, hinter klar definierten Schnittstellen versteckt sind.

2.2.2 Das Gesetz von Conway

Wenn es um die Teamaufteilung in IT Projekten geht, wird in den meisten Fällen sich an die historischen Herangehensweise orientiert und die Entwickler hinsichtlich ihrer Spezifikation eingeteilt. So werden Datenbankexperten, Frontend- und Backendentwicklern in jeweils einzelne Teams aufgegliedert.

Entscheidet sich jedoch ein Unternehmen für eine Microservicearchitektur, bedeutet diese Aufteilung, dass für jeden einzelnen Service Absprachen zwischen den Teams entstehen. Folglich etabliert sich eine Menge an einzelnen Kommunikationsbeziehungen, welche die Umsetzung von neuen Funktionen verlangsamt.

Der amerikanische Mathematiker Melvin Edward Conway umschrieb das Dilemma wie folgt: *„Organisationen, die Systeme designen, können nur solche entwerfen, welche die eigene Kommunikationsstruktur widerspiegelt.“* Er bezieht sich dabei auf die Tatsache, dass die Softwarearchitektur immer von den einzelnen Entwicklern mit entworfen wird und daher eingefahrene Kommunikationsstrukturen[14] sich in der Architektur wiederfinden lassen.

Nach Conway ist somit die Architektur direkt mit der Kommunikationsstruktur verbunden. Folglich sollte ein Unternehmen, welches das Gesamtsystem in einzelne Module teilt auch die gleiche Aufteilung in der Kommunikationsstruktur einführen. Konkret bedeutet dies für ein Unternehmen, dass die Entwickler nicht nach Spezifikation, sondern nach Kontext des Service geteilt werden. Diese Teams besitzen dabei möglichst kurze Kommunikationswege und haben wenig Abhängigkeiten zu anderen Teams. Im genaueren Definiert sich die Verknüpfung zu anderen Teams über die Schnittstellen der Services. Um die Kommunikation für ein Service auf das Minimum zu reduzieren, sollte nur ein Team pro Dient verantwortlich sein, wobei ein Team auch mehrere Services betreuen kann.

Um nun folglich eine Microservicesarchitektur umzusetzen, muss zum einem die Anwendung so Komplex sein, dass sie sich in einzelne Services teilen lässt und zum anderen so

[13] Dies bezieht sich auf das Gesetz von Constantine, welches besagt *„A structure is stable if cohesion is strong and coupling is low.“* (Endres und Rombach 2003, S. 43).

[14] Kommunikationsstrukturen ist dabei nicht zwingen gleich das Organigramm, da etabliere Kommunikationswege nicht zwingend von der Unternehmensstruktur abgebildet wurde.

viele Entwickler vorhanden sein, dass idealerweise jeder Dienst von einem Team betreut wird.

2.2.3 Domain Driven Design

Wie aus dem Abschnitt 2.2.2 hervorgeht, ist es das Ziel der Microservicesarchitektur möchglichst autonome Teams zu etablieren und unnötige Absprachen zu reduzieren. Um dies zu erreichen und gleichzeitig die hohe Kohärenz bei geringer Kopplung zu besitzen, bauen Microservices auf der Idee des Domain Driven Designs auf, welche von Eric Evans in seinem gleichnamigen Buch publiziert wurde (vgl. Newman 2019, Kap. 2.4). Es handelt sich dabei um eine Herangehensweise bzw. Denkweise zur Modellierung komplexer Systeme, bei der, dass Gesamtsystem anhand der einzelnen Geschäftsprozesse geteilt wird (vgl. Evans 2003, S. xix ff.).

Domain Driven Design ist dabei nicht exklusiv für Microserivces gedacht, sondern beschreibt vielmehr den Ansatz Softwareentwicklung und Geschäftsfunktion zu vereinen, welche in verschiedenen Architekturen umgesetzt werden kann. Nichtsdestoweniger ist Domain Driven Design wesentlich für die Bestimmung der Grenzen eines Services und somit entscheidet für Microservices (vgl. Wolff 2018, Kap. 4.3).

Ein Hauptbestandteil des Domain Driven Designs ist der Gedanke des Kontextes bzw. der Kontextgrenze (vgl. Fowler 2014). Eine Kontextgrenze ist dabei die Abgrenzung eines logischen Abschnittes (Kontext) innerhalb eines Unternehmens. Dadurch wird der Ansatz verfolgt, dass einzelne Funktionen und Inhalte nur innerhalb eines gewissen Bereiches Sinn ergeben. So ist zum Beispiel in einem E-Commerce Unternehmen bei der Bestellung von Ware die Anzahl, die Größe und das Gewicht entscheidend, während bei der Buchung Preis und Steuersatz wichtig sind. Des Weiteren kann je nach Kontext ein Begriff, wie Preis, unterschiedliche Bedeutungen haben (vgl. Evans 2003, S. 24 ff.). So bezieht sich dieser beim Bestellen von neuer Ware auf den Einkaufspreis, während er beim Verkaufen den Verkaufspreis meint. Somit hängt die Bedeutung des Wortes Preis vom dem zu lösenden Geschäftsprozess ab, in diesem Fall Bestellen von Ware oder Verkaufen von Produkten. Dies ist zwar nur ein einzelnes Beispiel, lässt sich jedoch auf Weitere übertragen. Demnach empfiehlt Domain Driven Design Kontextgrenzen so zu setzen, dass sie ein Geschäftsprozess umschließen und idealerweise genau ein spezifisches Problem lösen (vgl. Wolff 2018, Kap. 4.3).

