Hochschule München
Fakultät für Mathematik und Informatik

Seminararbeit

Konzeption und Prototyping eines hochskalierbaren Onlineshops - Am Beispiel von Microservices

Maximilian Auch geb. Spelsberg

Abgabe: 15 August 2016

betreut von: Prof. Dr. Peter Mandl *Hochschule München*

Martin Häusl *Hochschule München*

Eidesstattliche Erklärung

Erklärung gemäß § 15 Abs. 10 APO i. V. m. § 35 Abs. 7 RaPO.
Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, den 15.08.2015

(Maximilian Auch geb. Spelsberg)

Abstract

Für das Ziel eines hoch skalierbaren Systems, gilt eine horizontale Skalierbarkeit mit einer linearen Performance-Steigerung als eine optimale Eigenschaft. Für den im Zuge dieser Arbeit entwickelten Prototypen kommt daher als Lösungsansatz eine Microservice Architektur zum Einsatz. Dieses Vorgehen, welches neuerdings immer stärker Verbreitung findet, beschreibt das Schneiden des Systems nach der Domäne in kleine, unabhängige Services. Außerdem werden Prinzipien nach ROCA und Polyglot Persistence aufgestellt, um die Flexibilität und damit die Skalierbarkeit zu gewährleisten. Zudem wird eine starke Modularisierung durch die Einführung von horizontalen Schichten in einem bereits vertikal geschnittenen System beschrieben. Damit geht der Prototyp mit einer möglichst optimalen Änder- und Weiterentwickelbarkeit hervor. Für die Erzeugung dieser Microservices sind zudem Evaluationen, Abwägungen und Entwurfsentscheidungen für Architektur-, Design- und Code-Ebene festgehalten, um die gewählte Implementierung des Prototypen möglichst nachvollziehbar darzustellen. Für die meist kaufentscheidende Performance eines Onlineshops kommt Angular 2 zum Einsatz. Dieses, für das Frontend eingesetzte, Framework und die Integration der verschiedenen Microservices zu einem vollständigen System, werden ebenfalls ausgeführt und unter Aspekten wie der Sicherheit, der Performance und der Modularität bewertet. Hierbei wird für die Sicherheit eine Token-basierte Authentifizierung vorgeschlagen und eine mögliche Implementierung beschrieben. Abschließend erfolgt eine Evaluation der entstandenen Microservice-Architektur zu einem typischen, monolithischen System in Bezug auf die gestellten nicht-funktionalen Anforderungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

AMQP	Advanced Message Queuing Protocol	19
CORS	Cross-Origin Resource Sharings	22
HATEOAS	Hypermedia as the Engine of Application State	17
JWT	Java Web Token	22
KW	Kalenderwoche	10
LoC	Lines of Code	11
NPM	Node Package Manager	42
REST	Representational State Transfer	17
ROCA	Resource-oriented Client Architectur	16
RSA	Rivest, Shamir und Adleman	22
SSL	Secure Sockets Layer	22
TLS	Transport Layer Security	22

1. Einleitung

Dynamisch skalierende Applikationen in der Cloud finden nach Vaquero, Rodero-Merino und Buyya immer häufiger Aufmerksamkeit [vgl. Seite 45 VRB11]. Hierbei muss vor allem zwischen horizontaler und vertikaler Skalierbarkeit unterschieden werden[1]. Die horizontale Saklierbarkeit umfasst vor allem das replizieren neuer Server und die Lastverteilung zwischen allen Servern. Vertikale Skalierbarkeit ist dagegen das hinzufügen neuer Ressourcen (bspw. CPUs oder Speicher) zu dem gleichen System. Dies geschieht häufig auch on-the-fly, wobei viele Systeme dies nicht unterstützen und einen Neustart benötigen [vgl. Seite 45 VRB11]. Außerdem muss ab einer gewissen Größe typischerweise auf Spezialhardware zurückgegriffen werden. Dabei ist vor allem eine lineare Skalierbarkeit anzustreben. Durch diese hat das System die Möglichkeit, bei einer Verdopplung der Ressourcen eine gleich hohe Steigerung der Verarbeitung von Anfragen zu erzielen. Webanwendungen müssen heute sehr häufig eine Skalierbarkeit aufweisen, da die Auslastung meist nur schwer vorhersagbar ist.

1.1. Motivation

Diese zuvor in der Einleitung beschriebene Skalierbarkeit gilt es häufig in modernen Webanwendungen wie einem Webshop aufzugreifen und umzusetzen. Das geplante Shopsystem, das explizit eine hohe Skalierbarkeit aufweisen soll, ist hierbei mit diesem Ziel von Grund auf neu zu planen und zu entwickeln. Dies gibt die Möglichkeit, bereits innerhalb der Konzeption entsprechende Maßnahmen zu ergreifen, um dieser Anforderung bestmöglich nachzukommen. Hierfür kann ein sehr aktueller Architekturstil mit entsprechenden Konzepten, Tools und Frameworks zum Einsatz kommen, die zur Erreichung der Zielsetzung führen sollen.

1.2. Problemstellung

Im Zuge der Forderung nach Skalierbarkeit für Webanwendungen wie dem Onlineshop, gilt meist die individuelle Analyse, welche Architekturentscheidungen und verfügbaren Mittel den Anforderungen am geeignetsten entgegnen. Außerdem gilt hierbei, dass Skalierbarkeit die Komplexität des Systems in den meisten Fällen erhöht und Probleme entstehen können, die mit einer anderen Architektur ggf. nicht einher gehen. Die Kenntnis über entsprechende Probleme und mögliche Fallstricke fehlen häufig noch in den Unternehmen, um eine entsprechende Architektur zu konzipieren bzw. diese auch umzusetzen. Es gilt daher mögliche Fallstricke aufzudecken und entsprechende Maßnahmen zu ergreifen.

1.3. Zielsetzung

Im Fokus der Arbeit steht die technische Konzeption und die hierauf aufbauende Implementierung eines prototypischen, hoch skalierbaren Shopsystems. Es wird daher schritt-

[1] Vertikale Skalierbarkeit wird wird ebenfalls als Scaling-up, horizontale Skalierbarkeit dagegen als Scaling-out bezeichnet [vgl. Seite 3 PF04].

weise ein technisches Konzept erarbeitet, in dem Architektur und entsprechende Entwurfsentscheidungen erläutert sind. Im Zuge der Implementierungen ist zunächst ein generischer Microservice zu konzeptionieren und zu entwickeln, der die Grundlage für die in den Prototypen verwendeten Services darstellt und darüber hinaus auch in Zukunft für die Erweiterung des Systems durch zusätzliche Services als Template verwendet werden kann. Im Anschluss an das Prototyping gilt es, die verwendete Architektur mit der eines typischen, nicht skalierenden Shopsystems zu vergleichen.

1.4. Aufbau der Arbeit

Um die beschriebenen Ziele zu erreichen, wird zunächst in Abschnitt 2 auf die von der Technologie unabhängige, grundlegende Architektur eingegangen, die im Zuge der Konzeption gewählt wurde. Im Anschluss wird in Abschnitt 3 die Konzeption einzelner Services und deren Implementierung beschrieben. Hier stehen vor allem Technologiewahl, Designentscheidungen und deren Umsetzung im Fokus. Dies beinhaltet ebenfalls die in Abschnitt 3.2.4 ausgeführten Strategien für die Integration der Services in den Serviceverbund. Abschließend wird der beschriebene Prototyp eines Webshops noch auf die ausreichende Umsetzung der geforderten Qualitätsziele analysiert und bewertet.

1.5. Zeitplan

Für die Zielerreichung der zuvor beschriebenen Ziele lag ein bestimmter Zeitplan zugrunde. Dieser ist in Abbildung 1 mittels eines Gantt-Diagramms dargestellt bzw. in der Tabelle 2 noch detaillierter ausgeführt.

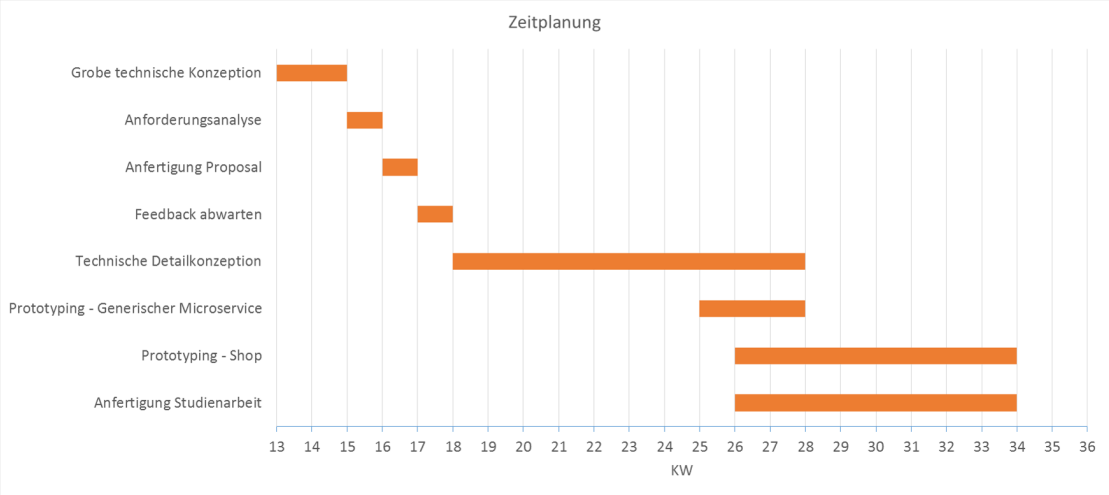

Abb. 1: Zeitplan für die Umsetzung der Zielsetzungen in Form eines Gantt-Diagramms

Aufgabe	Detailbeschreibung	Start KW	Ende KW
Grobe technische Konzeption	Festhalten der ersten Architektur- und Entwurfsentscheidungen. (aufgrund von Annahmen)	13	15
Anforderungsanalyse	Bis dahin eingegangene Anforderungen analysieren - Weitere Anforderungen oder Rahmenbedingungen werden als Annahme formuliert.	15	16
Anfertigung Proposal	Vorbereitung eines Proposals zur Absprache	16	17
Feedback abwarten	Feedback und Rücksprache zu dem angefertigten Proposal abwarten und weiteres Vorgehen klären (Detailabsprache).	17	18
Technische Detailkonzeption	Anfertigung eines detaillierten technischen Konzepts auf Basis der fachlichen Anforderungen.	18	28
Prototyping - Generischer Microservice	Entwicklung und Test eines generischen Microservices in einer geplanten Testumgebung.	25	28
Entwicklung eines lauffähigen Shop-Prototypen	Entwicklung und Fertigstellung eines lauffähigen Prototypen, der grundlegende Funktionen eines Webshops besitzt und den geforderten Qualitätskriterien entspricht.	26	34
Anfertigung Studienarbeit	Finale Fertigstellung der Studienarbeit.	26	34

Tab. 2: Detailbeschreibung zu dem in Abbildung 1 dargestellten Gantt-Diagramm.

2. Einführung in die Architektur hoch-skalierbarer Systeme anhand des Fallbeispiels Shopsystem

Um eine zuverlässige Skalierbarkeit anzustreben, gilt es ein paar Prinzipien zu beachten und umzusetzen. Hierzu gehört zunächst eine möglichst hohe Zustandslosigkeit. Das soll die Möglichkeit schaffen, Services beliebig in das System hinzu- und abzuschalten [vgl. Seite 49 VRB11]. Diese Zustandslosigkeit gilt neben dem System auch für die Kommunikationswege zwischen Client und Server sowie zwischen verschiedenen, möglichen externen Systemen. Zustände sollten daher nur clientseitig gehalten oder als Ressourcen persistiert werden.

Mögliche Schwankungen der Anfragen betreffen allerdings meist nicht alle Funktionalitäten und Ressourcen eines Shops. Im Fall eines Onlineshops werden bspw. Artikel viel häufiger als detaillierte Nutzerdaten abgefragt. Daher sollte eine möglichst hohe Modularisierung angestrebt werden. Idealerweise lassen sich auch somit einzelne Funktionalitäten skalieren. Die Modularität ist neben dem Aspekt der Skalierbarkeit auch aus Designgründen auf mehreren Ebenen einzuhalten. Hierzu zählen die Code-Ebene und die Ebenen der Komponenten- und Systemarchitektur [vgl. Seite 26 Fil12]. Auf Ebene der Systemarchitektur soll daher nun auf eine mögliche Modularisierung mittels Microservices eingegangen werden.

