Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jonas Johannsen

Umsetzung komplexer Geschäftsprozesse in Verteilten Systemen mit Docker

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jonas Johannsen

Umsetzung komplexer Geschäftsprozesse in Verteilten Systemen mit Docker

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 16.02.2018

Jonas Johannsen

Thema der Arbeit
Umsetzung komplexer Geschäftsprozesse in Verteilten Systemen mit Docker

Stichworte
Agile Softwareentwicklung, Verteilte Systeme, Docker, Continuous Delivery, Microservices, REST, Geschäftsprozesse, Softwareintegration

Kurzzusammenfassung
In dieser Arbeit wird mit Hilfe von Docker der Prototyp einer Anwendungslandschaft aus dem Bereich der Energiewirtschaft erstellt. Der Inhalt eignet sich zur Orientierung für den Aufbau ähnlicher Anwendungslandschaften und beschreibt diverse Schritte von der Analyse bis hin zur Implementierung. Verwendet wird die Anwendungslandschaft von der Forschungsgruppe „Projekt HAWAI“ der HAW Hamburg, um die Container Technologie und ihre Möglichkeiten genauer zu untersuchen. Der Entwicklungsprozess hat dabei gezeigt, dass der Einsatz von Docker sehr gut geeignet sein kann, um agile Prozesse zu unterstützen. Einmal korrekt eingerichtet konnte der Fokus auf die kontinuierliche Implementierung von Features gelegt werden. Nicht behandelt wird der Einsatz von Docker in Produktionsumgebungen.

Jonas Johannsen

Title of the paper
Development of complex business processes in distributed systems with Docker

Keywords
Agile Software Development, Distributed Systems, Docker, Continuous Delivery, Microservices, REST, Business Processes, Softwareintegration

Abstract
The object of this thesis is to build the prototype of a distributed system from the energy sector with the help of Docker. The content can be used as an orientation to build similar systems and describes various development steps from the analysis to the implementation. To evaluate the benefits of the technology the system will be used by the research group "‘Projekt HAWAI"’ of the HAW Hamburg. The development process demonstrated, that Docker can improve agile processes. Once configurated properly the resources could be spent to continuously implement features. Not part of this thesis is how to use Docker in a production environment.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Aufgrund der sich häufig ändernden Anforderungen an Softwaresysteme werden in der Wirtschaft seit einigen Jahren agile Entwicklungsmethoden eingesetzt. Diese haben gegenüber vorherigen Methoden wie dem Wasserfallmodell den Vorteil, dass auf Kundenwünsche schneller eingegangen werden kann und neue Technologien früher integriert werden können. Was auf der einen Seite einen enormen Vorteil mit sich brachte, hat auf der anderen Seite zu Komplikationen geführt, da die Abläufe agiler Methoden bisher nicht gut durch Software unterstützt wurden. So musste der ohnehin schon aufwendige Prozess der Integration von Systemen in Test- oder Produktionsumgebungen zunächst weiterhin manuell umgesetzt werden und zudem sehr viel häufiger als noch zuvor. Diese Schritte zu automatisieren und eine einheitliche Integration zu ermöglichen hat sich das 2013 erschienene „Docker“ zum Ziel gesetzt. Docker ermöglicht Anwendungen in leichtgewichtigen Containern zu virtualisieren und unterstützt Entwickler bei der mühsamen Arbeit, Anwendungsumgebungen aufzubauen, unabhängig davon, auf welchem Host-System die Umgebungen laufen sollen. Obwohl Virtualisierung und Verteilte Systeme keine Neuheiten sind, besitzt Docker doch einige interessante Funktionalitäten, die die Entwicklung im Team, aber auch die Skalierungsmöglichkeiten von Systemen erleichtern.

Inzwischen sind viele Firmen als Unterstützer und Partner von Docker eingestiegen. Darunter befinden sich große und namenhafte Firmen wie CISCO, Hewlett-Packard Enterprise, IBM und Microsoft. (vgl. Docker 2017b) Auch das Labor für Anwendungsintegration (Projekt HAWAI) der Hochschule für Angewandte Wissenschaften Hamburg forscht in diesem Bereich. Für eine Forschungsgruppe, deren Schwerpunkt auf der Anwendungsintegration liegt, ist diese Technologie besonders interessant und soll in dieser Arbeit anhand eines Anwendungsbeispiels näher untersucht werden.

1.1 Ziel der Arbeit

Das Projekt-HAWAI versucht nicht nur die Docker Technologie besser zu verstehen, sondern untersucht auch gewachsene Anwendungslandschaften und Entwicklungsprozesse. Ziel dieser Arbeit ist es mit Hilfe von Docker Containern eine reale, gewachsene Anwendungslandschaft zu simulieren, die im weiteren Projektkontext ausgebaut werden kann. Da der Wandel der Energiewirtschaft in der Informatik derzeit eine besondere Stellung einnimmt, sollte die Anwendungslandschaft aus dem Bereich der Energiewirtschaft stammen. Entstanden ist die prototypische Anwendungslandschaft eines fiktiven Energieanbieters, bestehend aus diversen Datenbanken und Services, die durch eine Webanwendung bedient werden können.

Ein erfolgreicher Abschluss des Projekts setzt voraus, dass die Lösung verständlich und wartbar ist sowie innerhalb eines Continuous Delivery Prozesses weiterentwickelt werden kann.

1.2 Struktur der Arbeit

In Kapitel 2 werden die Grundlagen geschaffen. Dazu gehört ein Einblick in den Aufbau der Energiewirtschaft sowie eine Einführung in die Themen Geschäftsprozesse, Continuous Delivery und Docker.
Anschließend werden in Kapitel 3 die Anforderungen an die exemplarische Anwendungslandschaft untersucht, die in Kapitel 4 dann entworfen werden soll.
Neben dem Entwurf werden in Kapitel 4 außerdem die Designentscheidungen des Systems, basierend auf den in Kapitel 3 beschriebenen Anforderungen und den in Kapitel 2 dargelegten Grundlagen, ausführlich begründet.
In Kapitel 5 wird dann die Umsetzung beschrieben. Dazu gehört eine Erläuterung des Projektaufbaus sowie eine Beschreibung, wie die Anwendungslandschaft gestartet und erweitert werden kann.
Im letzten Kapitel 6 werden die Ergebnisse der Arbeit abschließend nochmal zusammengefasst. Des Weiteren wird ein Ausblick gegeben, welche Erweiterungen oder Änderungen in zukünftigen Projekten umgesetzt werden könnten.

2 Grundlagen

Im Folgenden werden nun die für das Verständnis der Arbeit benötigten Grundlagen geschaffen.

2.1 Anwendungskontext - Energiewirtschaft

Die deutsche Energiewirtschaft muss sich zurzeit großen Herausforderungen stellen. Zum einen muss sie mit der Energiewende umgehen und zum anderen mit der Digitalisierung ihrer Prozesse. Einige dieser Prozesse sind durch Gesetze vorgeschrieben, andere wiederum können erforderlich sein, um im liberalisierten Markt wettbewerbsfähig zu bleiben. Die schiere Größe des Netzes und die Menge an Systemen und an Akteuren in den Systemen sorgen dabei für eine hohe und schwer zu beherrschende Komplexität. Stefan Kapferer, aktueller Vorsitzender der Hauptgeschäftsführung des Bundesverbands der Energie- und Wasserwirtschaft e.V. (kurz BDEW), spricht von der Energiewende sogar als, „[...] das größte nationale IT - Projekt aller Zeiten“ (BDEW 2016, S. 5). „In keiner Branche fallen perspektivisch mehr Daten an, deren Auswertung für eine sichere und effiziente Versorgung unserer Kunden sorgt“, beschreibt Kapferer die Situation weiter. Um die Energiewirtschaft besser verstehen zu können und Lösungen zu finden, möchte sich das Projekt HAWAI in Zukunft zunehmend mit ihren Problemen auseinandersetzen. Die im Rahmen dieser Arbeit entwickelte Anwendung bildet für dieses Projekt den Grundbaustein einer wachsenden Anwendungslandschaft. Entwickelt werden soll eine Webanwendung für den fiktiven Energielieferanten „**Greenworld Energies GmbH**“, über dessen Benutzerschnittstelle potenzielle Kunden unter anderem einen Stromliefervertrag abschließen können und Administratoren neue Stromtarife erstellen. Für die Umsetzung der Anwendung ist ein Grundverständnis der Energiewirtschaft notwendig. Dieses Kapitel gibt deshalb einen kurzen Einblick in die vorhandenen Akteure und den grundlegenden Aufbau der Energiewirtschaft. Der Fokus liegt dabei auf dem Unternehmen „Greenworld Energies“ und auf dem Zusammenspiel des Unternehmens mit anderen Akteuren des Marktes. Obwohl der Gashandel zwar ebenfalls ein Teil der Energiewirtschaft ist, kann dieser im Kontext der Arbeit vernachlässigt werden.

2.1.1 Akteure in der Energiewirtschaft

Christian Aichele unterteilt die Energieversorgung in sechs Teilbereiche: Energieerzeugung, Übertragungsnetz, Verteilnetz, Energievertrieb, Energiehandel und Messstellenbetrieb/Messdienstleistung. Diese Teilbereiche werden in der heutigen Energiewirtschaft von einer Vielzahl verschiedener Akteure umgesetzt. Einige davon sollen an dieser Stelle kurz vorgestellt werden. (vgl. Aichele 2012)

Netzbetreiber

Netzbetreiber kümmern sich um die Infrastruktur der Stromnetze zur elektrischen Energieübertragung. Dabei unterscheidet man zwischen **Übertragungsnetzbetreibern** (ÜNB) und **Verteilnetzbetreibern** (VNB). Während sich die ÜNB um Hoch- und Höchstspannungsnetze und den damit verbundenen Stromtransport über große Entfernungen kümmern, sind VNB für das Management des regionalen Strom-Verteilnetzes zuständig. Die VNB kümmern sich also um die eigentliche Belieferung der Verbraucher und sorgen dafür, dass der Strom zu den Verbrauchern gelangt. (vgl. Aichele 2012, S. 6) Der ÜNB nimmt außerdem die Rolle des **Bilanzkreiskoordinators** ein. (vgl. Panos 2007, S. 41) Der Bilanzkreiskoordinator ist dafür verantwortlich, innerhalb seines Netzbereiches, „Abweichungen zwischen Einspeisungen und Entnahmen durch ihre Durchmischung zu minimieren und die Abwicklung von Handelstransaktionen zu ermöglichen“. (BGBL 2005, § 3 Begriffsbestimmungen)

Energieerzeuger

Bei der Energieerzeugung geht es unter Anderem um die Umwandlung schlecht nutzbarer Energieformen in Elektrizität. (vgl. Aichele 2012, S. 5) Die Elektrizität gewinnt man z.B. aus thermischer Energie (z.B. Kohlekraftwerke) oder durch mechanische Energie (Windenergie, Wasserkraft). Klassisch wurde die Elektrizität von wenigen **Unternehmen mit großen Kraftwerken** erzeugt. Heutzutage ist die Stromerzeugung durch Einspeisungen aus der **Industrie** und aus erneuerbaren Energien (Windenergie, Solarenegie) sowie aus **privaten Haushälten**, allerdings sehr viel dezentraler geworden und die Menge an Erzeugern stark angestiegen. (vgl. Aichele 2012, S. 6) Da erneuerbare Energien von Umweltfaktoren abhängen, die schwer vorhersehbar sind und zu starken Schwankungen der Stromerzeugung führen können, erhöht die Einspeisung der erneuerbaren Energien und die zunehmende Dezentralität der Einspeisungen die Komplexität der Stromversorgung. Durch das Erneuerbare Energien Gesetz wurde es Kunden sogar möglich überschüssigen Strom aus erneuerbaren Energien an die Netzbetreiber zu verkaufen. Auf Grund dieser Komplexitätssteigerung gibt es viele Forschungsarbeiten, die sich mit Konzepten wie „Intelligente Stromzähler“, „Intelligente Netze“ oder „Dynamische

Tarife“ auseinandersetzen. Dabei handelt es sich um Konzepte und Visionen, bei denen sich Netze selber steuern und „Kunden zum Verbraucher und Erzeuger in einer Person“ werden. (Aichele 2012, S. 2) Die „dezentral erzeugten regenerativen Energien“ sollen auf diese Weise „zum richtigen Zeitpunkt in der richtigen Menge zur Verfügung“ gestellt werden können. (Aichele 2012, S. 2)

Letztverbraucher

Letztverbraucher schließen Lieferverträge mit Lieferanten ab und verwerten den erhaltenen Strom. Die Nachfrage der Verbraucher ist der Grund, aus dem die Energiewirtschaft überhaupt existiert. Dabei kann man Verbraucher in zwei Kategorien unterteilen. **Großkunden** sind Verbraucher, die viele tausend Kilowattstunden (Kwh) pro Jahr benötigen und größtenteils aus der Industrie oder öffentlichen Einrichtungen kommen. Im Gegensatz dazu sind **Haushaltskunden** „Letztverbraucher, die Energie überwiegend für den Eigenverbrauch im Haushalt oder für den einen Jahresverbrauch von 10.000 Kilowattstunden nicht übersteigenden Eigenverbrauch für berufliche, landwirtschaftliche oder gewerbliche Zwecke kaufen“. (BGBL 2005, § 3 Begriffsbestimmungen)

Energielieferanten (Vertrieb/Handel)

„Lieferanten können Kraftwerksbetreiber oder Händler sein, die Strom in eigener Regie kaufen und verkaufen.“ (Panos 2007, S. 39) „Greenworld Energies“ ist ein **Energiehändler**. Das bedeutet, dass sich das Unternehmen um die Vermarktung von Strom kümmert, ohne dass es ein eigenes Kraftwerk besitzt. Es bildet also die Schnittstelle zwischen der Stromerzeugung und den Verbrauchern. Dafür verpflichtet es sich in einem **Bilanzkreisvertrag** mit dem Bilanzkreiskoordinatoren (siehe Netzbetreiber) dieselbe Menge an Strom von Erzeugern zu kaufen und in das Netz einzuspeisen, wie seine Verbraucher dem Netz entnehmen. Aus diesem Grund wird der Verbrauch von Großkunden regelmäßig ermittelt. Bei Haushaltskunden hingegen wird der Verbrauch geschätzt und erst beim nächsten Ablesezeitpunkt mit dem realen Verbrauch abgeglichen, da die Schwankungen beim jährlichen Verbrauch aufgrund der geringen Menge in der Regel nicht so gravierend und geschäftsschädigend sind.

Messstellenbetreiber/Messdienstleister

„Der Messstellenbetreiber (MSB) und der Messdienstleister (MDL) sind für den Betrieb der Messstelle und den Service der notwendigen Geschäftsprozesse wie Ablesung, Zählertausch und Kommunikation der Zählerinformationen zuständig.“ (Aichele 2012, S. 7)

2.1.2 Aufbau der Energiewirtschaft

Mit dem Gesetz zur Neuregelung des Energiewirtschaftsrechts von 1998 wurde das bis dahin geltende Gesetz von 1935 abgelöst, woraufhin sich der deutsche Energiemarkt in den letzten zwei Jahrzenten drastisch reformiert hat. Zuvor war es in der Regel so, dass für ein bestimmtes Versorgungsgebiet der Netzbetreiber gleichzeitig der Lieferant war. Durch die Lieberalisierung wurde der Markt geöffnet und die Verbraucher konnten sich seitdem ihre Lieferanten frei wählen. (vgl. BGBL 1998) Grund hierfür war das sogenannte Unbounding der Tätigkeitsbereiche der Energieversorgung und die Einführung einer staatlichen Regulierungsbehörde für den Netzzugang. Die vertikal strukturierten Unternehmen, die den Markt bis dahin beherrscht haben, wurden so gesetzlich verpflichtet ihre Tätigkeitsbereiche (Energieerzeugung, Energietransport und Energieverkauf) zu „entflechten“, um möglichst viele Teile der Lieferkette dem freien Wettbewerb zu unterstellen. Die Entflechtung umfasst besonders die rechtliche Trennung der Netze von den anderen Tätigkeitsbereichen. Netzbetreiber sind dadurch verpflichtet ihre Netze jedem Netznutzer gleichermaßen und zu einem fairen Preis zur Verfügung zu stellen und dürfen sich nicht an der Erzeugung und dem Kauf- und Verkauf von Strom beteiligen. (vgl. Aichele 2012, S. 3-8)

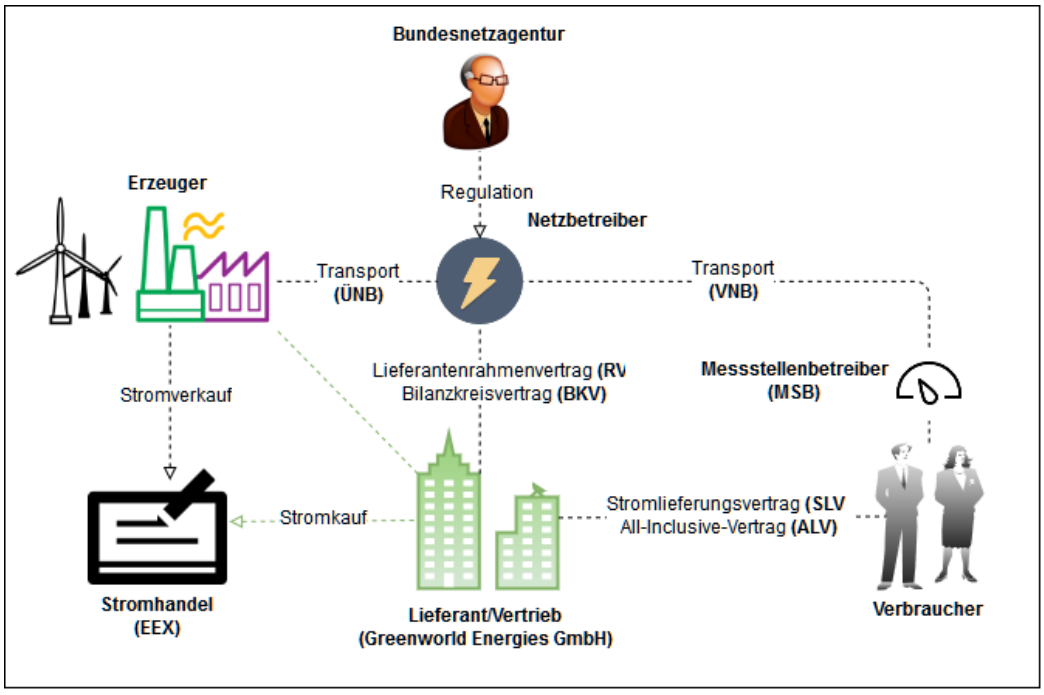

Abbildung 2.1: Der Aufbau der Energiewirtschaft

Abbildung 2.1 zeigt den Aufbau der Energiewirtschaft in einer vereinfachten Form und die Entflechtung der Unternehmen in Erzeuger, Netzbetreiber und Lieferanten. Durch die Entflechtung der Unternehmen, zusammen mit der Liberalisierung der Versorgungsnetze, sollten die Monopolstellungen der großen privaten Unternehmen aufgelöst und Verbraucher zu den „[...] günstigsten Konditionen marktgerecht versorgt werden“ können. (Aichele 2012, S. 3-4) Da die Versorgungsnetze allerdings nicht sinnvoll dem freien Wettbewerb unterzogen werden können, haben die Netzbetreiber hier nach wie vor eine Monopolstellung. Um ein Ausnutzen der Monopolstellung seitens der Netzbetreiber zu verhindern, wurde die staatliche Regulierung der Netze durch die **Bundesnetzagentur** eingeführt. Die Bundesnetzagentur hat in erster Linie die Aufgabe, die Preise für die Nutzung der Netze zu kontrollieren und zu regulieren. Dadurch sollen alle Netznutzer die Netze unter denselben Bedingungen und zu fairen Preisen nutzen können, wodurch eine diskriminierungsfreie Behandlung aller Netznutzer gewährleistet werden kann. Die Energiepreise für den Endverbraucher unterliegen allerdings weiterhin dem freien Wettbewerb. Lediglich die Preise für die Netznutzung werden reguliert. (vgl. Aichele 2012, S.4-5)

Obwohl es noch einige klassische Energieversorgungsunternehmen wie EON und Vattenfall gibt, die Strom erzeugen und diesen auch selber vertreiben, sind durch die Liberalisierung des Strommarktes auch neue Konzepte wie der Handel mit Strom enstanden. Durch den Handel, der zum Beispiel über den **European Energy Exchange (EEX)** abgewickelt wird, sind viele Stromhändler wie „Greenworld Energies“ entstanden, die „weder über Kraftwerke, noch über eigene Leitungen noch über eigenen Bedarf vefügen“. (Aichele 2012, S. 7) Auf diese Weise ist ein „wichtiges neues Geschäftsfeld“ entstanden, in dem alle größeren Unternehmen der Branche und auch viele unabhängige Händler mit eingestiegen sind. (vgl. Aichele 2012, S. 7)

Damit der eingekaufte Strom bei der Entnahmestelle des Verbrauchers ankommt, muss der Lieferant Netznutzungsentgelte an den zuständigen Netzbetreiber zahlen. Die Abrechnung der Entgelte werden in einem **Lieferantenrahmenvertrag (RV)** zwischen dem Lieferanten und dem Netzbetreiber abgeschlossen. Der Vertrag regelt „alle Rechten und Pflichten in Zusammenhang mit der Belieferung der Kunden des Lieferanten in Netzgebiet des Netzbetreibers. Darin wird u.a. der Umfang in Form von Fahrplänen sowie Datenaustausch zwischen dem Lieferanten und der Verteilnetzbetreiber geregelt.“ (Panos 2007, S. 41)

Des Weiteren sind Lieferanten i.d.R. bilanzkreisverantwortlich. Zu diesem Zweck wird ein **Bilanzkreisvertrag (BKV)** zwischen dem bilanzkreisverantwortlichen Lieferanten und dem Bilanzkreiskoordinatoren (Übertragungsnetzbetreiber) abgeschlossen, in dem sich der Lieferant unter anderem verpflichtet, in einem 1/4 stündigen Takt im Voraus Fahrpläne an den Bilanzkoordinator abzuliefern und eine ausgeglichene Bilanz zwischen Einspeisungen und Entnahmen in seinem Bilanzkreis einzuhalten. (vgl. Panos 2007, S. 41) „Die Aufstellung der Fahrpläne erfolgt auf der Basis von historischen Daten für typische Tage, Wetterprognosen und sonstigen Ereignissen, die den Lastverlauf beeinflussen können." (Panos 2007, S. 342) Durch die Prognosen sollen Netzbetreiber besser planen und auf Schwankungen besser reagieren können.

Da es wie im Kapitel zuvor erwähnt verschiedene Arten von Verbrauchern gibt, gibt es auch unterschiedliche Arten von Lieferverträgen. „Einen Stromlieferungsvertrag (SLV) schließt der Kunde mit dem Lieferanten seiner Wahl ab. Meistens wird dies aber nur von Großkunden praktiziert, die Lieferverträge mit mehr als einem Lieferanten abschließen und auch Strom von der Börse beziehen können (Portfolio-Management). Kleinere und die Mehrzahl der mittelständischen Kunden schließen einen sogenannten All-Inclusive-Vertrag (AIV) mit einem Lieferanten ab, der auch die Netznutzung beinhaltet. Wegen des erheblichen Aufwandes für eine genaue Prognose des Bedarfs, der insbesondere zur Fahrplanerstellung für jede 1/4 Stunde erforderlich ist, ist ein Vollversorgungsvertrag auch im liberalisierten Markt für die Mehrzahl der Stromkunden der Regelfall." (Panos 2007, S. 41)

2.2 Komplexe Geschäftsprozesse

Es gibt verschiedene Definitionen darüber, was einen Geschäftsprozess ausmacht. Die Trennung, wo ein Geschäftsprozess anfängt und wo er aufhört, ist dabei allerdings häufig nicht klar definiert. Grundsätzlich lässt sich folgendes sagen:

Als Geschäftsprozess wird die Abfolge von logisch verknüpften Einzeltätigkeiten bezeichnet, die durch ein Eingangsereignis ausgelöst werden und ein bestimmtes geschäftliches oder betriebliches Ziel erreichen sollen. (vgl. Wikipedia) Das Gabler Wirtschaftslexikon definiert Geschäftsprozesse ähnlich, nämlich als eine „Folge von Wertschöpfungsaktivitäten (Wertschöpfung) mit einem oder mehreren Inputs und einem Kundennutzen stiftenden Output". (Lackes u.a. 2009) Als komplex werden Geschäftsprozesse in dieser Arbeit nun bezeichnet, sobald mehrere Systeme und Abteilungen beteiligt sein müssen, damit der Prozess erfolgreich abgeschlossen werden kann. Die Komplexität entsteht also durch eine Verteilung des Ablaufes

auf unterschiedliche Akteure.

Ein Beispiel für einen komplexen Geschäftsprozess wäre der Prozess „Lieferbeginn Strom“. Damit der Beginn der Stromlieferung in die Wege geleitet werden kann, müssen mehrere Einzeltätigkeiten von unterschiedlichen Akteuren durchgeführt werden. Im ersten Schritt muss der Kunde einen Stromliefervertrag mit dem Lieferanten abschließen. Bei einem erfolgreichen Vertragsabschluss kann der Lieferant im zweiten Schritt die Entnahmestelle des Kunden beim entsprechenden Netzbetreiber anmelden. Der Prozess besteht also bereits aus mindestens drei Akteuren, die miteinander interagieren müssen, damit der Kunde mit Strom beliefert werden kann. In der Realität besitzen die Akteure „Lieferant“ und „Netzbetreiber“ zudem vermutlich weitere Abteilungen, die ebenfalls einzelne Aufgaben übernehmen müssen.

2.3 Agile Softwareentwicklung

In dem „Agile Manifesto“ von Kent Beck (u. a.) wird als wichtigstes Ziel der agilen Softwareentwicklung genannt, „den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufrieden zu stellen“. (Beck u.a. 2001) Langfristige Entwicklungsprozesse sollen also durch schnelle Entwicklungszyklen ersetzt werden. Zu den bekanntesten agilen Methoden gehören „Scrum“ und „Kanban“. Allerdings existieren auch innerhalb dieser Entwicklungsmethoden Zyklen. (Mouat (2016)) Während Scrum Iterationszyklen von mehreren Wochen besitzen kann, ist Kanban eher auf die schnelle Implementierung von Features ausgelegt. Durch die Geschwindigkeit mit der neue Features in Produktion gebracht werden können, „[…] ergibt sich auf der Geschäftsseite ein wesentlicher Vorteil: Es ist viel einfacher, auf Änderungen am Markt reagieren zu können“. (Wolff 2016, S. 17) Wichtig ist dabei allerdings die kontinuierliche Einhaltung von Sicherheitsmechanismen, wie z.B. die Ausführung verschiedener Tests, um zu verhindern, dass sich Fehler bei der Integration in die Produktionsumgebung einschleichen. Ohne eine **Automatisierung** dieser Prozesse, kann dies sehr zeitintensiv sein und manuelle Unterstützung durch IT-Mitarbeiter benötigen, was ebenfalls eine Fehlerquelle sein kann.

„[…] Knight Capital beispielsweise hat aufgrund eines fehlerhaften Software-Rollouts 440 Mio. € verloren […]. Die Folge war die Insolvenz der Firma“. (Mouat 2016, S. 21)

Das Beispiel veranschaulicht, welche Risiken die Integration von Software in eine Produktionsumgebung besitzen kann. Um diese Risiken zu minimieren, können die beschriebenen Prozesse durch einen Continuous Delivery Ansatz automatisiert werden.

2.3.1 Continuous Delivery

„Continuous Delivery ermöglicht es, Software schneller und mit wesentlich höherer Zuverlässigkeit in Produktion zu bringen als bisher. Grundlage dafür ist eine Continuous-Delivery-Pipeline, die das Ausrollen der Software weitgehend automatisiert und so einen reproduzierbaren, risikoarmen Prozess für die Bereitstellung neuer Releases darstellt.“ (Wolff 2016, S. 1)

Die Pipeline besteht dabei aus einer sequentiellen Folge verschiedener Phasen. Jede Phase führt definierte Aufgaben aus, wie z.B. die Integration der Anwendung in eine Review- oder Staging-Umgebung. Da das Ausführen einer Phase, den Erfolg der vorhergehenden Phase voraussetzt, wird der Prozess abgebrochen, sobald eine Phase fehlschlägt. Auf diese Weise kann das Risiko minimiert werden, dass Fehler in Produktion gehen. (vgl. Wolff 2016, S. 24-26)

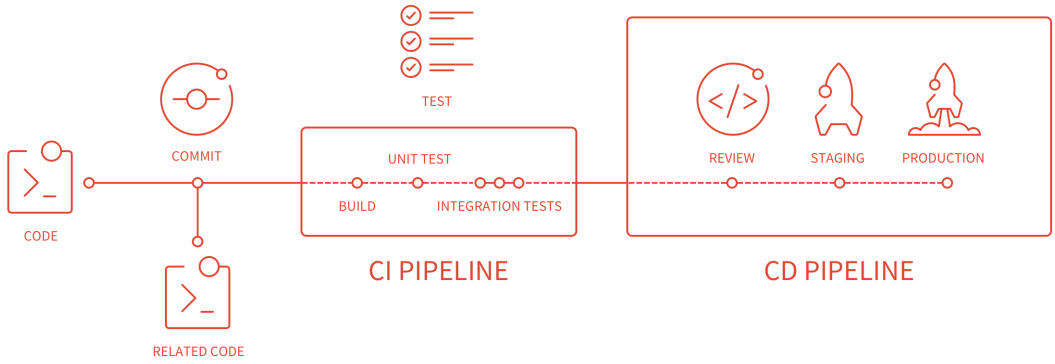

Abbildung 2.2: Aufbau der CI/CD-Pipeline
Quelle: https://docs.gitlab.com/ee/ci/ Zugriff: 31.01.2018

Continuous Delivery setzt in der Regel den Prozess der Continuous Integration (CI) voraus. Abbildung 2.2 veranschaulicht den Zusammenhang dieser Prozesse. Ausgangspunkt für die Ausführung kann der Commit des Quellcodes in das Repository der verwendeten Versionskontrolle, z.B. „GitLab.com“, durch einen Entwickler sein. Infolgedessen werden dann Build-Prozesse oder verschiedene Tests automatisiert ausgeführt. Dies können Unit-Tests sein bei denen die einzelnen Komponenten der Software getestet werden sollen sowie Integration-Tests, die das Zusammenspiel der Komponenten testen. Wenn die CI-Pipeline durchgelaufen ist, wird die Anwendung in verschiedenen Umgebungen getestet, bevor sie in der Produktionsumgebung veröffentlicht wird. Eine Review-Umgebung könnte z.B. genutzt werden, um „Akzeptanz-“ oder

„Kapazitätstests“ durchzuführen und somit die „Absicherung fachlicher Features“ sowie die „Performance“ des Systems sicherzustellen. (Wolff 2016, S. 138, S.157) Die Staging Umgebung könnte dagegen zum Beispiel. für „Explorative Tests“ und die Abnahme durch den Kunden genutzt werden. Anschließend wird der Prozess, vom Commit bis zur Integration der Anwendung in die Produktionsumgebung so oft wiederholt, bis alle Phasen erfolgreich ausgeführt wurden.

Die Beschreibung des Ablaufs wird dabei häufig durch verschiedene Scripts und Konfigurationsdateien ermöglicht. „GitLab.com“ stellt für die Konfiguration der Pipelines z.B. die Konfigurationsdatei „gitlab-ci.yml“ zur Verfügung. Die Datei bietet die Möglichkeit, in Form einer YAML-Datei, unterschiedliche Phasen (hier „stages“ genannt) zu beschreiben, die in einer bestimmten Reihenfolge abgearbeitet werden sollen und sogenannte „Jobs“ ausführen. Ausgeführt wird die Datei durch einen entsprechenden Executor, auch „Runner“ genannt, der von GitLab zur Verfügung gestellt wird und mit dem Projekt verlinkt werden kann. Listing 2.1 zeigt ein vereinfachtes Beispiel der Dateistruktur.

```
1  image: example-image
2  stages:
3    - build
4  #   - unit-test
5  #   - integration-test
6  job1:
7    stage: build
8    only:
9      - master
10   script:
11     - execute-job1-script
```

Listing 2.1: gitlab-ci.yml

Die Pipeline besteht in dem Beispiel aus einer einzigen Stage, namens „build“, die den Job „job1“ ausführt. Das angegebene Script wird allerdings nur dann ausgeführt, wenn der Commit auf dem „Master“-Branch („only: -master“) durchgeführt wurde. Auf diese Weise können verschiedene Workflows realisiert werden, wie zum Beispiel. eine Branching-Strategie mit verschiedenen Feature Branches, da so verhindert werden kann, dass das Deployment bei einem Commit auf einen Feature-Branch angestoßen wird. Das Schlüsselwort „image“ beschreibt dabei, welches Basisimage der Executor für das Ausführen der Jobs verwenden soll. Was genau

das Basisimage ist, wird im nächsten Abschnitt über die Docker-Technologie näher beschrieben. Die Stages „unit-test“ und „integration-test“ wurden in dem Beispiel auskommentiert und sollen nur zeigen, wie der sequentielle Ablauf mehrerer Stages definiert werden könnte.

Des Weiteren enthält das gezeigte Beispiel nur einen Bruchteil der möglichen Konfigurationsmöglichkeiten. Gerade in sehr komplexen Systemen kann die Datei wesentlich umfangreicher werden und Verweise auf weitere Scripts beinhalten, um die Integration in verschiedene Laufzeitumgebungen zu realisieren. Die Beschreibung sollte nur ein allgemeines Verständnis für die Umsetzung eines Continuous Delivery Prozesses mittels GitLabs CI-Pipelines schaffen. Die vollständige Beschreibung findet man unter der folgenden URL:
https://docs.gitlab.com/ee/ci/

GitLab unterstützt einen zwar dabei, eine CI/CD-Struktur aufzubauen, indem verschiedene Stages und Jobs definiert werden können - die Integration der Software in verschiedene Laufzeitumgebungen erfordert dabei allerdings die Verwendung weiterer Technologien. Eine Lösung bietet die Docker Technologie, die im nächsten Abschnitt nun beschrieben werden soll.