Wie eingangs beschrieben, handelt es sich jedoch bei Domain Driven Design nur um eine Herangehensweise und nicht um klare Regeln. Daher liegt es im Ermessen jedes Unternehmens die Granularität der Kontexte zu setzen.

Der Fokus auf ein spezifische Problemstellung, ermöglicht jedoch eine einheitliche Sprache innerhalb eines Kontexts einzufügen, sowie eine enge Verbindung zwischen Geschäftslo-

gik und technische Entwicklung zu erzeugen. Somit vertritt Domain Driven Design den Ansatz, dass technische Modelle und Prozessbeschreibungen die gleiche Terminologie besitzen (vgl. Evans 2003, S. 24 ff.). Dadurch soll die Kommunikation zwischen Fachexperten und Softwareentwickler gestärkt werden.

Konkret für die Einteilung von Microservices bedeutet dies, dass ein Microservices ein Kontext abbildet, welche an ein Geschäftsprozess orientiert ist (vgl. Wolff 2018, Kap. 4.3; vgl. Newman 2019, Kap. 4).

Diese Eigenschaft deckt sich mit der von Sam Newman beschriebenden Definition, dass Microservices *„um eine Geschäftsdomäne herum modelliert sind“* (Abschnitt 2.2). Des Weiteren fordert Domain Driven Design, dass ein Team, welches für ein Service verantwortlich ist, auch Personen aus der Geschäftsabteilung hat und dass das Modell, durch ein fachlich diverses Team entsteht (vgl. Evans 2003, S. 32 ff.).

In der strengsten Umsetzung von Domain Driven Design, besteht folglich ein Team aus Fachexperten, sowie Softwareentwicklern und besitzt einen engen Kontakt zu Usern (vgl. Wolff 2018, Kap. 4.3).

Nun handelt es sich bei Domain Driven Design jedoch um eine Ansatz, welcher nicht immer absolut umgesetzt wird.

Die Umsetzung des Domain Driven Design hat wiederum einige Vorteile. So wird durch das abgrenzen in einzelne Kontexte, die Autonomität des Teams weiter bestärkt und der Austausch zwischen Fachexperten und Softwareentwicklern durch die einheitliche Sprache verstärkt. Dadurch entstehen eigenständige Services, die hinsichtlich ihrer verwendeten Daten, als auch der verwendeten Terminologie, entkoppelt sind. Eingeteilt wird anhand von Geschäftsprozessen.

2.3 Microservice

Wie aus dem vorhergehenden Abschnitt hervor geht sind Microservices einzelne, unabhängige Services, die um eine Geschäftsdomäne modelliert sind, welche auf Grundlage der klaren Schnittstellen entkoppelt und durch einzelne Kontext begrenzt sind. Die Kontexte ermöglichen das Verwenden von Service internen Terminologie und Verwendung ausschließlich benötigter Daten. Abschließend hat Abschnitt 2.2.2 gezeigt, dass nur ein Team für ein Service zuständig ist.

In seinem Buch *Monolith to Microservices* geht Newman darauf ein, dass wohl die wichtigste Eigenschaft von einem Microservice die unabhängige Einsetzbarkeit ist (vgl. Newman 2019, Kap. 2.1.1). Er sieht darin die Verkörperung von Unabhängigkeit und eigenständigen Teams, welche durch die Kommunikation der Services durch standardisierte Schnittstellen ermöglicht wird. Die meist verbreitete Art der Schnittstelle ist die nach

dem REST-Standard, welche sich an eine Kommunikation über HTTP richtet (siehe Abschnitt 2.1.4).

Da viele Programmiersprachen den Informationsaustausch über HTTP ermöglichen, ist ein Microservice unabhängig einer bestimmten Technologie. Somit kann jedes Team eigenständig entscheiden, welche Programmiersprache sie verwenden wollen (vgl. Wolff 2018, Kap. 1.2). Dadurch können Sprachen nach Präferenz und Problemstellen gewählt werden, ohne dass eine Inkompatibilität zum Rest des Systems entsteht.

Vielmehr ermöglicht der Austausch über standardisierten Kommunikationswege, dass Services erstellt werden, die austauschbar sind (vgl. Wolff 2018, Kap. 1.2). Somit können zum Beispiel einige Services durch Dienste von Drittanbieter ausgeführt werden und Entwicklungskosten gespart werden. Außerdem wird dadurch die Möglichkeit geschaffen auf bestehender Logik aufzubauen und ältere Systeme mit modernen Technologien zu verbinden.