2.1. Microservices und die Servicearchitektur

Bevor eine Architektur beschrieben wird, ist zunächst der Begriff Microservice abzugrenzen. Häufig kommt es hierbei zu Missverständnissen. Der Präfix *micro* kann bspw. etwas möglichst kleines suggerieren, das mit sehr wenigen LoC auskommt und nicht mehr als eine Entität behandelt. Der Präfix *micro* sollte allerdings vor allem im fachlichen Kontext gehandelt werden. Hierbei ist gemeint, dass der entsprechende Microservice lediglich eine einzige fachliche Funktion bereitstellt. Aus diesem Konzept ergibt sich eine aufgetrennte Domäne, die auf verschiedene Microservices aufgeteilt wird [vgl. Seite 9 Fam15]. Hinzu kommen allerdings noch weitere Eigenschaften, die einzuhalten sind. Ein Microservice sollte immer eine autonome Lauffähigkeit aufweisen und unabhängig von der Umgebung sowie anderer Services möglichst gleiche Anfragenverarbeitung vornehmen und Antworten liefern. Außerdem sollten Microservices isoliert behandelt werden. Sie sind demnach nicht nur auf Komponentenebene möglichst unabhängig zu halten, sondern auch nach dem Aspekt der Verteilung separat betrieben werden. Zeitliche Abhängigkeiten sind ebenfalls zu vermeiden [vgl. Seite 10 Fam15]. Releases sind daher bspw. weitgehend unabhängig voneinander zu planen. Bei Änderungen, wie an der Schnittstelle eines Microservices, sind notwendige Releases von abhängigen Services allerdings meist mit einzuplanen. Generell sollte die Einhaltung eines *autonomen und isolierten*[2] Microservices über das gesamte Projekt von dem Entwurf, über die Entwicklung, Tests, Deployments bis hin zum Release-Management verfolgt werden. Über klar definierte Schnittstellen,

[2]Nach Familiar verhält sich ein Service autonom, wenn dieser unabhängig von des Systemkontextes arbeitet. Isolierte Services arbeiten dagegen unabhängig von Ort und Zeit des Gesamtsystems [vgl. Seite 10 Fam15].

einem Datenkontrakt und Konfigurationen werden diese in einem Systemverbund dargestellt. Dies bringt verschiedene Vor- und Nachteile, ein Umdenken in der Architektur sowie im Entwicklungsprozess mit sich, weshalb diese Aspekte nachfolgend weiter ausgeführt sind.

2.1.1. Aufbau einer Servicearchitektur mittels Microservices

Der in Abbildung 2 dargestellte Vergleich verschiedener Systemarchitekturen zeigt von links nach rechts eine Desktopanwendung, eine ebenfalls monolithische Client/Server-Architektur, eine beispielhafte n-Schichtenarchitektur mit 3 Schichten und eine Microservice-Struktur nach dem „Microservice Architektur Pattern“[vgl. Seite 584 Vil+15].

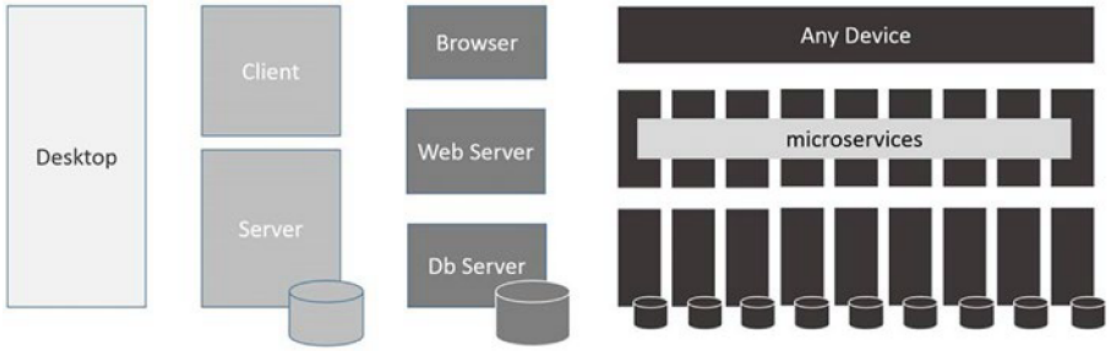

Abb. 2: Architekturvergleiche von unterschiedlichen Systemschnitten nach Familiar [Seite 6 Fam15]

Hierbei fällt auf, dass die Microservice-Architektur vertikal geschnitten ist, während die Schnitte der anderen Architekturen horizontal erfolgen. Die vertikal geschnittenen Microservices sollten hierbei isoliert und auf unterschiedlichen Systemen deployed werden, um sie tatsächlich unabhängig zu machen. Die voneinander unabhängigen Systemkomponenten besitzen damit grundlegend die Eigenschaften eines *verteilten Systems*[3].

2.1.2. Struktur, funktionale und nicht-funktionale Anforderungen des Shopsystems

Diese Microservicestruktur aus Abbildung 2 gilt es nun auf den Anwendungsfall eines Shopsystems zu übertragen. Zunächst sollten allerdings Anwendungsfälle festgehalten und Qualitätsziele erhoben werden, auf die sich Entwurfsentscheidungen und Bewertungen im Verlauf der Arbeit stützen.

[3]Ein verteiltes System zeichnet sich nach Schill und Springer dadurch aus, dass es sich „aus mehreren Einzelkomponenten auf unterschiedlichen Rechnern zusammen“ setzt, „die in der Regel nicht über gemeinsamen Speicher verfügen und somit mittels Nachrichtenaustausch kommunizieren, um in Kooperation eine gemeinsame Zielsetzung - etwa die Realisierung eines Geschäftsablaufs - zu erreichen.“ [Seite 4 SS12].

2.1.2.1. Anwendungsfälle des Prototypen

Der Prototyp soll möglichst allgemein gehalten werden, um als eine generische Beispielanwendung zu funktionieren. Diese soll die typischsten Anwendungsfälle zeigen und damit zukünftig möglicherweise mehreren Anwendungen als Grundlage zur Weiterentwickelung dienen. Das in Abbildung 3 dargestellte Usecase-Diagramm zeigt die wichtigsten Anwendungsfälle, die von dem Prototypen angeboten werden.

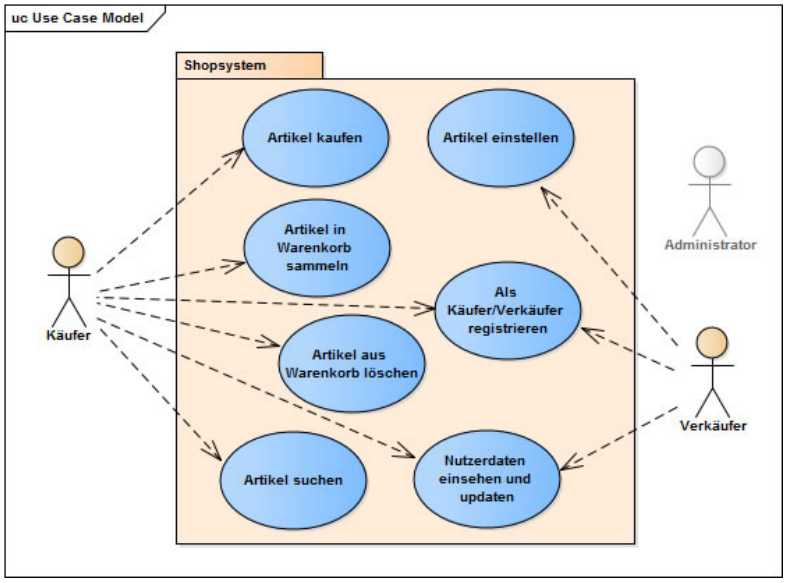

Abb. 3: Usecase-Diagramm der im Prototypen abgedeckten Anwendungsfälle.

Mögliche weitere wichtige Anwendungsfälle werden im Zuge dieser Arbeit abgegrenzt, da diese sonst den Rahmen sprengen würden.

2.1.2.2. Qualitätsziele

Als Qualitätsziel sind die für die Qualität der Software festzulegenden, nichtfunktionalen Anforderungen gemeint. Diese können unter anderem aus der ISO/IEC 9126[4] selektiert werden. Da Qualitätsziele allerdings miteinander korrelieren, also in Wechselwirkung stehen, muss eine Selektion und Priorisierung dieser Ziele erfolgen [vgl. Seite 92 Sta15].

Die für das System und den dafür entwickelten Prototypen definierten Qualitätsziele sind die folgenden:

1. Skalierbarkeit - Diese Anforderung ist aus der Zielsetzung der Arbeit gegeben und ist auch nicht explizit Bestandteil der ISO 9126.

2. Performance

[4]Die ISO/IEC 25010:2011 gilt zwar als der Nachfolger der ISO 9126, jedoch hat sie bisher in der Praxis kaum Einzug gehalten [vgl. Seite 41 Sta15]. Deshalb wird im Zuge dieser Arbeit die weit verbreitete ISO 9126 verwendet.

3. Änderbarkeit

Nachdem Performance noch als Anforderung an das System ausgemacht werden konnte, basiert die Änderbarkeit lediglich auf Annahmen. Hierbei wird angenommen, dass die Änderbarkeit eine Weiterentwicklung des Prototypen zu einem fertigen Endsystem unterstützt und damit als eines der übergeordneten Ziele gilt. Die zu Beginn des Kapitels 2 beschriebene, anzustrebende Modularität auf Code-Ebene sowie den Ebenen der Komponenten- und Systemarchitektur erleichtert die Erreichbarkeit des Qualitätsziels „Änderbarkeit“. Außerdem bestärken die, neben den in Abschnitt 2.1.3 beschriebenen Vorteile, die Entscheidung zur Nutzung einer Microservicestruktur.

2.1.2.3. Komponentensicht einer möglichen Endanwendung

Aus den zuvor festgelegten Anwendungsfällen könnte für ein weiterentwickeltes Shopsystem ein Serviceverbund nach der Abbildung 4 entstehen. Der Prototyp ist allerdings nicht nach den in der Abbildung dargestellten Komponenten geschnitten und trägt lediglich eine Teilfunktionalität. Die genauen Funktionen des Prototypen sind in Abschnitt 3.2 vermerkt. Diese Aufstellung aus Abbildung 4 soll ein Beispiel einer weiterentwickelten Shop-Plattform bestehend aus Microservices zeigen.

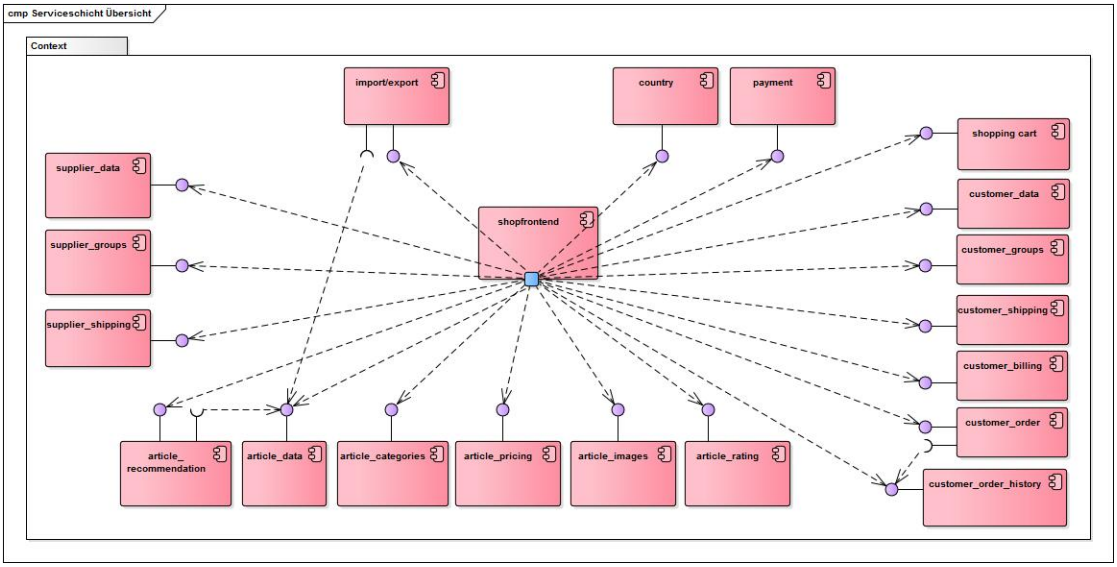

Abb. 4: Komponentendiagramm einer vertikal geschnittenen Microservicestruktur für ein beispielhaftes Shopsystem.

