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[1] im Backend und React.js[2]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[3](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[4] gespeichert und per Github Actions[5] an AWS geliefert. Dort wird die Applikation in das Elastic Container Service[6] (ECS) geladen und von Fargate[7] verwaltet. Die Daten werden in einer PostgreSQL[8] Datenbank abgespeichert, die auf einer Relational Database Service[9] (RDS) Instanz hinterlegt ist. Bilder und Tonaufnahmen werden in einem Simple Storage Service-Bucket[10] (S3) gespeichert und stehen der Webseite per URL zur Verfügung. Um automatisch zu jedem Beitrag Intros zu generieren, wurde eine AWS Lambda[11] Funktion geschrieben, die aus der Basis von Beschriftungstexten für

[1] Ruby on Rails ist ein quellenoffenes Webframework, welches für die Programmiersprache Ruby entwickelt wurde (*Ruby on Rails* 2020; vgl. Tate 2011, S. 24).

[2] 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).

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

[4] 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).

[5] 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).

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

[7] 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).

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

[9] 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.).

[10] 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).

[11] 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).

jede Audioaufzeichnung eine weitere Aufnahme für die Einleitung erstellt.

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 Plura-Polit 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

Der Abschnitt umfasst ca. 16 Seiten.

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

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 Bücher *Software Architecture in Practice und Documenting Software Architectures: Views and Beyond*, 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).

Somit definieren beide Softwarearchitektur als Strukturierung von einzelnen Komponenten, die untereinander in Beziehung stehen. 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 (vlg. 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. Davon getrennt sind die graphischen Darstellungen, welche durch Views definiert

werden. Sie erhalten ihre Informationen vom Model. Das Verwalten der Benutzereingabe, sowie das weiterleiten zwischen einzelnen Views geschieht über den Controller. 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. Abschließend updated er die erforderliche Darstellung.

Beim Model-View-Controller Ansatz handelt es sich um ein Muster, welches oft in der Softwarearchitektur verwendet 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 Architekturstile 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 fürs Web.

Dabei steht REST für *Representationl State Transfer*, welches ein 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).[12]

Ein Server ist dabei eine Komponente im Netzwerk, welches Services anbietet. Ein Service könnte 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.

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 gestellt hat, 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

[12]Hier kommt ein Text zur Einteilung des Begriffes Server-Client Architektur.

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 Server, 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 verwendeten 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 individuelle 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 ein Kommunikationsstandard auf, der sich im Internet etabliert hat.

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

Verwendet wird die REST-Architektur ausschließlich für Anwendungen im Internet, da es auf die Anwendung des Hypertext Transfer Protokoll[13] (HTTP) angewiesen ist. Dabei findet der Architekturstil, sowohl Anwendung für ganze Systeme, als auch in komplexen Anwendungen mit einer Vielzahl ein einzelnen 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 da mit 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 auf die Verbindung

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

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 vor, wenn die gesamte Funktionalität eines Systems gemeinsam über ein Deployment-Prozess bereitgestellt wird (vgl. Newman 2019, Kap. 2.2). Somit 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 ein Prozess abbildet. Dies bedeutet, dass jegliche Funktionalität auf einander aufbauend ist und nur eine Datenspeicherung für die gesamte Anwendung verwendet wird (vgl. Newman 2019, Kap. 2.2.1). 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 von monolithen System ist jedoch seltener und wird nur von einzelnen 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 (vgl. Newman 2019, Kap. 2.2.2). Unterschiedlich davon sind verteilte Monolithe. 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, da sowohl Nachteile auf Grund der Verteilung, als auch durch das gemeinsame Bereitstellen, entstehen.

Weder Stephens, als auch Newman geben Vorgaben hinsichtlich der Gliederung innerhalb eines Monolithischen Systems. Demnach kann eine Model-View-Controller Ansatz als Monolithisches System gelten, solange es einheitlich deployed wird. Anders ist es mit einem verteilten System, da nach Definition ein Monolithen System kein verteiltes System sein kann.