Abschnitt 2.2.3 hat gezeigt das Kontextgrenzen Microservices es ermöglicht Daten nach eigenem ermessen zu benennen und zu verwalten. Konkret besitzt jeder Dienst eine eigene Datenspeicherung, in Form zum Beispiel eines File-Storage-Systems oder einer Datenbank (vgl. Newman 2019, Kap. 2.1.3). Für das Gesamtsystem bedeutet dies, dass keine einheitliche Datenspeicherung vorliegt, sondern jeder Service die notwendigen Informationen verwaltet. Um jedoch die Funktionalität des gesamten Systems zu erhalten, müssen im Vorfeld bestimmt werden, wie der Informationsaustausch zwischen den Services abläuft (vgl. Wolff 2018, Kap. 4.1). Hierfür ist es wichtig zu bestimmen, welche Services welche Informationen verwalten und falls Abhängigkeiten zu anderen Diensten bestehen, die Schnittstellen festzulegen. Insbesondere die genauen Endpunkt und die zu erwarteten Informationen müssen bestimmt werden. Die Handhabung innerhalb eines Services, ist dann wieder dem einzelnem Team überlassen.

Gibt es ein Service, der Informationen von einem anderen Dienst benötigt, fordert dieser die Daten über öffentliche Schnittstellen aktive an. Dies führt jedoch dazu, dass gleiche Informationen von mehrere Diensten verwaltet werden und es zu unterschieden Datenständen kommt (vgl. Wolff 2018, Kap. 4.1). Es entsteht die Herausforderung Informationen zwischen Services Konsistent zu halten.

Volle Unabhängigkeit eines Microservice kann nur erreicht werden, wenn dieser auch für die Benutzeroberfläche verantwortlich ist (vgl. Wolff 2018, Kap. 4.4). In der Praxis ist dies jedoch schwierig, da eine Webseite Inhalte von mehreren Services anzeigt. Insbesondere nachdem AngularJS[15] veröffentlicht wurde, gibt es immer mehr JavaScript Frameworks die um die Benutzererfahrung zu verbessern mehr Logik in das Frontend legen. React.js

[15] AngularJS ist ein JavaScript Framework, welches 2010 von Google entwickelt wurde und eine Open-Source-Software ist (*AngularJS — Superheroic JavaScript MVW Framework* 2020).

ist eines dieser Frameworks. Die Beliebtheit hinter diesen Frameworks, liegt darin, dass jede einzelne Webseite nicht mehr als einzelne Seite betrachtet wird, sondern Informationen zwischen Benutzeroberflächen hinweg verwaltet werden (vgl. Wolff 2018, Kap. 9.1). Dadurch kann ein Datenabruf über mehrere Webseiten geteilt werden und somit die Performance verbessern. Erreicht wird dies, indem das Routing zwischen den Webseiten durch das Framework verwaltet wird. Dieser Ansatz wird als Single-Page-Application (SPA) bezeichnet und unterscheidet sich von der Idee des Model-View-Controller aus Abschnitt 2.1.3, da die nötigen Informationen aktive von der View geladen werden (*Single-Page-Webanwendung* 2019).

Da in der Praxis viele Webseiten Single-Page-Applicationen einsetzen, um eine gute Benutzererfahrung zu liefern, sollte in diesen Fällen über mögliche Teilung nachgedacht werden. Insbesondere wenn es sich um Applikationen handelt, die von mehrere Teams verwaltet werden. Ist dies nämlich der Fall, kann darunter die Unabhängigkeit der Teams leiden.

Auch kann es vorkommen, dass ein Microservices keine graphische Darstellung benötigt, da dieser zum Beispiel nur E-Mails verschickt oder die beliebtesten Filme ermittelt. Somit gibt es keine absolute Aussage darüber, ob ein Microservice eine Benutzeroberfläche haben sollte.

Insbesondere in puncto Skalierung bieten Microservices viele Vorteile (vgl. Newman 2019, Kap. 2.1.4). So lassen sich individuell, intensiv genutzte Services unabhängig vom Gesamtsystem skalieren. Dies kann ein großer Kostenvorteil sein, da nur ein einzelner Teil und nicht das gesamte System mehr Ressourcen zu geteilt werden muss. Auch können auf Basis der eingeteilten Teams weitere Entwickler angestellt werden und das Unternehmen hinsichtlich der Angestellten wachsen.

Während Microservices Skalierungen begünstigen, erschweren sie serviceübergreifende Veränderungen (vgl. Newman 2019, Kap. 2.15). Somit ist es auf Grund der technologischen Freiheit aufwendiger, Entwickler von einem Service zu einem Anderen zu überführen. Dies wird dadurch verstärkt, dass unterschiedliche Daten gespeichert und verschiedene Terminologien verwendet werden. Auch eine technologische Überführung von Funktionen wird erschwert, wenn verschiedene Programmiersprachen eingesetzt werden. Äquivalent ist das zusammenführen von Datenschemata, welches durch die unterschiedlichen Attributbezeichnungen boykottiert wird. Dies führt dazu, dass ein Zusammenschluss nur durch einen großen Aufwand erreicht werden kann.