2.1.3. Vorteile und Herausforderungen einer Microservicearchitektur

Typischerweise sind die Qualitätsziele, die an den Systemverbund gestellt werden, einzuhalten und beeinflussen die Qualitätsziele der einzelnen Microservices. Da jeder Microservice separat geplant und entwickelt wird, lassen sich die nicht-funktionalen Anforderungen zu jeder einzelnen Funktion separat betrachten. Außerdem sind Build- und

Testzyklen wesentlich schneller als bei monolithischen Systemen [vgl. Seite 6 Fam15]. Weitere Vorzüge für Microservices fast Familiar wie folgt zusammen:

- Evolutionär - Statt monatelangen oder sogar langjährigen Entwicklungszyklen können Services und deren Funktionalität priorisiert, möglichst unabhängig entwickelt und dem Serviceverbund nach und nach hinzugefügt werden. Dies ermöglicht eine durchgehende Erweiterung des Serviceangebots.
- Offen - Diese Art der Architektur lässt jedem Entwicklungsteam die Möglichkeit, neben den vorgegebenen Prinzipien, bestimmte Entscheidungen wie die Wahl der Programmiersprache, Zielsystem und Persistenz selbst zu treffen. Außerdem müssen geographisch verteilte und unterschiedliche, mit funktionalen Querschnittsaufgaben betreute Teams nicht mehr stark organisiert werden, sondern verständigen sich ebenfalls - im besten Fall - lediglich über die festgelegten APIs.
- High Velocity - Das Erweitern sowie aktive Entwicklungsarbeiten an dem System können beschleunigt verlaufen, da bspw. bei größeren Anwendungen keine Verarbeitung des Gesamtsystems im Buildprozess vorgenommen werden muss.
- Wiederverwendbar und Verbindbar - Microservices können jederzeit wiederverwendet und für unterschiedliche Systemverbunde herangezogen werden.
- Flexibilität - Es ist häufig einfacher für Entwicklerteams ein Entwicklungs- und Deployment-Konzept aufzustellen, zu dem auch Testing, Staging und die Produktivstellung gehören.
- Versionierbar und Ersetzbar - Es ist möglich, unterschiedliche Versionen eines Microservices zu betreiben, um somit Abwärtskompatibilität und eine erleichterte Migration zu ermöglichen.
- Besitz liegt bei lediglich einem Team. Idealerweise ist dieses Team auch an der Entwicklung beteiligt gewesen.

[vgl. Seite 13f. Fam15]

Diese Vorteile gehen allerdings nicht ohne einem gewissen Trade-off einher. Es entstehen bei der Verwendung von Microservices vor allem Herausforderungen bei der Integration der einzelnen Komponenten zu einem technisch und fachlich funktionierenden Systemverbund. Die Integration und mögliche Strategien werden daher am Beispiel des Prototypen in Abschnitt 3.2.4 noch einmal ausführlich erläutert. Ein System mit vielen verschiedenen Services und Entwicklungsteams erfordert ein konstantes Integrations- und Deploymentkonzept. Diese Aspekte können schnell sehr komplex werden, wenn kein nahtloser Übergang von Entwicklung zum Betrieb erfolgt. Daher sollte im Zuge der Verwendung von Microservices die Umstellung auf DevOps[5] in Betracht gezogen werden [vgl. Seite 13 Fam15].

[5] Unter DevOps wird das Vorgehen verstanden, die Verantwortung für Entwicklung, Deployment und Betrieb bei einem einzigen Team zu halten, statt diese getrennt in verschiedenen Organisationseinheiten umzusetzen [vgl. Seite 13 Fam15].

Generell kann bei dem Erstellen solcher verteilten Systeme auf bestimmte Regeln zurückgegriffen werden. Daher wird im Kontext der Entwicklung einer Webanwendung auf die Resource-oriented Client Architecture im Nachfolgenden eingegangen und evaluiert.

2.2. Resource-oriented Client Architecture

Die *Resource-oriented Client Architectur* oder auch ROCA stellt durch dessen Manifest klar definierte Richtlinien, Regeln und Ziele für das Entwickeln einer verteilten Systemarchitektur dar. Daher gilt es diese vorab für die zuvor in Abschnitt 2.1.2 beschriebene Microservice-Architektur und die Umsetzung des Prototypen in Betracht zu ziehen. Für das Backend sind die Punkte aus der Tabelle 3 als Prinzipien definiert.

Prinzip	Beschreibung
REST	Kommunikation über einen RESTful Webservice unter dem Einhalten aller Prinzipien.
Application Logic	Die gesamte Businesslogik liegt im Backend.
HTTP	Der Nutzer greift ausschließlich mittels HTTP-Anfragen auf die Services des Backends zu.
Link	Jede Ressource besitzt einen eindeutige ID.
Non-Browser	Der Service muss auch über andere Clients als dem Browser funktionsfähig sein. Hierzu zählen bspw. Kommandozeilenclients wie curl oder wget.
Should-Formats	Ressourcen sollten stets über verschiedene Repräsentationen bereitgestellt werden.
Auth	Es sollte immer auf eine Basic oder Digest Authentifizierung in Verbindung mit SSL gesetzt werden. Alternativ kann ein Formular-basierter Login eingesetzt werden. Bspw. wegen fehlender Unterstützung von Logout und Styling im Browser. Wenn im Zuge dessen Cookies verwendet werden, sollten diese alle zustandsbezogenen Informationen speichern.
Cookies	Sollten nur zur Authentifizierung oder zum Tracking verwendet werden.
Session	Jede Session ist immer zustandslos und sollte lediglich für simple Validierungen zur Authentifizierung einen Zustand halten.
Browser-Controls	Die Browserfunktionen „zurück“, „vorwärts“ und „aktualisieren“ sollten so funktionieren, wie sie von dem Browser vorgesehen sind.

Tab. 3: Zu beachtende Prinzipien für das Erzeugen eines Backends nach ROCA [vgl. Inn15a].

Ebenso definiert ROCA auch Prinzipien für das Frontend. Im Zuge der Entwicklung des Frontends für den Prototypen wurden diese Prinzipien allerdings nicht beachtet. Durch das Verfolgen der ROCA-Prinzipien, müssen bestimmte Kompromisse eingegangen werden. Bspw. können durch das serverseitige Herausreichen von HTML-Markup, das auf Clientseite lediglich durch Layout-Informationen und Verhalten erweitert und ausgegeben wird, gewisse Nachteile entstehen. Es muss zusätzlich zu den Ressourcen der HTML-Markup übertragen werden, was zu Performanceverlusten führen kann. Außerdem kann das Problem durch das POSH-Prinzip entstehen, dass verschiedene Clients unterschiedlichen HTML-Markup benötigen und die Kommunikation mit Nicht-ROCA-Clients erschwert wird [vgl. Inn15b]. In den Abschnitten 3.1.3 und dem Abschnitt 3.2.3 wird noch einmal konkret die Entwurfsentscheidung für die Authentifizierung und das Frontend durch eine alternative Lösung beschrieben. Die grundlegenden Prinzipien für das Backend werden für den Prototypen fast alle umgesetzt. Durch die gewählte Integrationsstrategie liegt ein Teil der Applikationslogik im Client, weshalb neben dem Prinzip *Application Logic* auch das Prinzip *Non-Browser* nur zum Teil auf den Prototypen zutrifft. Dafür wird eine clientseitige Validierung weitestgehend vermieden und lediglich innerhalb der Businesslogik im Backend angewendet. Ebenfalls wird im Zuge des Prototypen eine statuslose Kommunikation angestrebt und von ROCA durch eine Kommunikation mittels REST/HTTP vorgegeben. Darum ist diese nachfolgend noch einmal zusammengefasst.

2.3. Schnittstellenkommunikation nach REST

Representational State Transfer, auch bekannt als REST ist ein Architekturstil, der erstmals in einer Dissertation von Fielding Anfang des 21.Jahrhunderts aufgegriffen und durch dessen Arbeit geprägt wurde. Dort beschrieb er bereits das an die Web Architektur angelehnte Vorgehen für Webanwendungen [Fie00, vgl. Seite 76].

Um einen Webservice als RESTful bezeichnen zu können, sollte dieser im Grunde folgende Architekturprinzipien ausweisen:

- Eindeutige Adressierbarkeit: Anhand der URIs werden die durch den RESTful Service bereitgestellten Ressourcen mit eindeutig adressierbaren IDs versehen.
- Verknüpfungen/Hypermedia: Durch Links werden Ressourcen miteinander, ungeachtet des Applikationszustandes und des Serverstandortes, verknüpft. Dieses Vorgehen ist auch als HATEOAS bekannt.
- Ausschließlich HTTP-Standardmethoden: Zu den Methoden gehören die bekannten Befehle GET, POST, PUT, DELETE und andere meist weniger genutzte, zusätzliche Methoden wie bspw. HEAD und OPTIONS.
- Unterschiedliche Repräsentation: Durch das Akzeptieren von Anfragen und Bereitstellen von Ressourcen in unterschiedlichen Datenformaten, ist eine unabhängigere Kommunikation möglich. Client und Server müssen sich nicht auf ein Datenformat verständigen und lediglich dem zu Grunde liegenden Protokoll folgen.

- Statuslose Kommunikation: Der Zustand wird vollständig vom Client getragen oder von dem Server als eigenständige Ressource behandelt. Ein serverseitiger Sitzungsstatus ist nicht vorgesehen, da damit die zu erzielende Skalierbarkeit und Verringerung der Kopplung zwischen Client und Server gefährdet wird.

[Til11, vgl. Seite 11ff]

Diese von Fielding beschriebenen Prinzipien sollen alle für den Prototypen umgesetzt werden.

2.4. Datenhaltung innerhalb der Servicearchitektur mittels Polyglot Persistence

Die zuvor beschriebenen Prinzipien durch ROCA in Abschnitt 2.2 beschreiben keine Vorgehen zu dem Persistieren von Daten im Backend. Dennoch gilt es diese bei einem hoch skalierenden und verteilten System zu betrachten. Da die verschiedenen Microservices aus Sicht der Verteilung voneinander isoliert sind und damit auch deren domänenspezifische Funktionen und Entitäten typischerweise verteilt liegen, würde eine gemeinsame Datenhaltung die erzielten Vorteile der Architektur ggf. aufheben. Es müsste mit einem erneuten Auftreten eines nicht horizontal skalierenden Flaschenhalses bei Zugriffen auf die Datenbanken gerechnet werden. Außerdem kann es, nach dem getrennten Halten der Fachlichkeit in den darüber liegenden Schichten, zu einem erneuten Verstricken des Domänenmodells auf Datenebene kommen. Dies kann bei der Wartung und der Erweiterung des Systems durch verschiedene Teams zu zusätzlicher Komplexität und Problemen führen. Abgesehen von den zuvor beschriebenen Nachteilen stellt sich die Frage, welches Datenbankkonzept die optimale Lösung für den Anwendungsfall „hochskalierbares Shopsystem“ bietet. Ein Onlineshop hält meist sehr verschiedene Funktionen mit unterschiedlichen Daten und Datenoperationen bereit. Da diese Differenzen unter den Microservices gegliedert werden können, ist jeder Einzelne separat zu betrachten. Das von Martin Fowler bereits 2011 vorgestellte *Polyglot Persistence* kann an dieser Stelle den zuvor beschriebenen Nachteilen und Bedingungen entgegenwirken. Ebenfalls beschreibt dieser in seinem Blog eine beispielhafte Nutzung verschiedener Datenbanken im Kontext eines Onlineshops. Daraus geht die im Anhang A beigefügte Abbildung 11 hervor, die eine mögliche Aufstellung von Services und deren passend gewählten Datenbanken darstellt. Die Wahl des geeigneten Datenbankkonzepts und der passenden Lösung muss im Fall von jedem geplanten Microservice einzeln abgewogen werden. Daher sind die getroffenen Entscheidungen in Abschnitt 3 im Zuge der Konzeption der Services festgehalten. Für den ausgelieferten Prototypen wurde jeder Microservice lediglich auf eine H2-InMemory-Datenbank gesetzt. Dadurch kann der Prototyp leichter initial aufgesetzt und getestet werden. Mit dem Einsatz des Migrationsframeworks *Flyway*[6] und beispielhaften Konfigurationszeilen zur Integration einer Oracle-Datenbank, welche explizit angegeben und jederzeit einkommentiert werden können, lassen sich sehr schnell andere Datenbanktypen einbinden.

[6] Das Framework Flyway ist ein von *boxfuse* Open Source zur Verfügung gestelltes Datenbank Migrationstool [vgl. Seite 255 Gut16].

3. Konzeption und Implementierung der Services

Nachdem verschiedene Konzepte auf Architekturebene beschrieben und die grundlegende Architektur festgelegt wurde, gilt es nun den Prototypen auf Design- und Codeebene genauer zu betrachten.

3.1. Übergreifender Funktionsinhalt und Aufbau eines Microservices

Bevor auf den Prototypen und dessen Integrationsstruktur genauer eingegangen wird, sind vorab Aufbau eines Microservices und grundlegende Konzepte beschrieben. Hierzu zählen auch bestimmt Frameworks, mittels derer die Erzeugung eines Microservices wesentlich erleichtert wird. Für diese wird nachfolgend zunächst eine kurze Evaluierung auf einer Vorauswahl durchgeführt.

3.1.1. Entwicklungsunterstützung mittels Frameworks

Im Zuge der Entwicklung einzelner Services für den Prototypen wurde auf eine möglichst einheitliche Struktur und Wahl der Frameworks gesetzt. Damit soll das Aufsetzen, Analysieren und Weiterentwickeln des Prototypen erleichtert werden. Das Backend wird in Java entwickelt, da es hierfür bereits eine große Auswahl an Frameworks gibt. Detaillierteres zu den eingesetzten Frameworks ist im nachfolgenden Abschnitt 3.1.2 aufgeführt. Als Tool für das Build-Management kommt Apache Maven zum Einsatz. Für das erleichterte Erzeugen von Microservices mit entsprechender Konfiguration und Abhängigkeiten zu Bibliotheken sind ebenfalls Frameworks frei verfügbar. Daher gilt es diese kurz zu evaluieren und eine Entwurfsentscheidung für den Einsatz im Zuge des prototypings zu treffen. Die in Tabelle 4 beschriebenen Frameworks stellen hierbei eine kleine Auswahl sehr bekannter Lösungsansätze und deren Unterschiede dar.

Vergleichkriterien	Spring Boot	Dropwizard	Rest Express
Version	1.3.5	0.9.2	0.11.3
HTTP Container	Tomcat, Jetty, Undertow	Jetty	Netty
Service-kommunikation	HTTP/REST, JMS, AMQP und SOAP	HTTP/REST[7]	HTTP/REST
JSON	Jackson, GSON, json-simple	Jackson	Jackson, GSON
Logging	Lockback, Log4j, Log4j2, slf4j, Apache common-Logging	Jackson	Log4j

Tab. 4: Evaluation verschiedener Microservice-Frameworks, nach [Riz15, vgl.], [Zhi15, vgl.] und den POM-Dateien der Frameworks.

[7] Inoffiziell können noch weitere Arten unterstützt werden, jedoch sollen diese für den Prototypen und eine mögliche Endanwendung als nicht relevant gelten.

Für die Microservices des Prototypen wurde zugunsten von Spring Boot entschieden, da viele der hierbei zum Einsatz kommenden Bibliotheken aus dem Spring Framework selbst kommen und diese bereits aus dem Standard Spring Framework gut aufeinander abgestimmt sind. Außerdem bietet Spring die größte Flexibilität, da es am Beispiel von HTTP Container, Json-Verarbeitung und Logging die meisten unterschiedlichen Frameworks von Drittanbietern offiziell unterstützt. Diese große Auswahl verlangsamt den Service allerdings auch nicht, da Spring Boot sehr modular aufgebaut ist und eine Selektion für die Spring Boot Anwendung erfolgen kann. Dropwizard und RestExpress bieten grundsätzlich zunächst nur eine Kommunikation nach REST. Spring Boot bietet neben REST auch einen einfachen Support für JMS, Advanced Message Queuing Protocol und SOAP. Daraus lässt sich schließen, dass wenn ein Wechsel auf eine, möglicherweise zukünftige Kommunikationsmethode oder ein anderes Protokoll als HTTP angestrebt wird, dies mit Spring Boot am einfachsten funktioniert. Außerdem ist Spring Boot zu diesem Zeitpunkt die einzige Lösung, die über die Version 1.0 hinaus ist, was ein Indikator für die Reife des Produkts sein kann.

Für nachfolgend implementierte Services, welche im Zuge einer Weiterentwicklung später in das System hinzugefügt werden sollen, können die Frameworks und Technologien unabhängig von den Implementierungen des Prototypen gewählt werden. Ebenfalls kann auch ein kompletter Technologiewechsel erfolgen.

3.1.2. Aufbau eines Microservices für das Shopsystem

Während das Framework *Spring Boot* die passenden Bibliotheken liefert, ist die Architektur eines einzelnen Microservices gesondert zu betrachten. Auch wenn Microservices möglichst klein gehalten werden, bietet hierbei die Schichtenarchitektur innerhalb der Services gewisse Vorteile. Es werden demnach die aus Abbildung 2 vertikal geschnittenen Services noch einmal, nach der n-Schichtenarchitektur, horizontal geschnitten. Hierbei steht vor allem das Qualitätsziel der Änderbarkeit im Fokus. Wie Siedersleben bereits 2000 beschreibt, kann mittels der Verwendung einer Schichtenarchitektur eine Trennung von unter anderem anwendungsbasierter und technischer Software erreicht werden. Damit wird anwendungsbasierte Software wiederverwendbarer und technische Software leichter austausch- bzw. änderbar [vgl. Seite 248f. SD00]. Dieses Vorgehen wurde daher für den Prototypen, wie in Abbildung 5 beschrieben, umgesetzt. Die Trennung zwischen einer Kommunikationsschicht nach REST, einer Serviceschicht und einer DAO-Schicht ermöglicht damit bspw. den vereinfachten Austausch der technischen Kommunikationsart oder des Datenbankzugriffs.

Allerdings sind diese Vorteile nicht ohne damit verbundene Nachteile zu erreichen. Die Verwendung von Schichten und Frameworks wie *Jackson*[8], *Spring HATEOAS* und *Hibernate*[9] erfordern in der Rest-, Service und DAO-Schicht jeweils eigene Datentransportobjekte. Diese müssen wiederum aufeinander gemapped werden, was bei einem manuellen

[8] Jackson ist ein im Java-Kontext frei verfügbarer JSON-Parser, der POJOs in JSON-Format und wieder zurück konvertiert. Hierbei wird ebenfalls XML unterstützt.

[9] Hibernate ist ein ORM-Mapper, der zum Teil nach dem JPA-Standard implementiert ist. Hibernate ist nicht die Referenzimplementierung, jedoch sehr weit verbreitet.

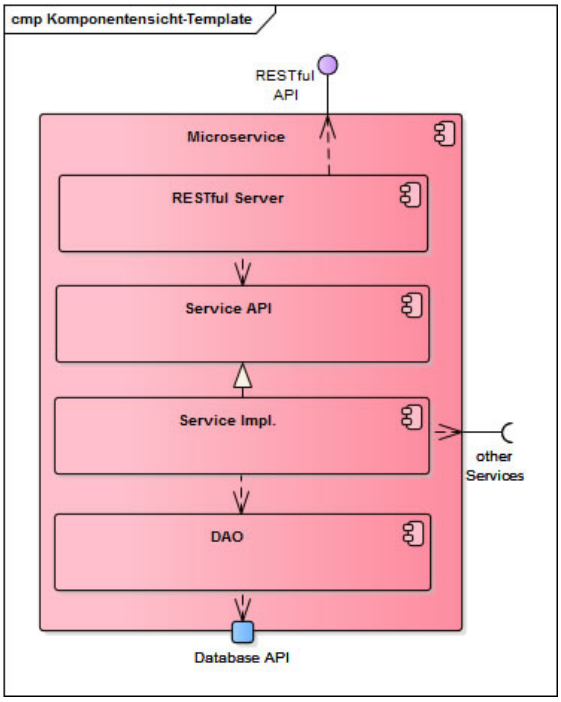

Abb. 5: Generell angestrebte Struktur aller für den Prototypen implementierten Microservices.

Vorgehen gegen das gesetzte Qualitätsziel der „Änderbarkeit“ wirkt. Es sollte daher auf ein automatisiertes Mapping mittels Framework zurückgegriffen werden. Hierfür kommt im Prototyp das Mapping-Framework *MapStruct* zum Einsatz, da dieses den notwendigen Code zur Compilezeit generiert, statt ein Mapping wie die meisten anderen Lösung mittels Java Reflections durchzuführen. Der generierte Code ist typischerweise schneller wie der mit Reflections zur Laufzeit ausgewertete Code. Damit konnte eine Minderung der Performance, welche als Qualitätsziel gilt, verhindert werden [vgl. WYS16]

Unabhängig von den Schichten hält auch jeder Microservice im Vergleich zu Monolithen gewissen redundanten Code, der die Applikation schwerer wart- und änderbar macht. Diese Redundanzen lassen sich allerdings bspw. mit einem als eigenständiges Maven-Projekt entwickeltes „Commons-Package“, das in jeden Microservice eingebunden wird, vermindern.

3.1.3. Authentifizierungskonzept

Zur Absicherung des Systems muss ein Authentifizierungs- und Authorisierungskonzept erstellt werden. Dieses wird auch grundlegend innerhalb des Prototypen eingesetzt. Im Gegensatz zu einer monolithischen Anwendung, besitzt das geplante Konzept typischerweise keinen zentralen Einstiegspunkt. Dies ist ein Problem, da die Authentifizierung und ein übergreifendes Berechtigungskonzept zentral gehalten werden sollte. Andernfalls würde es zu Redundanzen und Synchronisationsproblemen führen.

Es können für das verteilte System, bestehend aus vielen Microservices, unterschiedliche Authentifizierungskonzepte für den Einsatz erwogen werden. Zum einen kann, nach den ROCA-Prinzipien, bei jeder Clientanfrage an die Microservices in allen HTTP-Headern eine Basic- oder Digest-Authentifizierung übergeben werden. Die Microservices wiederum authentifizieren den Nutzer im Anschluss an einem Authentifizierungsservice. Dieses Vorgehen ist in Abbbildung 6 durch den Informationsfluss zwischen den Komponenten aus dem Punkt A dargestellt. Hierbei entsteht allerdings das Problem, dass die Skalierbarkeit eingeschränkt wird. Bei X clientseitigen Seitenaufrufen, die Y Microservices anfragt, wird der Authentifizierungsservice mit X*Y Anfragen belastet.

Um dieses Problem teilweise zu umgehen, besteht ein sogenanntes *Gateway API Pattern*, das den zentralen Zugriffspunkt für die Authentifizierung zurückbringt. Dieses ist in Abbildung 6 durch Komponenten in Punkt B definiert. Dieses Vorgehen wird unter anderem von Netflix unterstützt und voran getrieben [vgl. Ric15]. Das Gateway API liegt zwischen dem Client und den Services und bietet unter dem Aspekt der Sicherheit den Vorteil der zentralen Authentifizierung jedes Aufrufes und der möglichen Bündelung der Anfragen an den Authentifizierungsservice. Außerdem geht somit das bereits während der Beschreibung der Architektur erwähnte Problem des *Cross-Origin Resource Sharings* bzw. *CORS* verloren, da alle Anfragen nur an eine Domain gerichtet werden kann.

Dieses Gateway bekommt allerdings durch die ROCA-Architektur ebenfalls so viele Anfragen, wie der Authentifizierungsservice aus Punkt A der Abbildung 6. Außerdem würde das Vermeiden eines zentralen Zugriffspunktes die Skalierbarkeit und Ausfallsicherheit erhöhen, da kein hochverfügbarer Zugriffspunkt bereitgestellt werden muss. Daher wird für die Entwicklung des Prototypen der in Abbildung 6 beschriebene Punkt C als Authentifizierungsstrategie herangezogen. Die Idee ist hierfür einen Token-basierten Ansatz zu wählen. Der Client benötigt einen Token, den er sich einmalig über den Authentifizierungsservice besorgt. Hierbei wird das, ebenfalls durch ROCA vorgeschlagene, Basic-Verfahren in Verbindung mit SSL/TLS[10] verwendet. Der Token wird nach dem offenen JWT-Standard, welcher durch den RFC 7519 spezifiziert ist [vgl. IET15], mit Nutzerinformationen wie den Nutzer- und Gruppennamen, allerdings ohne dem Nutzerpasswort, generiert.

Der JWT-Standard beschreibt die Generierung von Token, bestehend aus drei Teilbereichen. Der erste Teil des Token ist der Header, in dem der Tokentyp mit dem unterstützten Algorithmus beschrieben wird. Grundlegend unterstützt JWT standardmäig das HMAC-Verfahren und eine RSA-Verschlüsselung [vgl. Seite 6 und 26 IET15]. Im Zuge des Prototypen wird auf das HMAC-Verfahren mit dem Hash-Algorithmus SHA-512[11] gesetzt. Der zweite Teil des Token umfasst die unverschlüsselten Nutzerdaten, wobei diese, selbst bei einem erfolgreichen Angriff, nur die zuvor beschriebenen, nicht kritischen Nutzerinformationen darstellen. Der dritte Block bildet eine Signatur, um den Token als valide zu prüfen und die Kopiersicherheit sicherstellen zu können. Die

[10] Da im Oktober 2014 eine schwere Sicherheitslücke in SSL 3.0 und den Vorgängern gefunden wurde, ist das als Nachfolger gehandelte Verschlüsselungsprotokoll TLS zu bevorzugen [USC, vgl.]. TLS 1.0 sollte allerdings ebenfalls ausgeschlossen werden, da es mit SSL 3.0 gleichzusetzen ist.

[11] Dieser seit 2002 veröffentlichte Algorithmus ist einer der Nachfolger von SHA-1 und arbeitet mit 512 Bits [DMO04, vgl. Seite 1].