Im Rahmen dieser Arbeit wird bei jeglichen weiteren Referieren auf den Begriff Monolithen System stets von einem Einzelprozess Monolithen ausgegangen, außer es wird expliziert von einem Modularen, oder verteilten Monolithen geschrieben. Dadurch sollen beide Definitionen berücksichtig werden.

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 Monolithen System einfacher, da es ein Bereitstellungsprozess für die gesamte Anwendung gibt. Wiederum führt

dies dazu, dass der Prozess deutlich länger dauert. Diese Tatsache ist insbesondere 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 Monolithen System dazu, dass bei vielen 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 Monolithen System begünstigt und können im Vergleich zu einem verteilten System einfacher umgesetzt werden (vgl. Newman 2019, Kap. 2.2.5).

2.1.6 Einordnung des aktuellen Systems von PluraPolit

2.2 Microservices

Der Begriff Microservices wird mehrdeutig verwendet. So wird je nach Perspektive entweder eine Softwarearchitektur oder eine Komponente einer solchen Architektur beschrieben. Eng verbunden mit diesem Begriff sind auch dynamische Systeme und Konzeptionierung von Unternehmensstrukturen. Demnach sieht Eberhard Wolf unter Microservices ein Modellierungskonzept, welches dazu dient größere Softwaresysteme in kleinere Einheiten zu teilen (vgl. Wolff 2018, Kap. 1.1). Dabei hat die Aufteilung Auswirkungen auf die Organisation als auch auf die Entwicklungsprozesse.

Nach Sam Newman ist ein Microservice ein *„eigenständige ausführbare Softwarekomponente, die innerhalb eines Anwendungssystems mit anderen Softwarekomponenten kollaboriert“* (Newman 2019, Kap. 2.1). Sie zeichnet sich durch das kommunizieren über definierte Netzwerkschnittstellen aus und formt in Vereinigungen eine Microservices-Architektur. Ein Microservice umfasst dabei die Datenspeicherung, Datenverarbeitung und Datendarstellung und besitzt eine gut definierte Benutzeroberfläche (vgl. Newman 2019, Kap. 2.1).

Ergänzend dazu schreibt Wolf, dass sich das Konzept der Microservices-Architektur aus der Philosophie vom Unix Betriebssystem ableitet, welches nach Peter H. Salus folgende drei Leitpunkte umfasst (Salus 1994; vgl. Wolff 2018, Kap. 1.1):

- Schreibe Programme, sodass sie nur eine Aufgabe erledigen und diese gut.

- Schreibe Programme, die zusammen arbeiten.
- Schreibe Programme, welche über definierte Schnittstellen (Textstream) kommunizieren.

Nach James Lewis sind Microservices kleine Anwendungen, die unabhängig bereitgestellt, getestet und skaliert werden. Ebenfalls wie Wolf beschreibt er diese Programme, als einfach zu verstehen, die nur eine Aufgabe übernehmen.

Zusammenfassend lässt sich sagen, dass der Begriff Microservices nicht einheitlich definiert ist und es sich zwei unterschiedliche Perspektiven ergeben. Zum einen beschreibt es ein Modellierungskonzept, welches Auswirkungen auf Unternehmenssturkur und Managemententscheidungen hat und zum anderen kennzeichnet es eine Software-Architektur, die in eigenständige Komponente geteilt ist.

In dieser Arbeit werden beiden Ansichten beleuchtet, um infolgedessen eine umfassende Einschätzung für PluraPolit zu geben.

2.3 Microservices als Modellierungskonzept

In diesem Abschnitt möchte ich mehr auf die Mircoservice-Architektur eingehen. Dabei möchte ich die Perspektive des verteilten Systems mehr beleuchten und die Unternehmsstruktur mehr vordern. Es wäre daher glaube sinnvoll den Abschnitt mit der Conways Law dem anzustellen oder einzu bauen.

2.3.1 Conway's Law

Die Abhängigkeit der Kommunikationsstruktur im Unternehmen zur verwendeten Software Architektur.