2.4 Kommunikation von Services

Aus Abschnitt 2.2 geht hervor, dass eine Microservice-Architektur aus einzelnen, unabhängigen Services besteht, welche voneinander entkoppelt sind. Um jedoch gleichzeitig die Funktionalität des Gesamtsystems zu gewährleisten, ist es notwendig, dass die einzelnen Services untereinander im Austausch stehen. Demzufolge stellt die Kommunikation eine Grundvoraussetzung für das Implementieren einer Microservice-Architektur dar und setzt voraus, dass ein Kommunikationsfähiges Netzwerk vorhanden ist.

Im Vergleich zu einem monolithen System hat jedoch der Austausch von einzelnen Services über ein Netzwerk einige Nachteile. So ist es Aufwendiger ein Systems aus einzelnen Komponenten richtig abzusichern, als ein einzelnen Rechner vor Angriffen zu schützen. Des Weiteren ist jede Schnittstelle, die öffentlich dem Netzwerk zur verfügung steht ein Risiko für potentielle Angriffe. Demzufolge erhöht die Umsetzung einer Microservice-Architektur die Gefahr für ein Angriff und ist gleichermaßen aufwendiger zu sichern. Anschließend ist der Aufruf über das Netzwerk langsamer und mit geringerer Bandbreite als ein Aufruf innerhalb eines Prozesses (vgl. Wolff 2018, Kap. 6.1).

Auch wenn bis lang nur auf die Kommunikation über den REST-Standard in dieser Ausarbeitung eingegangen wurde, ist es nicht die einzige Möglichkeit Schnittstellen zwischen Services zu definieren. So gibt es neben REST noch weitere Arten der Kommunikation, wie zum Beispiel GraphQL (*GraphQL* 2020) und gRPC (*gRPC* 2020). Beides sind Ansätze, welche in den letzten Fünf Jahren entwickelt wurden und deutlich andere Schwerpunkte setzen.

Im Unterschied zu REST muss bei GraphQL nicht für jeden Individuellen Client eine extra Schnittstelle geschrieben werden, sondern der Client gibt beim Aufruf die Informationen mit, welche er erhalten möchte (vgl. *GraphQL* 2020). Dabei wird im Vorfeld über ein Schema festgelegt, welche Informationen der Server zur Verfügung stellt. Auch können über GraphQL nicht nur Informationen von einer Relation (Tabelle) abgerufen werden, sondern über Verbindungen zwischen Relationen, auf Daten von anderen Tabellen zugegriffen werden. Auch hier wird die Spezifikation der Inhalte vom Client beim Aufrufen der Schnittstelle mitgegeben. Folglich entstehen wenige Programmschnittstellen, die jedoch eine Vielzahl Anforderungen bedienen können. Dies ist insbesondere für Aggregierte Informationen aus mehreren Tabellen wichtig, sowie für Systeme, in denen mehrere Clients unterschiedliche Informationen vom selben Service abrufen.

gRPC auf der anderen Seite ermöglicht es einem Client, Funktionen eines Servers übers Netzwerk aufzurufen, als währen sie in der gleichen Codebase. Hierbei ist gRPC Programmiersprachen unabhängig, sodass ein in Java geschriebener Client auf ein Python-Server zugreifen kann. Während REST explizit für Web-Anwendungen erstellt wurde, wurde gRPC speziell für den Austausch unter Services entwickelt. So werden zum Beispiel keine

Statuscodes oder andere Meta-Daten verschickt, sodass die Geschwindigkeit im Vergleich zum REST-Standard schneller ist. Des Weiteren ermöglicht gRPC unteranderem das Monitoren der Kommunikation zwischen Services, um auftretende Fehler schnell erkennen zu können. Auch nutzt gRPC den moderneren HTTP/2 Standard, durch welchen der Datenaustausch beschleunigt und optimiert wird.

Auch wenn 2003 Peter Rodgers Microservice-Architektur als ein System aus einzelnen REST-Services beschrieben hat, haben sich seitdem neue Technologien zur Kommunikation zwischen Services entwickelt. Insbesondere gRPC besitzt auf Grund der moderneren Technik bedeutende Vorteile hinsichtlich der Geschwindigkeit. Nichtsdestoweniger ist die Kommunikation zwischen entkoppelten Services langsamer, als Aufrufe innerhalb eines Prozesses und bringen einige Nachteile mit sich.

Demnach muss neben der Auswahl des Kommunikationsstandards ein Netzwerk vorliegen, welches einen hohen Durchsatz, als auch eine hohe Geschwindigkeit ermöglicht und vor Angriffe gesichert werden kann.