Signatur enthält einen privaten Schlüssel, den jeder Microservice aus der Servicestruktur kennt. Um die Sicherheit zu erhöhen, kann in weiterführenden Entwicklungsarbeiten bspw. mit unterschiedlichen privaten Schlüsseln für verschiedene Systembereiche gearbeitet werden. Damit lassen sich kritischere Systembereiche gesondert behandeln. Außerdem kann außerhalb des Leistungsumfanges des Prototypen, ein Verfahren zum regelmäßigen Wechseln des privaten Schlüssels auf allen Servern erwogen werden.

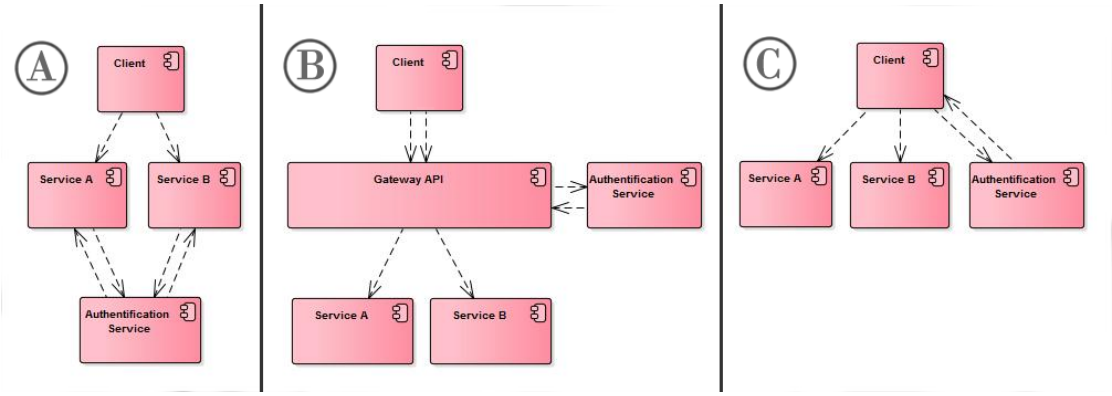

Abb. 6: Vergleich Basis Authentifizierung, Gateway API Pattern und Token Authentifizierung

Der für den Prototypen vorgesehene Token umfasst die folgenden, durch die ROCA-Prinzipien möglichst gering gehaltenen, Daten:

- Benutzername
- Benutzer-Id
- Nutzerrollen
- Erzeugungszeitstempel
- Aussteller des Token
- Verfallszeitpunkt
- Anzahl der fehlgeschlagenen Loginversuche seit der letzten erfolgreichen Anmeldung

Diese im Prototypen eingesetzte JWT-Schlüssel sind jederzeit auslesbar und müssen lediglich anhand der Signatur validiert werden. Dazu kommt eine angedachte, verschlüsselte Kommunikation über HTTPS. Davon abgesehen werden keine weitere Maßnahmen im Zuge der Prototypen-Entwicklung ergriffen.

3.2. Implementierung der Microservices im Kontext des Shopsystems

Abgeleitet von der in Abbildung 4 dargestellten Komponentensicht eines möglichen, umfassenden Shopsystems, sind in Abbildung 7 die Komponenten des implementierten Prototypen dargestellt. Um einen beispielhaften Onlineshop aufzusetzen, sind diese 7 Komponenten ausgewählt worden. Bei allen, bis auf den storefront-Service, handelt es sich um Backendkomponenten, die eine Struktur nach Abbildung 5 aufweisen sowie voneinander isolierte und autonome Services darstellen. Die Abhängigkeiten liegen hierbei hauptsächlich zwischen der Frontend-Komponente[12] und den anderen Komponenten und stellen die Integration der einzelnen Services dar. Details sowie Vor- und Nachteile zu der Integrationsstrategie werden in Abschnitt 3.2.4 genauer erläutert.

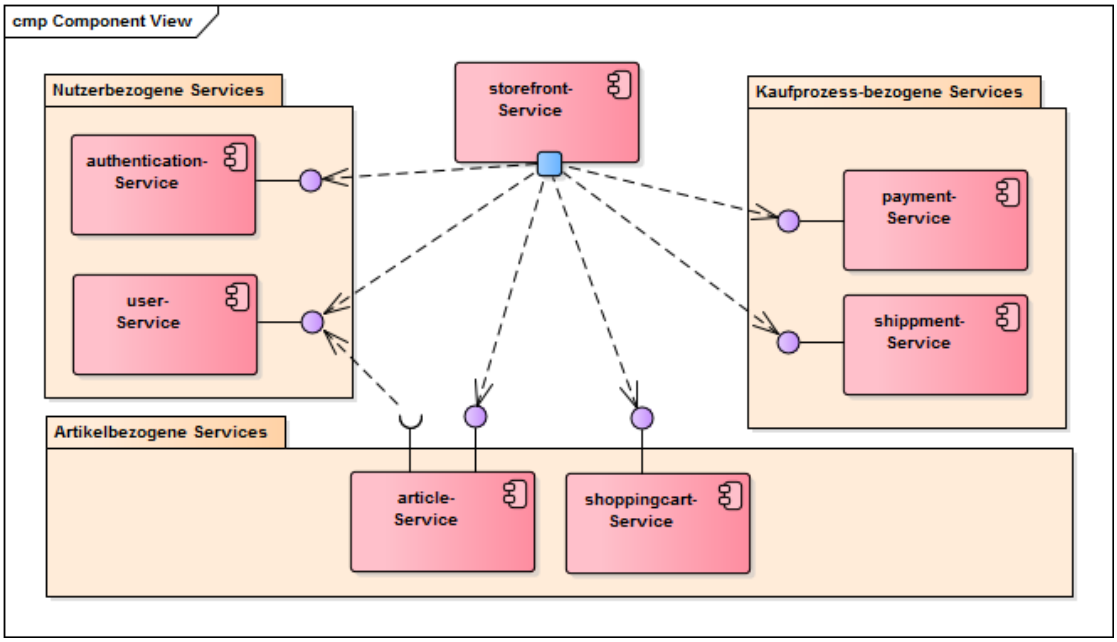

Abb. 7: Komponentensicht des IST-Zustandes des Prototypen.

3.2.1. Zweck und Verantwortlichkeit der Komponenten

Der Article-Service hält alle notwendigen Stammdaten eines Artikels. Daher liegt dieser Service sehr Zentral im Systemverbund. Alle Artikel-bezogenen Seiten benötigen diese oder einen Teil dieser Daten. Außerdem ist, unabhängig der gewählten Datenbank, eine Fremdschlüsselreferenz auf den Verkäufer hinterlegt. Details zu den gehaltenen Attributen des Services sind nachfolgend in der Schnittstellenbeschreibung festgehalten. Ebenfalls sehr zentral liegt der Authentication-Service, da dieser, wie bereits in Abschnitt 3.1.3 beschrieben, die Login-Credentials und die Berechtungsrollen hält und dem

[12] Der Storefront-Service liegt ebenfalls im Backend, dient allerdings lediglich dem Bereitstellen der Frontend-Masken für den Client.

Client einen Token zur Authentifizierung und Autorisierung an allen anderen Services bereit stellt. Dazu ergänzende Nutzerdaten werden dagegen separat im User-Service verwaltet. Der Prototyp ist hierbei so implementiert, dass der Authentication-Service die technische User-Id ausstellt, jedoch der User-Service die Verkäufer-Id. Der Shoppingcart-Service hält mit User-Id und Artikel-Id alle Warenkorbeinträge des Shops. Der Payment- und Shippment-Service liefern dagegen alle Daten zu Zahlungsoptionen und Versandoptionen, die der Verkäufer konfiguriert. Hierbei sind die Zahlungsoptionen übergreifend über alle Artikel auf den Verkäufer konfiguriert, während Versandoptionen direkt auf den Artikel konfiguriert werden können.

3.2.2. Schnittstellenbeschreibung

Diese Komponenten besitzen ein definiertes Domainen-Modell[13], welches in Abbildung 8 dargestellt ist. Die bereits dargestellte Schnittstellenbeschreibung der Backendservices aus Abbildung 8, soll nun noch einmal in den Tabellen 6, 7, 8, 9, 10 und 11 aus Anhang B detailliert beschrieben werden.

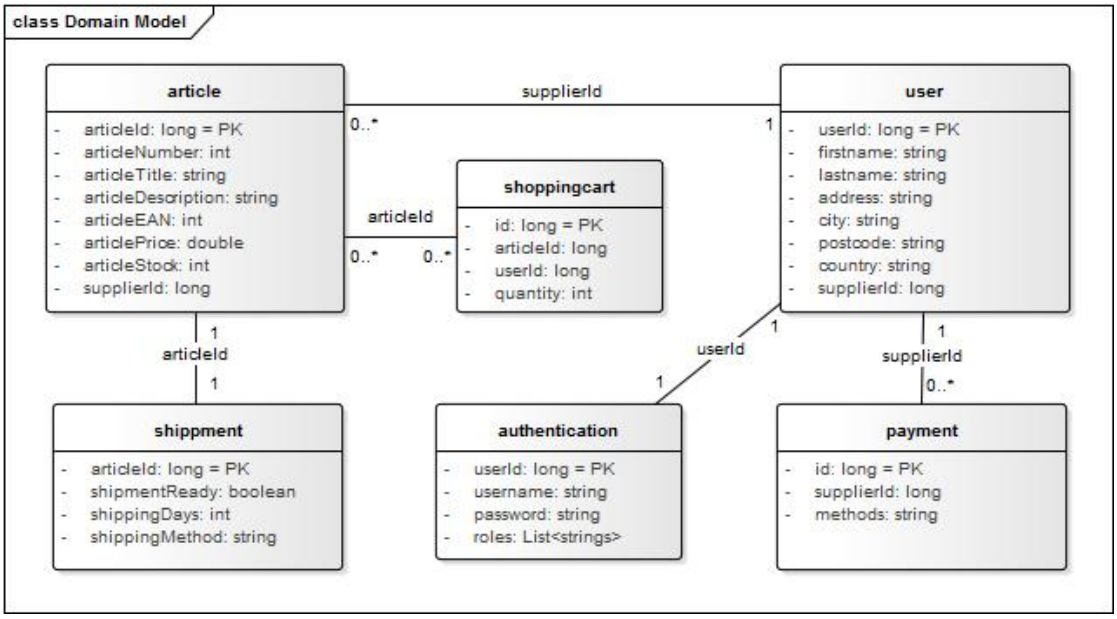

Abb. 8: Fachliches Domainmodell des Prototypen.

[13] Da es sich hierbei neben der Beschreibung des unterliegenden Datenbankmodells auch um die Schnittstellenbeschreibung zwischen den Services handelt, wurde von einem ER-Modell abgesehen und stattdessen lediglich ein Domainmodell mit Bezügen dargestellt. Im Zuge einer Weiterentwicklung des Prototypen sollten diese Sichten noch einmal differenziert werden.

3.2.3. Implementierung der Frontend-Komponente mittels AngularJS 2 und Typescript

Für die Umsetzung einer geeigneten Frontend-Lösung, muss ebenfalls eine Entwurfsentscheidung bezüglich der Technologiewahl getroffen werden. Solange die clientseitige Anwendung nicht das Qualitätsziel der Skalierbarkeit gefährdet, gilt es bei der Auswahl einer geeigneten Lösung auf Performance und Modularität für die Änderbarkeit und Wiederverwendbarkeit zu achten. Um die Performance zu verbessern sollte daher ein Javascript-Framework zum Einsatz kommen, das eine einfache asynchrone Kommunikation und einen asynchronen Seitenaufbau unterstützt. Es wurde daher auf das noch sehr junge Angular 2 gesetzt, das aktuell als Release Candidate[14] zur Verfügung steht.

Das Problem mit den noch nicht sehr stabilen Funktionen in Angular 2 wurde dafür in Kauf genommen, dass die Performance beim Rendering gegenüber der Vorversion grundlegend verbessert hat [vgl. Fai16]. Hierbei spielt vor allem die Möglichkeit der vereinfachten Verwendung von *Web Worker*[15] eine wichtige Rolle. Diese Web Worker ermöglichen die Umsetzung von Multithreading auf Clientseite [vgl. Seite 1 Gre12]. Damit blockieren bspw. große Tabellenblätter, welche sich unter anderem aus der Ausgabe von großen Artikellisten ergeben können, nicht den JavaScript-Thread für den restlichen Seitenaufbau.

Außerdem setzt Angular 2 unter anderem vollständig auf Typescript auf. Damit geht, im Gegensatz zur typischen Javascript-Entwicklung, eine feste Typisierung einher, die das System damit stabiler machen [vgl. Fai16].

3.2.4. Integrationsstrategie zwischen Microservices und Frontend

Zunächst gilt es festzuhalten, dass clientseitige Javascript-Aufrufe an verschiedene Microservices mit unterschiedlichen Domains gegen die in modernen Browsern hinterlegten „same origin“-Regeln verstößt. Es muss daher für die Integration auf CORS zurückgegriffen werden [vgl. Seite 8 Bar11], solange das in Abschnitt 3.1.3 beschriebene *Gateway API Pattern* nicht zum Einsatz kommt. Jede Anfrage des Prototypen-Clients die einen bspw. um Authentifizierungsdaten erweiterten Header besitzt, beginnt hierbei die Kommunikation mit einer browserspezifischen Preflight-Anfrage. Diese Anfrage checkt mittels der HTTP-Methode OPTIONS, ob der Server die Anfrage mit dem Header zulassen würde und welche HTTP-Methoden zur Verfügung stehen [vgl. Seite 5 Wil+15]. Hierfür besitzt jeder Microservice des Prototypen einen entsprechenden Filter, mittels dessen CORS-Anfragen zugelassen werden.

Wie bereits beschrieben, ist das Zielsystem nicht bekannt und soll auch möglichst generisch gehalten werden. Daher wird auf eine Dockerumgebung aufgesetzt, da mittels dieser ein virtuelles, verteiltes System einfach bereitgestellt werden kann. Docker ist open source und bietet eine virtuelle Plattform, um Applikationen in Container zu verpacken und relativ leichtgewichtig mit allen notwendigen Abhängigkeiten lauffähig

[14] Gilt in einem typischen Entwicklungskonzept als Stufe nach der Beta-Version und vor dem Final Release

[15] Die Spezifikation ist unter https://html.spec.whatwg.org/multipage/workers.html beschrieben.

zu machen. Die Container sind hierbei durch das Nutzen von Oracle VirtualBox oder VMWare Player komplett voneinander isoliert und besitzen ihr eigenes Netzwerk sowie Dateisystem. Es wird daher häufig auch von *Container as a Service (CaaS)* gesprochen [vgl. Seite 39 Voh16]. Da Microservices innerhalb eines Systems, im Rahmen bestimmter Funktionen, miteinander kommunizieren müssen, gilt es die Netzwerkkonfiguration jedes Dockercontainers unter Berücksichtigung des Sicherheitsaspekts anzupassen [vgl. Inc16]. Es wurde eine Integration der Services hauptsächlich auf Clientseite angestrebt. Dies hat mehrere Gründe. Die Wahl von Docker als Zielsystem basiert lediglich auf Annahmen, weshalb diese möglicherweise in der Folge von Weiterentwicklungen nicht die Zielplattform der Endanwendung gilt. Daher wird vermieden im Zuge dieser Arbeit nicht mehr als nötig auf Konfigurationen von Docker einzugehen. Außerdem ist der Prototyp im Backend damit einfacher zu verstehen sowie initial zu starten und zu integrieren. Durch die clientseitige Integration wird allerdings das Frontend komplexer und die Performance ggf. schlechter.

Das in Abbildung 9 dargestellte Diagramm zeigt damit beispielhaft einen im Prototypen umgesetzten Prozess mit einer hauptsächlich clientseitigen Integration. Im Zuge einer Weiterentwicklung zu einem produktiven System ist die Integration der Microservices im Backend allerdings nicht ausgeschlossen. Hierbei ist das Zielsystem ggf. bekannt und dementsprechend konfigurierbar. Dadurch lässt sich der für den Prototypen in kauf genommener Performanceverlust aufheben und entlastet die Komplexität des Storefront-Clients.

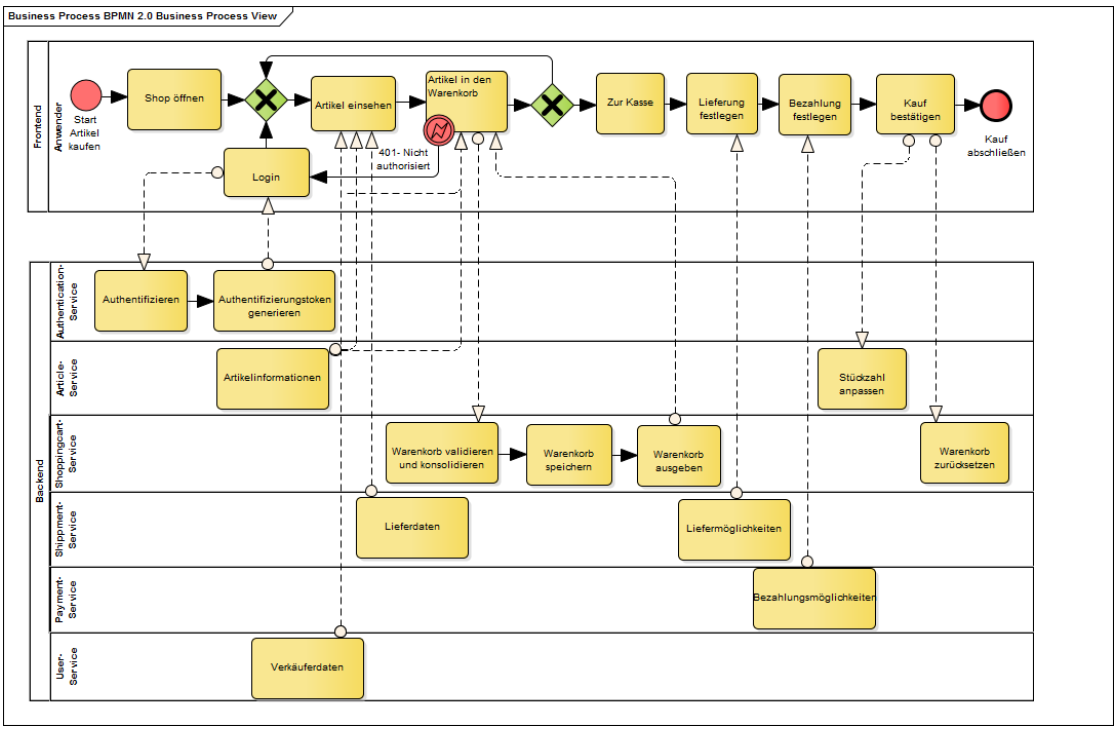

Abb. 9: Businessprozess für das Einkaufen eines Artikels in BPMN.

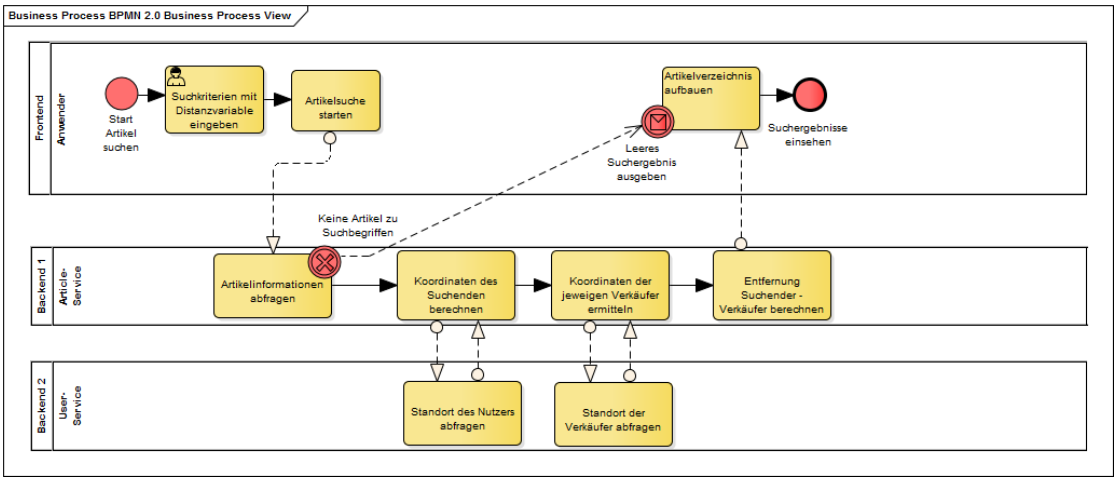

Abb. 10: Businessprozess für das Suchen eines Artikels in BPMN.

Um eine Integration der Microservices auf Serverseite vorzugeben, wurde ein unkritischer Prozess herausgegriffen, der eine beispielhafte Orchestrierung der Dockercontainer aufzeigt. Dieser Prozess und die Kommunikationswege sind in Abbildung 10 durch ein BPMN-Diagramm dargestellt. Der Artikelservice benötigt zur Berechnung der Distanz zwischen Käufer und Verkäufer aller gefunden Artikel die Adressdaten. Der Umweg über den Client, damit dieser die Adressdaten anfragt ist sehr ineffizient und möglicherweise fehleranfälliger. Die Registrierung eines neuen Nutzers sowie das Anlegen eines neuen Artikels sind weitere Prozesse, bei denen ebenfalls eine Orchestrierung der Microservices untereinander zu einer verringerten Komplexität und steigender Performance führen würden. Diese könnten im Zuge eines Folgeprojekts umgesetzt werden.

4. Evaluation des Prototypen zu einem monolithischen Shopsystems

Die zuvor getroffenen Entwurfsentscheidungen gilt es ebenfalls zu evaluieren. Hierbei soll vor allem die Wahl eines Microservice Architektur-Pattern in den Vergleich zu einem monolithischen System gestellt werden. Die in Abschnitt 2.1.3 hervorgehobenen Vorteile und Herausforderungen bilden die Grundlage der Entwurfsentscheidung zur Wahl des vorzuziehenden Patterns. Außerdem liefert es bereits erste Aspekte für die Evaluierung. Der in der nachfolgenden Tabelle 5 dargestellte Vergleich von Microservices und Monolithen greift diese Punkte unter anderem noch einmal auf und zeigt deren Vor- und Nachteile im Bezug auf die zu Beginn in Abschnitt 2.1.2.2 gewählten Qualitätsziele.

Qualitätsziele	**Kriterien**	**Microservicearchitektur**	**Monolithen**
Skalierbarkeit	horizontal / vertikal	Skaliert horizontal (Scale out) - Es kann meist leicht eine lineare Performancesteigerung erzielt werden.	Skaliert meist nur vertikal. (Scale up) Nur unter Umständen und größerem Aufwand horizontal skalierbar.
	Skalierung durch Unabhängigkeit	Systemkomponenten sind unabhängig. Die nicht-Skalierbarkeit einer Komponente behindert nicht die Skalierbarkeit der anderen Systemteile [vgl. Seite 6 Fam15].	Eine nicht-skalierbare Funktion gefährdet die Skalierbarkeit des gesamten Systems [vgl. Seite 6 Fam15].
	funktionsbezogene Skalierung	Es lassen sich einzelne Services und damit Systemteile bzw. bestimmte Funktionen skalieren.[16]	System lässt sich meist nur im gesamten skalieren.
Performance		Nicht so schnell wie Monolithen, jedoch können clientseitige Anfragen parallelisiert und mittels asynchronem Seitenaufbau dem Anwender schneller bearbeitbare Inhalte geliefert werden. Dies macht die Performanceverluste weniger bemerkbar.	Generell schneller, da nicht viele Kommunikationswege über das Netz erfolgen müssen. (Abhängig von Latenzzeiten.)

Die Performance und die Skalierbarkeit gilt es normalerweise anhand von Metriken zu testen. Diese sind allerdings stark individuell und neben den gewählten Architektur auch von Faktoren wie der Testumgebung, der Komplexität des zu testenden Prozesses und der gewählten Technologien für die Implementierung abhängig. Villarnizar und

[16] In dem Shop zur Verfügung gestellte Funktionen wie bspw. Blitzangebote, die eine kurzzeitige, hohe Last erzeugen können, lassen sich in eigenständige Microservices auslagern und zu Events wie dem Black Friday nach Bedarf skalieren. Damit ist ebenfalls die Stabilität des Systems gesichert, da die normale Shop-Funktionalität der erhöhten Last nur teilweise ausgesetzt ist und bei einem Ausfall des Angebot-Services weiterhin die Grundfunktionalitäten bereit gestellt sind.