- Begriff erklären
- Auswirkungen für die Teamstruktur beschreiben

2.3.2 Domain Driven Design

- Was ist DDD?
- Was sind Bounded Contexts?
- Bespiele dazu.
- Schlussfolgerung für PluraPolit

2.3.3 Cohesion und Coupling

In diesem Abschnitt möchte ich diese beiden Begriffe erklären und noch einmal den Zusammenhang zwischen Funktionlität hervorheben. Es soll klar werden wie Microservices geteilt werden und was es bedeutet Logig hinter definierten Schnitstellen zu verstecken.

Abschließend zu diesem Abschnitt soll die dritte Bedingung für Microservices erstellt werden: **eine in teamsgeteilte, gemischte Unternehmensstruktur**

2.4 Microservice

In diesem Abschnitt möchte ich Microservice definieren und die einzelnen Merkmale benennen. Es sollen zwei Eigenschaften deutlich als solche hervorstechen: Eigenständige Komponente und standatiesierte Kommunikation.

2.4.1 Eigenständige Komponente

In diesem Abschnitt möchte ich detalierter auf die Eigenschaft als eigenständige Komponente eingehen. Ich möchte beschreiben, welche Vorteile es für das einsetzen ergibt und was sich weitere Merkmale sich dadruch ergeben.

Benennen möchte ich dabei:

- deployment
- technology
- max ein Team ist verantwortlich
- DB
- Feature bezogen (nach UNIX Philo)
- wenige Abhängigkeiten -> bleibende Produktivität
- leichter Einstieg
- bedarf angepasste skalierung

Zum Ende des Kapitels möchte ich darauf eingehen, dass dies auch mit sich führt, dass der Aufwand fürs Refaktoren insbesondere bei Veränderungen über mehrere Services hinweg aufwendig ist. Die Aufwand steigt im vergleich zu unverteilten Systemen. Es stellt sich somit als ein deutlicher Nachteil heraus Funktionalität später großflächig zu ändern. Somit ist als Bedingung festzuhalten erst ein Produktmarketfit zu besitzen.

2.5 Produktmarketfit bestimmen

In diesem Abschnitt soll geklärt werden, wie ein Produktmarketfit bestimmt werden kann. Hierfür möchte ich viel aus Running Lean ziehen.

Es geht darum zum Einen Produktmarketfit zu definieren und zum anderen eine Methode zu gegeben an dieser ein solcher Zustand gemessen werden kann.

2.6 Kommunikation von Services

In diesem Abschnitt möchte ich darauf eingehen welche anforderungen eine entkoppelte Infrastuktur an das Netztwerk und an die Kommunikation über standards hat.

Ich möchte die unterschiedlichen Standards vorstellen und die zweite Anforderung für Microservices hinsichtlich des Netzwerks stellen.

Es sollen auf folgende Kommunikationsmethoden eingegangen werden.

- Request / Response
- REST
- HTTP-Operationen (GET, PUT, PATCH, POST, DELETE)
- gRPC (habe selbst keine Ahnung was das ist)

2.6.1 CAP - Theorem

System, Konsistenz, Verfügbarkeit und Partitionstoleranz können in einem verteilten System nicht gleichzeitig erfüllt sein. Die einzelnen Begriffe erklären

- Konsistenz
- Verfügbarkeit
- Partitionstoleranz

2.7 Bedingungen ableiten

Die Bedingungen aus den forhergehenden Kapiteln einführen und zusammenfügen.

3 Methodik

4 Auswertung der Interviews und Anwendung auf PluraPolit

- Public subscribe pattern
- SOA

5 Fazit

Literaturverzeichnis

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).

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

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).

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.

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).

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).

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).

Salus, Peter H. *A Quarter Century of UNIX*. Google-Books-ID: ULBQAAAAMAAJ. Addison-Wesley Publishing Company, 1994. 290 S. ISBN: 978-0-201-54777-1.

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).

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.

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