2.5 Start-up

Obwohl im Zentrum dieser Arbeit die notwendigen Bedingungen für eine Microservicearchitektur stehen, lässt sich das volle Maß der Umstellung nur verstehen, wenn auch die Situation eines Start-up verstanden wird. Insbesondere Abschnitt 2.2.3 hat gezeigt, dass Software Architektur und Geschäftsprozesse eng mit einander verbunden sind. Demnach sollten *„hart zu verändernde Entscheidungen“*[16] stets im Kontext der Unternehmenssituation getroffen werden. Demzufolge wird anschließend der Begriff Start-up definiert und einige Eigenschaften näher beschrieben.

Nach dem Wirtschaftslexikon Gabler ist ein Start-up, oder auch Start-up-Unternehmen, ein junges, noch nicht etabliertes Unternehmen, welches eine innovative Geschäftsidee verwirklichen möchte (vgl. Achleitner 2018). Folglich agiert ein Start-up in einem jungen oder noch nicht existierenden Markt und muss erst ein funktionierendes Geschäftsmodell entwickeln. Wurde ein solches gefunden und implementiert, gilt das Unternehmen allgemein nicht mehr als Start-up (vgl. *Start-up-Unternehmen* 2020).

Ein funktionierendes Geschäftsmodell zu finden, ist jedoch nicht trivial und eine Mehrheit der Start-ups melden nach nur wenigen Jahren Insolvenz an. Konkret scheitern neun von zehn Unternehmen innerhalb der ersten drei Jahre (vgl. Patel 2015).

Erfolgreiche Start-ups zeichnen sich daher durch adaptives Verhalten aus. So änderten Zweidrittel der erfolgreichen Start-ups drastisch ihre ursprüngliche Geschäftsidee (Mullins und Komisar 2009). Demnach lässt sich einen signifikanter Zusammenhang, zwischen

[16] Direkter Bezug zur Definition von Martin Fowler aus Abschnitt 2.1.1.

Erfolg und Flexibilität erkennen. Um erfolgreich mit einem Start-up zu sein, muss dieses also dynamisch und fokussiert arbeiten und schnell neue Erkenntnisse über die Bedürfnisse von Kunden sammeln.

Wie sich aus der Definition ableitet, ist es das Ziel eines Start-ups ein Produkt zu entwickeln, welches einen lukrativen Absatzmarkt besitzt. Demnach versucht jedes junge Unternehmen für den Kunden einen Mehrwert zu generieren, sodass er für diesen Geld bezahlt.

Um dies zu erreichen durchläuft ein Start-up drei gesonderte Phasen (vgl. Maurya 2012, S. 8 f.):

1. Problem-Lösung Fit
2. Produkt-Market Fit
3. Skalierung

In der ersten Phase (Problem-Lösung Fit) wird ermittelt, ob ein Problem vorliegt, welches Wert ist, gelöst zu werden. Um dies zu beantworten werden qualitative Kundenbefragungen und Kundeninterviews durchgeführt (vgl. Croll und Yoskovitz 2013, S. 170 ff.).

Anschließend wird in der zweiten Phase (Produkt-Market Fit) aus den Erkenntnissen der ersten Phase der minimale Funktionsumfang bestimmt und umgesetzt. Dieses sogenannte Minimum Viable Product (MVP) wird daraufhin für Anwendertests verwendet. Die Ergebnisse aus den Tests, werden abermals zur Anpassung des MVPs genommen, welches für weitere Kundentests eingesetzt wird. Dieser Prozess wird solange wiederholt bis ein Produkt entwickelt wurde, welches das ursprüngliche Problem löst, oder Geld und Ressourcen ausgehen (vgl. Croll und Yoskovitz 2013, S. 28).

In Phase Drei (Skalierung) wird das Start-up hinsichtlich Technologie, Struktur und Produkt angepasst, dass es hinsichtlich Kundenbedarf, Personal und Funktionsumfang ideal wachsen kann (vgl. Maurya 2012, S. 9).

Insbesondere Phase Eins und Zwei zeichnen sich durch schnelles fokussiertes Lernen aus, welches in kontinuierlichen Anpassungen des Geschäftsmodells endet. Folglich kommt es in diesen beiden Phasen zu ständigen Änderungen des Produkts und Geschäftsprozesses. Erst nach dem erreichen des Produkt-Market Fits stabilisiert sich das Produkt und der Fokus wird auf Skalierung gelegt.

Im Gegensatz zu den meisten Start-ups, ist PluraPolit ein gemeinnütziges Unternehmen, sodass der Fokus nicht auf dem erwirtschaften von Gewinn liegt, sondern auf der Maximierung der Kundeninteraktion. Da Interaktion sich ebenfalls aus der Akzeptanz der Anwender ergibt, ist PluraPolit ebenfalls daran bestrebt ein Produkt-Market Fit zu erreichen. Zum aktuellen Zeitpunkt befindet es sich jedoch noch in Phase Zwei.

2.6 Bedingungen ableiten

Nachdem in den vorangegangen Abschnitten ein Verständnis von Microservicearchitektur, sowie Software Architektur im allgemeinen geschaffen wurde, werden nun die Erkenntnisse in Bedingungen zusammen getragen. Bei den Bedingungen handelt es sich um Kernaussagen aus den einzelnen Abschnitte und dienen als Leitfaden für die Entscheidung von PluraPolit. Im nächsten Gliederungspunkt werden diese von Experten qualitativ bewertet.