2.4 Docker

Docker ist eine 2013 unter der Apache 2.0 Lizenz erschienene Open Source Virtualisierungs-Technologie und wurde seitdem von einer Vielzahl von namenhaften Unternehmen und Investoren gefördert. Da Docker eine noch sehr junge Technologie ist, die sich sehr schnell weiterentwickelt, gibt es noch nicht viel Literatur zu dem Thema und die Literatur, die es gibt, ist häufig bereits veraltet. Dennoch sind die Grundkonzepte von Docker bereits 2016 in dem Buch „Using Docker - Developing and Deploying Software with Containers“ von Adrien Mouat gut und detailliert beschrieben worden und nach wie vor gültig. (siehe Mouat 2016) In diesem Abschnitt werden nun die Grundlagen der Technologie anhand der Beschreibungen von Adrian Mouat vorgestellt und erklärt.

2.4.1 Die Frachtcontainer-Metapher

Eine bekannte Metapher für Docker ist die Frachtcontainer-Metapher. Wie viele Metaphern ist sie an einigen Stellen unsauber - dennoch hilft sie dabei, einen ersten Eindruck über die Ziele von Docker zu erhalten.

Bevor es die intermodalen Container im Transportgewerbe gab, war das Be- und Entladen von Schiffen und Transportern ein sehr aufwendiger Prozess. Bestimmte Container konnten nur von bestimmten Transportern transportiert werden und jeder Container hatte eigene Anforderungen auf die individuell und manuell eingegangen werden musste, z.B. durch die Arbeit von Dockarbeitern (engl. „Dockers"). Erst durch die Einführung von Standards wie einheitliche Containergrößen oder einheitliche Beschriftungen wurde der Aufwand minimal und konnte automatisiert werden. (vgl. Mouat 2016, S. 7-9)

Ähnliche Probleme existieren in der Softwareentwicklung. Wie der Abschnitt über „Continuous Delivery" gezeigt hat, müssen Anwendungen innerhalb der Pipelines häufig in verschiedene Umgebungen integriert werden. Sei es die Integration in eine lokale Entwicklungsumgebung auf den Endgeräten der Entwickler oder die Integration in eine Review- oder Staging-Umgebung: Es ist sehr schwer die verschiedenen Umgebungen einheitlich zu halten. Grund hierfür können unterschiedliche Versionen benötigter Bibliotheken oder unterschiedliche Betriebssysteme sein. Bei der Integration in andere Umgebungen ist dann häufig viel manuelle Arbeit nötig bis die Anwendung läuft. Die Idee von Docker ist es, wie in der Metapher beschrieben, einheitliche Bedingungen für das Erstellen, den Transport und die Integration von Softwaresystemen zu schaffen, um Anwendungen, unabhängig von der Hardware oder den Endsystemen auf denen sie laufen sollen, entwickeln und deployen zu können. Durch Docker wird die Automatisierung des Integrationsprozesses also zum Standard, wodurch die Risikowahrscheinlichkeit und die Kosten eines Softwarereleases sinken können. Es entsteht eine Reproduzierbarkeit des Integrationsprozesses. Die Integration in eine Review-, Staging- und Produktionsumgebung oder in die Entwicklungsumgebung der einzelnen Mitarbeiter verhält sich also gleich und kann deutlich schneller durchgeführt werden, als bei einer manuellen Integration. (vgl. Mouat 2016, S. 7-8)

2.4.2 Die Architektur

Ein weiterer Vorteil von Docker ist die schmale Architektur. So besteht Docker im Grunde aus fünf Grundbausteinen.

1. **Docker Container**
   „Container sind eine Verkapselung einer Anwendung und ihrer Abhängigkeiten". (Mouat 2016, S. 3) Dadurch sind Docker Containern den intermodalen Container im Transportgewerbe sehr ähnlich: Obwohl sie von außen gleich aussehen und gleich behandelt werden können, kann sich der Inhalt der Container unterscheiden.

2. **Docker Images**
   Docker Images sind Abbilder von Docker Containern, die gespeichert und portiert werden können. Sie enthalten sämtliche Anweisungen, die vom Docker Daemon benötigt werden, um einen Container zu erzeugen und lassen sich in einer Textdatei, dem sogenannten Dockerfile, beschreiben. Ein Beispiel-Image könnte das Abbild einer Ubuntu-Linux Distribution darstellen, in der ein bestimmter Satz an Bibliotheken vorinstalliert wurde. Man kann die Images also als den Inhalt oder die Ware der Container ansehen und mehrere Container mit der gleichen Ware erstellen. Dadurch wird eine horizontale Skalierung von Anwendungen möglich.

3. **Docker Registry**
   „In der Registry werden Images abgelegt und verteilt. Die Standard Registry ist der Docker Hub, auf dem tausende öffentlich verfügbare Images zur Verfügung stehen, aber auch kuratierte »offizielle« Images.“ (Mouat 2016, S. 36) Die Registry könnte damit dem Containerhafen oder verschiedenen Lagerhallen entsprechen, in denen die Waren gelagert werden. Da die HAW eine eigene Registry besitzt, wurde für dieses Projekt, aus Sicherheitsgründen, die private Registry der Hochschule verwendet.

4. **Docker Daemon/Engine**
   „Der Docker Daemon ist für das Erstellen, Ausführen und Überwachen der Container zuständig. Aber auch das Bauen und Speichern von Images fallen in seinen Zuständigkeitsbereich.“ (Mouat 2016, S. 36.) Abbildung 2.3 zeigt, dass der Daemon Images verwendet, um Container zu erstellen. Existieren die Images noch nicht auf dem Host System, so werden sie aus der Registry geladen. Der Daemon könnte also das Transportunternehmen sein, das das Be- und Entladen der Container und des Transportschiffes überwacht und den Transport steuert.

5. **Docker Client**
   „Der Docker Client [...] wird dazu genutzt, per HTTP REST mit dem Docker Daemon zu kommunizieren.“ (Mouat 2016, S. 36) Der Daemon besitzt zwar auch eine gut dokumentierte API, wodurch eine direkte Kommunikation möglich wäre. Für die meisten Anwendungen, einschließlich der in diesem Projekt entwickelten, reichen die Befehle und Funktionalitäten des Clients allerdings bereits aus. Um die Metapher abzuschließen: Der Client könnte der Kunde sein, der dem Transportunternehmen den Auftrag erteilt, bestimmte Frachten zu transportieren.

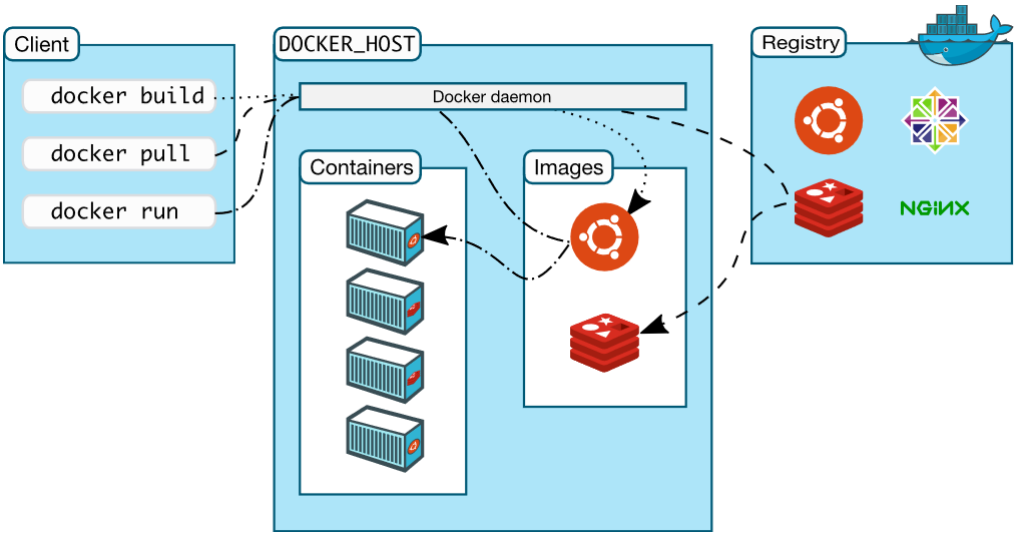

Abbildung 2.3: Architektur von Docker
Quelle: https://docs.docker.com/engine/article-img/architecture.svg

2.4.3 Dockerfile

Ein Dockerfile ist eine Textdatei, in der der Inhalt eines Images beschrieben werden kann. Der Daemon kann diese Datei verarbeiten und erstellt daraus das Image.

```
1 # Kommentar
2 FROM ubuntu # Basisimage
3 RUN apt-get install -y git # Installiere git
4 WORKDIR /bar # Ändere aktuelles Verzeichnis zu /bar
5 COPY ./foo # Kopiere Hostdateien von ./foo nach /bar
```

Listing 2.2: Ein Beispiel Dockerfile

Jedes Dockerfile besteht aus einer Reihe von Anweisungen, die an dieser Stelle nicht alle im Detail erklärt werden sollen. Die erste Anweisung muss dabei allerdings immer eine FROM-Anweisung sein. „Die From-Anweisung legt fest, welches Basisimage zu verwenden ist […]“ (Mouat 2016, S. 25) Ein Basisimage könnte zum Beispiel wie in Listing 2.2 ein Ubuntu-Image sein. Basierend auf diesem Basisimage können dann per RUN-Anweisung Ubuntu-Befehle ausgeführt werden, um benötigte Packages zu installieren. Jeder Container, der aus dem gleichen Image gebaut wird, ist auf diese Weise identisch aufgebaut und austauschbar. Die vollständige Spezifikation findet man unter der folgenden URL:
https://docs.docker.com/engine/reference/builder/

2.4.4 Docker vs. Virtuelle Maschinen

Wie die Architektur gezeigt hat, kapseln Docker Container die Anwendung inklusive ihrer Abhängigkeiten (benötigte Bibliotheken). Virtuelle Maschinen (VMs) übernehmen auf den ersten Blick dieselben Aufgaben. Im direkten Vergleich zeigen sich allerdings deutliche Unterschiede, besonders in der Performance. Adrian Mouat hat die wesentlichen Unterschiede in seinem Buch anschaulich beschrieben. (vgl. Mouat 2016, S.3-6) Im Folgenden werden die wichtigsten Punkte zusammengefasst.

Die Performance oder auch die Geschwindigkeit mit der Anwendungen gebaut und gestartet werden können kann besonders innerhalb eines Continuos Delivery Prozesses relevant sein. Man kann zwar mit Scripts versuchen das Erstellen und Starten der VMs ebenfalls zu automatisieren, dennoch wird man auf diese Weise nicht die Geschwindigkeit von Docker erreichen. Grund hierfür ist unter Anderem das nicht-Vorhandensein eines Hypervisors in der Docker Architektur. Der Hypervisor einer VM nimmt die priviligierten Maschinenbefehle der Gast-Systeme entgegen und verarbeitet sie. Damit stellt er eine Schnittstelle zwischen den Gast-Systemen und dem Host-System dar und koordiniert die Zugriffe auf die reale Hardware. Dies sorgt zwar aufgrund einer stärkeren Isolation der Gast-Systeme auf der einen Seite für wesentlich mehr Sicherheit, auf der anderen Seite aber auch für einen nicht zu vernachlässigenden Overhead, den ein Docker Container nicht besitzt.

Docker Container teilen sich die Ressourcen und den Kernel des Host-Systems und können aufgrund ihrer Leichtgewichtigkeit zu Dutzenden auf einem System parallel ausgeführt werden. Dies erleichtert und beschleunigt das Starten und Testen und damit das Entwickeln der Anwendungslandschaft im Team, da jedes Teammitglied die Anwendungslandschaft lokal auf dem eigenen System ausführen kann und zwar unabhängig von dem Host-Betriebssystem. Die gemeinsame Nutzung des Host-Kernels birgt allerdings auch Risiken für die Sicherheit der Container, da ein einzelner erfolgreicher Angriff auf den Host-Kernel sämtliche Container kompromitieren könnte. Aus dem Grund werden auch Kombinationen der beiden Technologien verwendet, bei denen Docker innerhalb einer virtuellen Maschine gekapselt ausgeführt wird.

Aufgrund der Automatisierung des Integrationsprozesses können Docker Container zudem leichter skaliert werden. Hierfür ist dann allerdings die Verwendung weiterer Technologien wie Docker Swarm oder Kubernetes erforderlich, die sich um die Orchestrierung der Container kümmern.

2.4.5 Docker Compose

„Docker Compose ist ein Tool zum Bauen und Ausführen von Andwendungen, die aus mehreren Docker-Containern bestehen. Es wird vor allem in der Entwicklung und beim Testen und weniger im Produktionsumfeld genutzt." (Mouat 2016, S. 38) Zu diesem Zweck wird die Anwendungslandschaft in Form einer YAML-Datei konfiguriert. Der Vorteil der Automatisierung mittels Docker Compose ist, dass eigene, fehleranfällige, scriptbasierte Orchestrierungs-Mechanismen abgelöst werden können. (vgl. Mouat 2016, S. 83) Listing 2.3 zeigt den Aufbau einer vereinfachten „docker-compose.yml"-Datei. Das Tool besitzt zwar sehr viel mehr Möglichkeiten, dennoch sollte das Beispiel ausreichen, um einige Grundprinzipien zu veranschaulichen. Die beschriebene Anwendungslandschaft besteht in diesem Fall aus den zwei Services „service-a" und „service-b".

```
1  services:
2    service-a:
3      build: ./service-a
4      ports:
5        - "8080:80"
6      volumes:
7        - ./service-a/src:/src
8    service-b:
9      image: docker-hub.informatik.haw-hamburg.de/abq307/greenworld-
           energies/service-b:latest
10     environment:
11       - VAR_1=some-value
```

Listing 2.3: Aufbau einer docker-compose.yml Datei

Zeile drei gibt an, dass „service-a" durch das Dockerfile im Verzeichnis „./some-service" gebaut werden soll. Mittels der „ports"-Anweisung kann man das Port Mapping des laufenden Containers beeinflussen. In diesem Fall wird der Port 8080 des Hosts auf den Port 80 des Containers gemappt. Ein Request, der an die Adresse „http://<host-ip>:8080" gerichtet ist sorgt demnach dafür, dass der Port 80 innerhalb des Containers angesprochen wird. Durch die „volumes"-Anweisung können Verzeichnisse, die sich auf dem Host-System befinden, innerhalb eines Containers verwendet werden. In Zeile sieben wird also das Verzeichnis „./service-a/src" im Verzeichnis „/src" des Containers verfügbar gemacht. Anstatt das Docker-Image selbstständig zu bauen, können auch bereits erstellte Images aus einer Registry geladen werden (siehe Zeile neun). In diesem Fall wird das Image „service-b" aus der Registry der Hochschule verwendet.

Da die Ausführung einiger Anwendungen, die Übermittlung von Umgebungsvariablen benötigen könnte, kann man diese mittels der „environment“-Anweisung übergeben. Nachdem die beschriebene „docker-compose.yml“-Datei erstellt wurde, kann die Anwendungslandschaft anschließend mittels des Befehls „docker-compose up“ gestartet werden. (vgl. Docker 2017a)

In diesem Fall erstellt das Tool zusätzlich ein Default-Netzwerk für die gesamte Anwendungslandschaft. Der Name des Netzwerkes ergibt sich dabei per Default aus dem Namen des Verzeichnisses, in dem die „docker-compose.yml“-Datei liegt. Das Tool sorgt dadurch dafür, dass eine gewisse Art der „Service-Discovery“ umgesetzt wird, in dem es die verschiedenen Container im Netzwerk für andere Container unter entsprechenden Hostnamen erreichbar macht, die den Service Namen entsprechen. Möchte der Service „service-b“ also z.B. mit dem Service „service-a“ kommunizieren, reicht ein Request an „http://service-a:80“ aus. Neben dem Default-Netzwerk stellt das Tool allerdings auch weitere Möglichkeiten zur Verfügung, um komplexere Netzwerke aufzubauen. Für diese Arbeit reicht die Default Lösung aber aus.

3 Anforderungsanalyse

Szenario

Seitdem der deutsche Energiemarkt im Jahr 1998 liberalisiert wurde, können die Verbraucher ihre Lieferanten frei wählen (vgl. BGBL 1998). Eine Langzeitstudie des BDEW zeigt, dass die Wechselquote in den Jahren von 2001 bis 2015 von 4.3% auf 39,5% gestiegen ist (vgl. BDEW 2002, S. 4 und BDEW 2015, S. 9). Das fiktive Unternehmen „Greenworld Energies“ ist Neueinsteiger auf dem Energieanbieter-Markt und möchte sich als feste Größe etablieren. Da die Anzahl von Energieanbietern seit der Liberalisierung stark anstieg, ist der Markt umkämpft. Um sich im Wettbewerb behaupten zu können, sollen bisherige Geschäftsprozesse nun durch eine moderne Webanwendung unterstützt und abgelöst werden. Wichtig ist dem Unternehmen die technische Unterstützung der „Zählerstandsermittlung“, der Gestaltung „attraktiver Tarife„ und des „Bestellprozesses“, um den Vertragsabschluss für Kunden interessanter zu machen.

In diesem Kapitel werden nun die Anforderungen an die zu entwickelnde Anwendungslandschaft ermittelt und in Form von funktionalen und nicht funktionalen Anforderungen näher spezifiziert. Aus diesem Grund soll zunächst der IST-Zustand verschiedener Geschäftsprozesse des Unternehmens aufgenommen werden. Die funktionalen Anforderungen beschreiben anschließend die Aufgaben, die die Systeme der Anwendungslandschaft für die Umsetzung der Geschäftsprozesse erfüllen sollen, während die nicht funktionalen Anforderungen verschiedene Qualitätseigenschaften der Lösung beschreiben. Anhand der Anforderungen kann zum Schluss der Arbeit nachvollzogen werden, ob die Lösung erfolgreich ist oder nicht.

3.1 Begriffsbestimmungen

Für das Verständnis der nachfolgenden Geschäftsprozesse müssen einige Begrifflichkeiten bestimmt werden.

3.1.1 Tarife

Tarife sind Bestandteile von Verträgen und geben an, zu welchem Preis ein Kunde eine entsprechende Leistung von einem Anbieter erhält. Im Falle eines Energieanbieters bestimmen Tarife, zu welchem Preis ein Kunde (Letztverbraucher) mit Strom beliefert wird und können dementsprechend als das Produkt angesehen werden, mit dem ein Energieanbieter handelt. Wie bereits in Kapitel 2.1.1 beschrieben, lassen sich Letztverbraucher je nach Stromverbrauch in mehrere Kategorien einteilen, den Großkunden und den Haushaltskunden. „Der durchschnittliche Tagesverbrauch der Kunden ist hierbei von Bedeutung, da diese in bestimmten Tageszeitspannen unterschiedlich viel Strom beziehen und dieser Strom zum einem physikalisch bereitgestellt sowie auch andererseits von Energieversorgungsunternehmen eingekauft werden muss." (Aichele 2012, S. 237) Aufgrund der unterschiedlichen Anforderungen der Kunden sind unterschiedliche Tarifstrukturen notwendig.

Laut Christian Aichele bestehen Stromtarife häufig aus einer grundlegenden „Kombination von nutzungsunabhängigen und nutzungsabhängigen Preiskomponenten, wobei eine weitverbreitete Methode die Gestaltung von zweiteiligen Tarifen ist". (vgl. Aichele 2012, S. 238) Bei einem klassischen **Basistarif** umfasst die nutzungsunabhängige Komponente dabei einen **Grundpreis**, der meistens fixe Verwaltungskosten des Unternehmens abdecken soll sowie einen nutzungsabhängigen **Arbeitspreis**, der als „Preis je verbrauchte Mengenheit in kWh angegeben wird". Bei einem **Volumentarif** wird neben einem Grundpreis ein **Volumenpaket in kWh** angeboten. Das Paket umfasst eine festgelegte Menge an Strom in kWh, „für eine bestimmte Tariflaufzeit zu einem hierfür definierten **Volumenpreis**". Im Falle, dass der Verbraucher das gesamte Volumen vor Ablauf der Tariflaufzeit verbrauchen sollte, „wird jede über das festgesetzte Volumen hinausgehende verbrauchte kWh mit einem sogenannten **Mehrarbeitspreis** abgerechnet". Im Falle eines Stromverbrauches von „über 30.000 kWh jährlich pro Zähler" und sofern der Kunde über eine „viertelstündige Leitungsmessung" verfügt, können Großkunden von Stromanbietern zudem ein „individuelles Angebot für ihren Stromverbrauch anfordern". (Aichele 2012, S. 242) Allerdings haben Tarifstrukturen von Gewerbetarifen in der Praxis „meist zu einem Großteil die gleichen Preisbestandteile und Merkmale wie die Tarifstrukturen, die für Haushaltskunden angeboten werden". (Aichele 2012, S. 242)

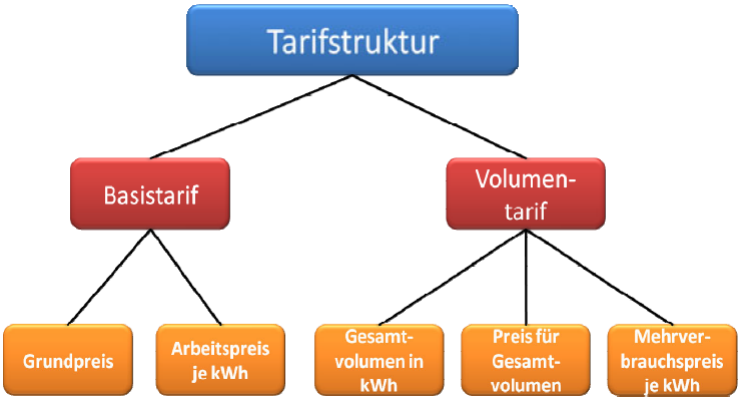

Abbildung 3.1: Preiskomponenten von Basis- und Volumentarifen
Quelle: Aichele 2012, S. 240

Interessanter werden in Zukunft **Dynamische Tarife**, bei denen Strompreise je nach Tageszeit oder aktueller Last des Verbrauchers variieren können, um „den Stromverbrauch der Endkunden einerseits zu verringern sowie andererseits den stattfindenen Verbrauch in andere tageszeitbezogene Zeitzonen zu verlagern“. (Aichele 2012, S. 245) Auf diese Weise könnten die Netzauslastung durch ein weiteres Mittel besser gesteuert werden und Anbieter kundenspezifischere Angebote bereitstellen, um sich von anderen Anbietern auf dem Markt zu unterscheiden. Da für die Umsetzung flächendeckend „intelligente Stromzähler“ benötigt werden und die energiewirtschaftlichen Rahmenbedingungen, zum Zeitpunkt der Veröffentlichung von Christian Aicheles Arbeit, wenig Möglichkeiten für die Gestaltung dynamischer Tarife geboten haben und es auch zum Zeitpunkt dieser Arbeit am Markt keine weite Verbreitung gibt, gilt es noch abzuwarten, wie sich dies in den nächsten Jahren entwickelt. (Aichele 2012, S. 246)

Es ist nicht nur durch dynamische Tarife möglich, Preisstrukturen individueller zu gestalten. Ein weit verbreitetes Mittel ist die Verwendung von **Tarifboni** und **Preisgarantien**, durch die die Kunden spezielle Vorteile erhalten. Tarifboni lassen sich im Grunde in drei verschiedene Arten kategorisieren. Die erste Art besteht aus einmaligen Gutschriften für Kunden, in der zweiten Art werden Volumenboni in Form von kostenlosen Kilowattstunden angeboten und die dritte Art besteht aus andauernden Rabatten. Dies können Rabatte auf den monatlichen Gesamtpreis (mGp) sein oder pro Kilowattstunde. Mit Hilfe einer Preisgarantie kann den Kunden zugesichert werden, dass sich der vertraglich festgelegte Energiepreis bis zu einem bestimmten Zeitpunkt nicht mehr ändern wird, auch wenn sich die Kosten (Stromeinkaufspreis, EEG-Umlage) für das Unternehmen in der Zeit verändern.

3.1.2 Entnahmestelle

Als Entnahmestelle wird die „Abnahmestelle mit all ihren physikalischen Messeinrichtungen, über die Energie eingespeist bzw. entnommen werden kann“ bezeichnet. „Eine Entnahmestelle wird durch eine Zählpunktbezeichnung definiert, die - solange die Entnahmestelle existiert - nicht mehr verändert wird.“ (BNetzA 2012, S. 4) Sie besteht immer aus einem Zählpunkt und wird für die Abrechnung des Stromverbrauches verwendet.

3.1.3 Stromliefervertrag

Obwohl es unterschiedliche Arten von Lieferverträgen gibt, wird der All-inclusive-Vertrag im Folgenden verallgemeinert als Stromliefervertrag (SLV) bezeichnet. Ein Stromliefervertrag wird zwischen einem Lieferanten und einem Kunden abgeschlossen und enthält laut K. Panos Details über die Liefer- und Messspannung in Volt, der Anschrift der Übergabestelle, der Preisregelung einschließlich der Preisänderung, der Entgelte für Messung, Datenbereitstellung und Abrechnung, der Abgaben und Steuern, der Abrechnung, sowei der Vertragslaufzeit und der Kündigung. (Panos 2007, S. 51) Die Preisstruktur eines Stromliefervertrages wird häufig durch Tarife definiert.

3.1.4 Lieferantenrahmenvertrag

„Ein Lieferantenrahmenvertrag (RV) wird zwischen dem Lieferanten und dem Netzbetreiber abgeschlossen und regelt alle Rechten und Pflichten in Zusammenhang mit der Belieferung der Kunden des Lieferanten in Netzgebiet der Netzbetreibers. Darin wird u.a. der Umfang in Form von Fahrplänen sowie Datenaustausch zwischen dem Lieferanten und der Verteilnetzbetreiber geregelt, und die Abrechnung von Netznutzungsentgelten und Messentgelt vereinbart. Der Lieferant ist i.d.R. bilanzkreisverantwortlich.“ (Panos 2007, S. 41)

3.1.5 Kundendaten

Als Kundendaten werden im Folgenden alle kundenspezifischen, für das Unternehmen relevanten Daten zusammengefasst. Dazu gehören Kontaktdaten wie Vorname, Nachname, Email-Addresse, Telefonnummer, Liefer- und Rechnungsaddresse sowie die Bankverbindung. Diese Daten werden vom Unternehmen gespeichert und für weitere Prozesse wie dem Lieferbeginn oder der reinen Kontaktaufnahme benötigt.

3.2 Ist-Zustand: Beschreibung der Geschäftsprozesse

Die Umsetzung der Anwendung erfordert die technische Unterstützung von Geschäftsprozessen des Unternehmens. Dazu gehört in erster Linie die Unterstützung des aus mehreren Subprozessen bestehenden „Bestellprozesses“, aber auch die Unterstützung des „Tariferstellungs-Prozesses“ sowie des Prozesses „Zählerstand- / Zählerwertermittlung“. Nicht betrachtet werden an dieser Stelle Abrechnungsprozesse, wie die Netznutzungsabrechnung oder Prozesse, die sich mit dem Einkauf von Elektrizität oder dem Erstellen von Verbrauchsprognosen für den Bilanzkreisausgleich auseinandersetzen. Die beschriebenen Prozesse decken demnach nur einen Bruchteil der Prozesse ab, die von einem Lieferanten umgesetzt werden müssen. Unter anderem sind die Prozesse aus den gesetzlich festgelegten **„Geschäftsprozessen zur Kundenbelieferung mit Elektrizität“ (GPKE)** (BNetzA 2012) sowie zum Teil aus der Arbeit von Christian Aichele abgeleitet worden (Aichele 2012). Ferner wurden die Prozesse angepasst, vereinfacht oder erfunden. Ziel dieser Arbeit ist es, eine realistische, gewachsene Anwendungslandschaft zu erstellen, die innerhalb des Projektkontexts ausgebaut werden kann. Damit die softwareunterstützte Umsetzung der Geschäftsprozesse erfüllbar bleibt und das Projekt erfolgreich abgeschlossen werden kann, werden diese minimal gehalten und haben nicht den Anspruch der Vollständigkeit.

3.2.1 Bestellprozess Strom

Der „Bestellprozess Strom“ ist einer der wichtigsten Geschäftsprozesse des Unternehmens, da es nur dann Gewinne machen kann, wenn Kunden einen Stromliefervertrag abschließen und vom Unternehmen mit Strom beliefert werden. Abbildung 3.2 auf der nächsten Seite zeigt, dass sich der Bestellprozess aus mehreren Prozessen zusammensetzt. Die genaueren Beschreibungen der Sub-Prozesse erfolgt auf den nachfolgenden Seiten.

Ausgangspunkt des Prozesses ist der Wunsch eines Kunden, mit Strom beliefert zu werden. Grund hierfür kann z.B. der Umzug in eine neue Wohnung oder der Wunsch nach einem günstigeren Tarif sein. Zu diesem Zweck verwenden Kunden häufig verschiedene Online-Vergleichsportale, die es ermöglichen, Anbieter und Tarife von Anbietern übersichtlich und schnell zu vergleichen. Das Unternehmen „Greenworld Energies“ bietet unabhängig von den Vergleichsportalen einen eigenen Dienst an, über den Kunden Tarifangebote erhalten. Dieser wird im „Angebotsbereitstellungs-Prozess“ näher beschrieben. Da ein Vertrag immer für einen verfügbaren Tarif abgeschlossen werden muss, setzt der Vertragsabschluss diesen Prozess voraus. Das Unternehmen wertet dabei die Daten des Kunden aus und liefert entsprechende

Tarifangebote. Hat sich ein Kunde für einen Tarif entschieden, kann er den „Stromliefervertrag“ ausfüllen und unterschrieben an das Unternehmen zurücksenden. Mit dem Erhalt des Vertrags wird der Geschäftsprozess „Vertragsabschluss“ ausgelöst. Wenn dabei keine Fehler aufgetreten sind und die Entnahmestelle des Kunden erfolgreich beim Netzbetreiber angemeldet werden konnte, erhält der Kunde eine Vertrags- und Lieferbestätigung und kann entsprechend der vertraglich festgelegten Konditionen zum Lieferbeginn mit Strom beliefert werden.

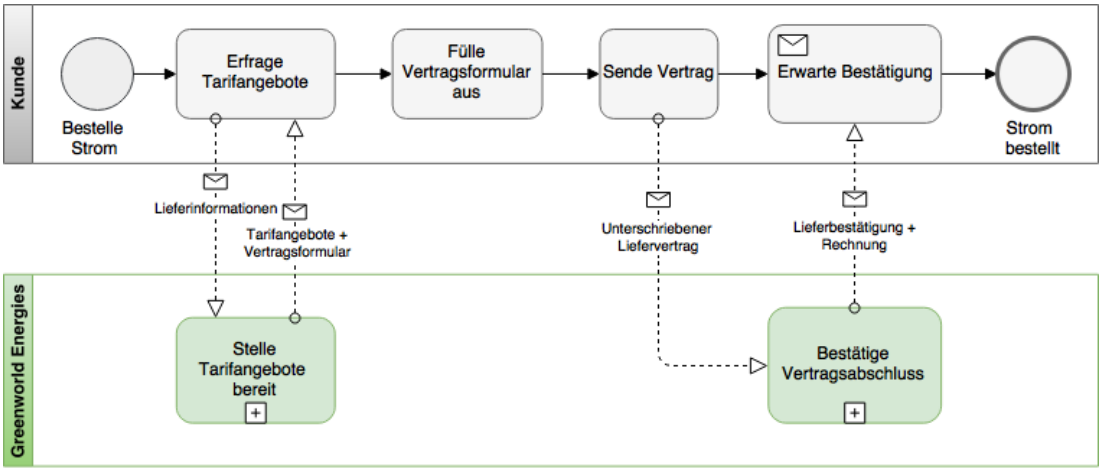

Abbildung 3.2: Bestellprozess Strom

3.2.2 Angebotsbereitstellung: Haushaltskunde

Der Prozess „Angebotsbereitstellung“ erspart den Kunden den Umweg über ein Vergleichsportal und ermöglicht ihnen, Tarifangebote einzuholen. Für eine korrekte Bearbeitung werden vom Kunden sowohl die Postleitzahl der Entnahmestelle, als auch der geschätzte Strom-Jahresverbrauch benötigt. Betrachtet werden in diesem Fall nur Haushaltskunden mit einem Strom-Jahresverbrauch, der 10.000 kWh nicht überschreitet. (Bei Großkunden, mit mehreren Tausend Kwh pro Jahr, wären individuelle Tariffverhandlungen möglich, die weitere Paramter benötigen. Für das Ziel dieser Arbeit reicht jedoch die Betrachtung von Haushaltskunden aus.) Die Verkaufsabteilung des Unternehmens nimmt die Anfrage vom Kunden entgegen und sucht daraufhin in der bisher verwendeten Excel-Tabelle nach entsprechenden Tarifen. Wenn keine Tarife gefunden wurden, kann die Entnahmestelle des Kunden nicht beliefert werden. Dies kann zum einen daran liegen, dass das Unternehmen aktuell nicht in der Lage ist, die gewünschte Region zu beliefern oder bestimmte Regionen nicht beliefert werden sollen. In

beiden Fällen wird der Kunde mit einer Nachricht darüber in Kenntnis gesetzt. Wurden Tarife gefunden, wird der Stromliefervertrag vorbereitet und zusammen mit den Tarifen an den Kunden gesendet. Da dieser manuelle Umweg über eine Verkaufsabteilung und eine Exceltabelle allerdings sehr zeitaufwendig ist, soll dieser Prozess durch die neue Webanwendung und einem neuen Tarifsystem automatisiert werden.