Qualitätsziele	**Kriterien**	**Microservicearchitektur**	**Monolithen**
Änderbarkeit/ Erweiterbarkeit	Elastizität	Neue Funktionen (Services) lassen sich je nach Anwendungsfall und Systemschnitt häufig unabhängig entwickeln und während des Betriebs einbinden und abschalten [vgl. Seite 6 Fam15].	Erweiterungen müssen direkt auf dem System entwickelt und dieses meist neu gestartet werden [vgl. Seite 6 Fam15].
	Änderungen/ Weiterentwicklung	Lassen sich einfacher dezentral von unterschiedlichen Teams entwickeln.	Wartung und Weiterentwicklung erfolgt meist von einem zentralen Entwicklerteam.
	Technologiewechsel	Grundlegender Technologiewechsel für Erweiterungen oder im Zuge von Wartungsarbeiten jederzeit möglich [vgl. 585 Vil+15].	Technologiewechsel meist nicht möglich [vgl. 585 Vil+15].
Test und Deployment	Übergreifende Integrationstests (automatisiert)	Schwerer zu realisieren, da die Services über mehrere Laufzeitumgebungen und ggf. verteilte Systeme liegen.	Leichter zu realisieren.
Betrieb		Konzepte wie DevOps, bei dem ein Entwicklerteam auch die Wartung übernimmt und nur für diesen Teil des Gesamtsystems verantwortlich ist, lässt sich einfacher umsetzen.	Betrieb meist an einer zentralen Stelle

Tab. 5: Evaluation einer Microservice-Architektur im Vergleich zu einer Monolithen-Architektur anhand des Prototypen.

andere haben ebenfalls eine Gegenüberstellung der beiden Architekturpattern durch Performancetests angestrebt. Das Ergebnis ist ein nur sehr geringer Performanceverlust bei der Verwendung von Microservices [vgl. Seite 588 Vil+15]. Hierbei kam allerdings, neben einer abweichenden Zielplattform, eine andere Architektur basierend auf dem „Gateway API Pattern“ sowie dem Framework Play statt Spring Boot zum Einsatz. Daher ist das Ergebnis nur bedingt im Kontext des Prototypen zu werten. Da vor allem eine mögliche Zielumgebung nicht definiert wurde, keine angemessene Testumgebung im Rahmen der Arbeit bereitgestellt werden konnte und ein ähnliches monolithisches Referenzprojekt zur Gegenüberstellung fehlte, wurde im Rahmen dieser Arbeit von entsprechenden Testvorgehen abgesehen. Das Erreichen der Qualitätsziele kann dennoch positiv bewertet werden, da diese grundsätzlich durch die verwendete Architektur unterstützt werden.

5. Fazit

Hervorgehend aus der Evaluierung kann abschließend festgehalten werden, dass mittels der Microservicestruktur ein wesentlich flexibleres und besser skalierendes System entwickelt werden konnte. Mittels der gewählten Architektur und Frameworks konnten zu Beginn festgelegte Qualitätsziele großteils ohne starke Wechselwirkung eingehalten werden. Hierfür wurde allerdings ein erhöhter Konzeptions- sowie initialer Entwicklungsaufwand in Kauf genommen. Die Entwicklung des Prototypen unter dem „Microservice Architectur Pattern“ forderte dazu einen höheren Aufwand bei der Integration der Services und dem entsprechenden Testen. Dadurch ist der Prototyp allerdings leichter änder- und erweiterbar, womit eine Weiterentwicklung im Zuge von nachfolgenden Projekten unterstützt wird. Der Prototyp stellt letzten Endes ein *Proof of Concept* mit mehreren vertikalen Durchstichen von Front- bis Datenbankebene dar, um eine Auswahl an verschiedenen Funktionalitäten bereit zu stellen. Die Integration der Funktionen zu einem beispielhaften Shopsystem ist hierbei rudimentär umgesetzt, unterstützt allerdings ein schnelles und vereinfachtes Aufsetzen des Prototypen. Eine Anleitung ist hierzu in dem Anhang D beigefügt. Ebenfalls liegt der Arbeit ein, aus den Architekturbeschreibungen hervorgehendes, Referenzprojekt bei. Dieses kann, wie in Anhang E beschrieben, als Maven Archetype genutzt und damit als funktionierender Architektur- und Technologievorschlag für weitere Microservices verwendet werden.

Mittels des *Microservice Architecture Pattern* und der daraus hervorgehenden Modullierung sowie dem Einhalten der Prinzipien nach ROCA wird eine horizontale Skalierbarkeit erreicht. Hierbei können einzelne Systemteile linear und damit optimal skaliert werden, weshalb auch von einem hoch skalierbaren System gesprochen werden kann. Da es sich allerdings um einen Prototypen handelt, gilt es, das System neben der fachlichen Funktionalität auch technisch weiter zu entwickeln. Um eine Weiterentwicklung voranzutreiben, sollte der nachfolgende Abschnitt *Ausblick* beachtet werden.

6. Ausblick

Bevor ein produktiver Einsatz des Shopsystems ermöglicht wird, müssen neben dem Entwickeln neuer Funktionen, auch bereits berücksichtigte Aspekte noch einmal beleuchtet werden. Hierunter fallen vor allem der Sicherheits- sowie Stabilitätsaspekt. Das System muss gegen viele mögliche Angriffe gehärtet werden. Dies bedeutet, dass neben dem eingesetzten Token-Verfahren bspw. auch eine Zwei-Faktoren-Authentifizierung für kritische Login-Bereiche oder Nutzerrollen in Betracht gezogen werden sollte. Außerdem muss das Berechtigungskonzept weiter verfeinert und alle eingehenden Daten auf die fachlichen und technischen Aspekte validiert werden. Ebenso kann das Authentifizierungsverfahren mittels der JWT-Token noch weiter gehärtet werden, so dass bspw. mit der Ausstellung eines neuen Schlüssels die alten als invalide gewertet werden. Die Stabilität sollte außerdem durch weitere Validierungsmassnahmen und Integrationstests sowie einem Update des Frontends auf eine stabilere Angular 2 Version erhöht werden. Angular 2 und einige eingebundene Hilfsmodule im Frontend sollten vorerst nicht produktiv eingesetzt werden, bis diese einen finalen Status erreicht und etwas stabiler sind. Des weiteren wurde eine für den Prototypen optimale Integrationsstrategie der Services gewählt, die für ein produktiv einzusetzendes System möglicherweise nicht in Frage kommen würde. Dies gilt es ebenfalls nachzubessern. Resultierend aus diesen Aspekten und der technischen Schulden beim Prototyping, die sich unter anderem aus der knappen Zeit ergaben, wurde eine strukturierte Problemliste eingeführt. Diese ist vor bzw. zu Beginn einer Weiterentwicklung zu berücksichtigen und befindet sich als Ausschnitt im Anhang 2 und vollständig sowie aktuell in dem öffentlichen Git-Respository als Issue-Liste, welches im Anhang C beschrieben ist.

7. Literatur

[Bar11] A. Barth. *RFC 6454 - The Web Origin Concept*. 2011. URL: https://tools.ietf.org/html/rfc6454 (besucht am 04. 08. 2016).

[DMO04] L. Dadda, M. Macchetti und J. Owen. "The design of a high speed ASIC unit for the hash function SHA-256 (384, 512)". In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*. Bd. 3. Feb. 2004, 70–75 Vol.3. DOI: 10.1109/DATE.2004.1269207.

[Fai16] Y. Fain. *Angular 2 and TypeScript - A High Level Overview*. 2016. URL: https://www.infoq.com/articles/Angular2-TypeScript-High-Level-Overview (besucht am 08. 08. 2016).

[Fam15] Bob Familiar. "Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions". In: Berkeley, CA: Apress, 2015. Kap. IoT and Microservices, S. 133–163. ISBN: 978-1-4842-1275-2. DOI: 10.1007/978-1-4842-1275-2_7. URL: http://dx.doi.org/10.1007/978-1-4842-1275-2_7.

[Fie00] R. Fielding. "Architectural Styles and the Design of Network-based Software Architektures". Diss. University of California, 2000.

[Fil12] U. Fildebrandt. *Software modular bauen: Architektur von langlebigen Softwaresystemen - Grundlagen und Anwendung mit OSGi und Java*. dpunkt.verlag, 2012. ISBN: 9783864911835. URL: https://books.google.de/books?id=m8dNDAAAQBAJ.

[Fow11] M. Fowler. *PolyglotPersistence*. 2011. URL: http://martinfowler.com/bliki/PolyglotPersistence.html (besucht am 08. 08. 2016).

[Gre12] I. Green. *Web Workers: Multithreaded Programs in JavaScript*. O'Reilly Media, 2012. ISBN: 9781449322090. URL: https://books.google.de/books?id=lEdt-AKB3iQC.

[Gut16] F. Gutierrez. "Spring Boot Actuator". In: *Pro Spring Boot*. Berkeley, CA: Apress, 2016, S. 245–281.

[IET15] Internet Engineering Task Force (IETF). *JSON Web Token (JWT)*. 2015. URL: https://tools.ietf.org/html/rfc7519 (besucht am 11. 08. 2016).

[Inc16] Docker Inc. *Understand container communication*. 2016. URL: https://docs.docker.com/engine/userguide/networking/default%5C_network/container-communication/ (besucht am 03. 08. 2016).

[Inn15a] InnoQ. *ROCA - Resource-oriented Client Architecture*. 2015. URL: http://roca-style.org/ (besucht am 13. 06. 2016).

[Inn15b] InnoQ. *ROCA - Resource-oriented Client Architecture*. 2015. URL: http://roca-style.org/faq.html (besucht am 13. 06. 2016).

[PF04] D. Pachico und S. Fujisaka. *Scaling Up and Out: Achieving Widespread Impact through Agricultural Research*. CIAT Economic and Impact Series. Centro Internacional de Agricultura Tropical (CIAT), 2004. ISBN: 9789586940641. URL: https://books.google.de/books?id=S0Qj4zfZfQUC.

[Ric15] C. Richardson. *Title: Building Microservices: Using an API Gateway*. 2015. URL: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/ (besucht am 08. 08. 2016).

[Riz15] U. Rizwan. *Dropwizard vs Spring Boot—A Comparison Matrix*. 2015. URL: https://dzone.com/articles/dropwizard-vs-spring-boot (besucht am 09. 08. 2016).

[SD00] J. Siedersleben und E. Denert. "Wie baut man Informationssysteme?Überlegungen zur Standardarchitektur". In: *Informatik-Spektrum* 23.4 (2000), S. 247–257. ISSN: 1432-122X. DOI: 10.1007/s002870000110. URL: http://dx.doi.org/10.1007/s002870000110.

[SS12] A. Schill und T. Springer. *Verteilte Systeme: Grundlagen und Basistechnologien*. eXamen.press. Springer Berlin Heidelberg, 2012. ISBN: 9783642257964. URL: https://books.google.de/books?id=VBQeBAAAQBAJ.

[Sta15] G. Starke. *Effektive Softwarearchitekturen - Ein praktischer Leitfaden*. 7. Carl Hanser Verlag München, 2015.

[Til11] S. Tilkov. *REST und HTTP - Einsatz der Architektur des Web für Integrationsszenarien*. 2. Aufl. dpunkt.verlag, 2011.

[USC] US-CERT/NIST. *Vulnerability Summary for CVE-2014-3566*. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3566; abgerufen am 16.04.2015.

[Vil+15] M. Villamizar u. a. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *Computing Colombian Conference (10CCC), 2015 10th*. Sep. 2015, S. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.

[Voh16] D. Vohra. *Kubernetes Microservices with Docker*. Apress, 2016. ISBN: 9781484219072. URL: https://books.google.de/books?id=rJUFDAAAQBAJ.

[VRB11] Luis M Vaquero, Luis Rodero-Merino und Rajkumar Buyya. "Dynamically scaling applications in the cloud". In: *ACM SIGCOMM Computer Communication Review* 41.1 (2011), S. 45–52.

[Wil+15] Andreas Wilke u. a. "A RESTful API for Accessing Microbial Community Data for MG-RAST." In: *PLoS Computational Biology* 11.1 (2015). URL: http://dblp.uni-trier.de/db/journals/ploscb/ploscb11.html#WilkeBHBDGMPWGD15.

[WYS16] M. WYSZOMIERSKI. *Mapping Dozer vs MapStruct*. 2016. URL: http://mariusz.wyszomierski.pl/en/mapping-dozer-vs-mapstruct/ (besucht am 08. 08. 2016).

[Zhi15] A. Zhitnitsky. *Java Bootstrap: Dropwizard vs. Spring Boot*. 2015. URL: http://blog.takipi.com/java-bootstrap-dropwizard-vs-spring-boot/ (besucht am 09. 08. 2016).

A. Anhang - Beispielhafte Wahl verschiedener Datenbanklösungen für Microservices im Onlineshop-Kontext

Abb. 11: Mögliche Auswahl an relationalen und NoSQL-Datenbanken für verschiedene Microservices im Onlineshop-Kontext [Fow11].

B. Anhang - Detaillierte Schnittstellenbeschreibung

Article-Service

Attribut	Datentyp	Beschreibung
articleId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Artikel. (Primärschlüssel)
articleTitle	String	Titel des Artikels, der nicht eindeutig ist und eine sehr kurze Artikelbeschreibung darstellt.
articleDescription	String	Detaillierte Beschreibung des Artikels
articleEAN	String	„European Article Number“ hilft bei der Identifizierung des Artikels außerhalb des Shopsystems. Sie ist nicht eindeutig, weshalb diese nicht als Schlüssel funktioniert. (Sollte für ein produktives System ggf. durch die neuere „Global Trade Item Number“ oder „GTIN“ abgelöst oder erweitert werden.)
articlePrice	Double	Der Bruttonpreis für einen Artikel. (Könnte ggf. in einen separaten Preisservice ausgelagert werden, der neben dem Bruttopreis auch Nettopreis, Steuer, Angebotspreis vor Nachlass, Rabatt, usw. hält.)
articleStock	Integer	Angebotener Bestand des Artikels
supplierId	Sequenz-nummer	Die technische Identifikationsnummer des Verkäufers. (Fremdschlüssel)

Tab. 6: Schnittstellenbeschreibung Article-Service.

User-Service

Attribut	Datentyp	Beschreibung
userId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Nutzers. (Fremdschlüssel auf den Authentication-Service)
firstname	String	Vorname des Nutzers.
lastname	String	Nachname des Nutzers.
address	String	Straße, Hausnummer und ggf. Zusatzadresse des Nutzers
postcode	String	Postleitzahl
country	String	Land des Wohnortes
supplierId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Verkäufer. (Primärschlüssel)

Tab. 7: Schnittstellenbeschreibung User-Service.

Shippment-Service		
Attribut	Datentyp	Beschreibung
articleId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Artikel. (Primärschlüssel)
shipmentReady	Boolean	Flag, ob der Artikel generell zum Versand bereit steht.
shippingDays	Integer	Tage, die der Versand benötigt.
shippingMethod	String	Die Art der Lieferung. (Bspw. Selbstabholung, Lieferunternehmen X, Container/Seefracht, ...)

Tab. 8: Schnittstellenbeschreibung Shippment-Service.

Authentication-Service		
Attribut	Datentyp	Beschreibung
userId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Nutzers. (Primärschlüssel)
username	String	Eindeutiger Nutzername (unique), den der Nutzer selbst einmalig anlegt.
password	String	Passwort für den Nutzeraccount.
roles	Collection an Strings	Liste an Rollen, die der Nutzer mit dem Account einnimmt. (Bisher ist lediglich die Rolle „user“ definiert und für alle Zugriffe verwendet.)

Tab. 9: Schnittstellenbeschreibung Authentication-Service.

Payment-Service		
Attribut	Datentyp	Beschreibung
supplierId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Nutzers. (Primärschlüssel)
methods	String	Zahlungsmethoden, die der Verkäufer unterstützt.

Tab. 10: Schnittstellenbeschreibung Payment-Service.

Shoppingcart-Service		
Attribut	Datentyp	Beschreibung
articleId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Nutzers. (Primärschlüssel)
userId	Sequenz-nummer	Eindeutige, technische Identifikationsnummer für den Nutzers. (Primärschlüssel)
quantity	Integer	Anzahl des gleichen Artikels, die der Nutzer in den Shop legt.

Tab. 11: Schnittstellenbeschreibung Shoppingcart-Service.

C. Anhang Software

Die folgenden Software-Komponenten sind im Rahmen der Arbeit implementiert und dieser beigefügt:

1. Der funktionsfähige Prototyp eines skalierbaren Shopsystems.

- Frontendservice - Mittels diesem Service wird die clientseitige Webanwendung bereitgestellt.
- Article-Service - Dieser Service liefert eine Artikelfunktion.
- Shoppingcart-Service - Warenkorbfunktionen.
- User-Service - Hält Stammdaten der Nutzer.
- Shippment-Service - Hält Daten zur Lieferung.
- Authentication-Service - Grundlegende Nutzerinformationen zur Authentifizierung und Autorisierung.
- Payment-Service - Hält Daten zum Bezahlungsprozess.

2. Ein Maven-Archetype-Template, mittels dessen weitere initial lauffähige Microservices erzeugt werden können. Dieses liegt als Referenz in Form eines Maven-Projekts vor und kann jederzeit in ein Archetype umgewandelt werden. Eine Beschreibung für die Generierung des Archetypes liegt in Anhang E vor. Durch die Bereitstellung des Projekts und der Source-Dateien können technische Weiterentwicklungen der Microservices in dieses Projekt einbezogen werden und neue Services stets auf der aktuellen Code-Basis aufbauen.

Diese Services bzw. Projekte liegen der Arbeit bei und sind zusätzlich in einem öffentlich zugänglichen Git-Repository frei unter der MIT-Lizenz mit folgender URI zugänglich:

https://github.com/CodeMax/Scalable-Shopsystem

Eine Liste aller offenen Punkte, Bugs und Verbesserungsvorschläge liegt in Anhang F und ebenfalls im Repository unter der folgenden URI bereit:

https://github.com/CodeMax/Scalable-Shopsystem/issues

D. HowTo: Das Aufsetzen und Betreiben des Prototypen

D.1. Microservices bauen

Bevor die Microservices mittels Docker bereits gestellt werden, macht es Sinn, zuvor diese testweise zu bauen. Hierfür kann folgendes vorbereitet und ausgeführt werden:

1. Clonen des Repositories oder der einzelnen Microservices.
2. Apache Maven in einer Version >3.1 herunterladen, installieren und dem Systempfad hinzufügen.
3. Bauen der Services mittels des Befehls:
   >>mvn clean install

D.2. Services mittels Docker bereitstellen

Für das testweise Aufsetzen des Prototypen, wird eine Docker-Maschine benötigt, da für diese die Konfigurationen vorgenommen wurden.[17] Für das lokale Deployment auf Docker können bspw. die in Listing 1 vorgenommenen Kommandozeilen-Befehle unter Windows eingegeben werden.

Listing 1: Maven und Docker Kommandozeilenbefehle

```
1 // Um Microservices deployen zu koennen, muss zunaechst ein Fat-Jar
      generiert werden. Dieses muss neben dem Service-Code alle verwendeten
      Frameworks enthalten. Dazu muss das Hauptverzeichnis des Services
      angesteuert und der folgende Befehl ausgefuehrt werden:
2
3     > mvn clean package
4
5 // Die fuer den Prototypen verwendeten Microservices besitzen ein
      untergeordnetes Maven-Modul (bez. "'-build"'), in dem das
      deployfaehige Jar erstellt wird. Daher gilt es in das /target-
      Verzeichnes des Untermoduls zu wechseln und dort ueber die
      Dockerkonsole den folgenden Befehl auszufuehren: (<service-name> ist
      der in der Maven POM vorkonfigurierte finale Name des Services)
6
7     > docker build -t <service-name> .
8
9 // Unter Angabe des Service-Namen und einem beliebigen Port kann nun der
      Container gestartet werden.
10
11    >   docker run -p <port>:<port> -t <service-name>
12
13 // Bei einer gewuenschten Verlinkung zwischen zwei Containern ist eine
      Moeglichkeit die Nutzung des Attributes --link. Damit konfiguriert
      Docker automatisch einen abgesicherten Kommunikationstunnel zu dem
      anderen Container auf.
```

[17] Für andere Virtualisierungstools müssen zunächst entsprechende Konfigurationen in dem Prototypen vorgenommen werden.

```
14
15      >   docker run -p <port>:<port> -t --link <name oder id des anderen
            gestarten Containers>:<name oder id des anderen gestarten
            Containers> <service-name>
16
17 // Hinweis: Fuer eine bidirektionale Kommunikation muss der Name fuer den
       ersten zu startenden Container bereits bekannt sein, weshalb fuer
       docker run der Zusatz -name <Name des Containers> hinzugefuegt werden
       muss.
```

Alternativ kann auch ein frei verfügbares Maven-Build-Tool, wie das von Spotify zur Verfügung gestellte „docker-maven-plugin“, verwendet werden. Damit übernimmt Maven direkt bei einem Build-Vorgang das Deployment auf Docker. Hierbei können allerdings Docker-Container meist nicht wie gewünscht automatisiert verwaltet werden, weshalb ein manuelles Vorgehen meist eher das gewünschte Ergebnis liefert. Außerdem ist dieses oder ähnliche Build-Tools meist noch nicht über eine 1.0 Version heraus und verfügen auch meist über keinen professionellen Support, weshalb von einem Einsatz im Enterprise-Bereich möglicherweise abzusehen ist.

Da innerhalb des Storefront-Services, welches das Frontend darstellt, die Services bisher hart Codiert sind, ist zu empfehlen, die jeweiligen IPs und Ports zu verwenden. Hierbei ist für die Docker-Container standardmäßig die folgende Host-IP-Adresse hinterlegt:

>>http://192.168.99.100

Alternativ kann diese auch in der globalen Variable im Storefront-Service abgeändert werden. Die Ports sind hierbei ebenfalls anzupassen oder nach dem folgenden Schema zu konfigurieren:

- 8083 - Article-Service
- 8084 - Shoppingcart-Service
- 8085 - Shipping-Service
- 8086 - Payment-Service
- 8087 - User-Service
- 8088 - Authentication-Service

Der, wie im Ausblick beschriebener Einsatz eines Servicediscovery-Services nimmt diese Konfigurationsschritte großteils ab. Hierfür müssen allerdings die IP-Tables der Container angepasst werden.

D.3. Frontend bereitstellen

Anders als das Backend, ist der Service zur Bereitstellung des Frontends in Angular 2 und damit in Typescript bzw. Javascript umgesetzt. Die Bereitstellung des Services funktioniert daher anders und ist im Folgenden für ein lokales Aufsetzen des Services beschrieben:

1. Clonen der Storefront-Komponente aus dem Repository.
2. Notwendige Frameworks nachladen.
   2.1. NodeJS sowie NPM herunterladen und installieren.
   2.2. Beide Lösungen dem Systempfad hinzufügen.
   2.3. Mit Konsole in das Storefront-Verzeichnis wechseln und folgendes ausführen:
   >>npm install
   Damit sollte ein neuer Ordner „./node_modules/“ angelegt werden und alle in der mitgelieferten packages.json definierten Abhängigkeiten standardmäßig in diesem Verzeichnis abgelegt werden.
3. Projekt kompilieren und lokal starten.
   3.1. Den Taskrunner Gulp herunterladen und installieren.
   3.2. Gulp in den Systempfad hinzufügen.
   3.3. Mit der Konsole in das Storefront-Verzeichnis wechseln und folgendes ausführen:
   >>gulp
   oder
   >>gulp serve-dev
   Damit beginnt Gulp den Buildprozess und startet im Anschluss direkt Browsersync und eine lokale Live-Demo im Browser.

E. HowTo: Generieren und Anwenden des Maven-Archetypes für die Weiterentwicklung

Das Maven-Projekt „Template“ aus dem Repository bzw. der beiliegenden Software gilt als der Referenzservice und besitzt bestimmte Konfigurationen und Namensgebungen, um ein Maven Archetype generieren zu können. Das Generieren erfolgt mittels des Befehls Create-From-Project aus dem Maven-Archetype-Plugin. Alternativ liegt im Root-Verzeuchnis des Projekts eine Batch-Datei bereit, die alle notwenigen Befehle ausführt und auf Wunsch ein passendes Projekt in dem gleichen Workspace erzeugt. Hierfür sind die folgenden, in Listing 2 beschriebenen Befehle zu verwenden.

Listing 2: Maven Kommandozeilenbefehle für die Verwendung des Archetypen

```
1  // Um aus dem Maven-Projekt ein Archetype zu generieren, ist der folgende
       Befehl auszufuehren:
2
3  > mvn archetype:create-from-project
4
5  // Den Archetypen gilt es im Anschluss mittels eines normalen Build-
       Prozesses in eine Jar-Datei umzuwandeln.
6
7  > cd /template-build/target/generated-sources/archetype
8  > mvn install
9
10 // Der Archetype kann, wenn gewuenscht, mit dem folgenden Befehl in den
       lokalen, standardmaessigen Archetype-Katalog hinzugefuegt und im
       Anschluss bspw. direkt in Eclipse fuer das Erstellen eines neuen Maven
       -Projekts verwendet werden.
11
12 >   mvn archetype:crawl
```

F. Auszug einer Liste an Bugs, Verbesserungsvorschlägen und nicht umgesetzter Funktionen

Diese nachfolgende Liste stellt einen Auszug an Bugs, Verbesserungsvorschlägen und nicht umgesetzter Funktionen dar. Diese sollte lediglich beispielhaft gewertet werden, da sie nicht vollständig, aktuell und ausführlich ist. Die ausführliche Liste liegt dem Repository bei.

1. Article-Service: Die Funktion „Artikel-Umkreis-Suche“ besitzt nicht ausreichend Unit- und Integrationstests.
2. In allen Backendservices sollte eine Validierung in der Businessschicht mittels Hibernate vorgenommen werden.
3. Authentication-Service: Zurücksetzen des Passworts bisher nicht möglich.
4. Storefront-Service: An einigen Stellen ist CSS im HTML eingebettet. Dieses sollte in eigene CSS-Dateien ausgelagert werden.
5. Storefront- und Article-Service: URIs und Ports bisher im Code hinterlegt. Diese sollten besser über DNS und ein Service-Register wie Eureka oder Consul aufgelöst werden.
6. Storefront-Service: Noch immer situationsabhängiges Fehlverhalten der Seitenverlinkung im Frontend. Dies könnte mit einem Update auf eine zukünftige Router-Package-Version behoben werden, welche keinen Beta-Status besitzt.
7. Neue Services wie ein Bestellungs-, Email- und ein Preisservice könnten integriert werden, um weitere essentielle Grundfunktionalitäten eines Shops bereit zu stellen.