Wie Abschnitt 2.2.2 gezeigt hat, ist das Ziel einer Microservice-Architektur, dass Teams möglichst unabhängig von einander Arbeiten können. Erreicht wird dies, indem die Verantwortung eines Services nur einem Team gegeben wird und Absprachen zwischen Teams sich auf die Festlegung der Schnittstellen begrenzen. Daraus ergeben sich zwei Bedingungen an das Unternehmen:

1. Es ist möglich die Verantwortung für ein Geschäftsprozess an ein Team zu geben und
2. Die Services greifen ausschließlich auf Ressourcen zu, die über die Schnittstellen erreicht werden können.

Anschließend zeigte der Abschnitt über Domain Driven Design, dass Microservices die Komplexität eines Unternehmens reduziert, indem die Geschäftsprozesse in Services geteilt werden. Dieser Ansatz der Trennung Anhang der Geschäftsdomainen fördert die Fähigkeit, dass Services unabhängig von einander wachsen können und gleichzeitig eine klare Verantwortung besteht. Es ergeben sich die Bedingungen, dass ein System vorliegt:

1. welches zum einen eine gewisse Komplexität aufweist und
2. zum anderen Geschäftsabläufe besitzt, die sich trennen lassen.

Aus Abschnitt 2.4 wird klar, das neben der Einteilung in einzelne Komponenten auch Bedingungen an das Netzwerk gestellt werden. So ergeben sich hinsichtlich des Netzwerks folgende Anforderungen:

1. Es muss möglich sein, dass Services untereinander kommunizieren können,
2. Zugriffen von Unbefugten verhindert werden und
3. Eine möglichst hohe Datengeschwindigkeit vorliegen.

Weiterführend setzt Abschnitt 2.4 voraus, dass die Services Programmierschnittstellen aufweisen, die Server zu Server Kommunikation erlauben.

Neben der Betrachtungen hinsichtlich der Umsetzung einer Microservice-Architektur hat Abschnitt 2.5 die Situation eines Start-ups betrachtet. Es wurde deutlich, dass ein Start-up in einem noch nicht existierenden Markt operiert und dadurch besonders dynamisch

agieren muss. Daraus ergibt sich die Anforderung, dass erst eine Microservice-Architektur umgesetzt werden kann, wenn ein Produkt-Market Fit vorliegt.

Zusammenfassend ergeben sich aus der Literaturrecherche folgende neun Bedingungen:

1. Es ist Möglich die Verantwortung für ein Geschäftsprozess an ein eigenständiges Team zu geben.
2. Die Services greifen ausschließlich auf Ressourcen zu, die über die Schnittstellen erreicht werden können.
3. Das System weist eine gewisse Komplexität auf.
4. Die Geschäftsabläufe lassen sich von einander trennen.
5. Das Netzwerk ermöglicht die Kommunikation zwischen den Services.
6. Der Zugriff aufs Netzwerk ist vor Unbefugten gesichert.
7. Das Netzwerk hat ein hohen Datendurchsatz.
8. Die Services verfügen über Schnittstellen, die Server zu Server Kommunikation ermöglichen.
9. Es liegt ein Produkt-Market Fit vor.

3 Methodik

Um herauszufinden, ob die Bedingungen aus der Literaturrecherche notwendig sind, bevor PluraPolit ihre Software Architektur ändern kann, wurde eine Experteninterview durchgeführt. Es wurde sich für diese Methode entschieden, um in Ermangelung der wenigen Literatur eine qualitative Einschätzung zu bekommen und abschließend eine Empfehlung für PluraPolit zu geben. Weiterführend wurde sich für ein semistrukturiertes Interview entschieden, um zum einen subjektive, als auch vergleichbare Erkenntnisse zu erhalten. Anschließend werden die Erkenntnisse in Abschnitt ... diskutiert.

3.1 Durchführung des Interviews

3.2 Auswahl der Experten

3.3 Erstellung der Interviewfragen

4 Auswertung der Interviews und Anwendung auf PluraPolit

- Public subscribe pattern
- SOA

Da die Implementierung einer Microservicearchitektur, serviceübergreifende Veränderungen erschwert und diese nur durch hohen Aufwand erreicht werden kann, sollte gewährleistet sein, dass kaum Änderungen dieser Art auftreten. Für die Einteilung der Services bedeutet dies, dass Kontextgrenzen so gesetzt werden, dass Veränderungen, die mehrere Services umfassen, ausgeschlossen sind. Wie Abschnitt ... zeigt, orientieren sich diese Grenzen an den Geschäftsprozessen, sodass folglich eine Stabilität des Geschäftsabschlaufs Grundlage für eine statische Einteilung der Services ist.