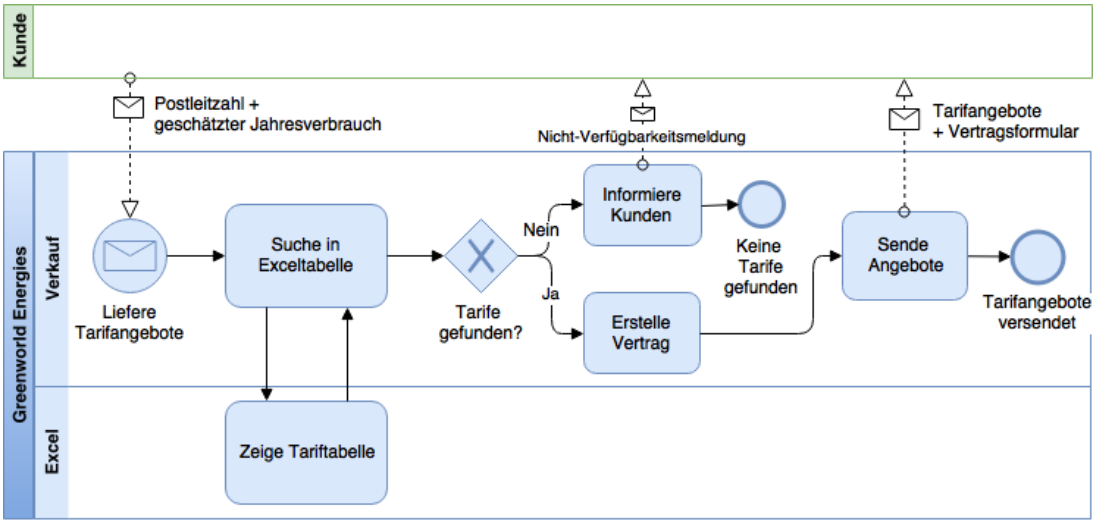

Abbildung 3.3: Ablauf der Angebotsbereitstellung

3.2.3 Vertragsabschluss

Der „Vertragsabschluss-Prozess“ sorgt dafür, dass der Kunde als Kunde registriert wird und entsprechend der vertraglich festgelgten Konditionen mit Strom beliefert wird. Er besteht wie der Bestellprozess aus diversen Subprozessen, die allerdings nicht alle im Detail umgesetzt werden sollen. Für das Verständnis der Subprozesse reicht die jeweilige Beschreibung als Black-Box an dieser Stelle deswegen aus.

Ausgangspunkt des Vertragsabschluss-Prozesses ist der Erhalt eines vom Kunden unterschriebenen „Stromliefervertrags“. Infolgedessen wird der Vertrag zunächst in der Verkaufsabteilung des Unternehmens überprüft. Dazu gehört die gültige und vollständige Angabe der Kundendaten sowie der Verweis auf den angebotenen Tarif, Adresse und Zählpunktbezeichnung der Entnahmestelle, der gewünschte Lieferbeginn und der geschätzte Jahresverbrauch. Zu diesem Zweck vergleicht der Mitarbeiter die Daten mit entsprechenden Exceltabellen. Wenn die Daten vollständig übermittelt wurden und gültig sind, überprüft die Finanzabteilung die Kreditwürdigkeit des Kunden. Erst nachdem diese Überprüfung ebenfalls erfolgreich war, wird

der Kunde akzeptiert und die Entnahmestelle beim Netzbetreiber angemeldet. Das Anmelden der Entnahmestelle ist durch den Prozess „Lieferbeginn“ gesetzlich von der Bundesnetzagentur am 11.07.2006 unter den „Geschäftsprozessen zur Kundenbelieferung mit Elektrizität, GPKE“ vorgestellt und beschlossen worden. (vgl. BNetzA 2012) Gleichzeitig wird die Rechnung für die Kosten der Anmeldung der Entnahmestelle erstellt. Nachdem der Lieferbeginn angemeldet wurde, wird der Vertragsabschluss validiert und der Kunde gilt als registriert. Das Erstellen der Rechnung und die Validierung des Vertragsabschlusses wird vom Unternehmen bisher ebenfalls durch Exceltabellen realisiert, während die Kundenverwaltung von einem CRM-System vorgenommen wird. Wenn alles erfolgreich abgelaufen ist und keine Probleme aufgetreten sind, wird dem Kunden die Rechnung zugesendet und der Prozess gilt als abgeschlossen. Ansonsten wird der Kunde bei einem Misserfolg über den aufgetreten Fehler in Kenntnis gesetzt. Fehler können zum Beispiel auftreten, wenn die Entnahmestelle dem Netzbetreiber unbekannt, der übermittelte Vertrag ungültig oder der Kunde nicht kreditwürdig ist.

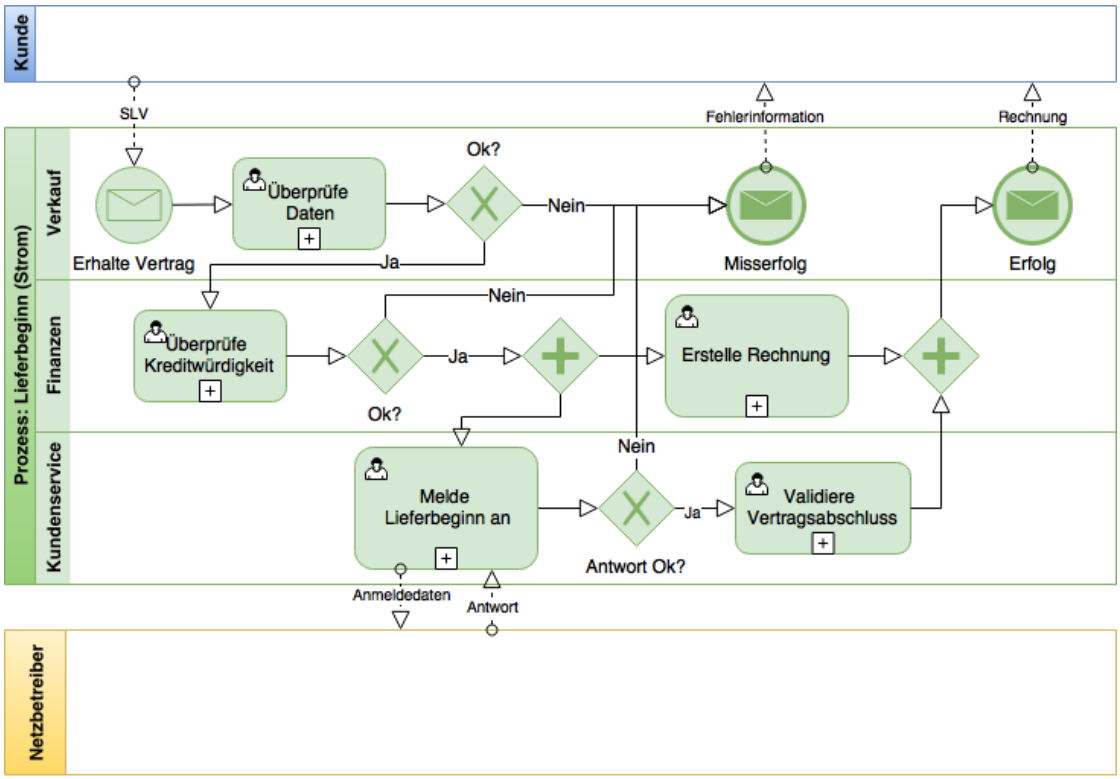

Abbildung 3.4: Ablauf des Vertragsabschlusses

3.2.4 Tariferstellung

Die Tariferstellung ist ein unternehmensinterner Prozess, der das Ziel verfolgt möglichst schnell neue Tarife auf den Markt zu bringen. Der hier beschriebene Prozess orientiert sich an dem Forschungsprojekt „e-configurator“, einem gemeinsamen Projekt der „movento GmbH [...] sowie des Fachbereich Betriebswirtschaft der Fachhochschule Kaiserslautern“, das in der Arbeit von Christian Aichele beschrieben wird und ergänzt den Ansatz um die Bindung von Tarifen an Regionen. (siehe Aichele 2012, S. 250) Betrachtet wird an dieser Stelle ausschließlich die Erstellung von klassischen Tarifen. Demnach lässt sich die Erstellung in mehrere Schritte unterteilen.

1. Wähle Tariftyp
   Ausgangspunkt ist die Aufgabe eines Mitarbeiters, einen neuen Tarif zu erstellen. Hierfür wird zunächst der Tariftyp ausgewählt. Dies kann zum Beispiel ein „Basis Tarif“, ein „Volumentarif“ oder die Erstellung eines neuen Tariftyps sein. Der Tariftyp stellt eine eindeutige Identifikation des Tarifes dar.

2. Wähle Preisstruktur
   Jeder Tariftyp besitzt eine Preisstruktur. Die Preiststruktur setzt sich aus einer Menge an Komponenten zusammen, die kombiniert werden können. Dazu gehören unter anderem die Komponenten „Grundpreis“, „Arbeitspreis“, „Volumen in kWh“ und „Mehrarbeitspreis“. Da für diese Arbeit nur die klassischen Tarife „Basis Tarif“ und „Volumentarif“ benötigt werden, reichen diese vier Komponenten bereits aus, um alle benötigten Tarife darstellen zu können. Arbeits- und Grundpreise bei „Hoch- und Niedertarifen“ wären an dieser Stelle auch denkbar, können aber in dieser Arbeit ignoriert werden.

3. Wähle optionale Preise
   Im nächsten Schritt werden optionale Preise hinzugefügt. Dies kann eine Kaution, eine Preisgarantie oder eine Vorauskasse sein. Da Anbieter häufig eine Mindestvertragslaufzeit angeben, um Kunden für längere Zeit an sich zu binden, kann diese ebenfalls an dieser Stelle ausgewählt werden.

4. Wähle Tarifboni
   Durch Tarifboni haben Unternehmen die Möglichkeit sich abgesehen von den Preisen von anderen Anbietern abzuheben. Tarife können auf diese Weise für Kunden interessanter gemacht werden. Boni können zum Beispiel ein einmaliger „Geld-“ oder ein „Volumenbonus“ sein.

5. Wähle Regionen
   Der letzte Schritt wurde in dieser Arbeit ergänzt und beschreibt die Bindung von Tarifen an Regionen. Da sich Strompreise häufig regional stark unterscheiden, sollen Tarife auch regional unterschiedlich angeboten werden können. Jeder Tarif kann damit in mehreren Regionen angeboten werden, die zusammen seinen Gültigkeitsbereich darstellen.

Abbildung 3.5 veranschaulicht den Prozess und zeigt, dass Tarife innerhalb der aktuellen Systemlandschaft ausschließlich in Excel-Tabellen und manuell durch die Mitarbeiter des Unternehmens gespeichert werden. Ein Zustand, der nun durch die neue Webanwendung und die Integration eines neuen Tarifsystems abgelöst und automatisiert werden soll.

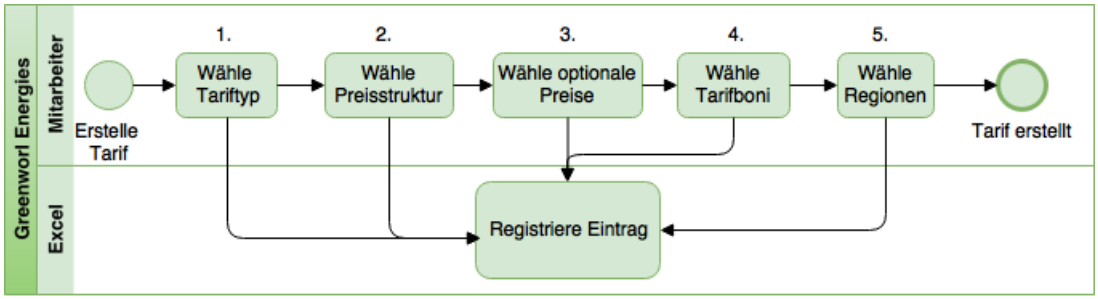

Abbildung 3.5: Tariferstellung

Ein exemplarischer Basistarif könnte demnach wie folgt strukturiert sein:

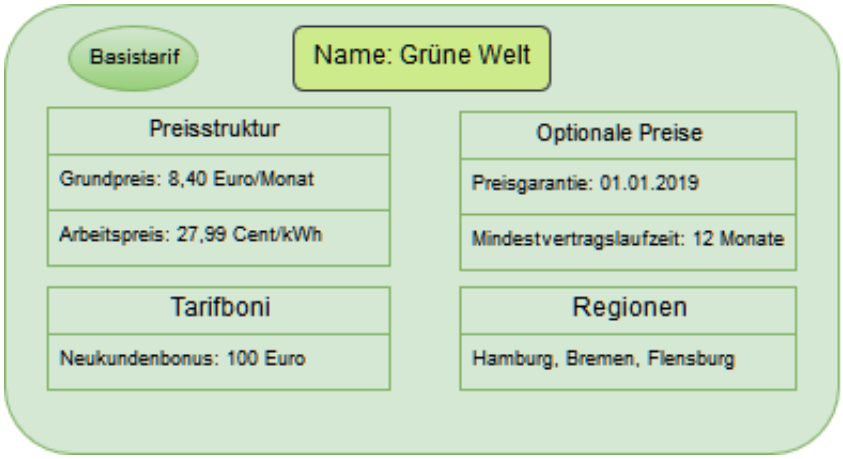

Abbildung 3.6: Darstellung des exemplarischen Tarifs „Grüne Welt“

3.2.5 Zählerstand- / Zählerwertübermittlung

Der Prozess „Zählerstandübermittlung“ wurde ebenfalls gesetzlich von der Bundesnetzagentur als einer der Geschäftsprozesse zur Kundenbelieferung mit Elektrizität (GPKE) beschlossen und beschäftigt sich mit der Übermittlung, der für die Netznutzungsabrechnung benötigten Stromzählerstände. (vgl. BNetzA 2012) Die Zählerstände können dabei auf drei unterschiedliche Arten ermittelt werden. Im ersten Fall handelt es sich um eine außerturnusmäßige Ablesung oder die Änderung des Ableseturnus. In beiden Fällen beauftragt der Lieferant (LR) den Netzbetreiber (NB) damit, den aktuellen Stromzählerstand einer Entnahmestelle zu übermitteln. Die Erhebung der Messwerte kann vom NB selbständig oder von einem entsprechenden Messstellendienstleister (MDL) durchgeführt werden. Im zweiten Fall werden die Messwerte entweder durch den MDL nach der Erhebung aufgrund eines vereinbarten Ableseturnus direkt an den Lieferanten übermittelt und im dritten Fall werden die Messwerte unmittelbar vom Kunden durch die Selbstablesung ermittelt.

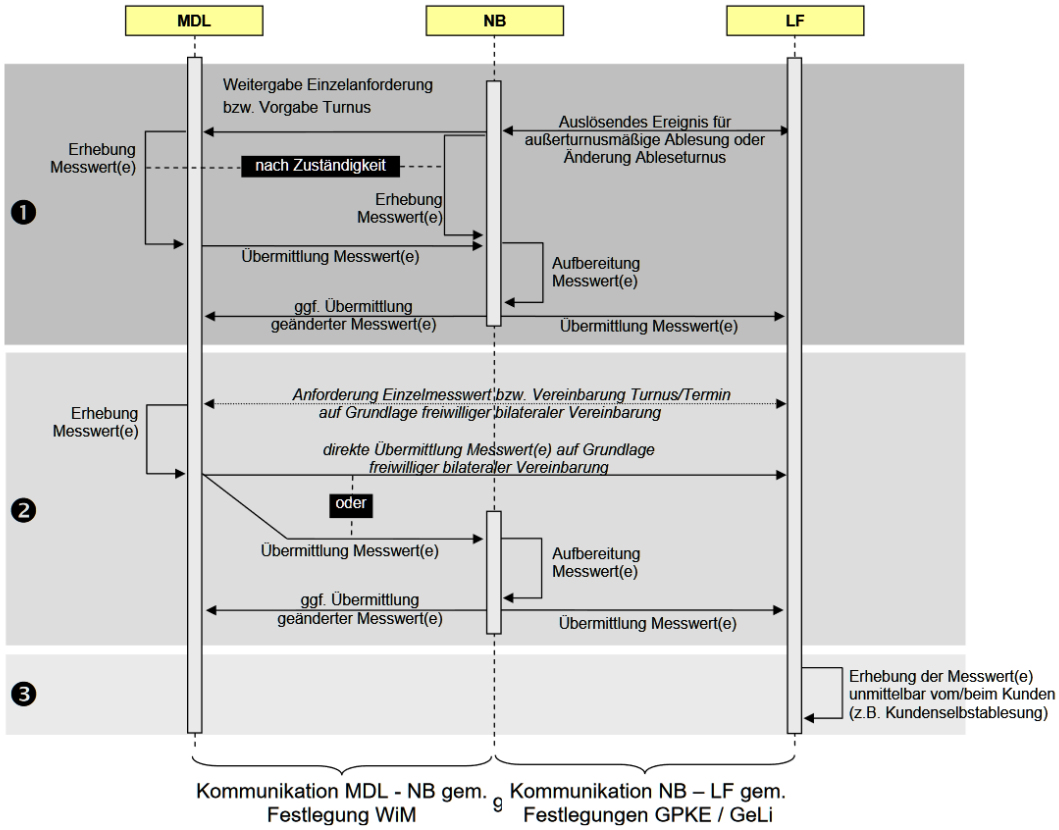

Abbildung 3.7: Darstellung der Zählerstandermittlung
Quelle: BNetzA 2012, S. 39

3.3 Funktionale Anforderungen

Um die vorgestellten Prozesse durch Software zu unterstützen, soll die alte Systemlandschaft bestehend aus Exceltabellen, dem Customer Relationship Management System „SugarCRM“ und einer rudimentären statischen Webseite durch neue Systeme und Funktionen ersetzt und erweitert werden. Dazu gehört ein neues Tarifsystem für die Erstellung von Tarifen, eine Benutzerverwaltung für die Verwaltung der Benutzer der neuen Anwendungslandschaft, ein Energiedaten Management System für die Erfassung des Stromverbrauches sowie eine neue Webanwendung, die als Benutzerschnittstelle für Kunden und Mitarbeiter dienen soll. Einige der Systeme werden komplett neu entwickelt, andere wiederum werden erweitert oder müssen in die Anwendungslandschaft integriert werden. An dieser Stelle sollen die Aufgaben der Komponenten nicht alle im Detail beschrieben werden. Vielmehr dient dieses Kapitel dazu, einen groben Überblick über die verschiedenen Komponenten und ihre Aufgabengebiete zu erhalten. Aus dem Grund werden die wesentlichen Aufgaben der einzelnen Komponenten im Folgenden nur kurz zusammengefasst. Ausnahmen bilden die Benutzerverwaltung und die Webanwendung, da die Anforderungen der beiden Komponenten komplexer sind und als Referenz dienen sollen.

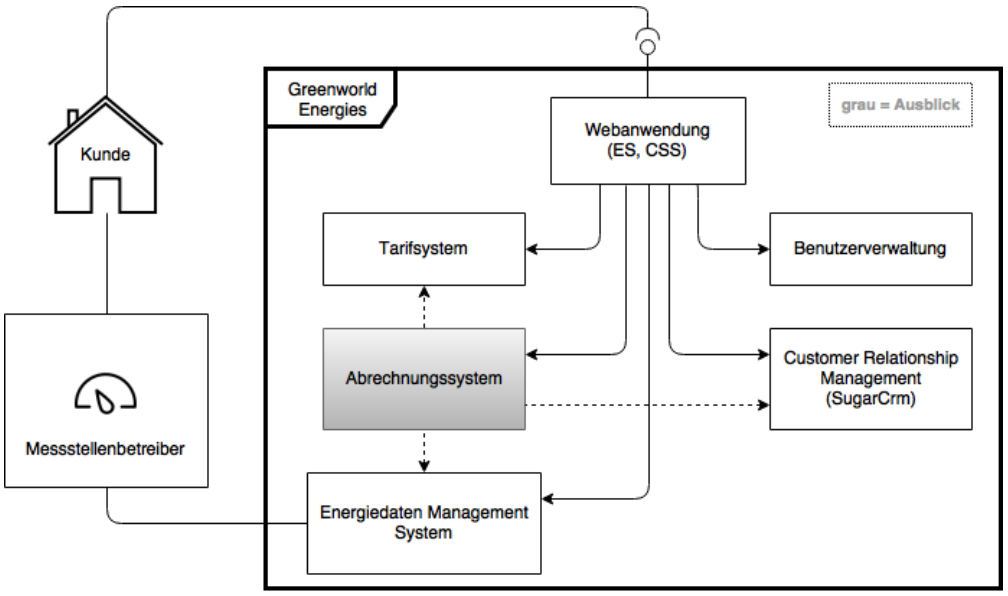

Abbildung 3.8: Übersicht der Systemlandschaft

Das Tarifsystem

Das Tarifsystem soll sowohl für die Erstellung als auch für die Verwaltung von Stromtarifen verwendet werden und die Hoheit über sämtliche Tarifpreise besitzen. Das System soll neu entworfen werden und die bisher verwendeten Exceltabellen ersetzen.

Das Energiedaten Management System (EDM)

Das Energiedaten Management System (kurz EDM) soll den tatsächlichen Stromverbrauch der Kunden verwalten. Die entsprechenden Daten dafür werden durch den Messstellenbetreiber und den Kunden selbst ermittelt und können vom Abrechnungssystem für Abrechnungen verwendet werden. (siehe 3.2.5) Für die Umsetzung soll die Komponente als Prototyp entwickelt werden, der dann zu einem späteren Zeitpunkt durch ein echtes EDM-System ersetzt werden kann. Wichtig ist zunächst, dass Kunden ihre Stromzählerstände aktualisieren können und dass gespeicherte Zählerstände ausgelesen werden können.

Das Abrechnungssystem

Das Abrechnungssystem kann in Zukunft für das Erstellen und Verwalten von Rechnungen verwendet werden und soll die Abrechnung des Stromverbrauches realisieren. Zu diesem Zweck werden in Zukunft vom Tarifsystem die tariflich festgelegten Preise und vom EDM-System der tatsächliche Verbrauch des Kunden benötigt. Obwohl das System in dieser Arbeit nicht umgesetzt werden soll, kann es dennoch sinnvoll sein es an dieser Stelle bereits zu erwähnen, um den Kontext der zu entwickelnden Systeme verständlicher zu machen. Durch die Integration wäre die Anwendungslandschaft vorerst vervollständigt.

Das Customer Relationship Management - System (CRM)

„SugarCRM“ ist das bisher vom Unternehmen verwendete Customer Relationship Management - System (dt. Kundenbeziehungsmanagement) und soll weiterhin verwendet werden. Ziel eines solchen Systems ist „die Steigerung des wirtschaftlichen Erfolgs über den Aufbau und die Pflege profitabler Kundenbeziehungen.“ (Aichele 2012, S. 180) Während die verwendete Version von „SugarCrm“ derzeit lediglich als Kundenverwaltungssystem genutzt wird und Kunden- sowie Vertragsdaten speichern soll, bieten neuere Versionen und andere CRM Systeme mehr Funktionalitäten. So wäre es denkbar, jede Transaktion eines Kunden zu speichern, um Kundenbedürfnisse, -verhalten und -motivationen analysieren zu können. Die durch die Analyse entstehenden Ergebnisse könnten dann von der Verkaufsabteilung für Marketingzwecke

genutzt werden, um den Gewinn des Unternehmens zu steigern. Des Weiteren könnten Mitarbeitern Aufträge, in Form von Tasks, zugewiesen werden, wordurch die Organisation im Team erleichtert werden kann. Obwohl das Unternehmen langfristig auf eine CRM-Strategie aufbauen möchte, reicht die aktuelle Version zunächst aus, um die vorgestellten Geschäftsprozesse zu realisieren. Zu diesem Zweck muss das System lediglich um die Speicherung und Verwaltung von Stromlieferverträgen erweitert werden und in die restliche Anwendungslandschaft integriert werden.

Die Benutzerverwaltung

Um eine grundlegende Sicherheit innerhalb der Systemlandschaft zu ermöglichen, ist ein Sicherheitsmechanismus notwendig, der Zugriffe auf Ressourcen authorisiert. Kunden sollten zum Beispiel nicht ohne weiteres einen sicherheitskritischen Bereich, wie den Kundenbereich eines anderen Kunden, betreten können. Aus diesem Grund soll eine Benutzerverwaltung eingeführt werden, die den einzelnen Benutzern der Anwendung verschiedene Rechte gewährt. Umgesetzt werden soll dies durch die Einführung eines Rollenmodells.

Das Modell wurde für diese Arbeit frei erfunden. Ähnlichkeiten zu anderen Modellen sind aber wahrscheinlich, da Rollen im „Identity- und Access- Management“ häufig Verwendung finden und es viele Arbeiten gibt, wie die Arbeit „Rollen und Berechtigungskonzepte - Elemente zur Berechtigungssteuerung“ von A. Tsolkas und K. Schmidt, die sich mit der Thematik auseinandersetzen. Auch hat dieses Modell nicht den Anspruch, die Sicherheit innerhalb der Anwendungslandschaft allumfassend zu gewährleisten. Es soll lediglich eine vereinfachte Lösung darstellen, die zu einem späteren Zeitpunkt durch eine bessere ersetzt werden könnte.

Durch das Modell soll jedem Benutzer der Systemlandschaft eine Rolle zugewiesen werden. Jede Rolle besitzt verschiedene Zugriffsrechte. Benutzer werden im System als **User** eingeführt, die **Aktionen** in der Systemlandschaft ausführen können. Jedem User kann genau eine **Rolle** zugewiesen werden. Basierend auf der Rolle darf er bestimmte Aktionen im System ausführen und andere wiederum nicht. Grundlegend lassen sich die Rollen in **Besucher, Kunde, Mitarbeiter** und **Administrator** unterteilen, wobei die Rolle Mitarbeiter noch spezifischer sein kann und sich von der Unternehmensstruktur von „Greenworld Energies“ ableiten lässt.

Die Unternehmensstruktur von „Greenworld Energies“ besteht aus diversen Abteilungen, dessen Angestellte jeweils unterschiedliche Aufgaben übernehmen. An dieser Stelle werden vereinfacht nur die Abteilungen Verkauf, Finanzen und Kundenservice betrachtet. Jede Ab-

teilung besitzt spezifische Rechte. So darf ein Mitarbeiter der Verkaufsabteilung zum Beispiel keine Rechnungen erstellen und ein Mitarbeiter der Finanzabteilung darf keine Zählerstände im Namen eines Kunden aktualisieren. Die verschiedenen Abteilungen werden damit durch die Rollen **Mitarbeiter: Verkauf, Mitarbeiter: Finanzen** und **Mitarbeiter: Kundenservice** repräsentiert. Diese Rollen reichen derzeit für die Umsetzung der Geschäftsprozesse aus - weitere Unterteilungen sind allerdings denkbar und innerhalb eines realen Unternehmens wahrscheinlich.

Die Rollen sind hierarchisch aufgebaut und es gilt: Rollen, die in der Hierarchie über einer anderen Rolle stehen, übernehmen sämtliche Rechte, die die in der Hierarchie unter ihnen stehenden Rollen besitzen.

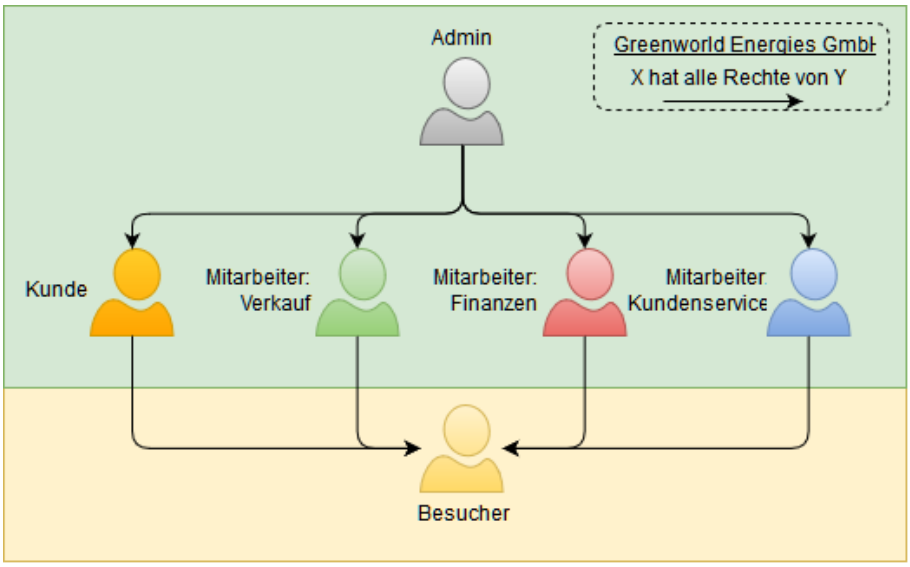

Abbildung 3.9: Übersicht des Rollenmodells

Administratoren besitzen die meisten Rechte im System. Dies ist notwendig, damit sie bei Problemen und Fehlern besser handeln können. User mit der Rolle „Besucher“ dürfen hingegen nur wenige Aktionen ausführen. Als Besucher werden alle Benutzer der Webanwendung bezeichnet, die nicht im System gespeichert sind, aber dennoch Zugriffsrechte besitzen. Besucher werden auch als „Potentielle Kunden“ bezeichnet und können zum Beispiel die Webanwendung nutzen, um Tarife zu suchen oder den Bestellprozess zu durchlaufen. Welche Rechte die verschiedenen Rollen allerdings genau besitzen, wird in dem nächsten Abschnitt näher beschrieben.

Die Webanwendung

Die neue Webanwendung ist die Benutzerschnittstelle der Systemlandschaft und soll sowohl einen sogenannten „Employee Service (ES)“, auch persönlicher Arbeitsbereich genannt, als auch einen „Customer-Self-Service (CSS)“, auch persönlicher Kundenbereich genannt, bereitstellen. Der ES wird von Mitarbeitern des Unternehmens verwendet. Eingeloggte Administratoren und Mitarbeiter können darüber die Entitäten des verteilten Systems verwalten und neue Tarife erstellen. Außerdem können sie einen vom Kunden erhaltenen Vertrag als validiert kennzeichnen. (siehe 3.2.3) Der CSS wird von Kunden und Besuchern verwendet. Besucher haben darüber die Möglichkeit, wie innerhalb des „Bestellprozesses“ beschrieben (siehe 3.2.1), neue Tarife zu finden und können entsprechende Stromlieferverträge abschließen. Registrierte Kunden können sich in ihren persönlichen Kundenbereich einloggen und dort ihre Vertragsdaten einsehen sowie ihre Stromzählerstände aktualisieren.

In diesem Abschnitt sollen nun die wichtigsten Anwendungsfälle detailierter betrachtet werden. Dazu gehört der Login, das Erstellen und Finden von Tarifen, das Durchlaufen des Bestellprozesses sowie das Aktualisieren der Zählerstande. Das Verwalten der Benutzer des Systems sowie das Verwalten der für die Tariferstellung benötigten Regionen und Tarifboni ist ähnlich aufgebaut und muss für das weitere Verständnis der Anwendung nicht unbedingt betrachtet werden.

(Hinweis: Die folgenden Tabellen können aufgrund ihrer Größe mehrere Seiten umfassen.)

Name:	UC2.01.
Beschreibung:	User mit der Rolle „Kunde“ können sich in der Webanwendung einloggen und gelangen so in ihren persönlichen Kundenbereich (CS).
Akteure:	User, Webanwendung, Benutzerverwaltung
Vorbedingungen:	Der User muss bereits ein Kundenkonto und ein entsprechendes Userkonto besitzen.
Nachbedingung:	Der User ist eingeloggt und gelangt in seinen persönlichen Kundenbereich (CS).

Standardablauf:	1. Der User öffnet die Homepage der Webanwendung. 2. Der User klickt auf den Button „Anmeldebereich“ und gelangt zu dem „Login-Formular“. 3. Der User füllt im Formular die Felder Email-Addresse und Passwort aus und klickt auf den Button „Login“. 4. Der Webserver der Webanwendung nimmt die Anfrage entgegen und leitet sie an die Benutzerverwaltung weiter. 5. Die Benutzerverwaltung prüft die Anfrage und gewährt den Zugriff. 6. Die Webanwendung aktualisiert die Seite und zeigt den persönlichen Kundenbereich an.

Tabelle 3.1: Login: Kundenbereich

Der Login der Mitarbeiter des Unternehmens erfolgt auf dieselbe Weise wie der Login der Kunden. Aus dem Grund kann der Anwendungsfall an dieser Stelle ignoriert werden. Der einzige Unterschied besteht darin, dass Mitarbeiter kein Kundenkonto benötigen und durch den Login in ihren persönlichen Arbeitsbereich gelangen und nicht in einen Kundenbereich.

Name:	UC2.02.
Beschreibung:	User mit den Rollen „Mitarbeiter: Verkauf“ oder „Administrator“ können den ES der Webanwendung verwenden, um Tarife zu erstellen.
Akteure:	User, Webanwendung, Tarifsystem (TS)
Abhängigkeiten:	UC2.01.
Vorbedingungen:	Der User muss entweder die Rolle „Mitarbeiter: Verkauf“ oder „Administrator“ besitzen und muss bereits im ES eingeloggt sein. Es müssen Regionen im System vorhanden sein.
Nachbedingung:	Der neue Tarif ist im TS gespeichert und kann durch den Angebotsbereitstellungs-Prozess gefunden werden.

Standardablauf:	1. Der Benutzer öffnet die „Tarifverwaltungs-Seite“ der Webanwendung. 2. Der Benutzer erhält eine Übersicht der vorhandenen Tarife und ein Formular zum Erstellen von neuen Tarifen. 3. Der Benutzer fügt im Formular die benötigten und innerhalb des „Tariferstellungsprozesses“ beschriebenen Informationen ein und klickt auf „Erstellen“. (siehe 3.2.4) 4. Das TS prüft die Anfrage und speichert den neuen Tarif. 5. Die Webanwendung aktualisiert die Seite und der neue Tarif wird in der Übersicht angezeigt.

Tabelle 3.2: Tarife erstellen

Das Erstellen von Regionen, Boni und anderen Entitäten verhält sich ähnlich wie das Erstellen von Tarifen. Auf eine detaillierte Beschreibung wird deswegen an dieser Stelle verzichtet.

Name:	UC2.03.
Beschreibung:	User können mit Hilfe der Webanwendung nach Tarifen suchen und erhalten eine Übersicht der verfügbaren Tarife.
Akteure:	User, Webserver, Tarifsystem (TS)
Abhängigkeiten:	Angebotsbereitstellung in 3.2.2.
Vorbedingungen:	Es müssen Tarife im TS vorhanden sein.
Nachbedingung:	Der User erhält eine Übersicht der verfügbaren Tarife.
Standardablauf:	1. Der User öffnet die Homepage der Webanwendung in seinem Browser. 2. Der User gibt im entsprechenden Suchformular die Postleitzahl der Entnahmestelle und den geschätzten Strom Jahresverbrauch an und sendet es ab. 3. Der Webserver nimmt die Anfrage entegegen und leitet sie an das TS weiter. 4. Das TS prüft die Anfrage und liefert die gefundenen Tarife. 5. Der Webserver nimmt die gefundenen Tarife entgegen und zeigt sie dem User in einer Tabelle an.

Tabelle 3.3: Tarife suchen

Name:	UC2.04.
Beschreibung:	User können mit Hilfe der Webanwendung Strom bestellen.
Akteure:	User: „Besucher“, Webanwendung, Benutzerverwaltung, EDM-System, SugarCrm
Abhängigkeiten:	Bestellprozess in 3.2.1, UC2.03.
Vorbedingungen:	Der User hat zuvor nach Tarifen gesucht und einen ausgewählt.
Nachbedingungen:	Der User wird als Kunde im System registriert und kann sich mit seinen Userdaten im Kundenbereich einloggen. Im Kundenbereich kann er den ausgefüllten Vertrag ausdrucken und ihn somit für den vollständigen Abschluss des Bestellprozesses unterschrieben an das Unternehmen senden.
Standardablauf:	1. Dem User wird das Bestellformular angezeigt. 2. Der User füllt die benötigten und innerhalb des Prozesses „Vertragsabschluss“ beschriebenen Informationen aus und sendet das Formular ab. (siehe 3.2.3) 3. Der Webserver der Webanwendung nimmt die Anfrage entgegen und bearbeitet sie. 3.1. Der Webserver legt bei der Benutzerverwaltung ein Userkonto mit der Rolle „Kunde“ an. 3.2. Der Webserver legt bei dem EDM-System einen neuen Stromzähler für das erstellte Userkonto an. 3.3. Der Webserver legt bei SugarCRM ein neues Kundenkonto für das erstellte Userkonto an, wodurch der Besucher als Kunde im System gespeichert ist. 3.4. Der Webserver speichert bei SugarCRM die Vertragsdaten für das neue Kundenkonto.

Tabelle 3.4: Strom bestellen

Durch das Absenden des Bestellformulars wurden alle benötigten Entitäten in den Systemen gespeichert. Allerdings fehlt dem Unternehmen für den vollständigen Abschluss des Bestellprozesses der vom Kunden handschriftlich unterschriebene Vertrag. Dieser muss wie im „Bestellprozess“ beschrieben (siehe 3.2.1) vom Kunden mit der Post an das Unternehmen gesendet werden. Dort nimmt ein Mitarbeiter den Vertrag entgegen und kann ihn nach erfolgreichem Anmelden der Entnahmestelle im CRM validieren.

Name:	UC2.05.
Beschreibung:	User mit der Rolle „Mitarbeiter: Kundenservice“ oder „Administrator“ können über die Webanwendung offene Verträge validieren.
Akteure:	User: „Mitarbeiter: Kundenservice“ / „Administrator“, Webanwendung, Benutzerverwaltung, SugarCrm
Abhängigkeiten:	Bestellprozess in 3.2.1, UC2.01. und UC2.04.
Vorbedingungen:	Ein Kunde hat Strom bestellt und den Vertrag unterschrieben an das Unternehmen gesendet. Die Entitäten „Userkonto“, „Kundenkonto“, „Stromzähler“ und der „Stromliefervertrag“ wurden bereits in den Systemen gespeichert. Die übermittelten Daten wurden erfolgreich von den entsprechenden Abteilungen des Unternehmens überprüft und die Anmeldung beim Netzbetreiber war ebenfalls erfolgreich.
Nachbedingungen:	Der Stromliefervertrag wurde validiert und der Kunde wird entsprechend der vertraglich festgelegten Konditionen mit Strom beliefert.
Standardablauf:	1. Der User loggt sich in seinem persönlichen Arbeitsbereich ein und öffnet die Kundenverwaltungsseite. 2. Der User klickt auf das gewünschte Kundenkonto und erhält vom Webserver eine Übersichtsseite des Kunden, auf der er, seinen Rechten entsprechend, Kundendaten, Verträge und Stromzähler einsehen kann. 3. Der User klickt auf den entsprechenden Vertrag und erhält vom Webserver die Vertragsinformationen. 4. Der User klickt auf den Button Vertrag validieren. 5. Der Webserver verarbeitet die Anfrage und speichert die Validierung bei SugarCRM.

Tabelle 3.5: Stromliefervertrag validieren

Name:	UC2.06.
Beschreibung:	User mit der Rolle „Kunde“ können in ihrem persönlichen Kundenbereich ihre Stromzählerstände aktualisieren.
Akteure:	User: „Kunde“, Webanwendung, SugarCrm, EDMS
Abhängigkeiten:	Bestellprozess in 3.2.1, UC2.01. und UC2.04.
Vorbedingungen:	Der Prozess „Strom bestellen“ wurde erfolgreich abgeschlossen.
Nachbedingungen:	Der Stromzählerstand wurde aktualisiert.
Standardablauf:	1. Der User loggt sich in seinem persönlichen Kundenbereich ein und öffnet die Seite „Meine Stromzähler“. 2. Der Webserver der Webanwendung kommuniziert mit SugarCrm, um herauszufinden für welche Stromzähler der Kunde einen Vertrag abgeschlossen hat und beschafft sich die entsprechenden Stromzählerstände vom EDMS. 3. Der User erhält eine Übersicht aller Stromzähler, für die der Kunde einen Stromliefervertrag abgeschlossen hat. 4. Der User trägt in dem Formular „Zählerstand aktualisieren“ des entsprechenden Stromzählers den aktuellen Zählerstand ein und sendet das Formular ab. 5. Der Webserver verarbeitet die Anfrage und speichert den aktualisierten Zählerstand im EDMS.

Tabelle 3.6: Stromzählerstand aktualisieren

3.4 Nicht funktionale Anforderungen

Die Formulierung der nicht funktionalen Anforderungen muss differenziert werden. Zum einen ist das Ziel der Arbeit, die Realisierung einer verständlichen, wartbaren Anwendungslandschaft, die nach Abschluss des Projekts innerhalb eines Continuous Development Prozesses ausgebaut werden soll, zum anderen besitzt das fiktive Unternehmen ebenfalls bestimmte nicht funktionale Anforderungen an die Anwendungslandschaft. Einige der Anforderungen überschneiden sich mit dem Ziel der Arbeit, andere sind im Sinne dieses Szenarios ergänzt worden.

1. Skalierbarkeit
   Das Unternehmen geht davon aus, dass die Anforderungen an die Performance der Anwendungslandschaft in den kommenden Jahren stark steigen werden. Aus dem Grund soll die Landschaft so entworfen werden, dass die einzelnen Komponenten bei einer stärker werdenden Auslastung gut skaliert werden können.

2. Wartbarkeit
   Da sich das Entwicklerteam in den nächsten Jahren häufig ändern kann, soll das System besonders wartbar sein. Aus dem Grund soll eine ausführliche Dokumentation der Schnittstellenspezifikationen, der Umsetzung des CI/CD-Prozesses, sowie der einzelnen Komponenten der Anwendungslandschaft erfolgen.

3. Portierbarkeit
   Um die Entwicklung im Team zu erleichtern und die Umsetzung des CI/CD-Prozesses zu ermöglichen, soll die Anwendung plattformübergreifend ausgeführt werden können.

4 Systemdesign

Nachdem in Kapitel 3 die Anforderungen an die zu entwickelnde Anwendungslandschaft formuliert wurden, soll diese nun entworfen werden. Die Umsetzung erfordert die Entwicklung diverser neuer Komponenten sowie die Realisierung der Kommunikation zwischen den einzelnen Komponenten.

Im Folgenden sollte zunächst das Entity Relationship Modell vorgestellt werden sowie das daraus resultierende Datenmodell. Diese bilden das Fundament der Anwendungslandschaft und helfen beim weiteren Verständnis. Anschließend wird die Architektur beschrieben und diesbezügliche Designentscheidungen begründet. Dazu gehört einerseits das Herausarbeiten der Vorteile von serviceorientierten Architekturen gegenüber monolithischen, und andererseits ein Vergleich verschiedener Kommunikationsarten. Zum Schluss werden die einzelnen Services dann genauer vorgestellt und deren Schnittstellen beschrieben.

Die Komponenten und Entitäten werden von nun an in die englische Sprache übersetzt. Sowohl die Implementierung, als auch die Dokumentation der Systeme erfolgt auf Englisch, um von einem internationalen Team weiterentwickelt werden zu können.

4.1 Entity Relationship Modell

In den Anforderungen in Kapitel 3 wurden diverse Entitäten beschrieben, die den einzelnen Komponenten der Anwendungslandschaft zugeordnet werden können. So besteht die Anwendungslandschaft im Grunde aus den acht Entitäten Benutzer (User), Kundenkonto (Account), Stromliefervertrag (Contract), Stromtarif (Tariff), Tarifregion (Region), Tarifbonus (Bonus), Stromzähler (ElectricMeter) und Zählerstand (MeterReading). Das folgende ER-Modell beschreibt die Zusammenhänge der genannten Entitäten.

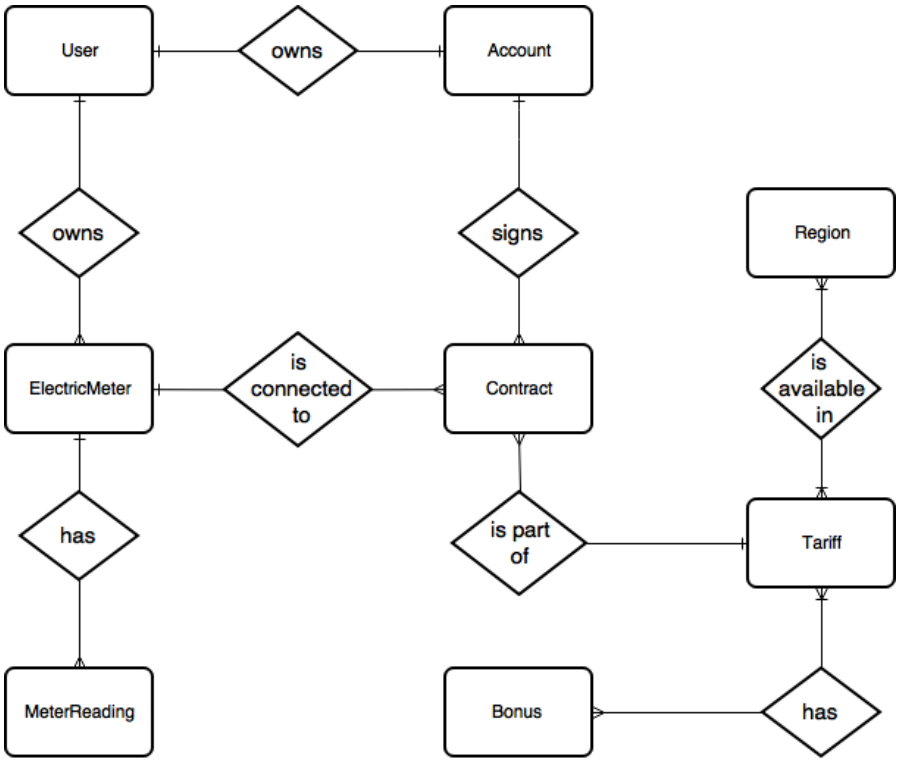

Abbildung 4.1: ER-Modell aus der Sicht der Kunden

Abbildung 4.1 zeigt, dass jeder registrierte Kunde genau ein Kundenkonto besitzt sowie einen Benutzer, dem dieses Konto gehört. Das Kundenkonto enthält sämtliche Informationen über den Kunden und existiert bereits in der Datenbank von SugarCrm. Der Benutzer wird für die Umsetzung der Zugriffskontrolle benötigt. (siehe Kapitel 3.3) Wie die Zugriffskontrolle allerdings im Detail umgesetzt werden kann, wird im entsprechenden Abschnitt in Kapitel 4.4.2 beschrieben.

Jedes Kundenkonto kann mehrere Stromlieferverträge abschließen. Jeder Stromlieferverträg ist allerdings nur einem Kundenkonto zugewiesen. Auf diese Weise kann ein Vertrag eindeutig einem Kunden zugeordnet werden.

Für den Abschluss eines Stromliefervertrages ist die Auswahl eines Stromtarifes notwendig. (siehe „Bestellprozess“ in 3.2.1) Aus diesem Grund wird jedem Stromliefervertrag genau ein Stromtarif zugewiesen. Da unterschiedliche Kunden einen Vertrag für denselben Tarif abschließen können, können Tarife mehreren Verträgen gleichzeitig zugewiesen werden.

Da das Unternehmen nicht jedes Gebiet in Deutschland beliefern kann oder nicht beliefern möchte, und vor allem, da sich die Strompreise regional häufig stark unterscheiden, sollen sich Tarife in bestimmten Gebieten ebenfalls unterscheiden und in anderen gar nicht angeboten werden. (siehe „Tariferstellungsprozess“ in Kapitel 3.2.4) Um dies zu gewährleisten, werden Tarife an Regionen gebunden. Für eine individuellere Gestaltung der Tarife sollen zudem Tarifboni eingeführt werden. Jeder Tarif kann somit mehrere Boni enthalten. Da Regionen und Boni zusätzlich in mehreren Tarifen gleichzeitig vorkommen können, wird dies jeweils als N-zu-M-Beziehung umgesetzt.

Für die Abrechnung des Stromverbrauches werden die Daten des Stromzählers der Entnahmestelle benötigt. Um zusätzlich bessere Verbrauchsprognosen durchführen zu können, wird jeder aktualisierte Zählerstand des Stromzählers festgehalten. Der Stromzähler ist wie das Kundenkonto für die Umsetzung der Zugriffskontrolle an das Benutzerkonto gebunden. Da ein Kunde mehrere Stromzähler besitzen könnte und für jeden ein eigener Liefervertrag abgeschlossen werden müsste, wird der Stromzähler an den entsprechenden Vertrag gebunden. Durch einen Mieterwechsel kann es allerdings passieren, dass mehrere Verträge mit demselben Stromzähler verbunden sein könnten. Genauso können sich dadurch die Besitzverhältnisse des Stromzählers ändern. Damit es nicht zu Problemen bei der Abrechnung und den Zugriffsrechten kommt und z.B. mehrere Kunden für denselben Verbrauch zahlen müssten, müssen diese Fehlerfälle entsprechend beachtet werden.

4.2 Das Datenmodell

Im Folgenden wird nun der Entwurf des Datenmodells vorgestellt und die diesbezüglichen Designentscheidungen begründet. Das Modell ist aus dem beschriebenen ER-Modell hervorgegangen und erweitert die genannten Entitäten um die benötigten Daten. Aufgrund der Einheitlichkeit des Modells und einer damit verbundenen verbesserten Verständlichkeit, orientiert sich das Modell an den Konventionen der SugarCrm Datenbank. Die Abkürzungen PK und FK stehen an dieser Stelle für die Konzepte Primär- und Fremdschlüssel.

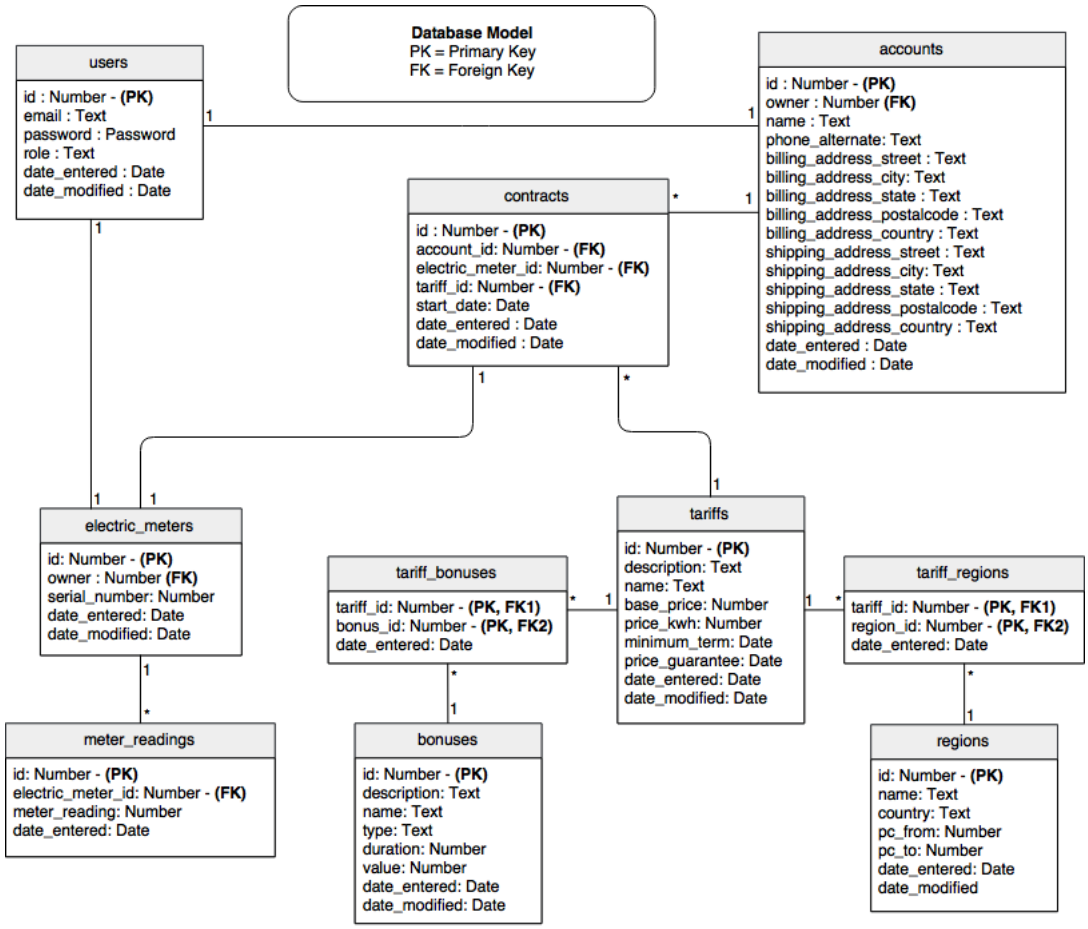

Abbildung 4.2: Darstellung des Datenmodells der Anwendungslandschaft

4.2.1 Benutzer

Die Benutzer des Systems werden in der „users“-Tabelle gespeichert. Die Tabelle enthält Informationen über die Email-Adresse und das Passwort sowie die Information darüber, welche Rolle der jeweilige Benutzer besitzt. Email-Adresse und Passwort werden für den Login in den Kunden- oder Arbeitsbereich benötigt. Anhand der Rolleninformation kann geprüft werden, welche Rechte der Benutzer im System besitzt.

4.2.2 Kundenkonto

Sämtliche Kundendaten werden in der „accounts“-Tabelle der Datenbank von SugarCrm gespeichert. Das abgebildete Modell wurde an dieser Stelle auf die wichtigsten Informationen reduziert und zeigt nur einen kleinen Ausschnitt der tatsächlichen Tabelle. Zudem wurde die Tabelle um das Feld „owner“ ergänzt. Dieses verweist auf einen Benutzer in der „users“-Tabelle und gibt an, wem das Kundenkonto gehört. Diese Information wird zusammen mit der Rolleninformation der Benutzer für die Zugriffskontrolle verwendet.

4.2.3 Stromlieferverträge

Die benötigten Daten für die Stromlieferverträge werden in der „contracts“-Tabelle gespeichert. Obwohl im beschriebenen Modell nicht alle notwendigen Daten aufgelistet werden, reichen die genannten Informationen bereits größtenteils aus, um die geforderten Geschäftsprozesse umsetzen zu können. Die Tabelle enthält durch die Fremdschlüssel-Beziehungen Informationen darüber, welches Kundenkonto den Vertrag abgeschlossen hat, welcher Stromzähler für die Abrechnung verwendet werden soll und zu welchen tariflichen Konditionen die Abrechnung und die Belieferung erfolgen soll. Die Angaben über Liefer- und Messspannung in Volt, Entgelte für Messung der Abgaben und Steuern sowie der Kündigungfrist sind allerdings nicht enthalten. Diese Angaben wären für den vollständigen und korrekten Abschluss der Verträge notwendig und können jederzeit ergänzt werden.

4.2.4 Stromtarife

Der Aufbau von Stromtarifen wurde bereits in den Kapiteln 3.1.1 und 3.2.4 beschrieben und kann beinahe direkt in der „tariffs“-Tabelle umgesetzt werden. Dies liegt daran, dass die meisten der benötigten Informationen durch einen atomaren Wertebereich abgedeckt werden können und nur maximal einmal für einen Tarif definiert werden müssen. Aus diesen Gründen werden die Komponenten „Grundpreis“ (base_price), „Arbeitspreis“ (price_kwh), „Mindestvertragslaufzeit“ (minimum_term) und „Preisgarantie“ (price_guarantee) durch entsprechende Felder in der

Tabelle repräsentiert. Da es sich dagegen bei der Bindung von Tarifen an Regionen und Boni um komplexere Datentypen handelt, muss dies genauer modelliert werden.

Tarifboni

Wie in Kapitel 3.1.1 beschrieben, kann man drei verschiedene Arten von Boni unterscheiden: Einmalzahlungen, Volumenboni und andauernde Rabatte. Zusammenfassend lassen sich diese Arten durch die drei Faktoren Typ („type“), Laufzeit („duration“) und Wert („value“) darstellen. Ein Tarifbonus der ersten Art kann somit z.B. durch einen Bonus vom Typ „Einmalzahlung“ mit einer Laufzeit von „einem Monat“ und einem Wert von „20,00 €“, ein Bonus der zweiten Art durch das Dreier-Tupel „(Volumenbonus, 0 Monate, 200 Kwh)“ und ein Bonus der dritten Art durch das Dreier-Tupel „(mGp-Rabatt, 12 Monate, 10 €)“ dargestellt werden.

Regionen

Die Regelung des fünfstelligen Postleitzahlen-Systems, das seit dem 01.07.1993 in Deutschland gültig ist, sorgt bereits für eine Einteilung Deutschlands in Regionen und ist deswegen gut geeignet, um Verfügbarkeitsgebiete für Tarife zu erstellen. So ist Deutschland auf Postleitzahlenebene in zehn verschiedene Zonen unterteilt, die durch die erste Ziffer der Postleitzahl angegeben werden und aus den Ziffern null bis neun bestehen können. Zusammen mit der zweiten Ziffer, die die Region angibt, bilden die ersten beiden Ziffern die Postleitregion, anhand derer eine Einteilung möglich wird.

Abbildung 4.3 auf der nächsten Seite zeigt die Postleitzahlenkarte von Deutschland und verdeutlicht diesen Zusammenhang noch einmal. Die fettgedruckten Zahlen geben dabei die zehn Zonen an und die zweistelligen Zahlen zeigen die entsprechenden Regionen.

Um einen Tarif in „Norddeutschland“ anbieten zu können, kann man also eine Region festlegen, die sämtliche Postleitzahlen mit den beiden Anfangsziffern „20“ bis „29“ umfasst. Ein Tarif für „Ostdeutschland“ könnte durch den Bereich „01“ bis „19“ abgedeckt werden oder durch die zwei Regionen, „01“ bis „09“ und „10“ bis „19“. Das System ermöglicht aber durch die weiteren Zahlen auch feingranularere Einteilungen von Regionen, z.B. die Unterteilung Hamburgs in drei unterschiedliche Regionen: Die erste Region umfasst die Postleitzahlen „20095-21149“, die zweite, die Postleitzahlen „22041-22769“ und die dritte, die auf sich selber verweisende Postleitzahl „27499“, womit durch den Bereich „27499-27499“ ebenfalls die politisch zu Hamburg gehörende Insel Neuwerk abgebildet werden könnte, obwohl sie geografisch gesehen ca. 120 km Luftlinie nordwestlich entfernt von Hamburg liegt.

Abbildung 4.3: Postleitzahlenkarte von Deutschland
Quelle: https://www.welt-atlas.de/karte_von_detuschland_postleitzahlen_1-990 (26.09.2017)

Stromzähler

Die Stromzählerinformationen werden in der „electric_meters“-Tabelle gespeichert und bestehen aus der realen Zählernummer sowie dem Verweis auf den aktuellen Besitzer der Ressource. Im Falle eines Mieterwechsels können die Stromzähler auf einen anderen Namen angemeldet werden. Da die Information, für welchen Stromzähler ein Kunde einen Liefervertrag abgeschlossen hat, in der „contracts“-Tabelle referenziert wird, kann das „owner“-Feld verändert werden, ohne dass die Information gänzlich verloren geht. Vorraussetzung für die Aktualisierung der Besitzverhältnisse wäre allerdings, dass der Stromvertrag bereits gekündigt wurde und der Kunde nicht mehr mit Strom beliefert wird. So könnte man z.B. vor Abschluss eines Stromliefervertrages, den frühestmöglichen Lieferbeginn überprüfen. Die Belieferung der Entnahmestelle könnte dann erst ab dem Zeitpunkt erfolgen, ab dem der bisherige Vertrag abgelaufen ist. Die Besitzverhältnisse würden sich somit erst bei Lieferende und -beginn ändern. Um den Verbrauch des Kunden besser analysieren zu können, wird jede Aktualisierung des Zählerstandes in der „meter_readings“-Tabelle festgehalten. Die Tabelle enthält den Zeitpunkt der Aktualisierung sowie den neuen Zählerstand.

4.3 Architektur und Kommunikation

Die Anwendungslandschaft wird durch eine serviceorientierte Architektur umgesetzt. Jede Komponente, die in Kapitel 3.3 beschrieben wird, besteht dabei aus einem oder mehreren Webservices, die unabhängig voneinander entwickelt und deployed werden können und REST-Schnittstellen bereitstellen. Bevor diese Entscheidungen allerdings begründet werden, soll zunächst der Aufbau des Systems beschrieben werden.

4.3.1 Systemaufbau

Abb. 4.4 auf der folgenden Seite zeigt den Aufbau der Anwendungslandschaft aus der Sicht der Webservices. Die Userverwaltung wird dabei durch das „User Management System“ (UMS), bestehend aus den zwei Services „Authentication Service“ und „User Service“, die Tarifverwaltung durch das „Tariff System“ (TS), bestehend aus den drei Services „Region Service“, „Tariff Service“ und „Bonus Service“ und das Energiedaten Management Systems durch das „Energy Data Management System“ (EDMS), bestehend aus dem „EDM Service“, umgesetzt. Der Webserver dient dagegen als Hauptschnittstelle des Systems und stellt die Ressourcen der verschiedenen Backendsysteme durch eine entsprechende Benutzeroberfläche zur Verfügung, die über die entsprechenden Clients bedient werden kann.

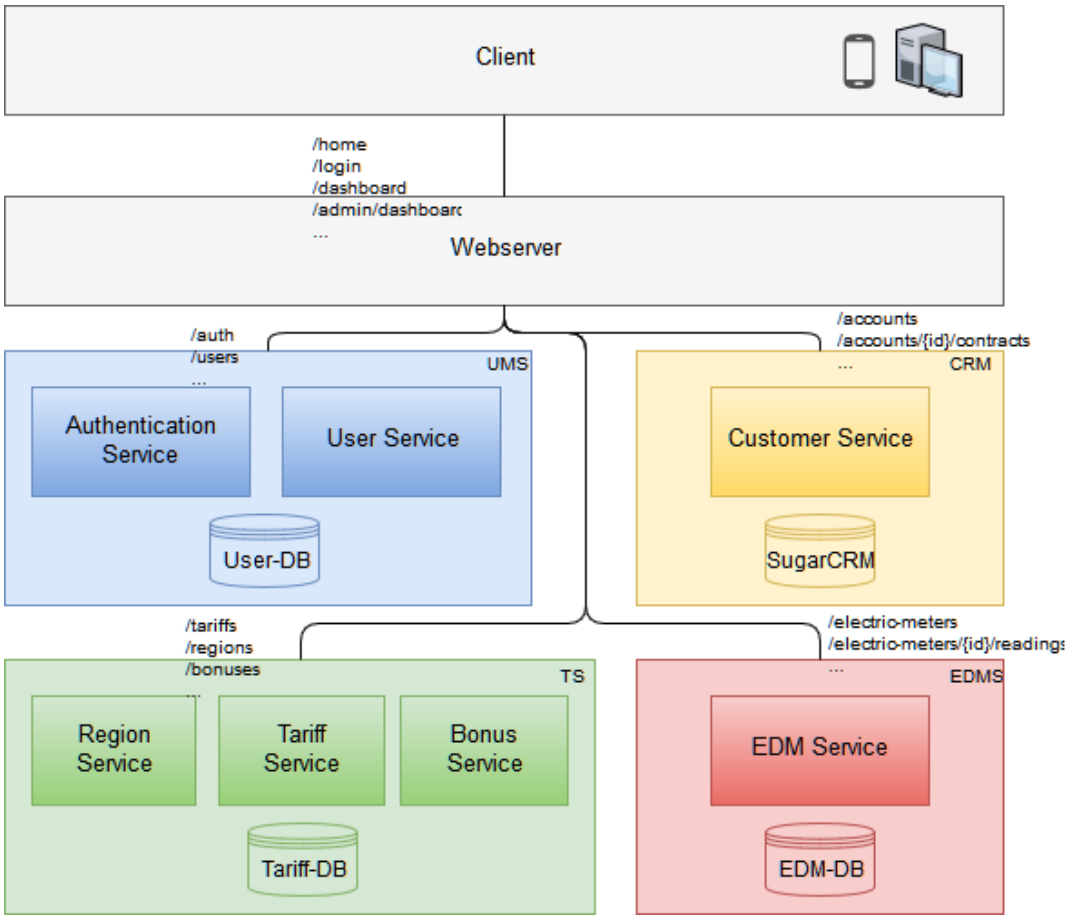

Abbildung 4.4: Systemarchitektur

Die Schnittstelle von SugarCrm hat einige Schwächen im Design und in der Benutzbarkeit. So besitzt sie z.B. nur einen einzigen REST-Endpunkt, der zudem nur einen Get-Request verarbeiten kann. Die Wahl der Methode und die Übergabe von Parametern muss dabei immer innerhalb des Bodies angegeben werden, wodurch die Schnittstelle gegen sämtliche Konventionen verstößt und weit davon entfernt ist, REST-konform zu sein. Des Weiteren ist es aufgrund einer unzureichenden Dokumentation und einer verworrenen Systemarchitektur sehr aufwendig, die Schnittstelle zu erweitern. Da die geforderte Erweiterung viel Zeit beanspruchen würde, wurde die Entwicklung eines eigenen „Customer Services“ favorisiert. Dieser dient als Wrapper für SugarCrm und kann leichter erweitert werden. Da die Systemlandschaft aus Webservices besteht, die nur lose gekoppelt sind, kann eine korrekte Erweiterung SugarCrms zudem jederzeit nachgeholt werden.

Darüber hinaus besitzt jedes Backendsystem eine eigene Datenbank, die von den entsprechenden Services geteilt wird. Dies hat zur Folge, dass das Datenmodell (siehe Kapitel 4.2) auf mehrere Datenbanken aufgeteilt werden muss und dass die Fremdschlüssel durch entsprechende Referenzen ersetzt werden. Eine Replikation der Datenbanken wird allerdings nicht umgesetzt. Die folgende Abbildung zeigt das überarbeitete verteilte Datenmodell.

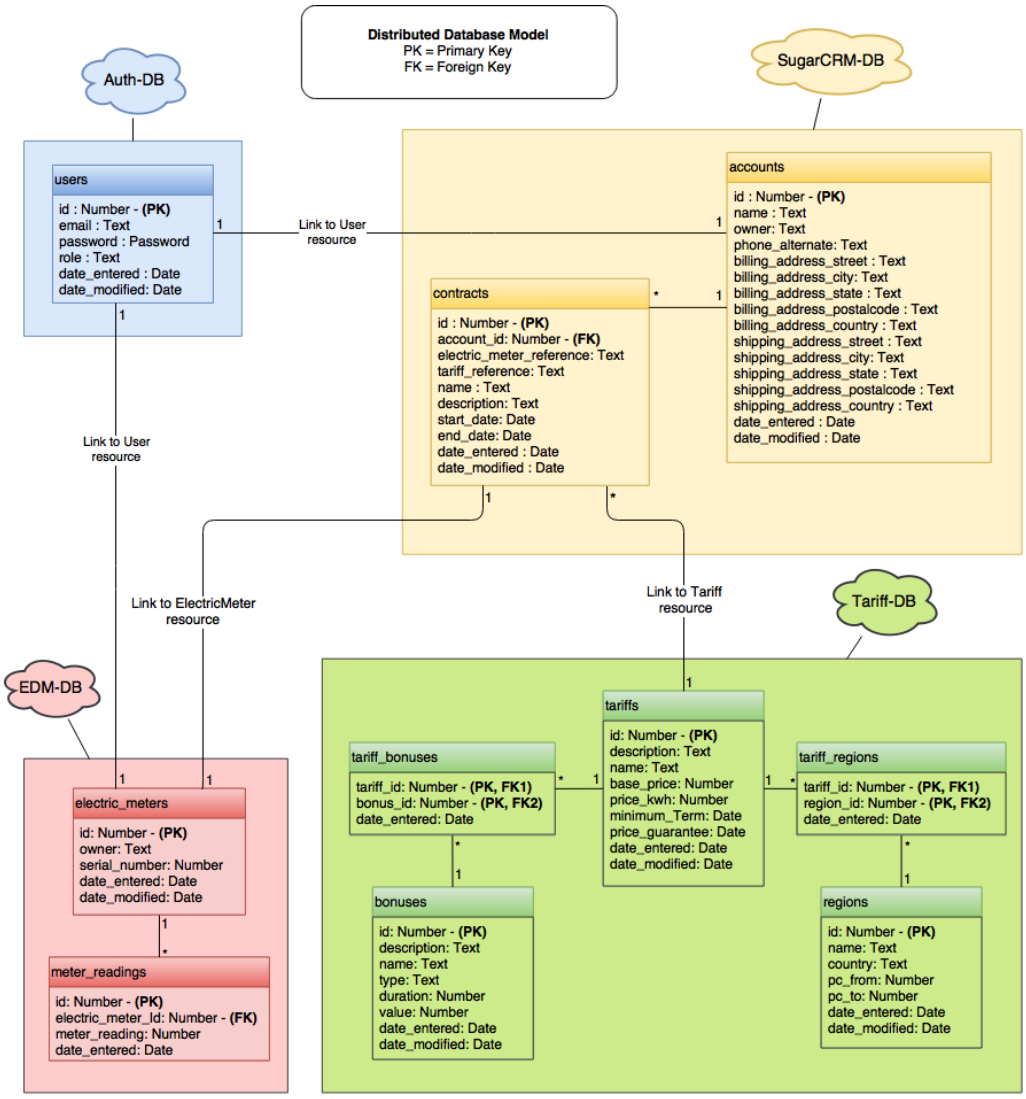

Abbildung 4.5: Das verteilte Datenmodell

4.3.2 Vor- und Nachteile einer Microservice Architektur

Eine Möglichkeit der Realisierung Verteilter Systeme ist die Umsetzung durch eine serviceorientierte Architektur, im Speziellen einer Microservice Architektur (MSA). Im Folgenden sollen nun die Vor- und Nachteile einer solchen Architektur herausgearbeitet werden und es ausführlich begründet werden, warum sie sich für die Umsetzung der Anwendungslandschaft eignet.

1. Geringe Kopplung
   Eine Microservice Architektur mit klar definierten und gut dokumentierten Schnittstellen ermöglicht die Kopplung der einzelnen Komponenten so gering wie möglich zu halten. Dies erleichtert das Entfernen alter Komponenten sowie das Hinzufügen neuer Komponenten innerhalb der Anwendungslandschaft. Ausßerdem erleichtert es auch das Austauschen verwendeter Technologien, da die Realisierung der Funktionalitäten hinter klar definierten Schnittstellen verborgen wird. Je geringer die Kopplung zwischen zwei Microservices, desto leichter können sie unabhängig voneinander deployed werden.

2. Hohe Kohäsion
   Durch die klare Trennung der Zuständigkeiten, verteilt auf mehrere kleine Komponenten, wird die Kohäsion der einzelnen Komponenten erhöht. Auf diese Weise können komplexe Systeme in mehrere einfache Systeme aufgeteilt werden, wodurch das System verständlicher werden kann. Dies führt zu einer besseren Wartbarkeit und erhöht die Wiederverwendbarkeit der entwickelten Dienste.

3. Horizontale Skalierung
   Serviceorientierte Architekturen bieten die Möglichkeit der horizontalen Skalierung. Durch die horizontale Skalierung kann die Last auf mehrere Instanzen desselben Services verteilt werden, wodurch das Problem der begrenzten und teuren vertikalen Skalierung umgangen werden kann. Mit Hilfe von Docker oder ähnlichen Technologien können die Services zudem in kürzester Zeit gestartet und auf diverse Maschinen verteilt werden. Für eine produktionsfertige Lösung mit Docker sind allerdings weitere Technologien wie Kubernetes oder Docker Swarm notwendig, die die Orchestrierung der Container und die Service Discovery übernehmen.

Natürlich besitzen Microservice Architekturen aufgrund der Verteilung der Zuständigkeiten auf mehrere Systeme auch Nachteile. Dazu gehört z.B. eine aufwendigere Suche und Behebung von Fehlern im System und eine aufwendigere Erstellung von Tests durch die Abwesenheit eines „Single point of controls“. Außerdem führt dies zu einer erhöhten Netzauslastung und erschwert die Speicherung von Daten. Insbesondere die Einhaltung der **ACID Prinzipien** (Atomicity, Consistency, Isolation, Durability) von Transaktionen wird dadurch erschwert. So muss z.B. der wahrscheinliche Fall abgefangen werden, dass einer der beteiligten Services während einer verteilten Transaktion ausfällt oder nicht mehr erreicht werden kann, wodurch inkonsistente Datenbestände entstehen können. Grund hierfür kann zum Beispiel ein Systemausfall oder ein Netzwerkfehler sein.

Laut dem **CAP-Theorem** von Eric Brewer ist es in Verteilten Systemen zudem generell unmöglich die drei Kriterien „Konsistenz“ (alle Systeme sehen die gleichen Daten), „Verfügbarkeit“ (akzeptable Antwortzeiten), sowie „Partitionstoleranz“ (das System kann mit Netzwerkpartitionen umgehen) gleichzeitig zu erfüllen. (vgl. Brewer 2000) Lediglich zwei der Kriterien können laut Brewer zur selben Zeit erfüllt werden. Da die horizontale Skalierung von Anwendungen allerdings eines der Hauptvorteile von Docker ist und diese in der Regel erforderlich ist, um mit einer großen Anzahl von Anfragen zurecht zu kommen, muss die Partitionstoleranz unbedingt erfüllt werden. Dasselbe gilt für eine hohe Verfügbarkeit. Sollte z.B. das Tarifsystem oder die Webanwendung ausfallen, könnten potentielle Kunden nicht mehr nach Stromtarifen suchen und würden vermutlich schnell bei der Konkurrenz nachschauen, wodurch wertvolle Kundenbindungen verloren gingen.

Aus dem Grund wird für bestimmte Anwendungsfälle häufig das schwächere **BASE-Prinzip**, anstatt des ACID-Prinzips, für die Einhaltung einer Datenkonsistenz verwendet. BASE steht für „Basically Available, Soft State, Eventual consistency“ und bedeutet im Grunde, dass das System für eine möglichst kurze Zeitspanne im Notfall inkonsistent sein darf, bevor es wieder einen konsistenten Zustand erreichen muss. Für die Suche nach Tarifen ist dies z.B. völlig ausreichend. Bei einer Banküberweisung könnte es allerdings zu einem Problem führen, wenn der Überweisende eine falsche Information erhält, obwohl die Überweisung schon durchgeführt wurde. Im schlimmsten Fall würde der Vorgang dann wiederholt werden und das Geld somit doppelt abgebucht werden.

Um die Transaktionen in dieser Arbeit nicht zu verkomplizieren, werden die einzelnen Datenbanken zunächst nicht repliziert. Aus dem Grund teilen sich die Services einer Komponente vorerst auch dieselbe Datenbank. Die Aufteilung des Datenmodells auf die einzelnen Datenbanken der verschiedenen Systeme führt bereits zu Fehlerfällen, die behandelt werden müssen. So muss im Bestellprozess z.B. zunächst ein Benutzerkonto für den Kunden angelegt werden, bevor das Kundenkonto im CRM angelegt werden kann. Fällt nun der Webserver oder das CRM während des Prozesses aus, also nachdem das Benutzerkonto erstellt wurde, entstehen Fehler beim erneuten Absenden des Formulares, da das Benutzerkonto bereits existiert. Wie solche Fehler in der Lösung behandelt werden könnten, wird in den entsprechenden Abschnitten in Kapitel 5 genauer beschrieben.

Natürlich wäre eine verteilte Architektur für diese spezielle Webanwendung grundsätzlich nicht unbedingt notwendig, da davon ausgegangen werden kann, dass die Zahl der Seitenaufrufe pro Tag nicht sehr hoch sein wird. Zudem wird die Aufenthaltsdauer der Benutzer als gering eingestuft, da die geforderten Funktionen wie die „Tarifsuche“, der „Bestellprozess“ oder die „Tariferstellung“ in wenigen Minuten durchgeführt werden können und sich die Zahl der Ausführungen vermutlich auf wenige Male im Jahr pro Benutzer beschränkt. Eine gute Skalierbarkeit des Systems könnte also eigentlich vernachlässigt werden. Da die Anwendung allerdings in erster Linie der Erforschung von Docker und der Anwendungsintegration dient und viele Konzepte von Docker, wie z.B. die Service Discovery oder die Orchestrierung mehrerer Container, per Definition auf Webservices ausgelegt sind, bietet sich eine serviceorientierte Architektur an. Ein Monolith würde in diesem Kontext wenig Sinn ergeben. Außerdem macht die fortschreitende Digitalisierung der Prozesse der Energiewirtschaft, Konzepte wie „Dynamische Tarife“ und „Smart Meters“ realisierbar und könnte somit dazu beitragen, dass Kunden energiebewusster agieren müssen. Zukünftige Webanwendungen von Energieanbietern könnten durch die zusätzliche Menge an ausgetauschten Daten aufwendiger werden und müssten häufiger mit entsprechenden Clients kommunizieren, wodurch die Auslastung der Systeme wieder steigen würde.

Grundsätzlich kann man also sagen, dass die Umsetzung verteilter Systeme und die Verwendung einer Microservice Architektur Risiken birgt. Wenn man sich dieser Risiken allerdings bewusst wird und entsprechende Lösungen einsetzt, hat man große Chancen, umfangreichere und komplexere Anwendungen zu realisieren. Gerade die Möglichkeit der horizontalen Skalierung von Anwendungen ist in der heutigen Zeit, in der das Mooresche Gesetz an seine Grenzen zu stoßen scheint, wichtiger denn je.

4.3.3 REST & SOAP vs. Messaging

Die Microservices der einzelnen Komponenten der Systemlandschaft sollen durch REST-Schnittstellen realisiert werden. (vgl. Fielding und Taylor 2000) Neben REST (Representational State Transfer) gibt es allerdings noch viele weitere Konzepte, die für die Interprozesskommunikation und die Beschreibung von Schnittstellen eingesetzt werden könnten. Zu den bekanntesten gehören derzeit SOAP (Simple Object Access Protocol) und Messaging. (vgl. Box u.a. 2000 und Banavar u.a. 1999) Obwohl an dieser Stelle kein ausführlicher Vergleich der verschiedenen Technologien erfolgen soll, kann ein grober Überblick über die wesentlichen Unterschiede der Technologien für die Begründung der Entscheidung durchaus sinnvoll sein.

Bei der Wahl der geeigneten Technologie spielen vor allem die Art der Kommunikation sowie die Umsetzung der Service Discovery eine wichtige Rolle. Die Kommunikationsart lässt sich in synchrone und asynchrone Kommunikation aufteilen und ist abhängig vom jeweiligen Anwendungsfall. Bei der synchronen Kommunikation muss der Client nach dem Versenden einer Nachricht auf die direkte Antwort des Servers warten, während er bei der asynchronen Kommunikation weiterarbeiten kann, ohne zu blockieren. Bei der Service Discovery geht es dagegen darum, wie sich Services in einem verteilten System gegenseitig „finden“ können, um die jeweiligen Schnittstellen zu verwenden.

Während REST und SOAP in erster Linie synchrone Kommunikationen ermöglichen, wird die Kommunikation beim Messaging über den Message Broker dagegen asynchron abgewickelt. Dies kann für bestimmte Anwendungsfälle wie dem Versenden eines Newsletters oder der Berechnung von „Verbrauchsprognosen“ für den Bilanzkreisausgleich sinnvoll und sogar erforderlich sein, wenn die Antwort irrelevant ist oder intensive Berechnungen erfolgen müssen. Eine synchrone Kommunikation sorgt zudem dafür, dass der Client Kenntnis über die Antworten des Servers besitzen muss, wodurch eine zusätzliche Kopplung der Systeme geschaffen wird, die es beim Messaging nicht gibt. Für andere Anwendungsfälle wie dem Login in den Kundenbereich kann die direkte Antwort allerdings erforderlich sein. Die Service Discovery ist dagegen durch den Message Broker naturgemäß nur beim Messaging vorhanden, da alle beteiligten Parteien lediglich Kenntnis über die Adresse des Brokers besitzen müssen. Bei REST und SOAP sind hierfür in der Regel weitere Konzepte und Technologien notwendig. Im Fall von Docker kann die Service Discovery z.B. durch Kubernetes oder Docker Swarm umgesetzt werden.

Die Benutzer der Webanwendung erwarten direkte Antworten auf entsprechende Eingaben. Aus dem Grund muss die Webserver Komponente in jedem Fall synchrone Kommunikationsmechanismen ermöglichen. Um die erste Version der Anwendungslandschaft nun nicht mit diversen Technologien zu überladen und um den Entwicklungsaufwand zu minimieren, wurde zunächst nur ein synchroner Ansatz für die Kommunikation innerhalb der Anwendungslandschaft gewählt. Dies bedeutet aber nicht, dass die asynchrone Kommunikation in einem zukünftigen Release ausgeschlossen werden sollte. Sie kann sogar erforderlich werden.

Im Folgenden sollen nun die Unterschiede von REST und SOAP beschrieben werden.

1. Einheitlichkeit der Schnittstellen
   Der Hauptunterschied zwischen REST und SOAP ist die Denkweise, mit der Schnittstellen spezifiziert werden. Während der Fokus bei SOAP auf den Methoden liegt, geht es bei REST in erster Linie um Ressourcen. Für jede Ressource stehen deshalb dieselbe Menge an Methoden zur Verfügung, während bei SOAP beliebige Methoden definiert werden können. Daduch bieten SOAP Schnittstellen auf den ersten Blick zwar mehr Möglichkeiten, auf den zweiten Blick können sie dadurch aber auch schwerer zu verstehen sein. Durch die Einheitlichkeit der REST Schnittstellen können Endpunkte leichter erstellt und erweitert werden, wodurch Entwickler wiederum entlastet werden können. Die einheitlichen Schnittstellen setzen sich aus eindeutigen Adressen der Endpunkte, einer einheitlichen Repräsentation der Ressourcen (in der Regel mittels XML oder JSON), dem Verwenden von Standardmethoden (siehe HTTP-Methoden) und dem Konzept HATEOAS („Hypermedia as the Engine of Application State“) zusammen.

2. HATEOAS und YAML vs. WSDL
   Es gibt zwar nicht wie bei SOAP eine WSDL-Datei („Webservice Definition Language“), die die Schnittstelle des Webservices verbindlich beschreibt - dafür gibt es diverse Markup Sprachen wie YAML oder RAML, die sich für das Erstellen von Schnittstellenbeschreibungen eignen. Wenn HATEOAS zudem korrekt umgesetzt wird, ist es für den Benutzer einer REST-Schnittstelle nur erforderlich, den Eingangspunkt zu kennen. Alle weiteren Aktionen können durch die in der Nachricht mitgelieferten Hyperlinks ausgeführt werden. Die vollständige Umsetzung von HATEOAS ist allerdings mit viel Aufwand verbunden, weswegen viele Schnittstellen nur „RESTish“ implementiert werden und dieses Konzept häufig vereinfacht umsetzen. Die WSDL-Datei bei SOAP stellt dagegen eine Art Vertrag zwischen Benutzer und Anbieter dar, an den sich beide Parteien halten müssen. Dies sorgt zwar auf der einen Seite für mehr Gewissheit, dass die

Kommunikation korrekt durchgeführt wird, auf der anderen Seite sorgt dies aber auch für eine engere Kopplung der beiden Parteien.

3. Performance
   Die WSDL-Datei und die Übermittlung der Daten durch XML sorgt bei SOAP für einen Overhead an mitgesendeten Daten, den es bei REST durch die Kompaktheit und die Möglichkeit der Übertragung von JSON Objekten nicht gibt. Aus dem Grund ist die Datenübertragung bei REST Schnittstellen schneller als bei SOAP Schnittstellen.

Beide Konzepte haben ihre Vor- und Nachteile und eignen sich beide für die Umsetzung der Schnittstellen. Aufgrund der persönlichen Empfindung, dass REST Schnittstellen verständlicher sind und aufgrund des geringen Overheads an mitgesendeten Daten, wurde sich für eine Kommunikation mittels REST-Schnittstellen entschieden. Die Verständlichkeit der Schnittstellen ist gerade in Hinblick auf die Erweiterbarkeit der Anwendungslandschaft wichtig, damit weitere Gruppen im Rahmen der Forschungsarbeiten im Labor für Anwendungsintegration der Hochschule für angewandte Wissenschaften Hamburg weniger Probleme haben, das System weiterzuentwickeln. Der vollständige und korrekte Entwurf einer REST-Schnittstelle, inklusive HATEOAS, wäre allerdings mit einem Aufwand verbunden, der nicht unbedingt erforderlich ist. REST bietet auch mit einer „abgespeckten“ Variante von HATEOAS ein gelungenes Konzept für die Umsetzung einheitlicher Schnittstellen. Und zusammen mit einer Schnittstellenbeschreibung, z.B. in Form einer YAML-Datei, reichen die Konzepte bereits aus, um komplexe Geschäftsprozesse umzusetzen. Aus dem Grund werden die Schnittstellen in einer „abgespeckten“ REST-Variante ohne die vollständige Einhaltung von HATEOAS umgesetzt.

4.4 Schnittstellendesign

In diesem Kapitel werden nun die Schnittstellen der Services aus Kapitel 4.3 entworfen. Da die vollständige Schnittstellenbeschreibung sehr viel Platz einnehmen würde, sollen an dieser Stelle nur die wichtigsten Designentscheidungen begründet werden. Für die vollständige Dokumentation sei auf das entsprechende Swagger.io-Projekt verwiesen, welches dem Anhang auf der CD beigelegt wurde und alternativ unter der folgenden URL erreicht werden kann:
`https://app.swaggerhub.com/apis/JonesPButter/Greenworld-Energies/1.0.0`

4.4.1 Das Fehlermodell

Die Kommunikation der REST-Schnittstellen über HTTP liefert verschiedene HTTP Statuscodes. Das Problem der Statuscodes besteht darin, dass sie missinterpretierbar und nicht

aussagekräftig sein können. So sagt der Statuscode „400 - Bad Request“ z.B. nur aus, dass die Nachricht falsch aufgebaut wurde. In bestimmten Situationen kann es aber sinnvoll sein, wenn mehr Informationen über den Fehler bekannt gegeben werden. Im Talk „Beautiful REST + JSON APIs“ von Les Hazlewood, zum Zeitpunkt der Präsentation Technikvorstand von Stormpath, einer API für Identity und User Management, wird genau diese Problematik aufgegriffen und eine Lösung vorgestellt. So sollten Fehler z.B. so „beschreibend“ und „detailiert“ wie möglich sein, um die Entwickler bei der Nutzung der Schnittstelle zu unterstützen. Das folgende Fehlerobjekt orientiert sich an dem von Hazlewood beschriebenen Modell und wandelt es dabei lediglich geringfügig ab. (vgl. HAZLEWOOD11, S. 70)

```
{
  "httpStatus": 401,
  "code": 40001,
  "name": "MissingHeaderError",
  "properties": {
    "Authorization": "Expected JSON-Web-Token token"
  },
  "message": "The Authorization Header was missing."
}
```

Listing 4.1: Aufbau des Fehlerobjekts bei einem unauthorisierten Request

Listing 4.1 zeigt die Antwort eines Services bei einem unauthorisierten Request. Statt der üblichen HTTP Statuscodes erhalten die Anwender nun zusätzlich ein Fehlerobjekt mit detaillierteren Informationen. Der HTTP Statuscode 401 („Unauthorized“) wird dabei um einen genaueren Code, in diesem Fall 40001, ergänzt. Zudem erhalten die Fehlerobjekte in diesem Modell einen Namen und eine Nachricht für die Entwickler, wodurch sie für Menschen leserlicher werden. Über das „properties“ Feld können die Felder näher beschrieben werden, bei denen Fehler aufgetreten sind.

Auf diese Weise können die Fehlerinformationen zur Verständlichkeit und Wartbarkeit des Systems beitragen, da die Entwickler mehr Informationen erhalten. Gerade in der Entwicklung eines verteilten Systems kann dies sinnvoll sein, um den Debugvorgang zu erleichtern, die Entwickler können gegebenenfalls nicht auf die Logs eines anderen Systems zugreifen. Aus dem Grund wird das Fehlerobjekt (auch ErrorResponse Objekt genannt) im Folgenden von jedem Service im HTTP-Response-Objekt übermittelt, sobald ein Fehler auftritt.

4.4.2 Die Zugriffskontrolle

Die Umsetzung der Anwendung erfordert die Absicherung einiger Ressourcen vor den Zugriffen verschiedener Benutzergruppen. (siehe Kapitel 3.3) Um dies zu realisieren hat man in verteilten Systemen im Grunde zwei Möglichkeiten zur Auswahl. Entweder man zentralisiert die Zugriffskontrolle oder man dezentralisiert sie.

Eine zentralisierte Lösung könnte durch die Erweiterung der Datenbank des UMS realisiert werden. Jeder Rolle X könnte man dann z.B. verschiedene Zugriffsrechte zuweisen, die angeben, welche Aktionen ein User mit der Rolle X innerhalb der Anwendungslandschaft ausführen darf. Die Webservices könnten daraufhin bei einem erhaltenen Request den Auth Service des UMS fragen, welche Rechte der User besitzt. Durch die Zentralität würde man einen „Single point of control“ schaffen, wodurch die Zugriffsrechte besser verwaltet werden. Abbildung 4.6 veranschaulicht diese Lösung. Der Client übermittelt in dem Fall bei jedem Request seine Credentials (Email, Passwort). Der Service leitet die Anfrage daraufhin an den Auth Service weiter und ergänzt die Information um die Angabe des Pfades der angefragten Ressource. Nun kann der Auth Service überprüfen, ob der User das richtige Passwort angegeben hat und ob er die nötigen Rechte besitzt, die Ressource anzufragen. Wenn dies nicht der Fall ist, antwortet der Auth Service mit einer entsprechenden Fehlermeldung und übergibt ein ErrorResponse Objekt. (siehe Fehlermodell 4.4.1) Konnte der User authentifiziert und authorisiert werden, liefert der Auth Service ein entsprechende Bestätigung und der Beispiel-Service stellt dem Client die angefragte Ressource zur Verfügung.

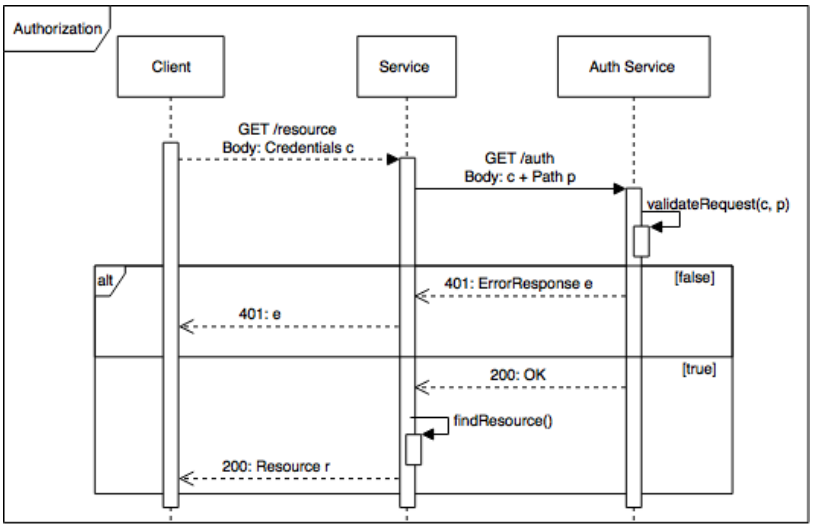

Abbildung 4.6: Hohe Kopplung + Hohe Netzauslastung

Das Problem dabei wäre allerdings, dass das UMS Kenntnis über sämtliche Aktionen innerhalb der Anwendungslandschaft besitzen müsste, wodurch man eine hohe Kopplung zwischen den Systemen schaffen würde. Außerdem würde dies die Netzauslastung erhöhen, da jeder Request einer abgesicherten Ressource zu einem weiteren Request beim UMS führen würde. Um die enge Kopplung zu umgehen, könnte man die Zugriffskontrolle auch dezentralisieren und die Authorisierung den einzelnen Services selbst überlassen. Diese würden lediglich Informationen über die Rolle und die Identität des Users benötigen und könnten daraufhin selbstständig entscheiden, welche Rechte der User besitzt und welche nicht. Das UMS wäre dann also nur noch für die Authentifizierung der User zuständig. Das Problem der Netzauslastung und der dadurch entstehende Overhead würde jedoch weiterhin bestehen.

JSON-Web-Token (JWT) löst das Problem der hohen Netzauslastung. JWT ist ein offener Standard, der es einer Authentifizierungsstelle wie dem „Authentication Service“ ermöglicht, Clients sogenannte Token auszustellen, die bei der nachfolgenden Verwendung eines Dienstes mit übergeben werden können. Die Token sind in sich geschlossen und besitzen eine Signatur, mit Hilfe derer die Datenintegrität und die Authentizität gewährleistet werden können. (vgl Jones u.a. 2015) Die User müssen sich demnach nur noch einmal bei dem „Authentication Service“ authentifizieren und erhalten dadurch Token, mit denen sie ihre Identität bei der Kommunikation mit anderen Services bestätigen können. Die Webservices können die Token daraufhin validieren und entscheiden, welche Rechte der jeweilige User besitzt. Ist ein Token nicht gültig, weil er abgelaufen ist, nicht die nötigen Rechte besitzt oder verändert wurde, kann der Zugriff verweigert werden. Die Token bestehen laut RFC7519 aus den drei Komponenten, „Header“, „Payload“ („Claim Set“) und „Signatur“, wobei jede Komponente „Base64url“ kodiert ist und mit einem Punkt konkateniert wird. Der Header beschreibt dabei die Art der Umsetzung und im Payload werden die Nutzdaten transportiert. (vgl Jones u.a. 2015) Die folgenden Listings zeigen den Aufbau der einzelnen Komponenten.

```
1 // Header
2 {
3   "typ": "JWT",
4   "alg": "HS256"
5 }
6 // Base64url kodierter Header
7 eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

Listing 4.2: JWT: Struktur des JOSE Header

Der Header enthält unter anderem Informationen über die kryptografischen Operationen, die auf dem Payload angewendet werden sollen. So gibt das Feld „typ“ z.B. an, ob der Payload verschlüsselt oder nur kodiert werden soll. Wird er lediglich kodiert, sollte man keine sensiblen Daten mitsenden, da die base64url-Kodierung von jedem, der im Besitz des Token ist, rückgängig gemacht werden kann. Eine passwortgeschützte Verschlüsselung würde allerdings zu einem größeren Rechenaufwand bei den Services führen, da sie die Daten wieder entschlüsseln müssten. Für die in dieser Arbeit beschriebenen Anwendungsfälle würde die Datenintegrität bereits ausreichen, da die Token keine sensiblen Daten wie Passwörter enthalten müssen. Aus dem Grund wird der Payload nicht zusätzlich verschlüsselt und nur die Signatur verwendet. Das Feld „alg“ beschreibt dabei, durch welchen Algorithmus die Signatur erstellt werden soll.

```
1  // Payload (Claim-Set)
2  {
3    "jti": "9a017ab1-0856-4c9d-9b88-5fd20a4b2bd9",
4    "iat": 1512644097,
5    "exp": 1512647697,
6    "user": {
7      "userRef":"http://user-service/users/1",
8      "role":"admin"
9    }
10 }
11 // Base64url kodierter Payload
12 eyJqdGkiOiI5YTAxN2FiMS0wODU2LTRjOWQtOWI4OC01ZmQyMGE0YjJiZDkiLC
       JpYXQiOjE1MTI2NDQwOTcsImV4cCI6MTUxMjY0ODAwOSwidXNlciI6eyJ1c2Vy
       UmVmIjoiL3VzZXJzLzEiLCJlbWFpbCI6InRlc3RAdGVzdC5kZSIsInJvbGUiOiJh
       ZG1pbiJ9fQ.CqRMVquaYkmx_cguMq48avhP219gKPI8PXs9GngUClA
```

Listing 4.3: JWT: Struktur des Payloads

Im Payload, auch Claim-Set genannt, werden Key-Value Paare übermittelt, die verschiedene Bedeutungen haben können. Einige davon sind standardisiert, andere wiederum können beliebig erstellt werden. Das Feld, auch „Claim“ genannt", „iat“ enthält z.B. die Auskunft über das Ausstellungsdatum des Token, während „exp“ angibt, wie lange der Token gültig ist. Die genaue Beschreibung der möglichen Claims findet man im angegebenen RFC. Durch den in dieser Arbeit erstellten Claim „user“ wird es nun möglich, die für die Authorisierung benötigten Informationen zu übertragen. So sind für die Authorisierung lediglich die Informationen wichtig, die beschreiben, um welchen User es sich handelt („userRef“) und welche Rolle dieser

besitzt („role“).

```
// Signatur = Alg((Base64Url(header).Base64Url(payload)), passwort)
CqRMVquaYkmx_cguMq48avhP219gKPI8PXs9GngUClA
```

Listing 4.4: JWT: Erstellen der Signatur

Die Signatur stellt die Datenintegrität sicher, in dem der kodierte Header zusammen mit dem kodierten Payload durch den im Header angegebenen Algorithmus verschlüsselt wird. Auf diese Weise kann nur derjenige, der das Passwort für die Verschlüsselung kennt, die Daten verändern. Die Passwörter für die Ver- und Entschlüsselung müssen die Services also vorher untereinander ausgehandelt haben.

Der Authentication Service ist somit nur noch für das Austelllen der JWT Token und damit für die Umsetzung der User-Authentifizierung zuständig.

Method	Endpoint	Body	Response	JWT
GET	/auth	Credentials	AuthResponse	Nein

Abbildung 4.7: Ausschnitt der Spezifikation des Authentication Services

Da die Anfrage keine Daten auf dem Server verändern muss und ein erneutes Ausführen zu dem gleichen Ergebnis führen würde, soll der Dienst durch einen „GET“-Request umgesetzt werden. Da es die Aufgabe des Authentication Services ist, User zu authentifizieren, erfordert der Request zudem keine Übermittlung eines JWT. Das „Credentials“-Objekt enthält die Emailadresse sowie das Passwort des zu authentifizierenden Users. Die Emailadresse wird benötigt, damit das entsprechende Userobjekt in der Datenbank gefunden werden kann. Für die Validierung des Users wird dann das übermittelte Passwort mit dem des gefundenen Userobjektes abgeglichen. War die Validierung erfolgreich, wird die Identität des Users durch die Übergabe eines entsprechenden Token bestätigt, ohne dass der Authentication Service die Information speichern muss. Der Token wird innerhalb des „AuthResponse“-Objekts übermittelt und ist nur für eine bestimmte Zeit gültig. Um den Token bei der Verwendung anderer Dienste nutzen zu können, muss der entsprechende Client diesen also selbst verwalten, wodurch das REST-Prinzip der Stateless-Server eingehalten werden kann. Konnte der User nicht authentifiziert werden, weil das übermittelte Passwort nicht mit dem gespeicherten Passwort übereinstimmte, erhält der Client eine entsprechende Fehlermeldung mit weiteren Details im ErrorResponse Objekt (siehe Fehlermodell in 4.4.1). Die folgende Abbildung 4.8 veranschaulicht diesen Prozess noch einmal.

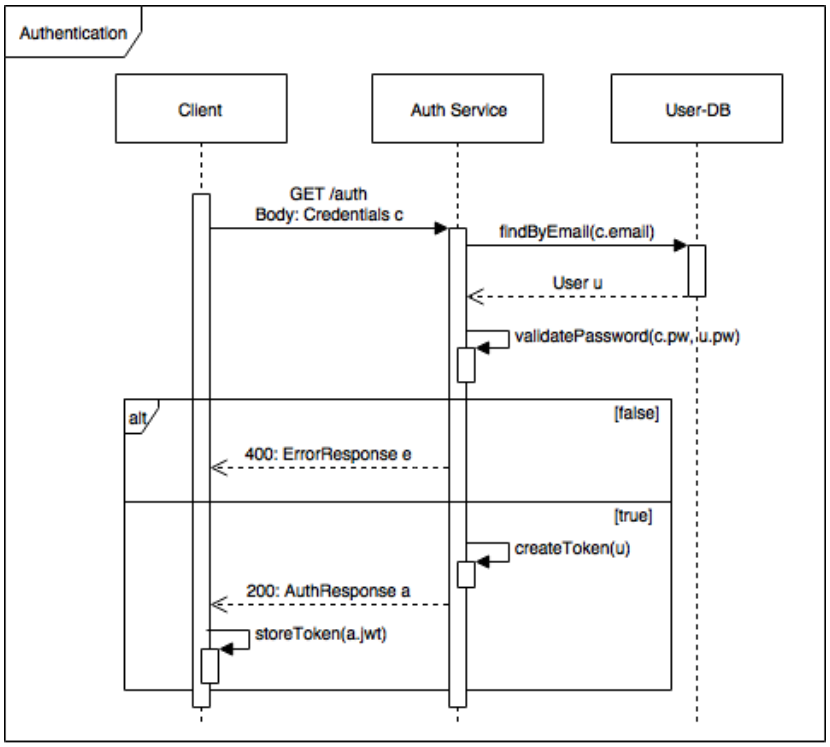

Abbildung 4.8: Ablauf der Authentifizierung eines im System gespeicherten Users

Da die einzelnen Services dezentral entscheiden, welche Ressourcen sie vor den Zugriffen einzelner Benutzergruppen absichern möchten, um die Kopplung der Systeme zu minimieren, erfordert der Zugriff auf abgesicherte Ressourcen die Übergabe eines JWT. In Abbildung 4.9 wird der Prozess der Authorisierung anhand eines fiktiven Services genauer beschrieben. Der Service besitzt dabei einen Endpunkt „/resources“ und kann einen „GET“-Request verarbeiten, um auf eine abgesicherte Ressource zuzugreifen. Damit dies gelingen kann, muss im „Authorization“-Header des Requests, der durch den „Authentication“-Prozess in Abbildung 4.8 ermittelte JWT angegeben werden. Dieser kann dann von dem entsprechenden Service validiert werden, indem der Service den Token mittels desselben Passwortes entschlüsselt, mit dem der Token vom Authentication Service verschlüsselt wurde. War die Validierung erfolgreich, kann die Ressource aus der Datenbank geladen werden. Ansonsten wird der Client über den aufgetretenen Fehler informiert.

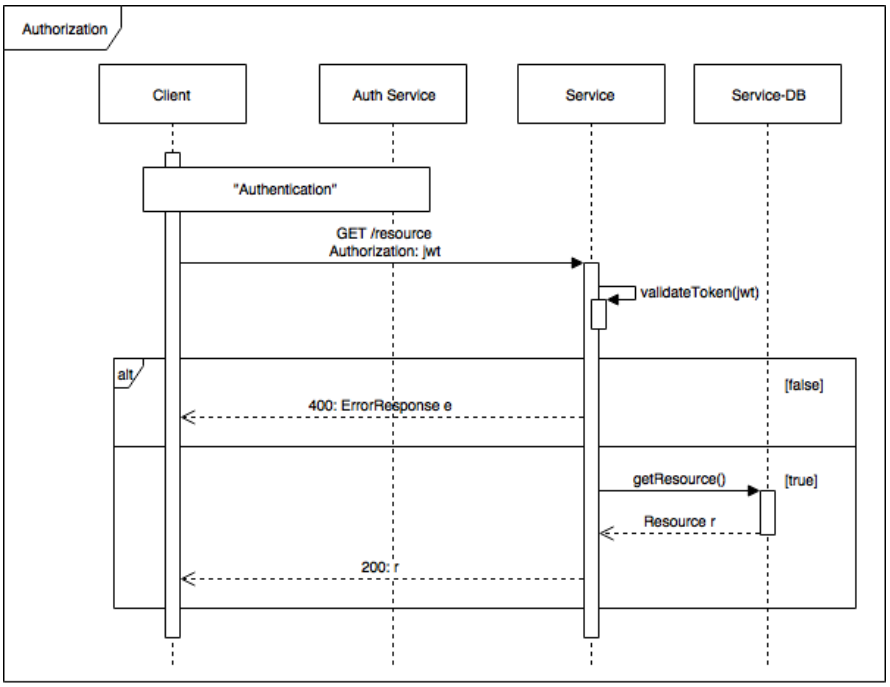

Abbildung 4.9: Ablauf des Authorisierungsprozesses

4.4.3 Ressourcen erstellen: POST vs. PUT

Die Services werden nach den REST-Prinzipien entworfen und kommunizieren über das HTTP-Protokoll miteinander. Obwohl das Protokoll mehr Methoden bereitstellt, können sämtliche Prozesse über die Methoden GET, POST, PUT und DELETE durchgeführt werden. Die Wahl der korrekten Methode ist dabei sehr wichtig, da ansonsten Fehler entstehen können.

Jede Methode besitzt im Grunde zwei Eigenschaften. Die erste Eigenschaft beschreibt die Sicherheit der Ausführung. So gilt eine Methode als sicher, wenn sie keine anderen Aktionen auf dem Server auslöst, außer der Beschaffung von Daten. Lediglich die Methoden GET und HEAD gelten somit als sicher. Die zweite Eigenschaft beschreibt die sogenannte Idempotenz. Eine Methode wird als idempotent betrachtet, wenn ein erneutes Ausführen zu demselben Ergebnis führt. Zu den idempotenten Methoden gehören die Methoden GET, HEAD, PUT und DELETE. (vgl. Fielding u.a. 1999, S. 33) Obwohl sichere Methoden bevorzugt ausgeführt werden sollten, reichen diese natürlich nicht aus, um sämtliche Anwendungsfälle abzudecken. Unsichere Methoden müssen ebenfalls verwendet werden. Die Idempotenz spielt im weiteren Verlauf eine wichtige Rolle, da im Falle eines Systemausfalls Nachrichten häufig

mehrmals gesendet werden müssen. Wird in so einem Fall eine nicht-idempotente Methode wie POST benutzt, können Fehler entstehen, die behandelt werden müssen. Am kritischsten wirkt sich die Wahl der geeigneten Methode auf das Erstellen der Ressourcen aus. Hierfür stehen laut Protokoll die Methoden POST und PUT zur Verfügung. Während POST im Grunde ausschließlich für das Erstellen von Ressourcen verwendet werden sollte, wird PUT neben der Erstellung häufig auch für das Aktualisieren einer Ressource eingesetzt. Dabei gilt frei übersetzt:

Wenn die Request-URI auf eine bestehende Ressource zeigt, soll die übergebene Ressource als eine modifizierte Version der alten Ressource betrachtet werden und die Ressource ersetzen, ansonsten soll eine neue Ressource an der entsprechenden Stelle angelegt werden.
(vgl. Fielding u. a. (1999, S. 36))

Wird eine Ressource jedoch mit einem POST Request erstellt und der Server fällt anschließend aus, könnte eine entsprechende Wiederholung der Client Anfrage zu einer Fehlermeldung führen. Dies kann zum Beispiel beim Erstellen eines Userkontos der Fall sein, wenn pro Emailadresse nur ein User im System existieren dürfte. Würde man stattdessen einen PUT-Request für das Erstellen der Ressource verwenden, würde ein erneuter Request nicht zu einem Fehler führen. Eine aufwändige Fehlerlogik seitens des Clients kann somit umgangen werden. Um das Erstellen der Ressourcen durch einen PUT-Request realisieren zu können, müssten die entsprechenden IDs durch den Client erzeugt werden. Dieser müsste dann allerdings Kenntnis über bereits im System existierende IDs besitzen. Aus dem Grund könnte man global eindeutige UUIDs einführen. Die Verwendung der UUIDs und die Erzeugung derselbigen sorgt allerdings nicht nur für eine höhere Kopplung von Client und Server, sondern verlagert auch die Zuständigkeiten. Des Weiteren sorgt dies für Effizienzprobleme bei den Datenbankzugriffen, wie die nachfolgende Perfomance Analyse veranschaulicht. Die Frage, ob PUT oder POST die richtige Methode für das Erstellen von Ressourcen ist, ist also viel mehr eine Frage der Effizienz der Datenbankzugriffe und der Zuständigkeiten.

Abbildung 4.10 zeigt die unterschiedlichen Typen, die für Primärschlüssel verwendet werden können und vergleicht deren Effizienz beim Einfügen mehrerer Millionen Einträge in einer MySQL Datenbank. Die X-Achse gibt dabei die benötigte Laufzeit in Minuten an und die Y-Achse zeigt die Anzahl der getätigten Einträge. Die fettgedruckten Linien stehen für die Anzahl der Einträge im Verlauf der Zeit und die gepunkteten Linien für die durchschnittliche Eintragsrate pro Sekunde.

Die Abbildung macht deutlich, dass die Effizienz bei der Verwendung ungeordneter UUIDs schon bei wenigen tausend Einträgen stark abnimmt und bei sehr großen Datenmengen fast unbenutzbar wird. Die Verwendung von Integer-Primärschlüsseln bleibt dagegen sehr effizient und steigt linear. Das Problem ist laut dem Verfasser der Statistik, das UUIDs nicht sequentiell indexiert werden können. Durch einen Workaround, bei dem der Verfasser den TIMESTAMP Part der UUIDs (Version 1, siehe Leach u.a. 2005) neu ordnet, konnte er ein lineares Eintragen der UUIDs erreichen. Dennoch müssen dadurch mehr Daten gespeichert werden und es gibt einen gewissen CPU Overhead durch die Umordnung, wodurch die Verwendung geordneter UUIDs ab einer bestimmten Größenordnung auch nur noch halb so effizient wäre wie die Verwendung von Integer-Primärschlüsseln.

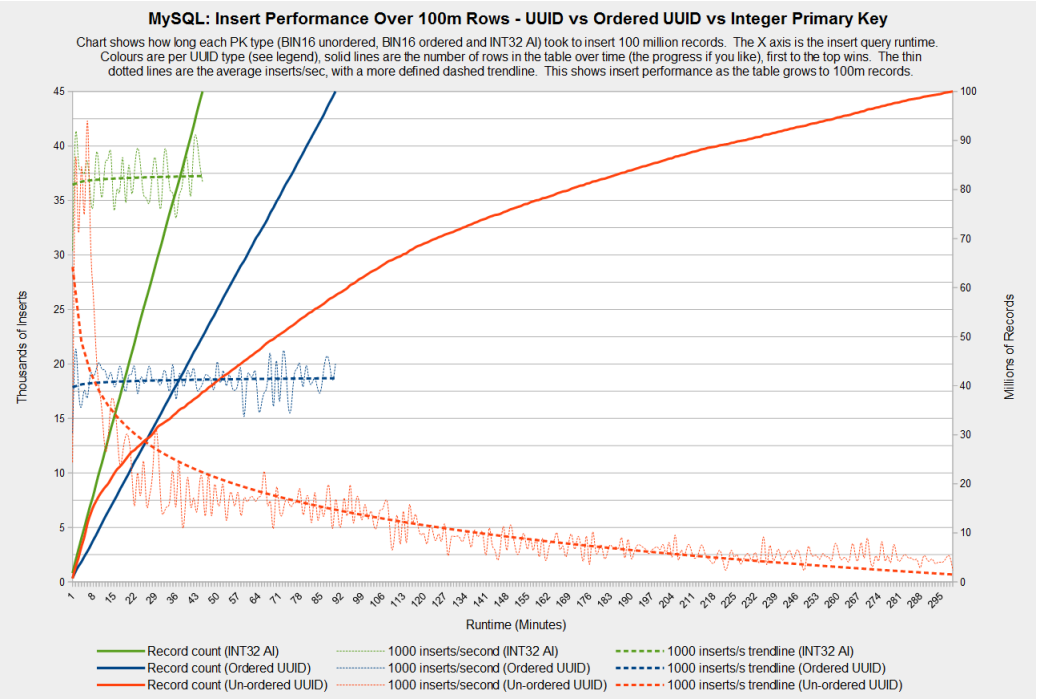

Abbildung 4.10: MySQL Effizienzvergleich: Integer vs. UUIDs
Quelle: https://cjsavage.com/guides/mysql/insert-perf-uuid-vs-ordered-uuid-vs-int-pk.html (20.11.2017)

Zusammenfassend ist festzustellen, dass die Verwendung von UUIDs nur dann eingesetzt werden sollte, wenn es unbedingt erforderlich ist. Dieselbe Schlussfolgerung kann man transitiv für das Erstellen von Ressourcen durch einen PUT Request ziehen, da die Methode unmittelbar von den UUIDs abhängig ist. Um die Zuständigkeiten den entsprechenden Services zu

überlassen, die Kopplung gering zu halten und die Effizienz der Datenbankzugriffe nicht zu beeinträchtigen, werden die Ressourcen im Folgenden also immer mittels eines POST-Requests erstellt und mittels eines PUT Requests aktualisiert.

Die entsprechenden Fehlerfälle, die bei einem erneuten Ausführen eines POST Requests auftreten können, müssen somit vom jeweiligen Client behandelt werden. Da die Webserver Komponente ein Client der verschiedenen Webservices ist, liegt es also an ihm, auf die Fehler entsprechend zu reagieren. (Über die Verwendung von PUT zur Aktualisierung einer Ressource ließe sich ebenfalls streiten, da die Aktualisierung meistens nur Teile der Ressource betrifft und demnach durch einen PATCH-Request durchgeführt werden sollte, wenn man es ganz genau nimmt. PUT ersetzt viel mehr die bestehende Ressource durch eine modifizierte. Da diese Entscheidung allerdings nicht so kritisch ist, wie die Verwendung von UUIDs, soll dies an dieser Stelle nur einmal erwähnt werden. Viele Anwender verstehen einen PUT Request zudem als das Aktualisieren einer Ressource, weswegen die Verwendung von PUT für diesen Fall akzeptabel wäre.)

4.4.4 N-zu-M-Beziehungen

Für das Erstellen der Tarife müssen Tarife an Regionen gebunden werden und optional an Boni. (siehe 4.2) Da es sich bei den Beziehungen jeweils um N-zu-M-Beziehungen handelt, gibt es besondere Anforderungen an die REST-Schnittstellen. So werden Tarife zum Beispiel nicht nur in mehreren Regionen angeboten, Regionen können zudem auch für mehrere Tarife gelten. Aus dem Grund kann es sinnvoll sein, für jede Entität einen eigenen Endpunkt zu definieren. Der Endpunkt „POST: /tariffs/{id}/regions“ würde zum Beispiel wenig Sinn ergeben, um Regionen im System anzulegen, da man eine Region gegebenenfalls in einem anderen Tarif wiederverwenden möchte. Sinnvoller wäre es die Endpunkte in „/tariffs“ und „/regions“ aufzusplitten und die Relationen zwischen den beiden Entitäten durch einen eigenen Endpunkt zu definieren, um die Endpunkte eindeutig und somit verständlicher zu machen.

Method	Endpoint	Body	Response	JWT
GET	/regions		List<Region>	Nein
	/regions/{id}		Region	Nein
POST	/regions	Region	ResourceLink	Ja
PUT	/regions/{id}	Region	ResourceLink	Ja
DELETE	/regions/{id}			Ja

Abbildung 4.11: Spezifikation des Region Services

Abbildung 4.11 zeigt, dass der Region Service fünf Endpunkte besitzt, die CRUD Operationen (Create, Read, Update, Delete) zur Verfügung stellen. Während die „GET“-Endpunkte keine Angaben eines JWT benötigen, erfordert das Erstellen, Aktualisieren und Löschen der Regionen die vorhergehende Authentifikation beim Authentication Service und die Übergabe des ermittelten JWT. Nur ein User mit der Rolle „Admin“ sollte solche kritischen Operationen durchführen können. (vgl. Anwendungsfall UC2.02. in 3.3) Im Objekt „Region“ sind jeweils die Daten des entsprechenden Tabelleneintrags der Datenbank enthalten. Zusätzlich wird das Objekt im Sinne der HATEOAS Prinzipien um die Referenz auf sich selbst ergänzt, damit der Client weiß, welchen Endpunkt er ansprechen muss, um Aktionen auf der Ressource auszuführen. Das Objekt „ResourceLink“ enthält dagegen nur die Referenz auf die erstellte, bzw. aktualisierte Ressource. (siehe Swagger.io Beschreibung für den genauen Aufbau der Objekte)

Während sich der Aufbau des Bonus Service von dem des Region Service kaum unterscheidet, ist der Aufbau des Tariff Service nun etwas umfangreicher.

Method	Endpoint	Body	Response	JWT
GET	/tariffs		List<Tariff>	Nein
	/tariffs/{id}		Tariff	Nein
POST	/tariffs	Tariff	ResourceLink	Ja
	/tariff-regions	TRRelation	ResourceLink	Ja
	/tariff-bonuses	TBRelation	ResourceLink	Ja
PUT	/tariffs/{id}	Tariff	ResourceLink	Ja
DELETE	/tariffs/{id}			Ja

Abbildung 4.12: Spezifikation des Tariff Services

Da Boni und Regionen im Grunde unabhängig von Tarifen verwendet werden können, umgekehrt Tarife aber an Regionen und optional an Boni gebunden werden müssen, kann man rechtfertigen, die Erstellung der Relationen, dem Tariff Service zu überlassen. So besitzt der Tariff Service neben den Standard CRUD-Operationen nun auch die Endpunkte „POST /tariff-regions“ und „POST /tariff-bonuses“, über die die Relationen „tariff_bonuses“ und „tariff_regions“ des Datenmodells umgesetzt werden können. Die Objekte TRRelation und TBRelation enthalten dabei jeweils die Relation in Form der entsprechenden IDs. Auf diese Weise können Tarife, wie in der folgenden Abbildung beschrieben, umgesetzt werden.

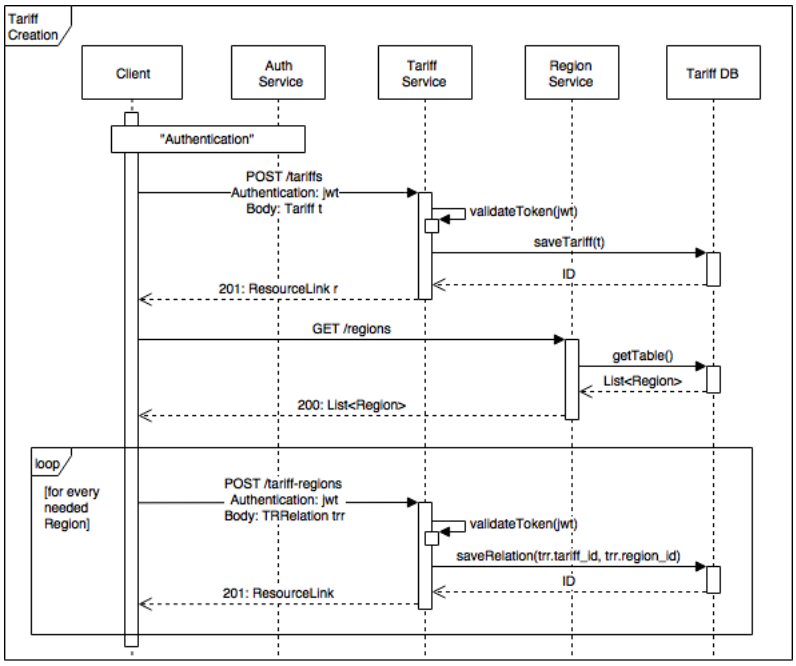

Abbildung 4.13: Ablauf der Tariferstellung

Das Erstellen von Tarifen setzt wie beschrieben die Übergabe eines JWT voraus. Aus dem Grund muss ein Client sich zunächst beim Authentication Service authentifizieren. Anschließend können die für die Erstellung benötigten Daten (siehe API Beschreibung) per „POST /tariffs“ an den Tariff Service gesendet werden. Dieser validiert den Token, speichert den übergebenen Tarif in der Datenbank und übergibt die entsprechende ID des erstellten Tarifs innerhalb des ResourceLink-Objekts an den Client. Für die Bindung von Regionen an Tarife muss der Client nun noch Kenntnis über die Regionen des Systems besitzen, weshalb er einen „GET“-Request an den Endpunkt „/regions“ sendet. Dieser wird vom Region Service bearbeitet und liefert eine Liste der Regionen im System. Auf diese Weise hat der Client Kenntnis über die IDs der Regionen sowie über die ID des Tarifs und kann durch einen „POST“-Request an „/tariff-regions“ die Relation erstellen. Die Bindung von Boni an Tarife kann auf dieselbe Weise durchgeführt werden.

Durch die Umsetzung eines entsprechenden Endpunktes für das Bearbeiten der n-zu-M-Beziehungen bleiben Tarife erweiterbar. Die Ressourcen und die Services des Systems sind in diesem Fall loser gekoppelt und könnten ebenso in anderen Systemen wiederverwendet werden. Sollten Tarife oder andere Ressourcen in Zukunft weitere N-zu-M-Beziehungen umsetzen müssen, kann dies auf dieselbe Weise getan werden.

4.4.5 Umsetzung der Anwendungsfälle

Im Folgenden soll nun gezeigt werden, wie die Schnittstellen der Microservices von der Webserver-Komponente genutzt werden, um die Anwendungsfälle zu realisieren. Aus Zeitgründen soll an dieser Stelle nur der Anwendungsfall „UC2.04 Strom bestellen“ beschrieben werden. (siehe 3.3) Durch den Prozess werden sämtliche übermittelten Daten im System gespeichert. Für den vollständigen Abschluss des „Bestellprozesses“ müsste der Kunde den Vertrag allerdings noch unterschrieben an das Unternehmen senden, wodurch der „Vertragsabschluss“-Prozess ausgelöst werden würde.

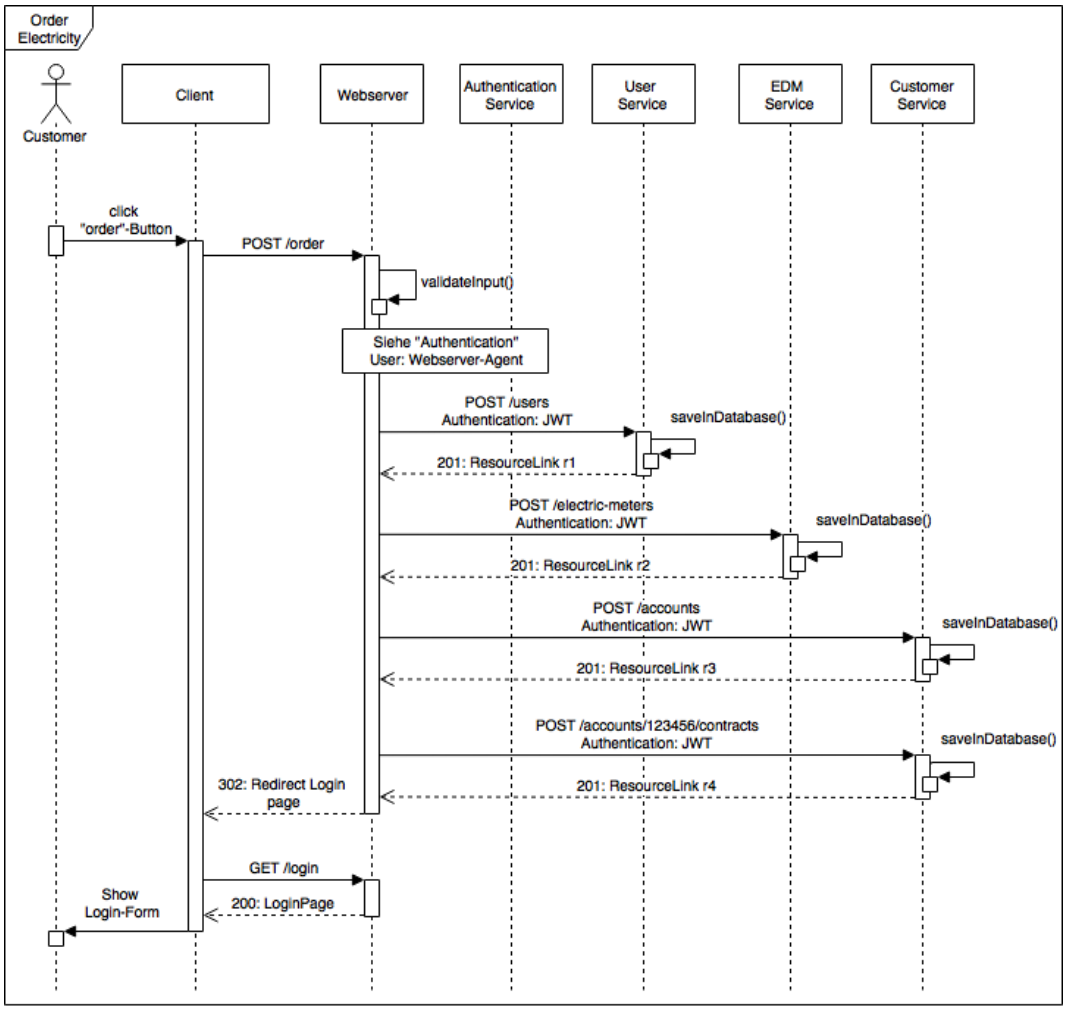

Abbildung 4.14: Speicherung der Bestellung

Abbildung 4.14 beschreibt die verteilte Speicherung der Bestellung. Ausgelöst wird der Prozess durch den Kunden, indem er das Bestellformular im Webbrowser absendet. Da die verteilte Kommunikation zu diversen Fehlern führen könnte, wird an dieser Stelle nur der Erfolgsfall gezeigt. Würde man jeden denkbaren Fehler bei einer fehlerhaften Kommunikation berücksichtigen, würde das Diagramm schnell unübersichtlich werden.

Nachdem die Webserver-Komponente den Request des Clients entgegengenommen hat, überprüft sie zunächst, ob alle Daten vollständig und korrekt übermittelt wurden. Wenn dies der Fall war, authentifiziert sie sich beim „Authentication Service“ als User mit der Rolle „Webserver-Agent“. Auf diese Weise erhält die Komponente ein gültiges JWT mit Adminrechten. Das JWT wird in der folgenden Kommunikation mit den entsprechenden Microservices im „Authentication“-Header übergeben und kann daraufhin von den einzelnen Services validiert werden. Da an dieser Stelle nur der Erfolgsfall dargestellt werden soll und die genauen Schnittstellenbeschreibungen in der „Swagger.io“-API zu finden sind, wird die Validierung des Token seitens der Services in dieser Darstellung weggelassen. Anschließend werden die übergebenen Ressourcen in den entsprechenden Datenbanken gespeichert. Die jeweiligen Antworten enthalten dabei die Referenz auf die erstellte Ressource.

Die Reihenfolge, mit der die Ressourcen gespeichert werden, muss eingehalten werden, da einige Ressourcen wie das Kundenkonto oder der Stromzähler an andere Ressourcen, wie z.B. das Benutzerkonto, gebunden sind. In dem „POST“-Request auf den Endpunkt „/electric-meters“ muss z.B. die im „ResourceLink r1“ enthaltene Referenz auf das Benutzerkonto übermittelt werden und für den „POST“-Request auf „/accounts/123456/contracts“ wird wiederum die Referenz auf den Stromzähler benötigt.

Wenn alle Ressourcen gespeichert wurden, setzt die Webserver-Komponente den Client darüber in Kenntnis, indem er per Redirect auf die „Login“-Seite weitergeleitet wird.

Da die Ressourcen allesamt wie beschrieben mit „POST“-Requests erstellt werden sollen, liegt es an dem Webserver, auf Fehler entsprechend zu reagieren. Ein Fehler kann z.B. auftreten, nachdem ein Service die Ressource in der Datenbank gespeichert hat. Fällt der Service daraufhin aus, kann es sein, dass der Webserver nicht mehr über den Erfolg informiert werden kann. In diesem Fall muss der Prozess abgebrochen und wiederholt werden. Da die Ressource bereits existiert, würde ein erneutes Absenden des Requests nun zu einem Fehler führen. Um den Fehler zu behandeln könnte sich der Webserver mit den vom Client übermittelten Userdaten

(Email, Passwort) beim „Authentication Service“ authentifizieren. War die Authentifizierung erfolgreich kann der Webserver sicher sein, dass die Daten durch den Kunden übermittelt wurden und kann die in der Antwort übermittelte Referenz verwenden, um auf die bereits erstellten Ressourcen per „GET“-Request zuzugreifen. Hierfür müssen zwei Fälle unterschieden werden:

1. **Der Webserver wird nicht darüber in Kenntnis gesetzt, dass der User bereits erstellt wurde.**
   In dem Fall reicht es aus, wenn der Webserver den User des Kunden beim „Authentication Service“ authentifiziert. Dadurch erhält der Webserver im „AuthResponse“-Objekt die Referenz auf den User und kann diese für weitere Requests verwenden.

2. **Der Webserver wird nicht darüber in Kenntnis gesetzt, dass eine andere Ressource nicht erstellt wurde.**
   Dieser Fall erfordert ebenso die Authentifizierung des Users beim „Authentication Service“. Anschließend kann der übermittelte JWT verwendet werden, um die erstellten Ressourcen per „GET“-Request zu erhalten. Da die ID der Ressource allerdings nicht bekannt ist, könnte ein „GET“-Request auf den exemplarischen Endpunkt „/resources“ mit der Angabe eines Query-Parameters: „?user_reference=<user_reference>“ erfolgen. Der entsprechende Service könnte dann in der Datenbank nach der Ressource suchen, die an das Userkonto gebunden ist. Der Zugriff wäre aufgrund der Suche mittels eines Strings zwar ineffizient, würde aber auch nur im Fehlerfall ausgeführt werden. Innerhalb der entsprechenden Antwort-Objekte befindet sich dann die Referenz auf die erstellte Ressource.

Die Lösung behandelt zwar einige Fehlerfälle, die durch den Ausfall der Microservices und auch durch den Ausfall der Webserver Komponente entstehen können, dennoch kann die Bestellung dadurch unvollständig gespeichert werden. In einer überarbeiteten Version könnte es deswegen sinnvoll sein, die Bestellung zunächst vollständig in einem „Bestellsystem“ zwischenzuspeichern und die anschließende verteilte Transaktion mit einer Messaging Lösung zu realisieren. Die in der Arbeit beschriebene Lösung wurde aus Zeitgründen ausgewählt und sollte nicht in Produktionsumgebungen eingesetzt werden.

4.5 Technologie Stack

Für die Umsetzung der Systemlandschaft müssen verschiedene Technologien verwendet werden. In der folgenden Abbildung wird das Zusammenspiel der verwendeten Technologien dargestellt.

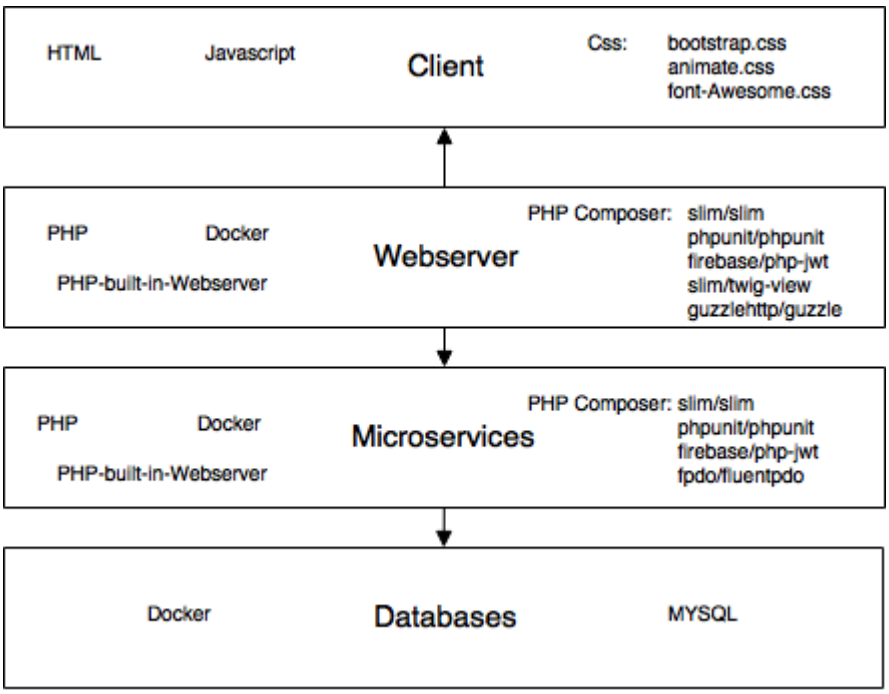

Abbildung 4.15: Übersicht der verwendeten Technologien

Für die Realisierung des Clients werden HTML, Javascript und verschiedene CSS Frameworks eingesetzt. Auf diese Weise kann die Anwendung über einen Webbrowser auf den Endgeräten der Nutzer ausgeführt werden. Während HTML unter Anderem für die Übermittlung der Form-Objekte und des Webcontents zuständig ist, werden Javascript und die CSS Frameworks lediglich für die Manipulation des Benutzerinterfaces eingesetzt. Bereitgestellt werden die entsprechenden Dateien durch die Webserver Komponente. Diese verarbeitet die Requests des Clients und kommuniziert dafür mit den entsprechenden Microservices. Für die Umsetzung der einzelnen Microservices und des Webservers kann es an dieser Stelle sinnvoll sein, dasselbe Framework zu verwenden, um die Komplexität nicht weiter zu erhöhen und den Fokus auf das Wesentliche zu reduzieren. Aus diesem Grund wurden folgende Kriterien an das Web-Framework formuliert:

1. Verständlichkeit
   Das Framework muss im weiteren Projektkontext von anderen Entwicklern verwendet werden können. Da der Projektfokus auf der Container-Technologie und der Anwendungsintegration liegt und die Projekte in der Regel zeitlich auf ein Semester begrenzt sind, sollte die Einarbeitungszeit nicht viel Zeit beanspruchen. Aus dem Grund sollte das Framework leicht verständlich sein und nicht mit vielen Features überladen sein.

2. Leichtgewichtigkeit
   Das Framework sollte wenig Ressourcen verbrauchen, um die Größe der Container zu reduzieren. Auf diese Weise können die Services im weiteren Projektkontext besser skaliert werden.

3. REST Schnittstellen
   Das Framework sollte die Umsetzung von REST-Schnittstellen ermöglichen, damit die Microservices wie beschrieben realisiert werden können.

4. Erweiterbarkeit
   Das Framework sollte leicht erweitert werden können und das Ergänzen fehlender Funktionalitäten ermöglichen. Dazu gehört die Integration vorhandener Bibliotheken sowie das Ergänzen eigenen Codes. Im besten Fall wird dies durch einen Dependency Injection Ansatz unterstützt.

Aus diesen Gründen ist die Wahl auf ein Microframework namens „Slim“ gefallen. In seinem Kern ist das Slim-Framework „ein Dispatcher, der einen HTTP-Request entgegennimmt, die entsprechende Callback-Routine ausführt und eine HTTP-Response zurückliefert“. (vgl. Josh Lockart 2017) Natürlich hätte man auch ein bekannteres Framework wie Spring (Java) oder Ruby on Rails (Ruby) verwenden können. Diese besitzen jedoch diverse Funktionalitäten, die für die Umsetzung nicht benötigt werden und erschweren dadurch gegebenenfalls den Einstieg.

```
1 // 1. Initialize application
2 $app = new \Slim\App();
3 // 2. Define routes
4 $app->get('/hello', function ($request, $response) {
5     return $response->write("Hello, World!");
6 });
7 // 3. Run application
8 $app->run();
```

Listing 4.5: Das „Hello, World“-Beispiel.

Listing 4.5 zeigt die kleinstmögliche Anwendung, die mit dem Slim-Framework erstellt werden kann. Die Anwendung ist dabei in drei Schritte unterteilt. Im ersten Schritt wird die Applikation initialisiert. Anschließend registriert der Befehl „$app->get()“ einen einzelnen Endpunkt. In diesem Fall beschreibt er die Callback Routine bei einem „Get-Request“, der an den Endpunkt „/hello“ gerichtet ist. Die Callback-Routine erhält ein PSR7-Request Objekt sowie ein PSR7-Response Objekt. (PSR7 stellt eine einheitliche Schnittstelle für das entsprechende HTTP-Objekt bereit.) Als Antwort auf den Request wird das berühmte „Hello, World!“ in das Response-Objekt geschrieben und zurückgegeben. Im letzten Schritt ist die Konfiguration der Anwendung abgeschlossen und die Anwendung wird ausgeführt.

Das Framework bietet zwar noch mehr Funktionalitäten, wie das Ergänzen von Middleware oder einen DI-Container für die Umsetzung einer Dependency Injection-Strategie - die Funktionalitäten reichen alleine allerdings noch nicht für die Realisierung der Microservices und der Webserver Komponente aus. Aus dem Grund müssen weitere Bibliotheken ergänzt werden. Dazu gehören die Bibliotheken „phpunit“ (https://phpunit.de/) für das Implementieren und Ausführen von Tests und „firebase/php-jwt“ (https://github.com/firebase/php-jwt) für das Erstellen und Entschlüsseln der JWT. Der Webserver verwendet außerdem „twig“ (https://twig.symfony.com/doc/2.x/) für das Erstellen der UI und „guzzle“ (http://docs.guzzlephp.org/en/stable/) als HTTP Client für die Kommunikation mit den Microservices. Die Microservices verwenden dagegen noch die Bibliothek „fluentpdo“ (https://github.com/envms/fluentpdo) für die MySQL Datenbankzugriffe. Das mag auf den ersten Blick nach vielen neuen Technologien aussehen - die verwendeten Schnittstellen der einzelnen Bibliotheken sind jedoch gut dokumentiert und nicht komplex. Über den Dependency Injection-Container (DI-Container) vom Slim-Framework können die Bibliotheken leicht in das Projekt integriert werden, wie in Kapitel 5 noch genauer gezeigt wird. Installiert werden die Bibliotheken dann durch PHP Composer. (siehe https://getcomposer.org/) PHP Composer ist ein Dependency Manager für PHP-Projekte, der die benötigten Bibliotheken auf Basis einer JSON-Datei in das Projektverzeichnis installiert.

Außerdem werden der Webserver, die einzelnen Microservices und die verschiedenen Datenbanken jeweils in Docker Containern isoliert. Auf diese Weise können Tools wie Docker-Compose oder Kubernetes verwendet werden, um die Anwendungslandschaft auszuführen und die Service Discovery zu realisieren. Als Datenbank wird MySQL eingesetzt, da eine Open Source Lösung für die Speicherung relationaler Daten benötigt wird.

4.6 Entwurf der Microservices/des Webservers

Da das Slim-Framework in gewisser Weise eine Architektur vorgibt, ist der Aufbau der Microservices und des Webservers in weiten Teilen ähnlich. Im Folgenden wird der Aufbau der Microservices beschrieben.

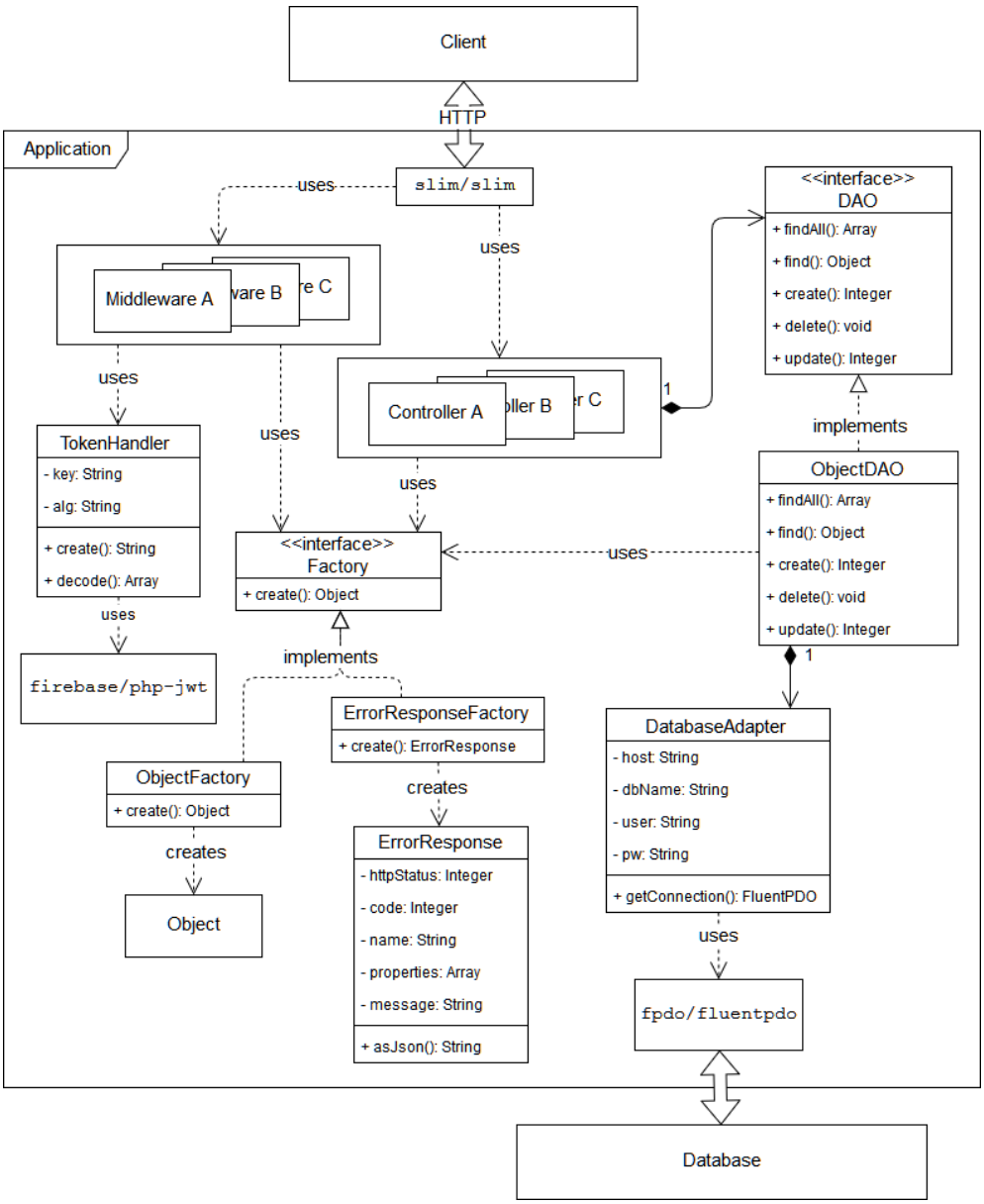

Abbildung 4.16: Aufbau der Microservices

Wie im letzten Abschnitt gezeigt wurde, übernimmt das Slim-Framework die Aufgaben eines Dispatchers und kann das HTTP-Request- und das HTTP-Response-Objekt an entsprechende Callback-Routinen weiterleiten. Um die Anwendung zu strukturieren, hat man die Möglichkeit, anstatt der Callback Routinen, Methoden innerhalb eines Controllers anzugeben, die die Callback-Routinen umsetzen sollen. Auf diese Weise kann die Anwendung im Sinne des Model-View-Controller-Patterns (MVC) realisiert werden. Da die Microservices keine View-Objekte übergeben müssen, werden die Views innerhalb der Microservices nicht integriert. Stattdessen müssen die Services verschiedene Geschäftsoperationen ausführen, wie z.B. das Erstellen oder Aktualisieren eines Kundenkontos. Diese Anforderungen ergänzen sich mit dem „Domain Driven Design"-Pattern (DDD). Das DDD-Pattern unterteilt die Model-Schicht des MVC Patterns in die Schichten „Domain" und „Infrastructure". Die Domain-Schicht enthält dabei die Geschäftslogik und die Infrastructure-Schicht kapselt die Datenbankzugriffe. Die Controller kommunizieren in diesem Fall nur noch mit entsprechenden Data Access Objects (DAO) und müssen sich keine Gedanken darüber machen, wie die Daten gespeichert wurden. Dies hat den Vorteil, dass die einzelnen Schichten austauschbar bleiben und wartbarer werden. Da die Operationen der DAOs häufig dieselben Ziele verfolgen, kann ein einheitliches Interface erstellt werden, dass CRUD-Operationen ausführt. Eine Implementation des Interfaces benötigt nur noch eine Verbindung zur MySQL Datenbank, um die Operationen durchzuführen. Hierfür wird die Klasse „DatabaseAdapter" verwendet, die die Datenbankverbindung herstellt, indem es ein Objekt der Klasse FluentPDO zurückgibt. FluentPDO wurde durch das Framework „fpdo/fluentpdo" integriert und ermöglicht das Ausführen verschiedener SQL-Queries.

Mittels des Factory-Patterns kann die Objekterstellung durch eine einheitliche Schnittstelle verborgen werden. Auf diese Weise müssen Clients nicht mehr wissen, wie die Objekte genau erstellt werden, wodurch die Kopplung geringer wird. Ändert sich etwas bei der Initialisierung, kann dies an einer zentralen Stelle verwaltet werden, wodurch ein Single-Point-Of-Control ermöglicht wird. Die Klasse „ErrorResponseFactory" wird von allen Microservices verwendet, um „ErrorResponse" Objekte zu erstellen. Diese setzen das Fehlermodell aus Abschnitt 4.4.1 um. Die Klasse „ObjectFactory" wurde nur zur Veranschaulichung ergänzt.

Des Weiteren ermöglicht das Slim-Framework das Ergänzen einer Middleware. Auf diese Weise können die Requests abgefangen werden, bevor die Controller sie verarbeiten und die Response-Objekte bearbeitet werden, bevor sie an den Client gesendet werden.

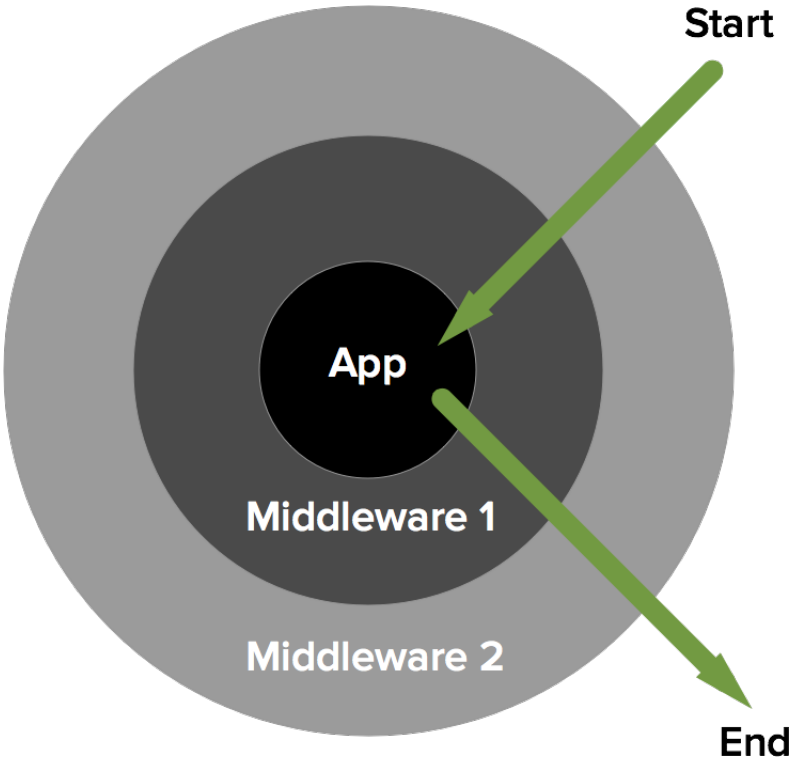

Abbildung 4.17: Funktion der Middleware innerhalb des Slim-Frameworks
Quelle: https://www.slimframework.com/docs/concepts/middleware.html (20.12.2017)

Auf diese Weise kann die Authorisierung mittels einer Middleware umgesetzt werden, indem überprüft wird, ob im Request-Objekt der „Authentication"-Header gesetzt wurde. Wenn dies der Fall war, wird der „TokenHandler" genutzt, um den Token zu validieren. Der TokenHandler kennt den „Key" mit dem die JWT vom „Authentication Service" verschlüsselt wurden und verwendet die Bibliothek „firebase/php-jwt", um die tatsächliche Entschlüsselung durchzuführen. Wenn der Token entschlüsselt werden konnte, kann die nächste Middleware, bzw. der entsprechende Controller ausgeführt werden.

Die Webserver-Komponente wird im Grunde ähnlich realisiert. Der Hauptunterschied ist, dass er keine Datenbankzugriffe durchführen muss, weswegen die DAOs wegfallen. Stattdessen integriert er den HTTP-Client „Guzzle", um mit den entsprechenden Microservices zu kommunizieren und wird um das Konzept der „Views" erweitert. Die Webserver-Komponente wird also durch eine klassische MVC-Architektur realisiert.

5 Realisierung

Nachdem die Anwendungslandschaft spezifiziert und entworfen wurde, soll nun die Realisierung beschrieben werden. Dazu gehört die Beschreibung des GitLab-Projekts und eine Beschreibung, wie die Anwendungslandschaft innerhalb eines Continuous Delivery Prozesses weiterentwickelt werden kann.

5.1 Umsetzung der Microservices und des Webservers

In diesem Abschnitt soll gezeigt werden, wie die Microservices und der Webserver realisiert wurden. Da sowohl die Microservices als auch der Webserver mit dem Slim-Framework entwickelt wurden, ist der Grundaufbau der Komponenten identisch. Der Aufbau orientiert sich dabei an dem offiziellen „Slim-Skeleton“ Projekt des Frameworks und wurde für diese Arbeit angepasst. (siehe https://github.com/slimphp/Slim-Skeleton)

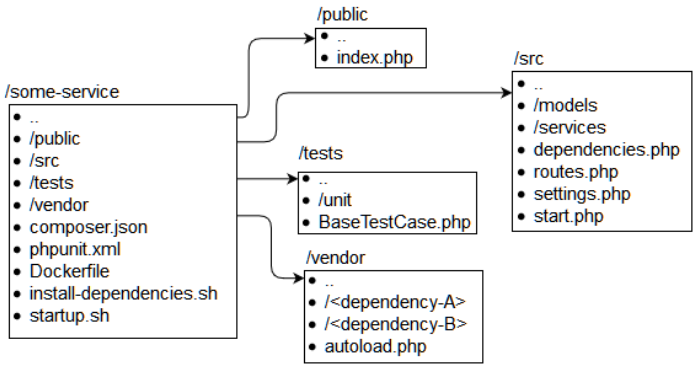

Abbildung 5.1: Projektstruktur der Microservices

5.1.1 “/public“

Das Verzeichnis „/public“ wird dafür genutzt, öffentlich zugängliche Dateien zu hinterlegen, wie zum Beispiel die Datei „index.php“, die den Eingangspunkt der Anwendung darstellt. In der aktuellen Version werden die Services und die Webserver-Komponente nur durch den

PHP-built-in-Webserver betrieben. Das Starten durch einen sichereren und leistungsfähigeren Webserver wie Apache kann jederzeit erweitert werden und wird in der Dokumentation des Frameworks näher beschrieben. In beiden Fällen muss die Datei als Einstiegspunkt angegeben werden. Die Webserver-Komponente der Anwendungslandschaft kann in dem „/public“-Verzeichnis zudem weitere Dateien hinterlegen, wie die CSS- oder Javascript-Dateien, die für das Frontend benötigt werden.

5.1.2 “/src“

Im Verzeichnis „/src“ wird der Source Code der Anwendung gespeichert. Da die Initialisierung der Anwendung zum Source Code gehören sollte, wird in der „index.php“ Datei nur die „start.php“ Datei des „/src“-Verzeichnisses aufgerufen. Die eigentliche Initialisierung der Anwendung wird also in der „start.php“ Datei durchgeführt.

“start.php“

Die Initialisierung einer Slim-Applikation wurde bereits in Abschnitt 4.5 angedeutet, für umfangreichere Anwendungen sind weitere wichtige Schritte notwendig. Jeder Schritt besitzt dabei eine wohl definierte Aufgabe.

```
1  // 1. Import autoloader for accessing classes
2  require __DIR__ . '/../vendor/autoload.php';
3  // 2. Start php $_SESSION (only the Webserver component)
4  \Source\Models\Helpers\SessionHelper::init();
5  // 3. Initialize DI-container with settings
6  $settings = require __DIR__ . '/../src/settings.php';
7  $container = new \Slim\Container($settings);
8  // 4. Initialize application
9  $app = new \Slim\App($container);
10 // 5. Add dependencies into DI-container
11 require __DIR__ . '/../src/dependencies.php';
12 // 6. Add routes and middleware
13 require __DIR__ . '/../src/routes.php';
14 // 7. Run application
15 $app->run();
```

Listing 5.1: Aufbau der start.php Datei.

Im ersten Schritt wird der Autloader von PHP Composer geladen, um die benötigten Klassen verwenden zu können. Wie dies erfolgt, wird im Abschnitt „composer.json“ näher beschrieben.

Im nächsten Schritt wird die PHP-Session initialisiert. Dies ist für die Webserver Komponente der Anwendung nützlich, um Informationen über den eingeloggten User über mehrere Requests hinaus festzuhalten. Die Microservices verzichten in der Regel auf die Verwendung der Session, um das REST-Prinzip der Stateless Services nicht zu verletzen. Wie bereits erwähnt, bietet das Slim-Framework einen DI-Container an, mit dem es möglich ist, Bibliotheken zu integrieren. Des Weiteren kann man den Container aber auch verwenden, um Einstellungen an einer zentralen Stelle zu verwalten, die sämtliche verwendete Bibliotheken und das Slim-Framework betreffen. Diese Einstellungen werden im dritten Schritt bei der Initialisierung eines neuen DI-Containers übergeben, der anschließend im vierten Schritt für die Initialisierung der Slim Anwendung verwendet werden kann. Im fünften Schritt werden die benötigten Bibliotheken in den DI-Container integriert und im sechsten Schritt werden die Endpunkte registriert. Abschließend kann die Anwendung ausgeführt werden. Was die einzelnen Dateien im Detail machen, wird nun im Folgenden erläutert.

“settings.php“

In der „settings.php“-Datei werden die Einstellungen der Anwendung und der verwendeten Bibliotheken verwaltet. Dies können Slim-spezifische Einstellungen sein, wie die Information, ob Fehler angezeigt werden sollen oder nicht, Datenbankeinstellungen, Angaben über das Passwort für das Erstellen der JWT oder Ähnliches. Auf diese Weise können Einstellungen an einer Stelle zentral verwaltet werden.

```
1  return [
2      'settings' => [
3          'displayErrorDetails' => true,
4          'jwt' => [
5              'key' => getenv("JWT_KEY"),
6          ],
7          'db' => [
8              'host' => 'tariff-db',
9              'name' => 'tariff_db',
10             'username' => 'root',
11             'password' => getenv('DB_ROOT_PW'),
12         ],
13     ],
14 ];
```

Listing 5.2: Aufbau der settings.php Datei.

Damit die Passwörter nicht in Klartext angezeigt werden, kann man in der „docker-compose.yml“-Datei Environment Variablen übergeben. (siehe „docker-conpose.yml“) Innerhalb des Codes kann mit der Funktion „getenv('name')“ auf die übergebenen Variablen zugegriffen werden.

“dependencies.php“

In der „dependencies.php“-Datei wird der DI-Container verwendet, um benötigte Bibliotheken an einem zentralen Ort zu verwalten und zu integrieren. Im folgenden Beispiel soll anhand des Logging Frameworks „Monolog“ gezeigt werden, wie weitere Bibliotheken integriert werden können. Da das Framework einige Einstellungen besitzt, die bei der Initialisierung mit angegeben werden müssen, müssen die Einstellungen zunächst in der „settings.php“-Datei ergänzt werden. Dazu gehört der gewünschte Name des Loggers sowie die Pfadangabe zu dem Verzeichnis, in dem die Log Datei gespeichert werden soll und das Level des Loggings. (siehe https://github.com/Seldaek/monolog)

```
return [
    'settings' => [
        'displayErrorDetails' => true,
        'jwt' => [
            'key' => getenv("JWT_KEY"),
        ],
        'db' => [
            'host' => 'tariff-db',
            'name' => 'tariff_db',
            'username' => 'root',
            'password' => getenv('DB_ROOT_PW'),
        ],
        'logger' => [
            'name' => 'slim-app',
            'path' =>__DIR__ . '/../logs/app.log',
            'level' => \Monolog\Logger::DEBUG,
        ],
    ],
]
```

Listing 5.3: Erweiterung der settings.php Datei

Anschließend kann der DI-Container des Frameworks geladen werden, um die Logging Komponente zu integrieren. Da die Einstellungen entsprechend erweitert wurden, können sie nun während der Initialisierung in Zeile sechs und sieben von Listing 5.4 verwendet werden:

```
1 // 1. Get di-container
2 $container = $app->getContainer();
3 // 2. Add Monolog-Logger
4 $container['logger'] = function ($container) {
5     $settings = $container['settings']['logger'];
6     $logger = new Logger($settings['name']);
7     $logger->pushHandler(new StreamHandler($settings['path'],
          $settings['level']));
8     return $logger;
9 };
```

Listing 5.4: Aufbau der dependencies.php-Datei

Da jeder Controller, wie im nächsten Abschnitt noch gezeigt wird, automatisch durch das Slim Framework mit dem DI-Container initialisiert werden kann, können die Controller die Bibliotheken problemlos aus dem DI-Container laden und verwenden.

```
1 class HomeController{
2     public function index($request, $response) {
3         // 1. load logger
4         $logger =  $this->container["logger"];
5         // 2. use logger
6         $logger->info("Logging passwords is a good idea.");
7         return $response->write("I've logged something.");
8     }
9 }
```

Listing 5.5: Verwendung der integrierten Bibliotheken innerhalb eines Beispiel-Controllers

“routes.php“

In der „routes.php“ werden sämtliche Endpunkte registriert. Ein Endpunkt besteht dabei immer aus einem Pfad, z.B. „/resources“ und einer Callback-Routine, die das HTTP-Request- und das HTTP-Response-Objekt verarbeiten kann. Um die Anwendung besser zu strukturieren, bietet

Slim zudem die Möglichkeit, anstatt der Callback-Routine, die Methode eines Controllers anzugeben, die den Request verarbeiten soll. Auf diese Weise kann man die Anwendung gemäß des Model-View-Controller-Prinzips strukturieren. Im Folgenden sollen einige Beispiel Endpunkte definiert werden, um Funktionalitäten des Slim-Frameworks vorzustellen.

```
1  // add group "/resources"
2  $app->group("/resources", function() use($container){
3      // add route with placeholder + controller method as callback
4      $this->get("/{id}", Controller::class.':getOne');
5      // add  route + controller method as callback
6      $this->post("", Controller::class.':create');
7      // add middleware
8  }->add(new AuthenticationMiddleware($container));
9
10 // add route with name
11 $app->get("/home", HomeController::class.':home')->setName("home");
```

Listing 5.6: Aufbau der routes.php-Datei

Durch die Funktion „group“ kann man mehrere Endpunkte in Gruppen zusammenfassen. Jeder Endpunkt, der einer Gruppe zugehörig ist, übernimmt dabei den Pfad der Gruppe. Im Listing 5.6 werden z.B. mehrere Endpunkte unter der Gruppe „/resources“ zusammengefasst. Zeile drei erstellt also einen „GET“-Request, der an den Endpunkt „/resources/{id}“ gerichtet ist und die Methode „getOne()“ der Klasse „Controller“ ausführt. Das Slim-Framework prüft daraufhin, ob es für die Klasse einen Eintrag im DI-Container gibt und erstellt gegebenenfalls ein Objekt der Klasse. Gibt es keinen Eintrag, ruft das Framework den Konstruktor der Klasse und übergibt den DI-Container. Benötigt ein Controller weitere Abhängigkeiten, so kann die Initialisierung in der „dependencies.php“-Datei ergänzt werden. Des Weiteren kann die Verwendung der „group“-Funktion nützlich sein, um mehrere Endpunkte mittels einer Middleware abzusichern. Die Middleware wird in Zeile sechs initialisiert und wird ausgeführt, noch bevor einer der registrierten Endpunkte ausgeführt werden kann. Auf diese Weise kann die Zugriffskontrolle aus Kapitel 4.4 realisiert werden, in dem die Middleware überprüft, ob ein gültiger JWT im „Authentication“-Header mitgesendet wurde. Erst wenn der Token vorhanden ist und validiert werden kann, wird der Request zur nächsten Schicht weitergeleitet, ansonsten wird der Client über den Fehler informiert. Listing 5.7 veranschaulicht, wie dies durch die AuthenticationMiddleware umgesetzt werden kann.

```
1  public function __invoke ($request, $response, $next) {
2      // 1. Get TokenHandler
3      $tokenHandler = $this->container["tokenHandler"];
4      // 2. Check Header existence
5      if ($request->hasHeader('Authentication')) {
6          $token = $request->getHeaderline('Authentication');
7          // 3. Validate Token
8          if($tokenHandler->validateToken($token)){
9              $decodedToken = $tokenHandler->getDecodedToken();
10             // 4. Add Token into Request object
11             $request = $request->withHeader("DecodedToken",
                   $decodedToken);
12             $response = $next($request, $response);
13         } else{
14                 $response = ErrorResponseFactory::
                       createInvalidTokenError($response, $token);
15         }
16     } else{
17             $response = ErrorResponseFactory::
                   createAuthenticationHeaderMissingError($response);
18     }
19     return $response;
20 }
```

Listing 5.7: Umsetzung der Authentifizierung

Mit Hilfe der im Listing 5.7 beschriebenen Middleware kann überprüft werden, ob der „Authentication“ - Header gesetzt wurde und ob das übergebene Token noch gültig ist. Die Validierung des Token wird dabei durch den TokenHandler durchgeführt. Aus dem Grund wird der TokenHandler zunächst in Zeile drei aus dem DI-Container geladen. Wenn die Validierung in Zeile sechs erfolgreich war, wird die nächste Middleware oder der entsprechende Controller mittels „$next“ aufgerufen. Damit eine weitere Middleware die Authorisierung umsetzen kann, wird der entschlüsselte Token vorher in das Request-Objekt geschrieben. Eine „AuthorizationMiddleware“ kann dann z.B. prüfen, ob der User die Rolle „Admin“ besitzt und entscheiden, ob er auf eine Ressource zugreifen darf oder nicht. Mit Hilfe der „ErrorResponseFactory“ werden im Fehlerfall entsprechende „ErrorResponse“-Objekte erstellt und in das Response-Objekt geschrieben. Auf diese Weise kann die Middleware den Prozess ignorieren, wie die Objekte erstellt werden.

5.1.3 “/tests“ und die „phpunit.xml“-Datei

Im Verzeichnis „/tests“ werden die Tests verwaltet. Die Klasse „BaseTestCase.php“ erweitert dabei die PHPUnit-Klasse „PHPUnit_Framework_TestCase“ und besitzt die beiden Funktionen „createRequest(some-parameters)„ und „runApp(request)“, über die Request-Objekte gemockt und die Applikation ausgeführt werden kann. Auf diese Weise kann die „BaseTestCase“-Klasse von anderen Klassen erweitert werden, um Tests wie folgt zu realisieren:

```
1 // @group unit-test
2 class ServiceTest extends BaseTestCase {
3     public function testHealthCheck() {
4       $requestObject = $this->createRequest('GET', '/health');
5       $response = $this->runApp($requestObject);
6       $this->assertEquals(200, $response->getStatusCode());
7     }
8 }
```

Listing 5.8: Umsetzung der Tests

Der Test erstellt zunächst ein Mock-Up-Request-Objekt für einen „GET“-Request, der an den Endpunkt „/health“ gerichtet ist. Obwohl im Beispiel nur zwei Parameter übergeben werden, stehen tatsächlich noch weitere Parameter für die Übergabe eines JWT und eines Body-Objektes zur Verfügung. Durch die „runApp“-Funktion wird dann die Applikation ähnlich wie in der „start.php“-Datei beschrieben aufgebaut und der Request verarbeitet. In diesem Fall wird nur getestet, ob der Service den Request verarbeiten kann und den Statuscode „200“ zurückgibt. Mit Hilfe der „@group“-Annotation kann PHPUnit so verwendet werden, dass nur bestimmte Gruppen von Tests ausgeführt werden sollen. Um einige Funktionen des Webservers zu testen, müssten zum Beispiel die anderen Services laufen. Mittels der Annotation „@group unit-test“ oder „@group integration-test“ kann spezifiziert werden, welche Art von Tests ausgeführt werden sollen. Über die „phpunit.xml“-Datei im Root-Verzeichnis kann man PHPUnit konfigurieren, um zum Beispiel das Verzeichnis anzugeben, in dem sich die Tests befinden.

5.1.4 “composer.json“

Mittels der „composer.json“-Datei installiert das Tool „PHP Composer“ die angegebenen Bibliotheken in das „/vendor“-Verzeichnis. Die „composer.json“-Datei beschreibt dabei, welche Bibliotheken integriert werden sollen und ermöglicht das Erstellen von Namespaces für das

automatische Laden von Klassen. Auf diese Weise können die im „/src“-Verzeichnis gespeicherten Klassen und die Klassen der integrierten Bibliotheken im Code verwendet werden. Über den Befehl „php composer install“ lädt Composer die beschriebenen Bibliotheken herunter und schreibt sie daraufhin in das Verzeichnis „/vendor“.

5.1.5 “install-dependencies.sh“

Das „install-dependencies.sh“ Script hat die Aufgabe PHP Composer herunterzuladen und anschließend die in der „composer.json“ Datei beschriebenen Bibliotheken lokal auf den Rechnern der Entwickler zu installieren. Dies kann sinnvoll sein, damit die verwendeten IDEs keine Fehler anzeigen, weil die verwendeten Bibliotheken nicht gefunden werden konnten.

```
1 #!/bin/bash
2 # 1. get php composer
3 php -r "copy('https://getcomposer.org/installer', 'composer-setup.
    php');" \
4 && php composer-setup.php --filename=composer.phar \
5 && php -r "unlink('composer-setup.php');"
6 # 2. install dependencies
7 php composer.phar install --prefer-source
```

Listing 5.9: Inhalt des install-dependencies.sh Scripts

5.1.6 “startup.sh“

Die Aufgabe des „startup.sh“ Scripts ist es, den PHP-built-in-Webserver mit der Adresse 0.0.0.0 auf Port 80 zu starten und die „index.php“ Datei als Einstiegspunkt zu verwenden. Möchte man einen einzelnen Service testen, ohne einen Docker Container zu bauen, müsste man zunächst das „install-dependencies.sh“-Script ausführen und anschließend das startup.sh“-Script. Da die Microservices das Passwort für die Ver- und Entschlüsselung der JWT benötigen, muss vorher die entsprechende Variable in das PHP Environment geladen werden. Außerdem würde die Datenbankkommunikation in diesem Fall nicht funktionieren, da der Service die Adresse des Datenbankhosts nicht kennt. (Stichwort: Service Discovery)

```
1 #!/bin/bash
2 php  -S 0.0.0.0:80 -t public public/index.php
```

Listing 5.10: Inhalt des startup.sh Scripts

5.1.7 “Dockerfile“

Das folgende Dockerfile beschreibt, wie aus einem Service ein lauffähiger Docker Container erstellt werden kann.

```
1  FROM docker-hub.informatik.haw-hamburg.de/abq307/greenworld-energies
       /php-7-base
2
3  # 1. Change directory
4  WORKDIR /var/www
5
6  # 2. Install dependencies
7  COPY composer.json /var/www
8  RUN php composer self-update && php composer install
9
10 # 3. Add files
11 COPY phpunit.xml /var/www
12 COPY startup.sh /usr/local/startup.sh
13 COPY ./tests /var/www/tests
14 COPY ./public /var/www/public
15 COPY ./src /var/www/src
16
17 # 34. Run tests
18 RUN vendor/bin/phpunit --configuration phpunit.xml --group unit-test
19
20 # 5. Expose port 80
21 EXPOSE 80
22
23 # 6. Create Entrypoint
24 User root
25 Run chmod 744 /var/www/startup.sh
26 ENTRYPOINT ["/bin/bash", "/var/www/startup.sh"]
27 CMD []
```

Listing 5.11: Aufbau des Dockerfiles

Das beschriebene Dockerfile kann sowohl vom Webserver als auch von den einzelnen Microservices verwendet werden. Als Basisimage dient ein angepasstes php-7-Image, das um einige Funktionalitäten erweitert wird. So besitzt das Basisimage bereits den PDO_MYSQL-Treiber, der das PHP Data Objects (PDO) Interface implementiert und den Zugriff auf die MySql-Datenbanken ermöglicht sowie die Bash-Shell für das Ausführen von Bash-Funktionen

innerhalb des Containers. Darüberhinaus wird PHP Composer innerhalb des „/var/www“ -Verzeichnisses vorinstalliert. Auf diese Weise können die Bibliotheken in Zeile acht mit dem Befehl „php composer install“ installiert werden. Da es vorkommen kann, dass eine neuere Version von PHP Composer existiert, wird vorher mit dem Befehl „php composer self-update“ dafür gesorgt, dass PHP Composer auf dem neuesten Stand ist. Im dritten Schritt werden alle benötigten Dateien und Verzeichnisse in das Image kopiert. Die Reihenfolge, mit der die Dateien in das Image kopiert werden, kann den erneuten Build-Prozess beschleunigen, da Docker nur Schichten nochmal baut, die sich verändert haben. Aus diesem Grund werden Dateien wie die „composer.json“-Datei oder die „phpunit.xml“-Datei vor Verzeichnissen wie dem „/tests“- oder dem „/src“-Verzeichnis in das Image kopiert. Im vierten Schritt werden die Unit-Tests ausgeführt. Auf diese Weise kann sichergestellt werden, dass der Container nur gebaut wird, wenn die Unit-Tests erfolgreich waren. Mittels der Anweisung „EXPOSE 80“ wird anschließend angegeben, dass der Container auf Port 80 auf Anfragen reagiert. Damit beim Starten des Containers der PHP-built-in-Webserver gestartet wird und die Anwendung ausgeführt werden kann, wird als Entrypoint das zuvor per „chmod744“ ausführbar gemachte „startup.sh“-Script übergeben und mit dem Befehl „/bin/bash /var/www/startup.sh“ ausgeführt.

5.2 Aufbau des GitLab Projekts

Für die Umsetzung der Versionskontrolle und des Continuous Delivery Prozesses wurde die GitLab Instanz der Hochschule verwendet. Im Folgenden soll der Aufbau des Projekts näher erläutert werden, um zu veranschaulichen, wie die Anwendungslandschaft ausgeführt und weiterentwickelt werden kann.

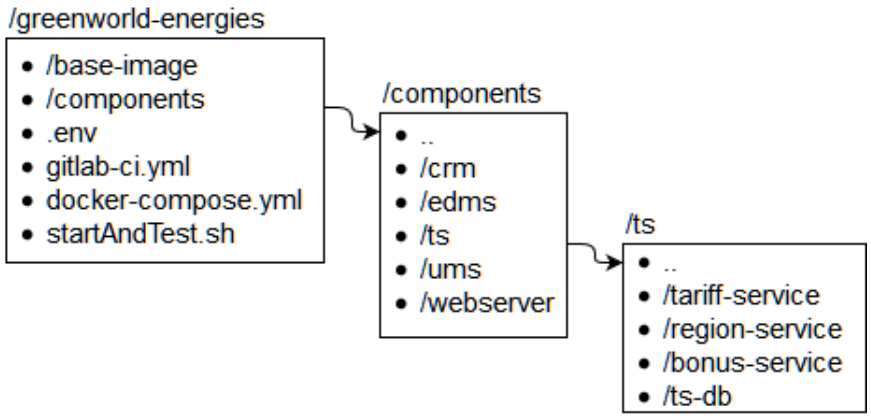

Abbildung 5.2: Projektstruktur: Teil 1

Das Projekt ist in vier Ebenen strukturiert. Im Hauptverzeichnis befinden sich die Dateien, die für das Starten der Anwendungslandschaft notwendig sind oder die den Continuous Delivery Prozess beschreiben. Die „gitlab-ci-yml“-Datei beschreibt dabei die Realisierung des Continuous Development Prozesses mittels der GitLab CI-/CD-Pipeline und die „docker-compose.yml“-Datei kann verwendet werden, um die Anwendungslandschaft lokal zu starten. Das „startAndTest.sh“-Script kann ebenfalls verwendet werden, um die Anwendungslandschaft lokal zu starten und ermöglicht zusätzlich die Ausführung von Integrationstests. Zu diesem Zweck verwendet das Script die „docker-compose.yml“-Datei. Im Unterverzeichnis „/base-image“ ist das Basisimage hinterlegt, das von allen Microservices und der Webserver Komponente verwendet wird. In der zweiten Ebene werden die Verzeichnisse der Komponenten, die in Kapitel 3.3 beschrieben wurden, gespeichert. Dazu gehört das Customer Relationship Management System (CRM), das Energiedaten Management System (EDMS), das Tarifsystem (TS), das User Management System (UMS) und die Webserver Komponente. Die dritte Ebene beschreibt den Aufbau einer Komponente. So besteht das Tarifsystem (TS) zum Beispiel aus den drei Services „Tariff Service“, „Region Service“ und „Bonus Service“ sowie aus der Datenbank „TS-DB“. In der vierten Ebene werden die Projekte der einzelnen Services, wie in Abschnitt 5.1 beschrieben, gespeichert. Die Datenbank-Verzeichnisse (siehe Abbildung 5.3) bestehen aus den Verzeichnissen „/data“ und „/scripts“. Im „/data“-Verzeichnis können die Daten persistiert werden und im „/scripts“-Verzeichnis werden die für die Initialisierung der Datenbank benötigten SQL-Skripte hinterlegt.

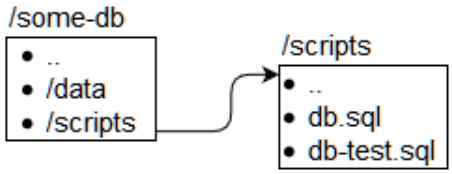

Abbildung 5.3: Projektstruktur: Teil 2

5.2.1 Umsetzung des CI-/CD-Prozesses

Damit die Anwendungslandschaft im Zuge eines Continuous Delivery Prozesses weiterentwickelt werden kann, wurde der CI/CD-Prozess von GitLab integriert. Der Prozess besteht dabei aus drei Stages, die im folgenden Listing ausführlich dargelegt werden. Für diesen Prozess ist es wichtig nachzuvollziehen, dass das Repository aus drei Branches besteht - namentlich „development“, „base-image“ und „master“. Damit nicht jeder Commit die gesamte Pipeline ausführt, werden einige Jobs an bestimmte Branches gebunden.

```
# Create Stages
stages:
  - buildBaseImage
  - dockerize
  - integrationTest

# Add Variables
variables:
  DOCKER_HOST: "tcp://localhost:2375"
  DOCKER_REGISTRY: "docker-hub.informatik.haw-hamburg.de"

# Use Docker-in-Docker Runner
image: nexus.informatik.haw-hamburg.de/docker:stable-dind
services:
  - nexus.informatik.haw-hamburg.de/docker:stable-dind

# 1. Stage: Build base image
build-base-image:
  stage: buildBaseImage
  only:
    - base-image
  script:
    - docker login -u $DOCKERHUB_USER -p $DOCKERHUB_PW
        $DOCKER_REGISTRY
    - docker build -t $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-
        energies/php-7-base ./base-images
    - docker push $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-
        energies/php-7-base

# 2. Stage: Dockerize services

dockerize-webserver:
  stage: dockerize
  only:
    - master
  script:
    - docker login -u $DOCKERHUB_USER -p $DOCKERHUB_PW
        $DOCKER_REGISTRY
```

```
35      - docker build -t $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-
            energies/webserver ./components/webserver
36      - docker push $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-
            energies/webserver
37
38  dockerize-customer-service:
39    stage: dockerize
40    only:
41      - master
42    script:
43      - docker login -u $DOCKERHUB_USER -p $DOCKERHUB_PW
            $DOCKER_REGISTRY
44      - docker build -t $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-
            energies/customer-service ./components/crm/customer-service
45      - docker push $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-
            energies/customer-service
46
47  # dockerize other services
48  ...
49
50  # 3. Stage: Test the integration
51  integrationTest:
52    stage: integrationTest
53    only:
54      - master
55    before_script:
56      - apk add --no-cache py-pip
57      - pip install docker-compose
58      - apk --update add bash
59      - apk --update add curl
60    script:
61      - docker login -u $DOCKERHUB_USER -p $DOCKERHUB_PW
            $DOCKER_REGISTRY
62      - bash startAndTest.sh
```

Listing 5.12: Inhalt der gitlab-ci.yml

Der Prozess besteht aus den Stages „buildBaseImage“, „dockerize“ und „integrationTest“. Da die Stages die Ausführung von Docker-Befehlen benötigen, werden die Jobs durch einen „Docker-in-Docker“-Executor ausgeführt, der in den Zeilen 13-15 integriert wird. Zeile 13 beschreibt

das Docker-Basis-Image des Build Containers, während in Zeile 15 die Docker Engine als Service gestartet wird. Damit der Build Container die Docker Enging verwenden kann, muss die „DOCKER_HOST“-Variable entsprechend gesetzt werden. Die Docker Engine kann über Localhost auf Port 2375 angesprochen werden. Aus diesem Grund muss der Wert der Variable „tcp://localhost:2375“ lauten. Die Variable „DOCKER_REGISTRY“ verweist auf die Registry, in der die Images gespeichert werden sollen, in diesem Fall ist es die Registry der Hochschule.

In der ersten Stage wird das Basis-Image erstellt, das von allen Microservices und der Webserver-Komponente verwendet wird. Da sich dieses Image nur sehr selten ändert, reicht es aus, wenn der Job „build-base-image“ nur bei einem Commit auf dem „base-image“-Branch ausgeführt wird. Das im Job ausgeführte Script besteht aus drei Schritten. Im ersten Schritt wird der Dockerhub User in der Registry eingeloggt. Die hierfür benötigten Variablen werden über GitLabs „Secret Variables“ hinzugefügt. Auf diese Weise müssen die Variablen nicht im Repository gespeichert werden und werden erst zur Laufzeit dem Runner übergeben. Anschließend wird das Basis-Image gebaut und getaggt und wenn alles fehlerfrei verlaufen ist, im dritten Schritt, in die Registry gepusht. Die Jobs der „dockerize“-Stage sind ähnlich aufgebaut, wie der „build-base-image“-Job, weswegen an dieser Stelle auf die Erläuterung verzichtet werden kann.

In der dritten Stage wird die Anwendungslandschaft durch das „startAndTest.sh“-Script gestartet und getestet. Da das Script über die „Bash“ ausgeführt wird und „docker-compose“ sowie „Curl“ verwendet, müssen diese Tools zunächst installiert werden. Wie das Script genau aufgebaut wird, zeigt der nächste Abschnitt.

Wenn die Integrationstests durchgelaufen sind, gilt der CI-Prozess in der aktuellen Version als erfolgreich abgeschlossen. Die nächste Stufe wäre z.B. die Integration der Updates in eine Staging- oder Produktionsumgebung. Aus Zeitgründen konnte dies nicht mehr realisiert werden.

5.2.2 Starten und Testen der Anwendungslandschaft

Um die Anwendungslandschaft in einer Entwicklungsumgebung ausführen und testen zu können, wird Docker-Compose verwendet - sei es durch die direkte Verwendung oder durch das „startAndTest.sh“-Script.

“docker-compose.yml“

Listing 5.13 zeigt den Aufbau der „docker-compose.yml“-Datei. Da die Integration der einzelnen Microservices und der entsprechenden Datenbanken gleich aufgebaut ist, werden an dieser Stelle nicht alle Services dargestellt.

```
1  version: '3'
2
3  services:
4
5    # Add webserver
6    webserver:
7      image: $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-energies/
           webserver:latest
8      ports:
9        - "80:80"
10     environment:
11       - JWT_KEY=$JWT_KEY
12     depends_on:
13       - authentication-service
14       - customer-service
15       - tariff-service
16       - region-service
17       - bonus-service
18       - edm-service
19     volumes:
20       - ./components/webserver/public:/var/www/public
21       - ./components/webserver/src:/var/www/src
22       - ./components/webserver/tests:/var/www/tests
23
24   # Add customer-service
25   customer-service:
26     image: $DOCKER_REGISTRY/$DOCKERHUB_USER/greenworld-energies/
           customer-service
27     ports:
28       - "8082:80"
29     environment:
30       - DB_ROOT_PW=$DB_PASSWORD
31       - JWT_KEY=$JWT_KEY
```

```
32      depends_on:
33        - sugarcrm
34        - sugarcrm-db
35      volumes:
36        - ./components/crm/customer-service/src:/var/www/src
37        - ./components/crm/customer-service/tests:/var/www/tests
38
39    # Add sugarcrm
40    sugarcrm:
41      image: 'bitnami/sugarcrm:latest'
42      ports:
43        - '8084:80'
44        - '443:443'
45      environment:
46        - SUGARCRM_PASSWORD=$DB_PASSWORD
47        - MARIADB_PASSWORD=$DB_PASSWORD
48        - MARIADB_HOST=customer-db
49      depends_on:
50        - sugarcrm-db
51
52    # Add sugarcrm-db
53    sugarcrm-db:
54      build: ./components/crm/sugarcrm-db
55      ports:
56        - "3306"
57      environment:
58        - MARIADB_ROOT_PASSWORD=$DB_PASSWORD
59
60    # Add other services
61    ...
```

Listing 5.13: Inhalt der docker-compose.yml

Eine fehlerfreie Ausführung setzt die Übergabe einiger Umgebungsvariablen voraus, wie z.B. die Angabe der Docker Registry, in der die Images gespeichert wurden, der Name des Dockerhub Users, der die Images gespeichert hat, der Schlüssel für das Ver- und Entschlüsseln der JWT oder die Datenbankpasswörter. Aus diesem Grund wurde im Hauptverzeichnis eine „.env“-Datei angelegt, die Key-Value-Paare enthält. Auf diese Weise kann Docker-Compose die Variablen durch die entsprechenden Werte in der „.env“-Datei ersetzen. Einfachheitshalber erhalten alle Datenbanken in der Entwicklungsumgebung zunächst dasselbe Passwort für

den User „root“. Die Anwendungslandschaft wird daraufhin mit dem Befehl „docker-compose -p dev up -d“ gestartet. Der Parameter „-p dev“ sorgt dafür, dass alle Container unter dem Projektnamen „dev“ als Prefix gestartet werden und der Parameter „-d“ ermöglicht, dass die Container im Hintergrund laufen. Folglich wird die Webserver Komponente im Beispiel im Container „dev_webserver_1“ gestartet.

Da einige Services wie der Webserver Abhängigkeiten zu anderen Services besitzen, werden diese mittels „depends_on“ festgelegt. Auf diese Weise werden zunächst die Services gestartet, bevor der Webserver gestartet wird. Zu beachten ist, dass Docker nicht mit dem Start des Webservers wartet, bis alle Abhängigkeiten vollständig gestartet wurden. Um dies zu realisieren, müsste man aktuell externe Lösungen finden.

Die Volumes können genutzt werden, um die Entwicklung zu beschleunigen. Da jede Änderung in einem der integrierten Verzeichnisse direkt vom jeweiligen Container verwendet wird, kann z.B. der Source Code zur Laufzeit verändert werden. Dies liegt daran, dass die PHP-Dateien nicht weiter kompiliert werden müssen. Aus diesem Grund werden beim Webserver die Verzeichnisse „/public“, „/src“ und „/tests“ als Volume integriert. Die Microservices benötigen dagegen nur die Verzeichnisse „/src“ und „/tests“.

“startAndTest.sh“

Das „startAndTest.sh“-Script wird genutzt, um die Anwendung lokal zu starten und um Tests auszuführen. Hierfür stehen verschiedene Optionen zur Verfügung, die bei der Ausführung des Scripts mit angegeben werden können. Die genaue Beschreibung der verschiedenen Optionen kann im entsprechenden Wiki-Eintrag nachvollzogen werden oder durch den Befehl „bash startAndTest.sh –help“. Das folgende Listing 5.14 zeigt einen bearbeiteten Ausschnitt des Scripts und legt den Fokus auf die wesentlichen Stellen.

```
1 #!/bin/bash
2
3 # initialize variables with default values
4 SKIPBUILD="#"
5 ...
6
7 # Get all passed options
8 ...
9
```

```
10 # 1. Start Containers?
11 if [ $SKIPBUILD == '#' ]
12 then
13   # 1.1. start containers
14   docker-compose -p ci up --build -d
15   # 1.2. wait for containers to be started (workaround)
16   sleep 20
17   # 1.3. check if SugarCrm is installed
18   result=$(curl -s -o /dev/null -L -w "%{http_code}" http://
         localhost:8084/index.php)
19   if [[ $result -ne 200 ]] && [[ $result -ne 404 ]]
20   then
21     printf "\nAn unexpected problem occured."
22     exit 1
23   fi
24   # 1.4. if Not installed: Install SugarCRM
25   while [[ $result -ne 200 ]]
26   do
27     sleep 15
28     result=$(curl -s -o /dev/null -L -w "%{http_code}" http://
           localhost:8084/index.php)
29     if [[ $result -eq 200 ]]
30     then
31       # extend and change SugarCrms database
32       source .env
33       ./components/crm/sugarcrm-db/postInstall.sh $DB_PASSWORD
34     fi
35   done
36 fi
37
38 # 2. Execute Tests?
39 if [ $TESTS == '--no-tests' ] || [ $TESTS == '--nt' ]
40 then
41   printf "\nNo tests will be executed...\n"
42 else
43   docker exec -i ci_webserver_1 vendor/bin/phpunit --configuration
         phpunit.xml
44   # others
45   ...
46 fi
```

```
47
48 # 3. Shall the containers continue to run?
49 if [ $RUNNING == '--keep-running' ] || [ $RUNNING == '--kr' ]
50 then
51   printf "\nThe containers are still running in the background...\n"
52 elif [ $SHUTDOWN == '--shutdown' ]
53 then
54   docker-compose -p ci down
55   docker volume prune
56 else
57   docker-compose -p ci stop
58 fi
59
60 exit 0
```

Listing 5.14: Inhalt des startAndTest.sh-Scripts

Der Default-Ablauf des Scripts besteht aus mehreren Schritten. Im ersten Schritt werden die Container mittels „docker-compose -p dev up -d“ gestartet. Daraufhin wartet das Script eine gewisse Zeit, bis alle Services gestartet wurden und überprüft anschließend, ob „SugarCrm“ bereits installiert ist. Zu diesem Zweck wird mittels Curl in Zeile 18 ein GET-Request an den Entrypoint von SugarCrm gesendet und der erhaltene Statuscode in der Variable „result“ gespeichert. Bei einer korrekten Integration von SugarCrm kann der Wert entweder 200 oder 404 sein. Sollte dies nicht der Fall sein, ist ein unerwarteter Fehler aufgetreten und das Script wird beendet. Tritt der Statuscode 404 auf, wird SugarCrm automatisch installiert. Anschließend wird in einer Schleife überprüft, ob SugarCrm die Installation vollständig beenden konnte. Wenn dies der Fall ist, wird ein entsprechendes „postInstall.sh“-Script ausgeführt. Die Aufgabe des Scripts besteht in der Ergänzung fehlender Datenbankeinträge und Tabellen. Hierfür wird über den Befehl „docker exec“ eine Shell innerhalb des Containers gestartet, die das MySql Command-Line Tool ausführt. Damit die Datenbank bearbeitet werden kann, muss der User „root“ mit dem entsprechenden Passwort eingeloggt werden. Die entsprechende Variable wurde zuvor im „startAndTest.sh“-Script über den Befehl „source .env“ in Zeile 32 zugänglich gemacht. Auf diese Weise kann das für die Installation benötigte Datenbankpasswort übergeben und im „postInstall.sh“-Script verwendet werden (siehe Abbildung 5.15).

```
1 #!/bin/bash
2 DB_ROOT_PW="$1"
3 # alter database
4 docker exec -it ci_sugarcrm-db_1 /bin/sh -c 'mysql -uroot -p'
    $DB_ROOT_PW' < /home/bitnami/alter-sugarcrm.sql'
```

Listing 5.15: Inhalt des postinstall.sh-Scripts

Wenn SugarCrm korrekt installiert wurde, wird die Anwendungslandschaft getestet. Zu diesem Zweck wird mittels „docker exec“ PHP-Unit innerhalb der einzelnen Container ausgeführt und als Konfiguration die „phpunit.xml“-Datei angegeben. (siehe Listing 5.14, Zeile 43) Es erfolgen sowohl Integrationstests als auch Unittests. Im letzten Schritt wird die Anwendungslandschaft nur noch gestoppt.

Um das Default-Verhalten des Script zu verändern, stehen einige Optionen zur Verfügung. Mittels „bash startAndTest.sh –kr“ kann man z.B. angeben, dass die Anwendungslandschaft nicht gestoppt werden soll und ein anschließender Aufruf von „bash startAndTest.sh –sb –kr“ würde dafür sorgen, dass nur die Tests ausgeführt werden. Auf diese Weise kann die Entwicklung sehr effizient unterstützt und beschleunigt werden. Es wird nur ein Befehl benötigt, um die gesamte Anwendungslandschaft zu testen. Möchte man das Projekt aufräumen und neu aufsetzen, bietet sich der Befehl „bash startAndTest.sh –shutdown“ an. Auf diese Weise werden sowohl die Container als auch die unbenutzten Volumes vom System gelöscht.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Anwendungslandschaft eines fiktiven Energieanbieters mit Hilfe von Docker innerhalb eines Continuous Delivery Prozesses prototypisch entwickelt. Ziel war es, eine möglichst reale Anwendungslandschaft zu simulieren, die im weiteren Projektkontext von anderen Projektteilnehmern ausgebaut werden kann. Im Folgenden sollen die Ergebnisse zusammengefasst werden sowie ein Ausblick gegeben werden, wie die Anwendungslandschaft in weiteren Projekten verbessert und erweitert werden könnte.

6.1 Zusammenfassung

Im Verlaufe des Projektes ist der Prototyp der Anwendungslandschaft eines fiktiven Energieanbieters entstanden, die in weiten Teilen in ähnlicher Form in der Wirtschaft existieren könnte. Die Geschäftsprozesse orientieren sich an realen Prozessen und die formulierten Tarifstrukturen ähneln ihren realen Vorbildern. Des Weiteren wurde die Umsetzung einer Zugriffskontrolle mittels Rollen und JSON Web Token eingeführt, um die Absicherung verschiedener Ressourcen innerhalb der Anwendungslandschaft zu ermöglichen. Und auch die einzelnen Systeme wurden so gewählt, dass die Umsetzung im Zuge eines Continuous Delivery Prozesses von einem Kunden wie „Greenworld Energies“ gefordert werden könnte. So besteht die Anwendungslandschaft aus bereits integrierten Systemen wie „SugarCrm“, Fremd-Systemen, die integriert werden mussten und neuen Systemen, die prototypisch entwickelt wurden.

Von den beschriebenen Geschäftsprozessen konnten sämtliche Prozesse zumindest prototypisch durch die Anwendungslandschaft unterstützt werden. Administratoren können Regionen, Boni und Tarife erstellen und Kunden können nach Tarifen suchen und den Bestellprozess durchlaufen. Nach einem erfolgreichen Vertragsabschluss können Kunden zudem im persönlichen Kundenbereich ihre Vertragsdaten einsehen und die Zählerstände des entsprechenden Stromzählers aktualisieren.

Um die Weiterentwicklung zu unterstützen, wurden die wichtigsten Designentscheidungen und der Projektaufbau ausführlich beschrieben. Ferner wurde neben der Integration von

Docker und des CI-/CD-Prozesses ein Start-und-Test-Script entwickelt, mit Hilfe dessen die Installation der Anwendungslandschaft und die Ausführung von Tests erleichtert werden kann. Zukünftige Projektteilnehmer müssen nur das GitLab-Projekt klonen und das beschriebene Script ausführen. Vorraussetzung hierfür ist lediglich, dass Docker und Git korrekt installiert und die Anweisungen des Wiki-Eintrages „how to setup“ korrekt umgesetzt werden.

Die Entwicklung der Anwendungslandschaft hat gezeigt, dass die Verwendung von Docker sehr gut geeignet sein kann, um agile Softwareentwicklung zu unterstützen. Einmal eingerichtet konnte das Hauptaugenmerk auf die Implementierung von Features gelegt werden. Besonders die Entwicklung Verteilter Systeme kann von der Automatisierung einzelner Schritte, wie der Installation der Komponenten, dem Aufbauen von Netzwerken und der Ausführung von Tests profitieren. Auf diese Weise konnte die Anwendungslandschaft auf unterschiedlichen Systemen mit verschiedenen Betriebssystemen nahezu problemlos getestet werden. Die meisten Fehler sind aufgrund des selbst entwickelten Start-und-Test-Scripts entstanden. Der Aufwand, der ohne die Automatisierung erforderlich wäre, steigt mit der Anzahl der Systeme. Durch Docker steht Entwicklern nun seit 2013 ein neues Tool zur Verfügung, mit dem der Aufwand gering gehalten werden kann und mit dessen Hilfe schnell Prototypen und neue Features entwickelt werden können.

Die Verwendung hat allerdings auch gezeigt, dass sich die Technologie noch in Entwicklung befindet. So sind vereinzelt Fehler aufgetreten wie Volumes, die nicht an Container gebunden werden konnten, Netzwerkfehler oder fehlerhafte Containerstarts. Abhilfe hat dann in der Regel ein Neustart von Docker geschaffen. In Zukunft ist allerdings davon auszugehen, dass nachfolgende Versionen noch stabiler laufen. Zusammen in Kombination mit weiteren Technologien wie Docker Swarm oder Kubernetes existieren zudem bereits Konzepte, um Docker auch in Produktion erfolgreich einsetzen zu können. Dies konnte aus Zeitgründen allerdings nicht mehr getestet werden.

6.2 Ausblick

Obwohl die beschriebenen Geschäftsprozesse weitestgehend durch die Anwendungslandschaft automatisiert unterstützt werden, gibt es an einigen Stellen noch Verbesserungspotential und Möglichkeiten zur Erweiterung. Außerdem sind im Verlauf der Entwicklung einige Fehler entstanden, die es zu überarbeiten gilt.

1. Umsetzung verteilter Transaktionen
   Die Webserver Komponente nimmt zwar die für die Bestellung benötigten Daten entgegen und speichert die daraus resultierenden Entitäten in den verschiedenen Systemen - die Lösung verhindert allerdings nicht, dass Bestellungen inkonsistent gespeichert werden. In Zukunft könnte es sinnvoll sein, dass der Webserver die Bestellung an ein Bestellsystem weiterleitet. Das Bestellsystem könnte die Bestellung daraufhin zwischenspeichern und anschließend die verteilte Transaktion korrekt durchführen, während sich die Webserver Komponente nur noch um die grafische Benutzerschnittstelle kümmern würde. Auf diese Weise könnte man ebenfalls die Zuständigkeiten optimieren.

2. Integration eines Abrechnungssystems
   Durch die Integration eines Abrechnungssystems könnte man die Anwendungslandschaft vervollständigen, so dass andere Geschäftsprozesse zur Kundenbelieferung mit Elektrizität wie z.B. die „Netznutzungsabrechnung“ (vgl. BNetzA 2012, S.45) realisiert werden könnten. Zusätzlich könnte der Vertragsabschluss Prozess weiter durch Software unterstützt werden. (siehe 3.2.3). Der Prozess setzt die Überprüfung der Kreditwürdigkeit der Kunden sowie die Erstellung von Rechnungen voraus. Diese Aufgaben könnten von einem Abrechnungs- oder Banksystem automatisiert werden.

3. Ausbau des CI-/CD-Prozesses
   Aufgrund der Fokussierung der Bachelorarbeit auf den Aufbau der Anwendungslandschaft und der Dokumentation der wichtigsten Designentscheidungen, konnte keine vollständige CD-Pipeline mehr realisiert werden. Die fehlenden Schritte wie die Integration von Kubernetes müssten in zukünftigen Versionen ergänzt werden, um die Anwendungslandschaft innerhalb einer Staging- oder Produktionsumgebung deployen zu können. Auf diese Weise können zukünftige Projektteilnehmer Konzepte wie „Blue-Green-Deployments“ oder „Rolling-Deployments“ näher untersuchen oder die Anwendungslandschaft für Akzeptanz- und Kapazitätstests nutzen.

4. Erweiterung SugarCrms
   Die tatsächliche Schnittstelle von SugarCrm wurde in der Arbeit durch einen „Customer Service“ ersetzt, der auf die Datenbank von SugarCrm zugreift. Grund hierfür war der große Mehraufwand, den eine Erweiterung SugarCrms ausgelöst hätte. Damit in Zukunft die tatsächliche Schnittstelle von SugarCrm verwendet werden kann, müsste man diese also korrekt erweitern. Für diesen Zweck bietet die Dokumentation von SugarCrm ein entsprechendes Kapitel, in dem die erforderlichen Schritte beschrieben werden.

Literaturverzeichnis

[Aichele 2012] Aichele, Christian: *Smart Energy - Von der reaktiven Kundenverwaltung zum proaktiven Kundenmanagement.* Wiesbaden : Springer Vieweg+Teubner Verl., 2012. – ISBN 978-3-8348-1570-5

[Banavar u. a. 1999] Banavar, Guruduth ; Chandra, Tushar D. ; Strom, Robert E. ; Sturman, Daniel C.: A Case for Message Oriented Middleware. In: *Proceedings of the 13th International Symposium on Distributed Computing.* London, UK, UK : Springer-Verlag, 1999, S. 1–18. – URL http://dl.acm.org/citation.cfm?id=645956.675943. – ISBN 3-540-66531-5

[BDEW 2002] BDEW, Bundesverband der Energie- und Wasserwirtschaft E.V.: *BDEW-Kundenfokus Haushalte 2002.* 2002. – URL https://bdew.de/internet.nsf/id/DE_BDEW-Kundenfokus_Haushalte_2002-2007. – Zugriffsdatum: 2017-07-14

[BDEW 2015] BDEW, Bundesverband der Energie- und Wasserwirtschaft E.V.: *BDEW-Kundenfokus Haushalte 2015.* 2015. – URL https://bdew.de/internet.nsf/id/DE_BDEW-Kundenfokus_Haushalte_2002-2007. – Zugriffsdatum: 2017-07-14

[BDEW 2016] BDEW, Bundesverband der Energie- und Wasserwirtschaft E.V.: *Die digitale Energiewirtschaft - Agenda für Unternehmen und Politik.* 2016. – URL https://bdew.de/internet.nsf/id/digitale-agenda-de. – Zugriffsdatum: 2017-07-14

[Beck u. a. 2001] Beck, Kent ; Grenning, James ; Martin, Robert C.: *Manifesto for Agile Software Development.* 2001. – URL http://agilemanifesto.org. – Zugriffsdatum: 2018-01-25

[BGBL 1998] BGBL, Bundesgesetzblatt: *Gesetz zur Neuregelung des Energiewirtschaftsrechts.* 1998. – URL http://www.bgbl.de/xaver/bgbl/start.xav?startbk=Bundesanzeiger_BGBl&jumpTo=bgbl198s0730.pdf. – Zugriffsdatum: 2017-07-14. – BGBL. 1 Nr. 23, S. 730 idF. v. 1998-04-28

[BGBL 2005] BGBL, Bundesgesetzblatt: *Gesetz über Elektrizitäts- und Gasversorgung*. 2005. – URL https://www.bmwi.de/Redaktion/DE/Gesetze/Energie/EnWG.html. – Zugriffsdatum: 2017-09-06

[BNetzA 2012] BNetzA, Bundesnetzagentur: *Darstellung der Geschäftsprozesse zur Anbahnung und Abwicklung der Netznutzung bei der Belieferung von Kunden mit Elektrizität - Geschäftsprozesse zur Kundenbelieferung mit Elektrizität, GPKE*. 2012. – URL https://www.bundesnetzagentur.de/DE/Service-Funktionen/Beschlusskammern/1BK-Geschaeftszeichen-Datenbank/BK6-GZ/2011/2011_0001bis0999/2011_100bis199/BK6-11-150/Konsolidierte_Lesefassung_GPKE.pdf. – Zugriffsdatum: 2017-09-14

[Box u. a. 2000] Box, Don ; Ehnebuske, David ; Kakivaya, Gopal: *Simple Object Access Protocol (SOAP) 1.1*. 2000. – URL https://www.w3.org/TR/2000/NOTE-SOAP-20000508/. – Zugriffsdatum: 2018-02-12

[Brewer 2000] Brewer, Eric: Towards robust distributed systems. In: *PODC 2000 - Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. New York, USA : ACM, 2000. – Folien unter: https://www.researchgate.net/publication/221343719_Towards_robust_distributed_systems. – ISBN 1-58113-183-6

[Docker 2017a] Docker: *Compose file version 3 reference*. 2017. – URL https://docs.docker.com/compose/compose-file/. – Zugriffsdatum: 2018-11-01

[Docker 2017b] Docker: *Find a Partner*. 2017. – URL https://www.docker.com/find-partner. – Zugriffsdatum: 2017-12-15

[Fielding u. a. 1999] Fielding, Roy T. ; Gettys, James ; Mogul, Jeffrey C.: Hypertext Transfer Protocol – HTTP/1.1 / RFC Editor. RFC Editor, June 1999 (2616). – RFC. – URL https://www.rfc-editor.org/rfc/rfc2616.pdf. https://www.rfc-editor.org/rfc/rfc2616.pdf. – ISSN 2070-1721

[Fielding und Taylor 2000] Fielding, Roy T. ; Taylor, Richard N.: Principled Design of the Modern Web Architecture. In: *Proceedings of the 22Nd International Conference on Software Engineering*. New York, NY, USA : ACM, 2000 (ICSE '00), S. 407–416. – URL http://doi.acm.org/10.1145/337180.337228. – ISBN 1-58113-206-9

[Hazlewood 2011] Hazlewood, Les: *Beautiful REST + JSON APIs*. 2011. – URL https://www.slideshare.net/stormpath/rest-jsonapis. – Zugriffsdatum: 2017-12-06

[Jones u. a. 2015] Jones, M. ; Bradley, J. ; Sakimura, N.: JSON Web Token (JWT) / RFC Editor. RFC Editor, May 2015 (7519). – RFC. – URL https://www.rfc-editor.org/rfc/pdfrfc/rfc7519.txt.pdf. https://www.rfc-editor.org/rfc/pdfrfc/rfc7519.txt.pdf. – ISSN 2070-1721

[Josh Lockart 2017] Josh Lockart, Rob Allan: *Slim-Framework Documentation.* 2017. – URL https://www.slimframework.com/docs/. – Zugriffsdatum: 2017-10-25

[Lackes u. a. 2009] Lackes, Richard ; Siepermann, Markus ; Schewe, Gerhard: *Definition Geschäftsprozess.* 2009. – URL http://wirtschaftslexikon.gabler.de/Archiv/5598/geschaeftsprozess-v12.html. – Zugriffsdatum: 2018-02-06

[Leach u. a. 2005] Leach, Paul J. ; Mealling, Michael ; Salz, Rich: A Universally Unique IDentifier (UUID) URN Namespace / RFC Editor. RFC Editor, July 2005 (4122). – RFC. – URL https://www.rfc-editor.org/rfc/pdfrfc/rfc4122.txt.pdf. https://www.rfc-editor.org/rfc/pdfrfc/rfc4122.txt.pdf. – ISSN 2070-1721

[Mouat 2016] Mouat, Adrian: *Software entwickeln und deployen mit Containern.* Heidelberg : dpunkt - Verl., 2016 (1. Aufl.). – ISBN 978-3-86490-384-7

[Panos 2007] Panos, Konstantin: *Praxisbuch Energiewirtschaft - Energieumwandlung, -transport, und -beschaffung im liberalisierten Markt.* Berlin Heidelberg : Springer-Verl., 2007. – ISBN 978-3-540-35377-5

[Wolff 2016] Wolff, Eberhard: *Continuous Delivery - Der pragmatische Einstieg.* Heidelberg : dpunkt - Verl., 2016 (2. Aufl.). – ISBN 978-3-86490-371-7

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16.02.2018 Jonas Johannsen