5 Fazit

Literaturverzeichnis

Achleitner, Prof Dr Dr Ann-Kristin. *Definition: Start-up-Unternehmen*. https://wirtschaftslexikon.gab unternehmen-42136. Library Catalog: wirtschaftslexikon.gabler.de Publisher: Springer Fachmedien Wiesbaden GmbH Section: economy. 14. Feb. 2018. URL: https://wirtschaftslexiko gabler.de/definition/start-unternehmen-42136/version-265490 (besucht am 14. 06. 2020).

Amazon Web Services (AWS) - Cloud Computing Services. Amazon Web Services, Inc. Library Catalog: aws.amazon.com. URL: https://aws.amazon.com/ (besucht am 27. 04. 2020).

AngularJS — Superheroic JavaScript MVW Framework. URL: https://angularjs.org/ (besucht am 12. 06. 2020).

Asynchrone Kommunikation. In: *Wikipedia*. Page Version ID: 187692564. 18. Apr. 2019. URL: https://de.wikipedia.org/w/index.php?title=Asynchrone_Kommunikation&oldid=187692564 (besucht am 16. 05. 2020).

AWS | Amazon EC2 Container & Konfigurationsmanagement. Amazon Web Services, Inc. Library Catalog: aws.amazon.com. URL: https://aws.amazon.com/de/ecs/ (besucht am 27. 04. 2020).

AWS Fargate – Container ausführen, ohne Server oder Cluster zu verwalten. Amazon Web Services, Inc. Library Catalog: aws.amazon.com. URL: https://aws.amazon.com/de/fargate/ (besucht am 27. 04. 2020).

AWS Lambda Data Processing - Datenverarbeitungsdienste. Amazon Web Services, Inc. Library Catalog: aws.amazon.com. URL: https://aws.amazon.com/de/lambda/ (besucht am 28. 04. 2020).

Balzert, Helmut. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Google-Books-ID: UqYuBAAAQBAJ. Springer-Verlag, 13. Sep. 2011. 593 S. ISBN: 978-3-8274-2246-0.

Baron, Joe, Hisham Baz und Tim Bixler. *AWS Certified Solutions Architect Official Study Guide: Associate Exam (Aws Certified Solutions Architect Official: Associate Exam)*. 1. Aufl. Sybex, 2016. 435 S. ISBN: 978-1-119-13855-6. URL: https://www.amazon.com/gp/offer-listing/1119138558/?tag=wwwcampusboocom587-20&condition=used (besucht am 27. 04. 2020).

Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (eBook)*. 2 edition. Addison-Wesley Professional, 2. Aug. 1995. 336 S.

Buxton, JN und B Randell. „Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, October 1969“. In: NATO Science Committee, 1970.

Clements, Paul. *Comparing the SEI's Views and Beyond Approach for Documenting Software Architecture with ANSI-IEEE 1471-2000:* Fort Belvoir, VA: Defense Technical In-

formation Center, 1. Juli 2005. DOI: 10.21236/ADA441291. URL: http://www.dtic.mil/docs/citations/ADA441291 (besucht am 25. 05. 2020).

Clements, Paul u. a. *Documenting Software Architectures: Views and Beyond*. 2 edition. Addison-Wesley Professional, 5. Okt. 2010. 592 S.

Croll, Alistair und Benjamin Yoskovitz. *Lean Analytics: Use Data to Build a Better Startup Faster*. 1 edition. O'Reilly Media, 8. März 2013. 556 S. ISBN: 978-1-4493-3567-0.

Dragoni, Nicola u. a. „Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Hrsg. von Manuel Mazzara und Bertrand Meyer. Cham: Springer International Publishing, 2017, S. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12 (besucht am 23. 04. 2020).

Duden | Monolith | Rechtschreibung, Bedeutung, Definition, Herkunft. In: Library Catalog: www.duden.de. URL: https://www.duden.de/rechtschreibung/Monolith (besucht am 16. 05. 2020).

DWDS – Digitales Wörterbuch der deutschen Sprache. In: *DWDS*. Library Catalog: www.dwds.de. URL: https://www.dwds.de/wb/Monolith (besucht am 16. 05. 2020).

Endres, Albert und Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories: A Handbook of Observations, Laws and Theories*. 1. Aufl. Addison Wesley Pub Co Inc, 13. März 2003. 352 S. ISBN: 978-0-321-15420-0.

Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1 edition. Addison-Wesley Professional, 22. Aug. 2003. 563 S. ISBN: 978-0-321-12521-7.

Features • GitHub Actions. GitHub. Library Catalog: github.com. URL: https://github.com/features/actions (besucht am 27. 04. 2020).

Fielding, Roy Thomas. „Architectural Styles and the Design of Network-based Software Architectures". Diss. UNIVERSITY OF CALIFORNIA, 2000. 180 S. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

Fowler, Martin. *Martin Fowler: BoundedContext*. martinfowler.com. Library Catalog: martinfowler.com. 15. Jan. 2014. URL: https://martinfowler.com/bliki/BoundedContext.html (besucht am 13. 06. 2020).

– *Who needs an Architect?* design. Aug. 2003. URL: https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf.

GraphQL: A query language for APIs. Library Catalog: graphql.org. URL: http://graphql.org/ (besucht am 15. 06. 2020).

gRPC. gRPC. Library Catalog: grpc.io. URL: https://grpc.io/ (besucht am 15. 06. 2020).

Hartl, Michael. *Ruby on Rails Tutorial: Learn Web Development with Rails (eBook)*. 4 edition. Addison-Wesley Professional, 17. Nov. 2016. 806 S.

Hilliard, Rich. *ISO/IEC/IEEE 42010 Homepage*. ISO/IEC/IEEE 42010 Homepage. Library Catalog: www.iso-architecture.org. URL: http://www.iso-architecture.org/42010/ (besucht am 06. 05. 2020).

Leach, Paul J. u. a. *Hypertext Transfer Protocol – HTTP/1.1*. Library Catalog: tools.ietf.org. 12. Mai 2020. URL: https://tools.ietf.org/html/rfc2616 (besucht am 12. 05. 2020).

Maurya, Ash. *Running Lean: Iterate from Plan A to a Plan That Works*. 2 edition. O'Reilly Media, 24. Feb. 2012. 240 S. ISBN: 978-1-4493-0517-8.

Mullins, John und Randy Komisar. *Getting to Plan B: Breaking Through to a Better Business Model*. Harvard Business Review Press, 8. Sep. 2009. 272 S.

Newman, Sam. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith (eBook)*. 1 edition. O'Reilly Media, 14. Nov. 2019. 272 S.

Node.js. *Node.js*. Node.js. Library Catalog: nodejs.org. URL: https://nodejs.org/en/ (besucht am 28. 04. 2020).

Patel, Neil. „90% Of Startups Fail: Here's What You Need To Know About The 10%". In: *Forbes* (16. Jan. 2015). Library Catalog: www.forbes.com Section: Entrepreneurs. URL: https://www.forbes.com/sites/neilpatel/2015/01/16/90-of-startups-will-fail-heres-what-you-need-to-know-about-the-10/ (besucht am 14. 06. 2020).

PostgreSQL: The world's most advanced open source database. URL: https://www.postgresql.org/ (besucht am 27. 04. 2020).

React – Eine JavaScript Bibliothek zum Erstellen von Benutzeroberflächen. Library Catalog: de.reactjs.org. URL: https://de.reactjs.org/ (besucht am 27. 04. 2020).

Rodgers, Peter. *Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity | CloudEXPO*. Mai 2018. URL: https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883 (besucht am 23. 04. 2020).

Ruby on Rails. Ruby on Rails. Library Catalog: rubyonrails.org. URL: https://rubyonrails.org/ (besucht am 27. 04. 2020).

Ruby on Rails Doctrine. Ruby on Rails. Library Catalog: rubyonrails.org. URL: https://rubyonrails.org/doctrine/ (besucht am 18. 05. 2020).

Single-Page-Webanwendung. In: *Wikipedia*. Page Version ID: 185777481. 17. Feb. 2019. URL: https://de.wikipedia.org/w/index.php?title=Single-Page-Webanwendung&oldid=185777481 (besucht am 12. 06. 2020).

Starke, Gernot. *Effektive Softwarearchitekturen (eBook)*. Carl Hanser Verlag GmbH & Co. KG, 7. Juli 2015. 458 S. ISBN: 978-3-446-44361-7. DOI: 10.3139/9783446444065. URL: https://www.hanser-elibrary.com/doi/book/10.3139/9783446444065 (besucht am 06. 05. 2020).

Start-up-Unternehmen. In: *Wikipedia*. Page Version ID: 197628824. 10. März 2020. URL: https://de.wikipedia.org/w/index.php?title=Start-up-Unternehmen&oldid=197628824 (besucht am 14. 06. 2020).

Stephens, Rod. *Beginning Software Engineering*. Google-Books-ID: SyHWBgAAQBAJ. John Wiley & Sons, 2. März 2015. 482 S. ISBN: 978-1-118-96916-8.

Synchrone Kommunikation. In: *Wikipedia*. Page Version ID: 182185824. 27. Okt. 2018. URL: https://de.wikipedia.org/w/index.php?title=Synchrone_Kommunikation&oldid=182185824 (besucht am 16. 05. 2020).

Tanenbaum, Andrew S. und Maarten van Steen. *Verteilte Systeme: Prinzipien und Paradigmen*. 2 edition. Pearson Studium, 1. Nov. 2007. 760 S.

Tate, Bruce A. *Sieben Wochen, sieben Sprachen*. 1 edition. Beijing: O'Reilly Verlag GmbH & Co. KG, 1. Juni 2011. 360 S. ISBN: 978-3-89721-322-7.

What is a Container? | App Containerization | Docker. Library Catalog: www.docker.com. URL: https://www.docker.com/resources/what-container (besucht am 27. 04. 2020).

Wolff, Eberhard. *Microservices: Grundlagen flexibler Softwarearchitekturen (eBook)*. 2 edition. dpunkt.verlag, 25. Juli 2018. 384 S.

Abbildungsverzeichnis

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift