Konzeptionierung und Implementierung eines Remote-Controllers mithilfe einer Microservice-Architektur zur Remote-basierten Nutzung einer On-Premise Software am Beispiel von Royal Render

Bachelorarbeit

Hochschule Hamm-Lippstadt
Computervisualistik und Design

vorgelegt von

Robin Dürhager
Matrikelnummer: 2150495

Erstprüfer
Prof. Dr. Darius Schippritt

Zweitprüfer
Prof. Stefan Albertz

Drittprüfer
Holger Schönberger

24. März 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

1. Grundlagen der Softwarearchitektur

1.1. Architekturstile

1.1.1. Monolith

Unter den Softwarearchitekturen ist der Monolith ein weit verbreiteter und genutzter Architekturstil. Der Monolith mit seiner Analogie zu einem einheitlichen, massiven Stein, birgt gewisse Vor- und Nachteile gegenüber anderen Architekturen. Diese, als auch der grundlegende Aufbau einer solchen Architektur, werden im Folgenden erläutert.

Ein monolithisches System vereint jegliche verwendete Technologien in einem einzigen Prozess (Newman, 2019; Gallipeau & Kudrle, 2018, S. 21). Laut Takai entsteht dadurch „[...] ein einschichtiges, untrennbares und technologisch homogenes System, das verschiedene Services in sich vereint“ (Takai, 2017, S. 17). Newman unterscheidet zwischen vier verschiedenen Arten des Monolithen:

- Einzelprozess-Monolith
- Modularer Monolith
- Verteilter Monolith
- Drittanbieter-Black-Box-Systeme

Einzelprozess-Monolith

Der *Einzelprozess-Monolith* ist die am häufigsten auftretende Form der monolithischen Softwarearchitektur. Wie vorher schon durch Takai beschrieben, besitzt dieser die Eigenschaft verschiedene Technologien in sich und unter einem Prozess zu vereinen. Somit bietet der Einzelprozess-Monolith den Ursprung des monolithischen Architekturstils. Wie weiter oben aufgelistet, entwickeln sich aus diesem weitere Varianten, die für die Entwicklung eines Softwareproduktes genutzt werden können.

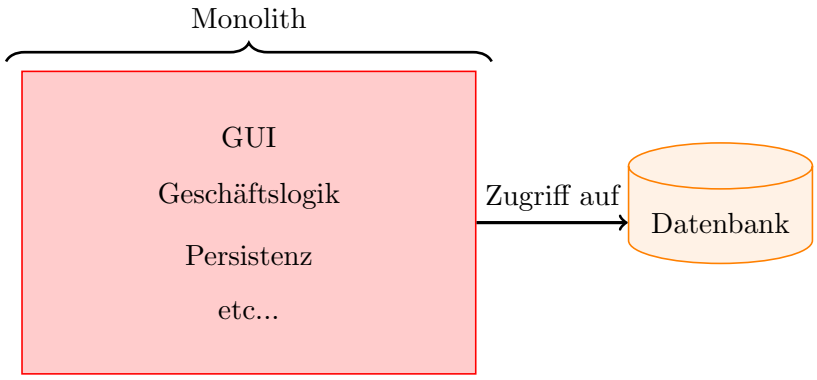


Abbildung 1.1.: Skizze eines Einzelprozess-Monolithen (Newman, 2019)

Wenn zukünftig in dieser Arbeit der Monolith benannt wird, so ist damit der Einzelprozess-Monolith gemeint, da dieser zu der bekanntesten Art der monolithischen Softwarearchitektur gehört. Eine derartige Architektur wird in Abbildung 1.1 skizziert.

Modularer Monolith

Eine Variation des Einzelprozess-Monolithen ist ein *modularer Monolith*. Anders als bei einem gewöhnlichen Monolithen, wird dieser zur Entwicklungszeit in Module untergliedert, welche unabhängig voneinander weiterentwickelt werden können. Zum Distributieren der entstehenden Software müssen diese Module jedoch wieder in einen einzigen Prozess zusammengefügt werden. Solange ein modularer Monolith klar getrennte Module aufweist, kann dies für ein Unternehmen gut funktionieren. Es ist hierbei zu beachten, dass die Datenbank, genau wie der modulare Monolith selbst, in dessen Module dekomponiert werden soll. Laut Newman wird dies allerdings nur selten realisiert. Abbildung 1.2 skizziert den grundlegenden Aufbau eines modularen Monolithen (Newman, 2019).

Als Beispiel fungiert hierfür das Unternehmen *Shopify*, welches einen gewöhnlichen Monolithen in einen modularen Monolithen umstrukturieren konnte. Ersteres führte nach Wachstum von Software und Unternehmen zu Problemen in den Bereichen Wartbarkeit, Testbarkeit und Erweiterbarkeit. Der Übergang von einem Monolithen zu einer Microservice-Architektur wäre für Shopify zu aufwändig gewesen, da dies eine Neuentwicklung des Produktes erfordert hätte (Westeinde, 2019).

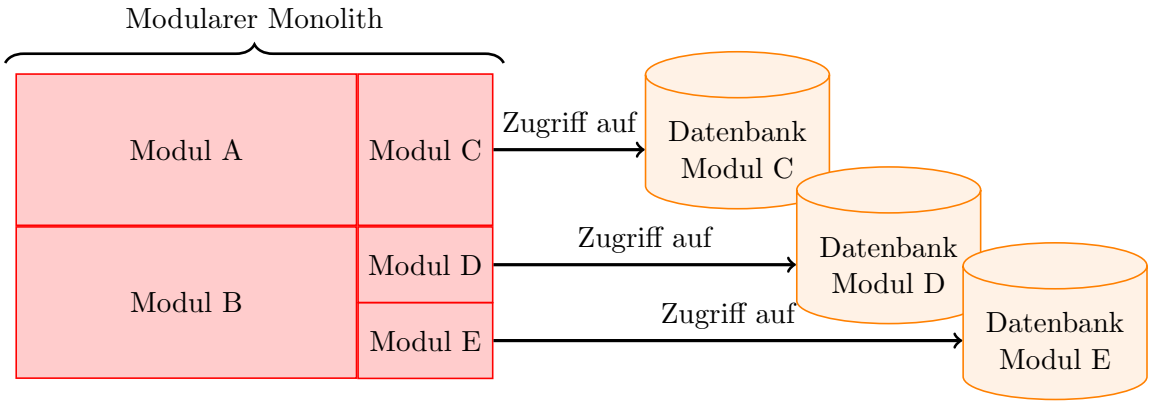


Abbildung 1.2.: Skizze eines modularen Monolithen (Newman, 2019)

Verteilter Monolith

Der verteilte Monolith ist weniger ein Architekturstil, als vielmehr ein unerwünschtes Artefakt, welches aus nicht eingehaltenen, spezifischen Entwurfsprinzipien der Service-Oriented Architecture (SOA) entsteht. Die Kernessenz von SOA besteht darin verschiedene Services zu gestalten, welche jeweils als eigene Prozesse unabhängig voneinander existieren und agieren können. SOA wird in Unterabschnitt 1.1.2 weiter erläutert (Newman, 2019; Erl, 2005).

Verteilte Monolithen entstehen oftmals in Umgebungen, in welchen zu wenig Zeit und Fokus in die Abstraktion von *Services* als auch deren Kohäsion von Geschäftslogik investiert wurde. Dadurch entstehen mehrere Services mit verwischten *Servicegrenzen*, wodurch eine stark gekoppelte Architektur entsteht, in welcher eine Veränderung in einem Service das ganze System beeinflussen kann. Im Idealfall sollte nur der veränderte Service von dessen Modifizierung beeinflusst werden. Beispielsweise sollte ein SOA System, bestehend aus einem *User-Service*, *Renderjob-Service* und *Download-Service*, nicht zusammenbrechen sobald der User-Service modifiziert wurde. Der modifizierte Service selbst kann abstürzen, jedoch sollte in einem solchen Fall der Rest des Systems, bestehend aus dem Renderjob-Service und dem Download-Service, fortbestehen können. Da ein Entwurfsprinzip von SOA die Partitionierung von Ressourcen und somit die lose Kopplung von Komponenten eines Systems ist, kann ein verteilter Monolith daher nicht die Versprechen von SOA einhalten (Newman, 2019; Richardson, 2018).

Drittanbieter-Black-Box-Systeme

Als letzte Art des Monolithen gelten laut Newman Drittanbieter-Black-Box-Systeme. Diese sind extern entwickelte Services. Solche können sowohl als Open Source Systeme in der eigenen Infrastruktur, als auch als Software as a Service (SaaS) Produkte über eine *API* oder ein *SDK* eingesetzt werden. Beispiele für solche Systeme wären der Objektspeicher *MinIO* und Google's Backend as a Service (BaaS), *Firebase*. MinIO gewährt dabei als Open Source Produkt Einblick in dessen Quellcode welchen man modifizieren kann, auch wenn dies mit einem gewissen Aufwand verbunden ist. Somit läuft man gerade bei Google's Firebase Gefahr, dass dieses BaaS Produkt ein Monolith ist, der eine typische Black-Box darstellt. Dort wird über eine API verdeutlicht, welche Funktionalität die Black-Box bereitstellt, jedoch nicht deren Art und Weise um die Funktionalität zu erfüllen. Dies ist besonders dann ein Problem, wenn für die entwickelte Software bestimmte Konditionen herrschen. Ein Beispiel dafür wäre ein Wert, der in einem bestimmten Typ zurückgegeben werden muss. Die externen Services MinIO und Firebase werden für diese Arbeit genutzt und dementsprechend in dem **??** weiter erläutert (Newman, 2019).

Vor- und Nachteile von Monolithen

Monolithen werden oftmals als problematisch eingestuft, obwohl diese Art der Softwarearchitektur ein valider Stil zum Entwickeln von Software ist. Vorteile von Monolithen finden sich durch die zentralisierte Codebasis in der Reduzierung der Komplexität des DevOps-Bereichs wieder.

Eine monolithische Java Webapplikation kann zum Beispiel mithilfe einer einzelnen Web Archive (WAR) Datei auf einem Server installiert werden. Die Applikation benötigt also nur einen Kompilierungs- und Installationsprozess. Im Falle eines verteilten Systems müssen allerdings, wie in Unterabschnitt 1.1.2 und Unterabschnitt 1.1.3 beschrieben, mehrere solcher Kompilierungen und Installationen durchgeführt werden, da jeder Service als eigenständiges System zu betrachten ist. Ein Monolith kann auch zu simpleren Workflows für Entwickler führen. Da der ganze Code für einen Monolithen in einem Prozess zu finden ist, können auftretende Fehler in verschiedenen Teilbereichen der Software in der selben Codebasis behoben werden, während sich der Kompiliervorgang deswegen nicht ändern muss. So lassen sich signifikante Änderungen an einer monolithischen Software effizient vornehmen. Verteilte Systeme hingegen benötigen für jeden Service einen individuell angepassten Kompiliervorgang, da diese einen eigenen Technologie-Stack besitzen können. Der Monolith beschreibt ebenfalls einen klaren Weg zum Testen von Software. So kann eine Monolithische Software mithilfe des End-To-End (E2E) Verfahrens in jedem ihrer

Teilbereiche getestet werden. Da sich diverse Teile der Software hier in einem Prozess wiederfinden, kann zum E2E Testen immer sofort die komplette Software getestet werden. Ein weiterer Vorteil liegt bei der Simplizität der Skalierung eines Monolithen vor. Da ein Monolith nur aus einem Prozess besteht, kann auch nur dieser als ein solcher skaliert werden. Deshalb können bei monolithischen Webapplikationen mit Problemen bei der Performance mehrere Instanzen dieser installiert werden. Ein Load Balancer könnte zwischen diesen dann einkommende Hypertext Transfer Protocol (HTTP)-Anfragen verteilen. Diese Vorteile sind besonders für kleinere Softwareprojekte gegeben (Newman, 2019; Richardson, 2018; Namiot & Sneps-Sneppe, 2014, S. 24).

Auch wenn kleinere Monolithen eine geringe Komplexität für einzelne Entwickler oder kleinere Entwicklerteams bergen, so kann diese proportional zur Größe des Monolithen wachsen. Somit besitzen größere Monolithen wiederum einige Nachteile, welche im nächsten Abschnitt aufgezählt werden.

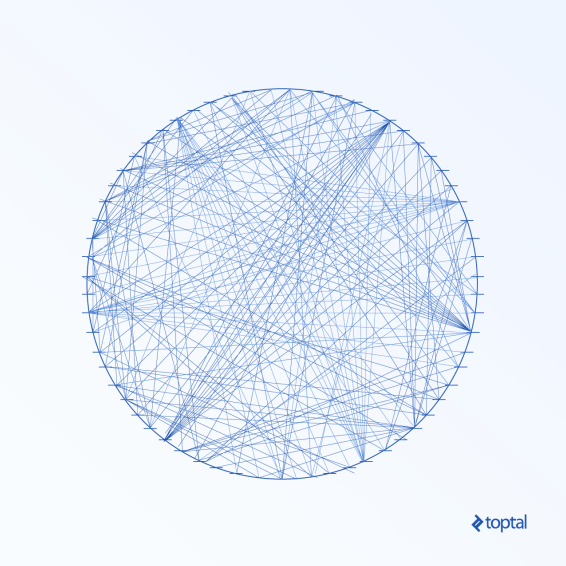


Abbildung 1.3.: Darstellung der grenzenlosen Kommunikation der Komponenten eines *Big Ball of Mud* (Gadzinowski, 2017)

Bei größeren Monolithen spricht man umgangssprachlich von einem *Big Ball of Mud* (Takai, 2017, S. 17). Ein solches System besitzt keine inneren Grenzen, sodass weit entfernte Komponenten des Systems auf direktem Weg Informationen teilen. Eine solche Kommunikation ist in Abbildung 1.3 dargestellt. Dies geht laut Foote und Yoder so weit, dass später wichtige Informationen im globalen Umfang oder sogar dupliziert im Code vorhanden sind (Foote & Yoder, 1997). Jene Entwickler eines Monolithen müssen einen Großteil der Codebasis verstehen, damit an dieser Änderungen vorgenommen werden können. Die

grenzenlose Kommunikation der Systemkomponenten bei einem großen Monolithen jedoch erschwert ein solches Vorhaben. Für neue Entwickler wird es zunehmend schwieriger sich in das System einzuarbeiten, was ein immer weniger produktives Unternehmen zur Folge hat. Das Verwischen dieser Systemgrenzen führt ebenfalls dazu, dass Entwickler in verschiedenen Teilbereichen der Software arbeiten müssen. Dies führt oft zu Unklarheiten bei den Entwicklern bezüglich der Zugehörigkeit und Zuständigkeit des geschriebenen Codes. Auch wenn ein modularer Monolith seine Systembereiche voneinander abtrennt, kann keiner dieser Bereiche unabhängig voneinander skaliert werden. Nimmt man sich wieder das Beispiel des verteilten Monolithen mit den Komponenten *User-Service*, *Renderjob-Service* und *Download-Service*, so könnte man hier jenen Service, der droht überstrapaziert zu werden, mit einer weiteren Instanz ausstatten. Im Falle des Monolithen ist man allerdings durch die starke Kopplung der Komponenten gezwungen eine weitere Instanz der kompletten Software zu starten. Abbildung 1.4 stellt die Art der Skalierung für beide Architekturstile dar. Dabei ist zu beachten, dass der User-Service, Renderjob-Service und Download-Service ein verteiltes System komponieren, während der Monolith diese Services stark gekoppelt in sich vereint und deshalb nicht in seinen Komponenten dargestellt wird. Die starke Kopplung eines Monolithen wird diesem im Falle eines Updates zum Verhängnis, denn für jedes Update muss die komplette Applikation neu veröffentlicht werden. Mit einem wachsenden Monolithen wird es schwieriger für das Entwicklerteam die Entwicklungsframeworks zu ändern, weshalb ein monolithisches Projekt zunehmend unflexibel wird (Richardson, 2018; Namiot & Sneps-Sneppe, 2014, S. 24; Gallipeau & Kudrle, 2018, S. 21–22).

1.1.2. SOA

Im Jahr 2000 kam der Architekturstil der Service-Oriented Architecture auf und bietet eine Alternative zur monolithischen Architektur. Dieser Stil „[...] wurde durch das Platzen der Dotcom-Blase in 2001 beflügelt, als man feststellte, dass die Serviceorientierung Marktvorteile bietet [...]“ (Takai, 2017, S. 12). SOA gilt als ein Vorreiter von Microservices, weshalb sich zwischen diesen beiden Stilen auch einige Ähnlichkeiten feststellen lassen.

Im Gegensatz zum Monolithen achtet man bei diesem Stil darauf mehrere Systeme zu entwickeln, welche isoliert voneinander existieren und miteinander kommunizieren können. Dabei stellt ein Service die primäre Quelle der Geschäftslogik dar. Die Kommunikation der Services findet nach Regeln eines Vertrages statt, wobei es keine Rolle spielt, welche Technologie die kommunizierenden Services benutzen, um ihre Anwendungsbereiche zu erfüllen (Takai, 2017, S. 12; Erl, 2005; Newman, 2019; Gallipeau & Kudrle, 2018, S. 22).

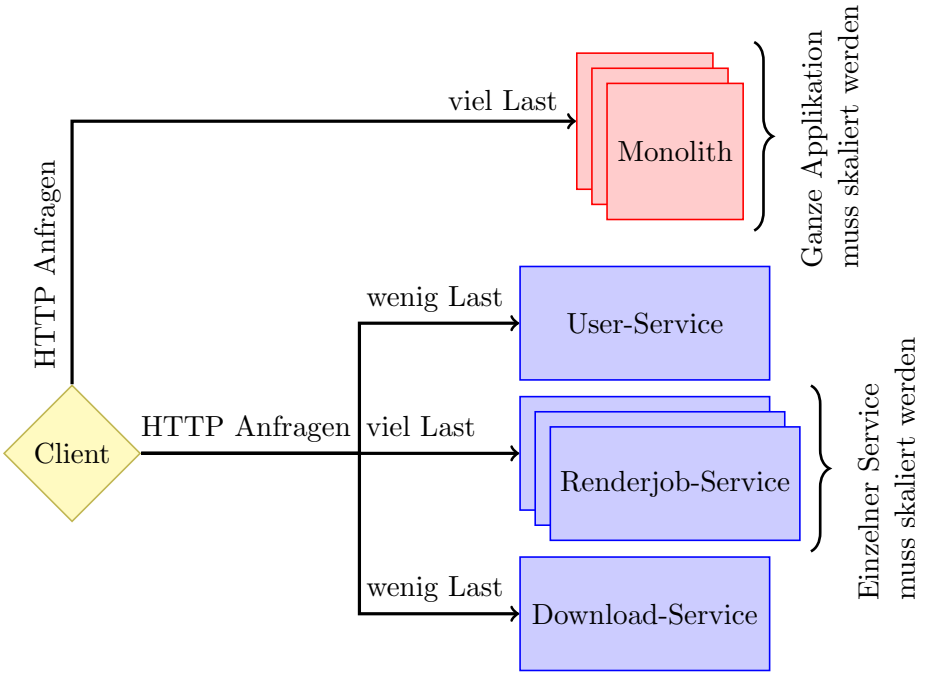


Abbildung 1.4.: Gegenüberstellung der Skalierungsform eines Monolithen und eines verteilten Systems

Laut Takai erwuchs SOA aus zwei Strömungen:

- Objektorientierte Analyse und Design
- Webservices

Ersteres schulte Softwarearchitekten ihre Systeme so zu entwerfen, damit diese erweiterbar, wiederverwendbar, flexibel und robust sind, um ihre Geschäftsziele genau abzudecken. Letzteres leitete die Funktionalität ein, dass Services mithilfe von genormten Kommunikationsprotokollen wie HTTP miteinander kommunizieren können (Takai, 2017, S. 12).

Laut Erl gelten für eine SOA acht Entwurfsprinzipien:

- **Servicewiederverwendbarkeit**: Ein Service sollte eine potenzielle Wiederverwendbarkeit mit sich führen. Ziel ist es hierbei einen Servicekatalog zu entwickeln, in welchem Services vorhanden sind, die von verschiedenen Akteuren genutzt werden können.

- **Servicevertrag**: Mithilfe eines Vertrages kann ein Service darstellen, welche Methoden seine API zur Verfügung stellt und spezifizieren, welche Art von Eingabe- und Ausgabematerial unterstützt wird. So können auch Regeln und Charakteristika des Services selbst und dessen Operationen erläutert werden. So weiß ein anfragender Service, was er von einem angefragten Service mit welcher Eingabe erhält.
- **Lose Kopplung**: Ein Service sollte beim Anfragen eines anderen Services von diesem entkoppelt bleiben. Dies wird durch Serviceverträge erreicht, da damit der anfragende Service nur mit stark eingeschränkten Parametern mit dem angefragten Service kommunizieren kann. Eine Kopplung der Services ist hierbei allerdings nicht komplett zu vermeiden.
- **Serviceabstraktion**: Services werden bei der SOA als Black-Box-Services entwickelt, welche mithilfe des Servicevertrages nur ihre nötigsten Informationen veröffentlichen. Dabei spielt die Größe der darunterliegenden Infrastruktur keine Rolle.
- **Service Composability**: Services sollten so gestaltet werden, dass sie effektiv von anderen Services konsumiert werden können. Eine Komposition aus verschiedenen Services kann wiederum eine komplexe Geschäftsanwendung widerspiegeln.
- **Serviceautonomie**: In der Servicegrenze sollte jeder Service seine Operationen handlungsfrei ausüben können.
- **Servicezustandslosigkeit**: Bei einem zustandslosen Service „muss ein Akteur nichts über seine Historie wissen, um eine Anfrage platzieren zu können“ (Takai, 2017, S. 13). Bleibt ein Service so schnell und lange wie möglich zustandslos, kann dieser die Anfragen weiterer Akteure schneller behandeln.
- **Service Discoverability**: Mithilfe einer *Service-Discovery* können Services dynamisch und automatisch von anderen Services gefunden werden. Mit einem solchen Prinzip lassen sich Services besser skalieren. Service-Discovery wird in **??** weiter beschrieben.

(Takai, 2017, S. 13–14; Erl, 2005)

Diese Entwurfsprinzipien nach Erl sorgen für ein System in welchem die Komponenten unabhängig voneinander agieren können. Ebenfalls werden Services wiederverwendbar und lassen sich in verschiedenen Geschäftsanwendungen nutzen. Dabei können diese über festgelegte Kommunikationsprotokolle miteinander kommunizieren und Daten versenden,

welche dann von weiteren Services konsumiert werden können. Viele dieser Entwurfsprinzipien treffen auch auf Microservices zu. Allerdings konnte SOA nicht von Beginn an in der IT-Branche florieren (Takai, 2017, S. 14–15; Erl, 2005; Gallipeau & Kudrle, 2018, S. 22).

Laut Takai machte SOA viele Versprechen, die zu jener Zeit nicht eingehalten werden konnten. Unter anderem fehlte die Technologie, um eine solche Architektur nachvollziehbar umsetzen zu können. Jeder Service braucht seine eigene Laufzeitumgebung, was in der Zeit ohne Virtualisierung impliziert, dass für jeden Service ein Server eingekauft werden musste. Ebenfalls gab es nun Services, die von allen konsumiert wurden statt von nur einer Abteilung. Darunter litten die Performance und die Skalierbarkeit der Services, was zu einer Verlangsamung dieser führte. Takai stellt allerdings besonders heraus, dass der Fokus der Unternehmen darauf saß, schwergewichtige Standards wie das Simple Object Access Protocol (SOAP) zu etablieren, wobei der geschäftliche Nutzen von SOA unerforscht blieb. Diese Lücke zwischen IT und Geschäft lässt sich allerdings durch Domain-Driven-Design (DDD) bei dem Entwurf eines verteilten Systems schließen (Takai, 2017, S. 16).

1.1.3. Microservices

Nach dem Aufstieg und Verfall von SOA fiel bei einem Workshop von Softwarearchitekten in der Nähe von Wien im Jahr 2011 das erste Mal der Begriff des *Microservice*. Im Mai 2012 entschied dieselbe Gruppe von Softwarearchitekten, dass der Terminus des Microservice der passendste Ausdruck für die in diesem Unterabschnitt erklärte Softwarearchitektur sei. Microservices ist zum Zeitpunkt dieser Arbeit ein noch junges Themengebiet, welches allerdings in kurzer Zeit viel Aufmerksamkeit bekommen hat (Fowler & Lewis, 2014).

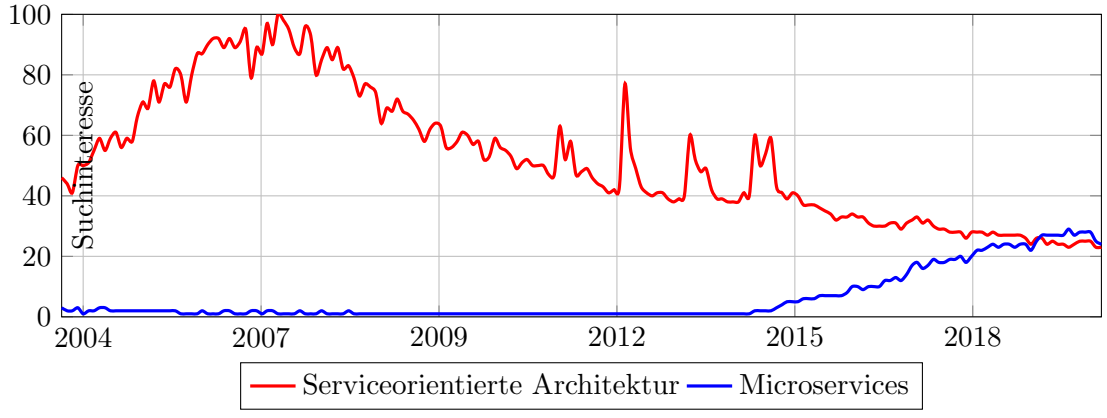


Abbildung 1.5.: Suchinteresse an den Architekturstilen Microservices und Serviceorientierte Architektur ab 2004 im weltweiten Vergleich (Google LLC, 2020)

Wie Abbildung 1.5 zeigt, erhöhte sich die Suchanfrage in Google nach SOA stetig bis zu einem Hochpunkt im Jahr 2007. Wie in Unterabschnitt 1.1.2 bereits vermerkt, lässt sich vermuten, dass das Platzen der Dotcom-Blase das Interesse an SOA positiv beeinflusste. Ab diesem Zeitpunkt verlor SOA kontinuierlich an Aufmerksamkeit, mit Ausnahme von vier lokalen Hochpunkten jeweils im September der Jahre 2011 bis 2014. Obwohl Microservices bis 2014 kaum Aufmerksamkeit bekamen, überholte letztenendes die Suchanfrage nach Microservices die von SOA im Jahr 2019 und macht somit diese zu einem aktuellen Diskussionsthema.

Oftmals werden Microservices als feinkörniges SOA beschrieben, da diese sich in ihrer Grundstruktur von SOA nur geringfügig unterscheiden. Aus diesem Grund wird eine Microservice-Architektur als eine leichtgewichtige Untermenge des SOA Architekturstils angesehen. Auch wenn SOA und Microservices viele Gemeinsamkeiten aufweisen, unterscheiden sich diese doch in einigen wenigen jedoch bedeutsamen Punkten (Takai, 2017, S. 20; Villamizar et al., 2015, S. 584).

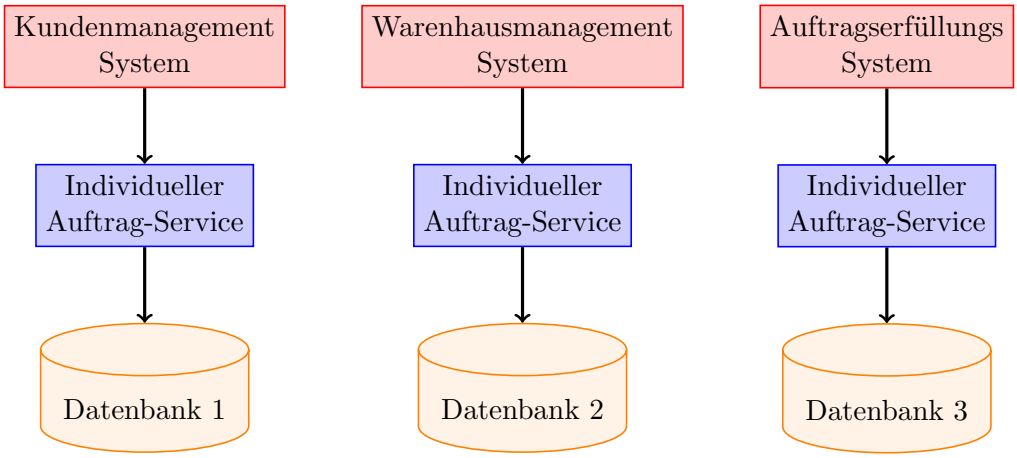


Abbildung 1.6.: Verschiedene Unternehmenssysteme, die jeweils ihre eigene Datenbank und Implementierung eines Auftrag-Service besitzen (Richards, 2016)

SOA benutzt einen *share-as-much-as-possible* Grundsatz, während sich Microservices auf einen *share-as-little-as-possible* Stil beziehen. Abbildung 1.6 beschreibt eine Unternehmenssoftware, die aus den drei Systemen *Kundenmanagement*, *Warenhausmanagement* und *Auftragserfüllung* besteht. Jedes dieser individuellen Systeme besitzt einen eigenen *Auftrag-Service*, da Aufträge je nach System unterschiedlich prozessiert und in der eigenen Datenbank abgespeichert werden müssen. Die Systeme können somit autark arbeiten. Die gleiche Benennung der Auftrag-Services in den verschiedenen Systemen lässt allerdings auf repetitiven Code schließen.

Das Don't Repeat Yourself (DRY) Prinzip besagt jedoch, dass Redundanzen weitestgehend reduziert werden sollten. Dieses Problem soll nach SOA durch eine geteilte Servicekomponente mithilfe von kombinierten Datenbanken, wie sie in Abbildung 1.7 dargestellt ist, gelöst werden (Richards, 2016; Thomas & Hunt, 2019).

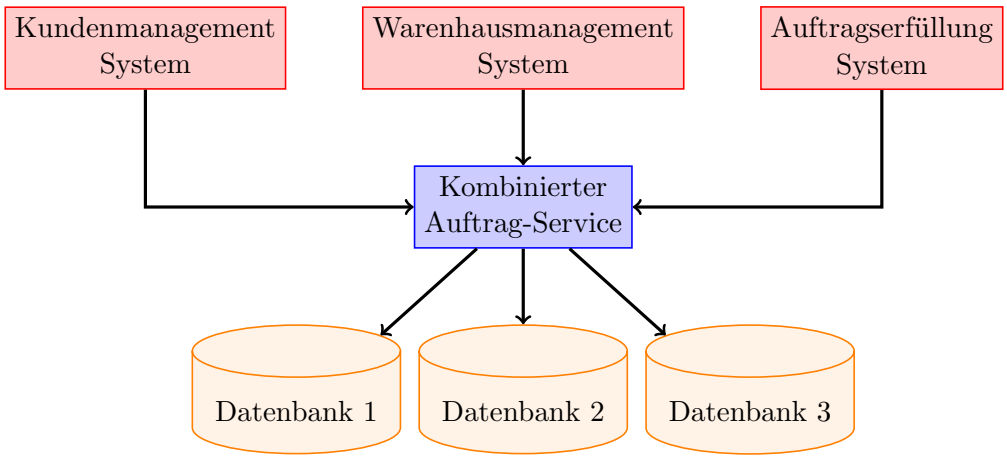


Abbildung 1.7.: Verschiedene Unternehmenssysteme, die eine Servicekomponente teilen, welche alle gebrauchten Datenbanken kombiniert (Richards, 2016)

Durch die Kombinierung der Datenbanken der jeweiligen Systeme wird der Auftrag-Service gezwungen mehrere Informationen von *Kundenmanagement*, *Warenhausmanagement* und *Auftragserfüllung* zu besitzen. Der Service weiß, welche Daten in welcher Datenbank vorhanden sind, abgespeichert, gelöscht und aktualisiert werden müssen, während gleichzeitig der Service alle drei Datenbanken miteinander in Synchronisation halten muss. Ergebnis ist dabei eine starke Kopplung des Auftrag-Service mit allen drei Unternehmenssystemen. Obwohl ein Service in der SOA beliebig viele Aufgaben übernehmen darf, verstößt eine Kopplung des Service mit den Unternehmenssystemen gegen das Entwurfsprinzip der losen Kopplung von SOA, welche in Unterabschnitt 1.1.2 erläutert wurde. Die Software droht damit sich, wie in Unterunterabschnitt 1.1.1 beschrieben, zu einem verteilten Monolithen zu entwickeln (Takai, 2017, S. 20; Richards, 2016).

Um eine solche Kopplung bei Microservices zu vermeiden, werden diese mithilfe des *Single-Responsibility-Prinzips*, ein Prinzip der SOLID-Prinzipien, entworfen. Takai beschreibt das Prinzip wie folgt: „Das Prinzip besagt, dass jede Klasse nur eine einzige Aufgabe haben sollte und sich auch nur aus diesem Grund verändern darf. Diese Aufgabe soll die Klasse kapseln und damit gleichzeitig eine hohe Kohäsion erzeugen“ (Takai, 2017, S. 18).

Wie der Name *Microservice* aussagt, beherrscht dieser, im Gegensatz zu einem Service der SOA, nur einen kleinen Teilbereich der geschäftlichen Funktionen. Diese werden von dem Microservice allerdings sehr gut ausgeführt. Eine solche Aufgabe wäre in dem vorherigen Beispiel das Speichern und Wiederfinden von Aufträgen mittels einer Datenbank. Zu der Größe eines Microservice wachsen antiproportional dessen Vor- und Nachteile. Kleinere Microservices bedeuten mehr Microservices, die einen Teilbereich des Geschäfts genauer abdecken können. Jedoch bedeutet dies auch eine höhere Komplexität im DevOps-Bereich, da alle Microservices auch miteinander agieren können müssen. Des Weiteren wird ein Microservice sowohl in seiner Codebasis, als auch architektonisch von anderen Microservices abgekapselt. Da jeder Microservice seine eigene Laufzeitumgebung besitzt, kann dieser als eigenes System angesehen werden. Somit kann einem Microservice zur Versionskontrolle ein eigenes *Repository* zur Verfügung gestellt werden, in welchem dann servicespezifische Kompilier- und Installations-Scripts ausgeführt werden können. Der Kompilier- und Installationsprozess ist somit individuell pro Microservice anpassbar. Gleichzeitig ist jeder Microservice für die Speicherung seiner Daten selbst verantwortlich, was im Umkehrschluss bedeutet, dass ein Microservice entweder seine eigene oder keine Datenbank besitzen sollte. Benutzt man hier beispielsweise eine verteilte Datenbank, so läuft man Gefahr einen verteilten Monolithen zu entwickeln, da sich hier Service- und Kontextgrenze (engl. *Bounded Context*) der Services vermischen können (Gallipeau & Kudrle, 2018, S. 22–23; Fowler & Lewis, 2014).

Die Kontextgrenze wird im Entwurf von Microservices benutzt, um zu ermitteln welche Komponenten und Daten eines Service gekoppelt werden können. Newman behauptet, dass sich eine Servicegrenze an einer Geschäfts- oder auch einer Kontextgrenze orientieren sollte, damit es offensichtlich erscheint, in welchem Teil der Servicekomposition welcher Code existiert. Fowler ergänzt hier, dass man wegen Conway's Law anhand von crossfunktionalen Teams, statt mit typischen funktionalen Teams, Systeme entwerfen sollte. Melvin Edward Conway behauptet nämlich, dass die Softwarearchitektur die entwerfende Organisation selbst widerspiegelt. Fowler redet hier von einem *Conway-Manöver*, also: „Die Veränderung eines Entwicklungsteams und der Architektur eines Systems, um sie besser mit der Zielorganisation in Einklang zu bringen [...]“ (Takai, 2017, S. 114). Eine Kontextgrenze ist ein Teil von DDD und wird in Abschnitt 1.3 behandelt. Mithilfe des Representational State Transfer (REST) Schnittstellenmodells kann ein Anfragender- bzw. *Upstream-Microservice* einen Angefragten- bzw. *Downstream-Microservice* nicht direkt beeinflussen, sondern nur über dessen API nutzen. Die Kommunikationsstile REST und GraphQL werden in **??** kurz erläutert. Weitere Entwurfsmuster von Microservices werden in **??** behandelt (Takai, 2017, S. 18–20; Conway, 1968, S. 31; Newman, 2015; Fowler & Lewis, 2014).

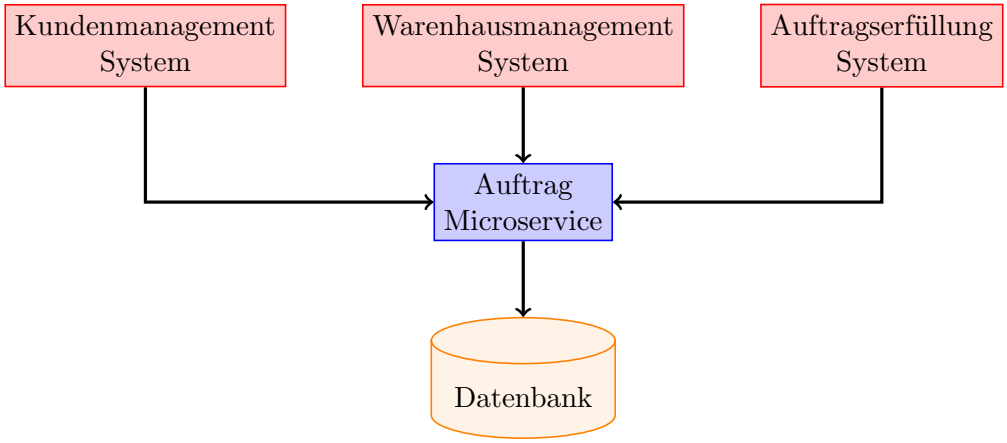


Abbildung 1.8.: Verschiedene Unternehmenssysteme, die einen Microservice teilen, welcher eine Datenbank besitzt und mithilfe von REST eine bestimmte Datenform von einkommenden Anfragen erwartet

Abbildung 1.8 stellt einen solchen entkoppelten Microservice dar. In dieser Abbildung nutzen die Unternehmenssysteme zwar weiterhin den Auftrag-Service, jedoch besitzt dieser eine einzige Datenbank, welche auf den Service selbst abgestimmt ist. Mittels eines durch REST spezifizierten Servicevertrages sind die Unternehmenssysteme gezwungen ihre Daten dem Microservice in einer bestimmten Form zu übermitteln. Ergebnis davon ist, dass der Microservice autark von den Unternehmenssystemen agieren kann. Der Service braucht somit kein spezielles Wissen über das Geschäft und dessen Systeme, sondern kann innerhalb seiner eigenen Kontextgrenze existieren.

Abbildung 1.9 hingegen beschreibt eine typische Microservice-Architektur, wie sie in einem *E-Commerce System* zu finden ist. In dieser Abbildung hat jeder Service eine eigene Datenbank und stellt über REST eine API bereit, welche durch ein *API Gateway* oder eine *WebApp* benutzt werden kann. Durch das API Gateway kann die mobile App ihre Benutzerschnittstelle mit Daten füllen und mit dem Microservice System interagieren. Die *Storefront WebApp* ist allerdings ein Teil des Microservice Systems und stellt eine eigene Benutzerschnittstelle bereit, welche durch einen Browser genutzt werden kann. Dadurch muss die Storefront WebApp nicht zwingend das API Gateway nutzen.

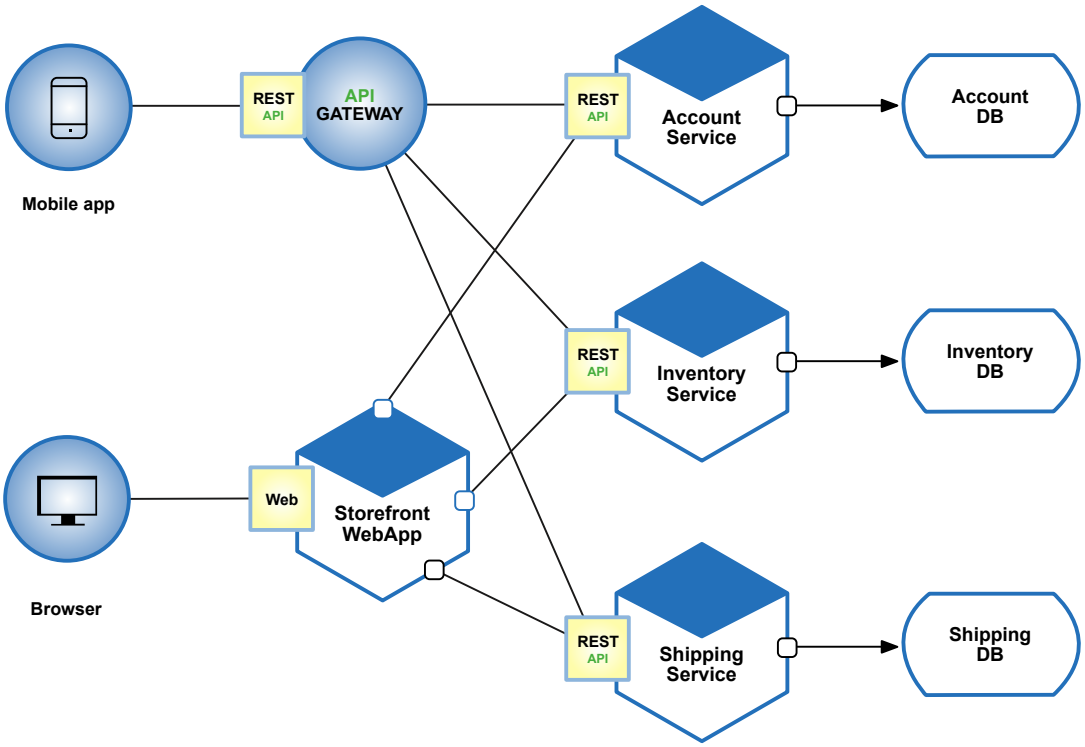


Abbildung 1.9.: Eine typische Microservice-Architektur eines fiktiven E-Commerce Systems (Richardson, 2019b)

Wie vorhin beschrieben, bieten Microservices auch eine Vielzahl an Vor- und Nachteilen, welche für eine Softwarearchitektur vor der Implementierung abgewägt werden müssen. Folgende Punkte definieren die Vorteile von Microservices:

- **Technische Heterogenität:** Bei einer Microservice-Architektur ist man nicht auf eine einzige Programmiersprache angewiesen. Da jeder Service seine eigene Laufzeitumgebung besitzt, kann pro Service eine andere Programmiersprache genutzt werden, solange diese über eine Kommunikationsschnittstelle wie beispielsweise REST einen Servicevertrag bereitstellen kann. Diesen Stil zum Schreiben von Software nennt man auch *Polyglot Programming*, also mehrsprachiges Programmieren. Der Grundgedanke ist hierbei, dass man eine bestimmte Programmiersprache für einen bestimmten Anwendungsfall benutzt.

  Entwickelt man zum Beispiel eine Software auf Basis der Microservice-Architektur, die unter anderem ressourcenintensive Bildverarbeitungsalgorithmen nutzt, so könnte man einen Service definieren, der über Webtechnologien wie *JavaScript* ankommende Bilddaten annimmt und an einen *Bildverarbeitungs-Service* weitergibt, indem diese dann mithilfe von *C++* oder *Rust* performant verarbeitet werden. Die Spra-

chen C++ und Rust eignen sich hierbei zum Verarbeiten von Bilddaten, da diese hardwarenah ausgeführt werden, während sich JavaScript in der Webentwicklung etabliert hat.

Dieser Vorteil bietet also eine Flexibilität beim Entwickeln von Services, da man je nach Anwendungsbereich der entwickelten Software verschiedene Services mit verschiedenen Programmiersprachen und deren jeweiligen Vorteilen nutzen kann.

- **Zuverlässigkeit:** In einer monolithischen Software beeinflusst jeder Teilbereich des Monolithen jeden anderen. Aus diesem Grund kann auch die gesamte Software fehlschlagen, sobald ein Teilbereich fehlschlägt. Bei einer Microservice-Architektur sind allerdings alle Teilbereiche voneinander abisoliert. Wenn in dem genannten Beispiel der Bildverarbeitungs-Service abstürzt, kann der *Annahme-Service* noch auf HTTP-Anfragen antworten und den Nutzer mit nötigen Informationen über das System versorgen. In der Zwischenzeit könnte dann eine neue Instanz des Bildverarbeitungs-Service gestartet werden. Dadurch wird ein System widerstandsfähig und zuverlässig.

  Um allerdings in den Genuss des Vorteils der Zuverlässigkeit von Microservices zu kommen, müssen neue Hürden überwunden werden. Diese äußern sich unter anderem in Form von Netzwerkkomplikationen bei der *Inter-Service-Kommunikation*.

- **Skalierbarkeit:** Wie schon in Abbildung 1.4 dargestellt, muss bei einer hohen Auslastung einer monolithischen Webapplikation die komplette Applikation skaliert werden. Dies sorgt dafür, dass auch Module skaliert werden, die keine hohe Auslastung haben, aber ressourcenintensiv sind.

  Microservices hingegen können pro Service skaliert werden. Wenn also in dem Beispiel der Bildverarbeitung der C++- oder Rust-Service derzeit mit einer Prozessierung von Bilddaten ausgelastet ist, kann eine weitere Instanz dieses Service gestartet werden, der weitere Bilddaten entgegennehmen kann. Solange der JavaScript-Service nicht mit HTTP-Anfragen ausgelastet ist, braucht dieser nicht zu skalieren. Dies erhöht die Ressourceneffizienz und auch die Kostenoptimierung eines Systems.

- **Leichte Installationen:** Mit einem wachsenden Monolithen wird es zunehmend schwerer einen solchen auf einer Maschine ohne Probleme zu installieren. Selbst eine einzige veränderte Zeile Code führt dazu, dass die komplette Applikation wieder kompiliert und installiert werden muss.

Bei Microservices stellt sich dieses Problem nicht, da diese als eigenständige Systeme zu betrachten sind. Wenn in dem obigen Beispiel der Bildverarbeitungs-Service ein Update benötigt, kann dieser unabhängig von dem JavaScript-Service aktualisiert werden.

- **Innovation:** Eine Microservice-Architektur vereinfacht das Einführen von neuen Technologie-Stacks und Frameworks, da jeder Service für sich steht. Gleichzeitig kann so ein vielversprechend aussehendes, aber möglicherweise riskantes Framework in einem Service zur Probe eingesetzt werden. Für den Fall, dass das Framework nicht die Anforderungen erfüllt, kann ein anderer Service mit einem anderen Framework eingesetzt werden.
- **Gesetz von Conway:** Wenn ein System in Microservices unterteilt ist, kann ein Entwicklerteam mit wenig Personal für jenes System einfacher eingeteilt werden. Dies führt zu effizienteren Kommunikationswegen und mehr Flexibilität im Unternehmen und der entwickelten Software.
- **Einfache Benutzbarkeit:** Ein Microservice legt eine primitive API offen, die durch festgelegte Operationen einfach zu benutzen ist. Diese Simplizität lädt andere Entwickler dazu ein, die API zu nutzen statt eine ähnliche Funktionalität zu entwickeln.
- **Effiziente Entwicklung:** Ein Service kann effizienter entwickelt werden, da dieser nur einen Bruchteil der Geschäftslogik widerspiegeln muss. Dadurch können Services mit relativ wenig Entwicklungszeit relativ viel Umsatz hervorrufen.
- **Automatisches Testen:** Ein Microservice kann durch seinen Minimalismus einfach getestet werden. Tests lassen sich in den jeweiligen Teilbereichen des entwickelten Systems detaillierter definieren, was eine erhöhte Qualität der Services hervorruft.
- **Effiziente Betreibbarkeit:** Ein Service kann durch seine geringe Komplexität einfach betrieben werden. Mittels einer funktionalen Virtualisierung oder Containerisierung, wie sie in Abschnitt 1.2 behandelt wird, fügt dem Ganzen eine effizientere Installationsmöglichkeit hinzu.

(Takai, 2017, S. 21–22; Newman, 2015; Allspaw & Robbins, 2010; Gallipeau & Kudrle, 2018, S. 23–25)

Microservices sind zwar ein neuer Ansatz, jedoch keine globale Lösung zum Entwickeln von Software. Diese bieten zwar viele Vorteile, enthalten allerdings auch einige Nachteile, welche im Folgenden aufgelistet sind:

- **Latenz:** Da Services in eigenen Laufzeitumgebungen im Netzwerk verteilt sind, entsteht in der Interkommunikation dieser eine erhöhte Latenz. Dadurch kann das System langsamer erscheinen. Bei Software mit regelmäßigen und vielen HTTP-Anfragen kann das Nutzererlebnis gestört werden.
- **Netzwerkkomplikationen:** Ein verteiltes System ist niemals fehlerlos. Microservices sollten immer mit dem Gedanken entwickelt werden, dass deren Operationen fehlschlagen und dementsprechend gehandelt werden muss. Durch das Hinzukommen der *Inter-Service-Netzwerkkomponente* können in einem verteilten System mehr Fehlersituationen entstehen als bei einem Monolithen.
- **Referenzielle Integrität:** Durch die Trennung der Datenbanken unter den Microservices kann die *referenzielle Integrität* nicht gewahrt werden. Diese besagt, dass nur Entitäten mit einer Referenz auf einer weiteren Entität in einer Datenbank abgespeichert werden können, wenn dieser Eintrag auch in dieser Datenbank einmalig existiert. Auf die referenzielle Integrität muss also manuell geachtet werden.
- **Neues Paradigma:** Aufgrund dessen, dass die Microservice-Architektur ein relativ neues Thema in der Softwareentwicklung ist, müssen Entwicklerteams sich neue Kompetenzen aneignen. Zwar verringern Microservices durch das Abkapseln ihrer Geschäftslogik in kleine Teil-Services deren Komplexität, jedoch entsteht dadurch eine höhere Komplexität im Komponieren dieser Services. Die Komplexität des DevOps-Bereichs wird erhöht und muss von Entwicklerteams beachtet werden.

(Takai, 2017, S. 22)

Zum Öffnen der On-Premise Software namens Royal Render, sodass diese remote-basiert benutzt werden kann, wird in dieser Arbeit eine Microservice-Architektur angestrebt. Für den zu entiwckelnden Remote-Controller wäre eine monolithische Architektur bei einer relativ kleinen Codebasis vollkommen legitim. Allerdings können die oben genannten Vorteile schon im Vorfeld auf das zu entwickelnde System abgebildet werden. Beispielsweise ist es nötig einen *File-Service* zu entwickeln, der gewisse Datenmengen in einem Dateisystem hinterlegen kann. Durch lang andauernde Datenübertragungen muss der Service länger einen Zustand bewahren, was darauf schließen lässt, dass es wichtig ist, dass dieser Service unabhängig von dem Rest der Software skaliert werden kann, damit die Software reaktiv bleibt. Royal Render bietet deutlich mehr Features, die durch Royal Render-Anbindungen

genutzt werden können. Es ist somit zu erwarten, dass die Software später mit mehr Services erweitert werden soll. Somit könnten die Features von Royal Render möglichst weitläufig und genau zur remote-basierten Nutzung abgedeckt und bereitgestellt werden.

1.2. Container vs virtuelle Maschinen

Wie in Unterabschnitt 1.1.2 beschrieben, waren die durch die Installation von Services mitgebrachten, kumulativen Kosten einer der Gründe des Scheiterns von SOA. Zu jener Zeit musste für jeden Service Hardware eingekauft werden, auf welcher dieser Service installiert werden konnte. Wie in Unterabschnitt 1.1.3 beschrieben, bilden Microservices anhand des Single-Responsibility-Prinzips nur einen kleinen Teilbereich in dem Geschäft ab und sind somit feingranularer als durch SOA entworfene Services. Microservices bedeuten also automatisch mehr Services im Vergleich zu SOA und gleichzeitig mehr Server, auf denen die Microservices installiert werden müssen.

Sowohl virtuelle Maschinen als auch Container bieten eine Form der Virtualisierung, um Software installieren und bereitstellen zu können. Dabei benutzen beide Varianten unterschiedliche Ansätze, um dieses Ziel zu erreichen. In Abbildung 1.10 sind diese Unterschiede grafisch dargestellt. Dabei sieht man links den container-basierten Ansatz, während man auf der rechten Seite den Virtual Machine (VM)-Ansatz zum Virtualisieren von Applikationen betrachten kann. Dabei wird Docker als Container-Manager betrachtet.

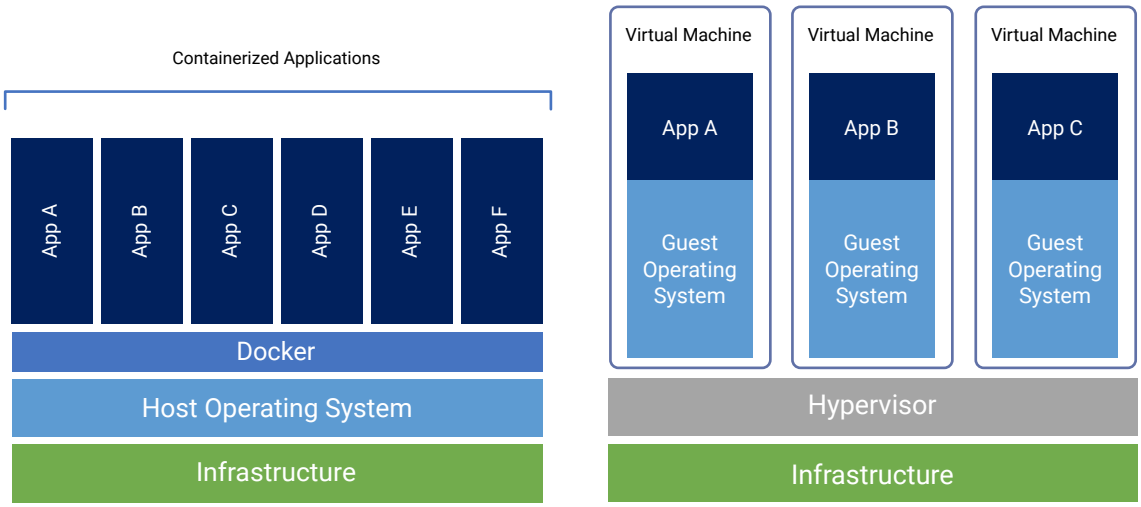


Abbildung 1.10.: Gegenüberstellung der Aufbauweise von Containern (links) und virtuellen Servern (rechts) (Fong, 2018)

Wie die Abbildung 1.10 zeigt, braucht die VM eine Hardware-Infrastruktur-Ebene, auf der diese aufgebaut werden kann. Darauf wird ein *Hypervisor* bereitgestellt, der als Monitor für die virtuellen Maschinen dient, die auf dem System installiert werden. Es gibt zwei Typen von Virtualisierung. Der erste Typ wird in der Abbildung gezeigt. In dieser stellt der Hypervisor gleichzeitig die Betriebssystemfunktionalität und die VM-Versorgung dar. Typ-2 injiziert zwischen Hypervisor und Infrastruktur ein Betriebssystem, auf dem der Hypervisor läuft. Dies beeinflusst allerdings die Performance von VMs, da dadurch mehr als nur die für den Hypervisor nötigen Betriebssystemfunktionalitäten angesteuert werden. Typ-2 wird dadurch oftmals für die Entwicklung genutzt, während Typ-1 für die tatsächliche Distribution von Software eingesetzt wird. Ein Beispiel für eine Typ-1 Virtualisierung bietet der Hypervisor *ESXi* der Firma *VMware*. Auf dem Hypervisor wird für jede Applikation eine virtuelle Maschine aufgesetzt, die ein eigenes Betriebssystem bereitstellt, auf der die Applikation installiert werden kann. Für den Fall, dass Linux als Betriebssystem gewählt wird, bekäme jede Applikation somit seinen eigenen Linux Kernel. Dies bedeutet im Umkehrschluss, dass jede virtuelle Maschine seinen eigenen Speicherslot bekommt und somit von anderen virtuellen Maschinen isoliert ist. Hinzu kommt, dass jede virtuelle Maschine auch komplett einzeln konfiguriert werden muss. Auch wenn sich VMs über die Zeit bewährt und weiterentwickelt haben, bieten Container einige weitere Vorteile (Chelladhurai et al., 2017; Kane & Matthias, 2018).

Container benutzen anders als VMs keinen Hypervisor, der Funktionalitäten eines Betriebssystems bereitstellt. Stattdessen laufen Container auf einem einzigen Betriebssystem. Container sind somit nur einzelne Prozesse, die allerdings durch *namespaces* und *interne Netzwerke* voneinander isoliert werden. Dies nennt man auch „operating system virtualization “ (Kane & Matthias, 2018). Anders als in Abbildung 1.10 dargestellt laufen Container nicht auf einer Docker Instanz, sondern direkt auf dem Betriebssystem, das bereitgestellt wird. Docker ist dabei ein Werkzeug, womit sich diese Prozesse verwalten lassen. Hierbei ist wichtig darauf zu achten, dass Container nur Applikationen betreiben können, die auch auf dem geteilten Kernel laufen können. Ein Linux Betriebssystem lässt also Container einen Linux Kernel teilen, wodurch Linux Container entstehen. Ein Windows Betriebssystem erstellt demnach Windows basierte Container. Ein Container braucht durch das Teilen des Kerns eines Betriebssystems für jede isolierte Arbeitslast kein komplettes Betriebssystem, wodurch die Installations- und Startzeit von Containern deutlich reduziert wird. Kane und Matthias benennen bei VMs folgendes Problem: „In a VM, calls by the process to the hardware or hypervisor would require bouncing in and out of privileged mode on the processor twice, thereby noticeably slowing down many calls“ (Kane & Matthias, 2018). Das Wegfallen einer Hypervisor-Ebene bei Containern impliziert, dass eine zusätzliche Indirektion beim Ausführen von Applikationen wegfällt, wodurch Container

an Performance gewinnen und somit für Microservices in Verbindung mit Continous Integration / Continous Deployment (CI/CD) in Frage kommen. Microservices ermöglichen dadurch ein schnelles und mehrmals am Tag erfolgendes Updaten, Kompilieren und Installieren, um Systemfehler auf schnellstmöglichem Weg zu eliminieren (Kane & Matthias, 2018; Chelladhurai et al., 2017).

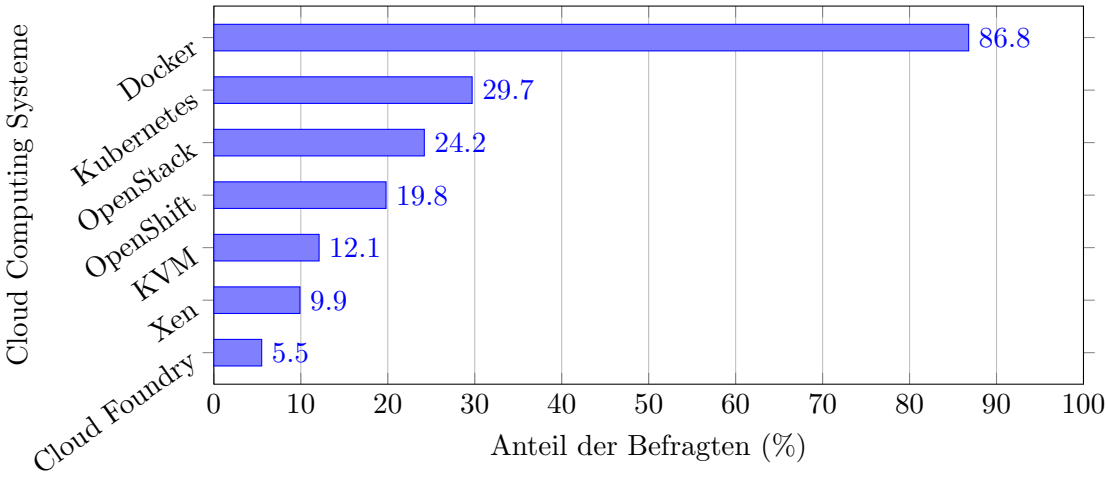


Abbildung 1.11.: Unter 91 Schweizer Unternehmen durchgeführte Umfrage zur Verwendung von Open Source Cloud Computing Systemen (Stürmer & Gauch, 2018)

Abbildung 1.11 zeigt eine Statistik aus dem Jahr 2018. Die Statistik war eine Umfrage zu dem Thema, welche Open Source Cloud Computing Systeme in den Organisationen verwendet werden. Dabei wird allerdings nicht nur Container-Software, sondern auch Software für VMs betrachtet. Dazu wurden 91 Schweizer Unternehmen befragt. Docker ist dabei mit einem Vorsprung von $57,1\%$ vor Kubernetes. Dabei muss herausgestellt werden, dass Docker eine Container-Plattform ist, während Kubernetes ein Container-Orchestrierer ist und somit Docker ergänzt. Kubernetes ist das Pendant zu Docker-Swarm, dem hauseigenen Container-Orchestrierer von Docker.

Durch die, wie in Abbildung 1.11 beschriebene, hohe Beliebtheit von Docker, wird diese Container-Plattform in dieser Arbeit zum Installieren und Entwickeln von containerbasierten Microservices benutzt. Die hohe Verwendungsanzahl von Docker lässt darauf schließen, dass es für diese Software am meisten Unterstützung in Form von Hinweisen, Lösungsvorschlägen und Werkzeugen gibt, was die Wartbarkeit und somit die Zuverlässigkeit des gesamten Systems erhöht. Gleichzeitig können mithilfe von Werkzeugen wie Kubernetes oder Docker-Swarm weitere Vorteile wie dynamische Skalierung von Services genutzt werden.

1.3. Domain-Driven-Design

Wie in Unterabschnitt 1.1.2 beschrieben, ist eine der Problematiken von SOA, dass für diesen Architekturstil mehr Wert auf den IT- und weniger auf den Geschäftsbereich gelegt wurde. Demnach konnten nur Systeme entwickelt werden, welche das Geschäft nicht optimal unterstützen konnten. Domain-Driven-Design (DDD) ist ein von Eric Evans geprägter Entwurfsstil, der die Lücke zwischen IT- und Geschäftsbereich schließen soll. In den nächsten zwei Unterabschnitten dieser Arbeit wird das Konzept von DDD und dessen Komponenten erklärt (Takai, 2017, S. 16).

1.3.1. Konzept

Wie der Name Domain-Driven-Design (DDD) schon sagt, versucht man komplexe Software anhand der Analyse von Geschäftsdomänen zu entwerfen. Software wird entwickelt, damit diese eine Domäne im Geschäft abdeckt oder einen Geschäftsbereich unterstützt. Zum Entwerfen und Entwickeln dieser Software werden Softwareentwickler benötigt, deren Domäneexpertise allerdings nicht auf den jeweiligen Geschäftsbereich zutrifft, sondern auf die Domäne der Softwareentwicklung. Zusammengefasst sollen also Softwareentwickler oftmals Probleme lösen, die bedingt mit der eigentlichen Domäneexpertise der Entwickler zu tun haben. Daraus entstehen gezwungenermaßen Modelle, die wiederum nicht auf den Geschäftsbereich passen, für welchen die Software entwickelt werden soll. Häufig werden Probleme zu technisch und zu wenig geschäftsabhängig betrachtet, wodurch Software entworfen wird, die zu komplex und unstrukturiert ist. Viele Entwickler haben allerdings auch kein Interesse daran die andere Domäne zu erlernen, um die individuelle Software besser modellieren zu können, da im schlimmsten Fall für jedes Softwareprojekt eine neue Geschäftsdomäne betrachtet und somit wieder erlernt werden muss (Evans, 2004, S. 4–6).

Stattdessen schlägt DDD eine Ansammlung von Werkzeugen vor, die mithilfe von Softwareentwicklern und Domäneexperten genutzt werden können, um Domänespezifische Software entwickeln zu können. Diese Werkzeuge sind in den beiden Herangehensweisen *Strategischer Entwurf* (engl. *Strategic-Design*) und *Taktischer Entwurf* (engl. *Tactical-Design*) getrennt. Der *strategische Entwurf* extrahiert durch die Analyse von domänespezifischen, semantischen, strategisch wichtigen Dingen eine Übersicht über die möglichen Anwendungsbereiche der Software in der Geschäftsdomäne. Der *taktische Entwurf* verfasst eine detailliertere, technische Ausprägung davon. Beim Entwerfen von *Domänenmodellen* wird explizit darauf geachtet, dass in diesen geschäftsspezifische Termini wiederzufinden sind (Evans, 2004, S. 3; Vernon, 2016).

1.3.2. Komponenten

Zum Entwerfen einer Software mit DDD sollten Domäneexperten mit Softwareentwicklern zusammenarbeiten, um eine bestmögliche Software für eine bestimmte Domäne zu entwickeln. Durch die verschiedenen Geschäftsdomänen, aus denen die Entwickler und die Domäneexperten stammen, müssen beide Parteien bereit sein aus den gegenseitigen Domänen kontinuierlich zu lernen und sich auf eine sprachliche Grundlage zu einigen. Diese linguistische Grundlage benennt Evans in seinem Buch als *allgemeine Sprache*. Sobald Domäneexperten und Entwickler den gleichen Jargon sprechen, kann dadurch die Kommunikation fließen, wodurch Anforderungen der Domäneexperten von Entwicklern wiederum besser interpretiert werden können. Die allgemeine Sprache findet sich dabei sowohl in Unterhaltungen zwischen den Entwicklern und Experten, als auch in Dokumentationen und im Code als Klassen und Operationen wieder, wodurch sich Domänenmodell und Code ergänzen. Der Begriff des Domänenmodells wird später in diesem Unterabschnitt erläutert (Evans, 2004, S. 24–27).

Nachdem eine linguistische Grundlage geschaffen wurde, kann das strategische Design genutzt werden, welches die Domäne (engl. Domain) und deren *Subdomänen* (engl. *Subdomains*) definiert. Subdomänen bilden dabei Teilbereiche der Domäne. Wenn ein Onlineshop, wie in Abbildung 1.12 gezeigt, entwickelt werden soll, dann definiert sich die Domäne als *E-Commerce System*, während Subdomänen davon *Bezahlung* (engl. *Purchasing*), *Inventar* (engl. *Inventory*), *Ressourcenplanung* (engl. *Ressource Planning*) und *Optimale Akquisition* (engl. *Optimal Acquisition*) sind. Subdomänen werden dabei in *Kerndomäne* (engl. *Core Domain*), *unterstützende Subdomänen* (engl. *Supporting Subdomains*) und *generische Subdomänen* (engl. *Generic Subdomains*) untergliedert. Die Domäne beschreibt den Geschäftsbereich oder den Problembereich, für den eine Software entwickelt werden soll. Die Kerndomäne ist eine Subdomäne, welche die Domäne am besten repräsentiert. Deshalb sollte in diese am meisten Arbeit fließen. In dem Beispiel des Onlineshops ist die Kerndomäne *Optimale Akquisition*. unterstützende Subdomänen sind Teilbereiche, welche die Kerndomäne direkt unterstützen, allerdings nicht Teil derer sind. Generische Subdomänen stellen Teilbereiche der Domäne dar, die für die generelle Geschäftslösung benötigt werden, jedoch die Kerndomäne nicht direkt unterstützen. Generische Subdomänen können oftmals extern eingekauft oder benutzt werden. Im E-Commerce System wäre *Bezahlung* eine unterstützende und *Ressourcenplanung* eine generische Subdomäne. Die Bezahlung eines Produktes ist für einen Onlineshop essenziell wichtig, während *Ressourcenplanung* indirekt die Domäne des Onlineshops unterstützt. Die Deklaration von Subdomänen einer Domäne stellt einen sogenannten *Problemraum* (engl. *Problem Space*) auf (Evans, 2004, S. 402, S. 406; Vernon, 2013, S. 52, S. 56–58; Vernon, 2016).

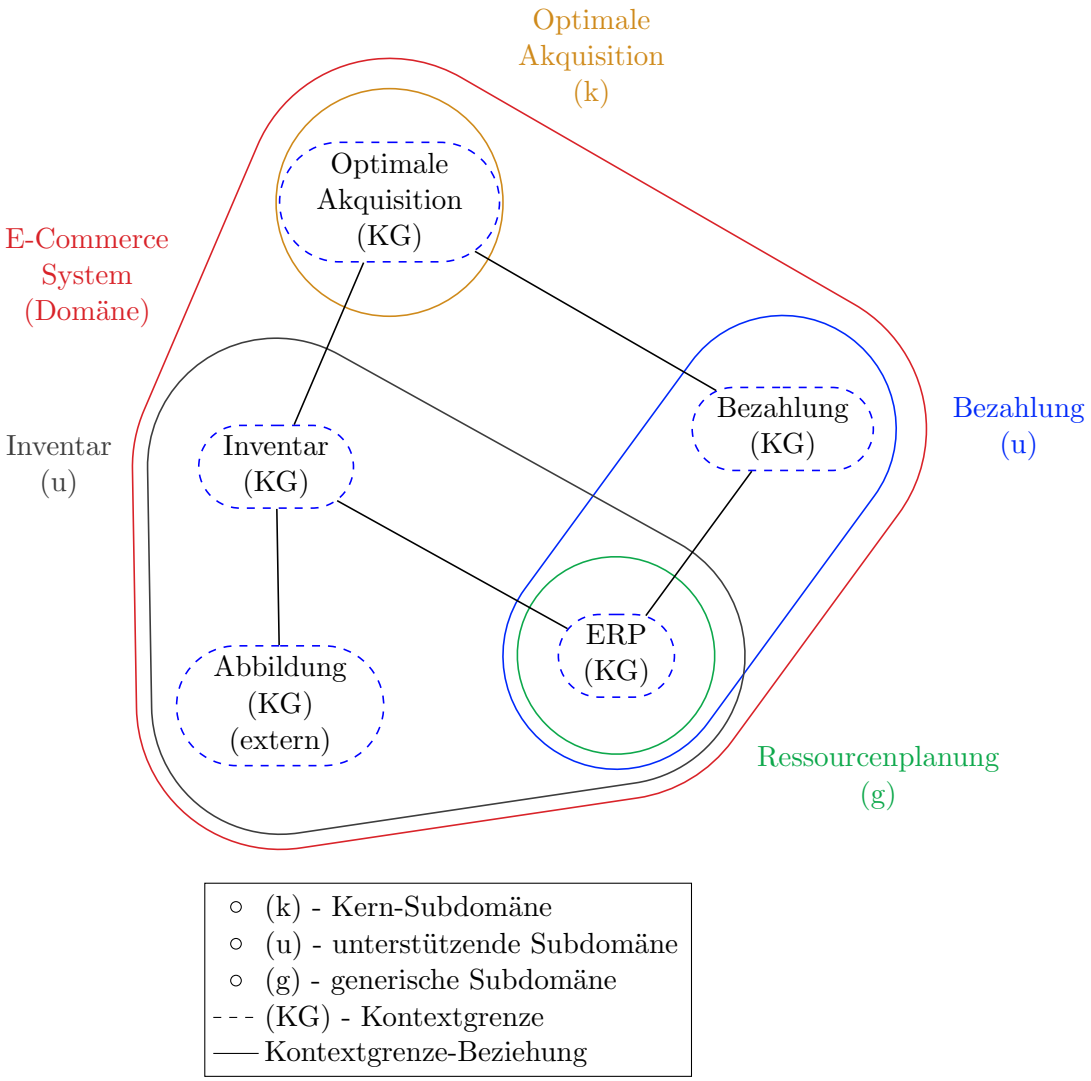


Abbildung 1.12.: Beispiel einer Kontextkarte anhand eines E-Commerce Systems mit einer Domäne, unterstützenden und generischen Subdomänen, einer Kerndomäne, Kontextgrenzen und Beziehungen zwischen den Kontextgrenzen. Die Subdomänen erstrecken sich teilweise über mehrere Kontextgrenzen (Vernon, 2013, S. 58)

Zum Überführen eines *Problemraumes* in einen *Lösungsraum* (engl. *Solution Space*) können Subdomänen durch Kontextgrenzen voneinander sprachlich abgegrenzt werden. Wie in Abbildung 1.12 dargestellt, können Subdomänen mehrere Kontextgrenzen benutzen. Eine Kontextgrenze definiert eine linguistische Grenze zwischen Subdomänen. Der Sinn von Kontextgrenzen ist hierbei, dass bestimmte Objekte in anderen Kontexten eine unterschiedliche Bedeutung haben. In dem Beispiel des Onlineshops könnte ein *Buch* für den Endnutzer ein Objekt mit Seitenzahlen und Inhalt sein, während für das System ein *Buch* aus einer Identifikationsnummer, einem Preis und einer Mengenangabe besteht. Kontextgrenzen können hierbei Beziehungen untereinander haben, um zu verdeutlichen, dass diese miteinander agieren. Erstellt man ein Diagramm wie in Abbildung 1.12, dann nennt man dieses Konstrukt eine *Kontextkarte* (engl. *Context Map*). Idealerweise orientiert man Subdomänen eins-zu-eins mit einer jeweiligen Kontextgrenze, sodass jede Subdomäne seine eigenen Definitionen der Objekte aufstellt und somit von anderen Subdomänen linguistisch klar getrennt ist. In einer Kontextgrenze kann man sich auf eine Darstellungsweise festlegen, welche diese repräsentiert. Die Überlappung der Subdomänen in Abbildung 1.12 ist linguistisch gemeint. Subdomänen wie *Inventory*, *Ressource Planning* und *Purchasing* nutzen Teile derselben Kontextgrenze, bleiben allerdings technisch eigenständige Subdomänen (Evans, 2004, S. 335–337; Vernon, 2013, S. 56–57).

Im *Lösungsraum* besteht eine solche Darstellung einer Kontextgrenze oft aus *Entitäten* (engl. *Entities*), *Wertobjekten* (engl. *Value Objects*), *Aggregaten* (engl. *Aggregates*), *Diensten* (engl. *Services*), *Fabriken* (engl. *Factories*), *Depots* (engl. *Repositories*) und *Domänenevents* (engl. *Domain Events*). Dort können die Relationen dieser Komponenten und deren Interaktionen untereinander repräsentiert werden. Dazu kann die Unified Modeling Language (UML) oder auch freies Zeichnen zum Erstellen von verschiedenen Diagrammen verwendet werden. Es wird also nicht das Domänenmodell dargestellt, sondern oftmals Relationen von Objekten in einer gewissen Kontextgrenze. Ein Domänenmodell beschreibt Evans wie folgt: „A domain model is not a particular diagram; it is the idea that the diagram is intended to convey. It is not just the knowledge in a domain expert's head; *it is a rigorously organized and selective abstraction of that knowledge*“ (Evans, 2004, S. 3, S. 35–37). Ein Domänenmodell ist also nicht nur ein Diagramm, sondern alles, was mit der Domäne in Verbindung steht. Dazu gehören zum Beispiel: Diagramme, Dokumentationen sowie Diskussionen zwischen Domäneexperten und Entwicklern.

Entitäten sind individuelle Domänenobjekte, die anhand eines bestimmten Parameters, zum Beispiel über eine Identifikationsnummer, genau identifiziert werden können. Diese sind meistens veränderlich, können jedoch auch unveränderlich sein. *Wertobjekte* hingegen sind Objekte, die als einfache Datencontainer fungieren und unveränderlich sind. Anders

als bei Entitäten kommt es dabei nicht auf die Individualität des Wertobjektes an. Domänenobjekte können, je nachdem in welcher Kontextgrenze sich diese befinden, ihre Rollen verändern. Ein *Buch* könnte demnach im Kontext des *Shops* eine Entität sein, da für jedes *Buch* ein individueller Preis festgelegt ist. Im Kontext der *Nutzung* ist ein *Buch* jedoch ein Wertobjekt, welches Seiten und Inhalt besitzt. Für die Nutzung ist nicht wichtig, welches exakte *Buch* genutzt wird, solange es denselben Inhalt hat. *Aggregate* sind Konglomerate von Entitäten und Wertobjekten. Eine *Aggregatwurzel* spiegelt dabei eine Entität wider. Ein Beispiel dafür wäre ein *User-Aggregat*, das auf die Entität *UserID* und das Wertobjekt *E-Mail* verweist. Aggregate sollten dabei nicht weitere Aggregate abspeichern, sondern lediglich deren Attribut, womit diese identifiziert werden können. *Dienste* sind Funktionen, die semantisch auf kein Aggregat oder Wertobjekt passen, jedoch für die Domäne gebraucht werden. Ein Beispiel hierfür wäre ein *Authentifizierungsdienst*. Eine *Fabrik* existiert, um Entitäten, Aggregate und Wertobjekte zu konstruieren. Zweck einer Fabrik ist es komplexe Objektkonstruktionen aus der Domänenlogik zu abstrahieren. Dabei muss jede Operation einer Fabrik atomar, also unabhängig von anderen Prozessen, laufen. Gleichzeitig ist eine Fabrik stark an ihre Parameter gekoppelt. Im Gegensatz zu Fabriken, wo neue Domänenobjekte erstellt werden, besitzen Depots schon vorher eine Anzahl an Objekten. Hier findet sich die Anbindung an eine Datenbank wieder. Letztlich findet man in einem Domänenmodell verschiedene *Domänenevents*. Diese sind Ereignisse im System, die für die Domäneexperten von Relevanz sind. Bei dem Onlineshop-Beispiel wäre dies, wenn ein favorisiertes *Buch* wieder eingelagert wurde. Durch Domänenevents können Domänenobjekte untereinander auf indirektem Weg Nachrichten übermitteln (Evans, 2004, S. 2–4, S. 89–108, S. 136–154; Vernon, 2013, S. 265–286, S.347–362, S. 389–401; Vernon, 2016).

In dieser Arbeit wird ein DDD-Ansatz genutzt, um für das System mögliche Microservices zu identifizieren. Ein ausführlicheres Erläutern von DDD und die Ausführung eines Event-Storming-Workshops, um mit Domäneexperten und Entwicklern Domänenevents des Systems zu finden, würde den Rahmen dieser Arbeit überziehen und kann zeitbedingt nicht genügend ausgeführt werden, weshalb darauf verzichtet wird.

Akronyme

BaaS Backend as a Service. 4

CI/CD Continous Integration / Continous Deployment. 20

DDD Domain-Driven-Design. 9, 12, 21, 22, 25

DRY Don't Repeat Yourself. 11

E2E End-To-End. 4, *Glossar:* E2E

HTTP Hypertext Transfer Protocol. 5, 7, 15, 17

REST Representational State Transfer. ii, 12–14

SaaS Software as a Service. 4

SOA Service-Oriented Architecture. 3, 6–10

SOAP Simple Object Access Protocol. 9

UML Unified Modeling Language. 24

VM Virtual Machine. 18–20

WAR Web Archive. 4

Glossar

DevOps Ein Prozessverbesserungsansatz, der verschiedene Werkzeuge und Prozesse einsetzt, um bei den Bereichen der Entwicklung, Qualitätssicherung und dem IT Betrieb für eine automatisierte Zusammenarbeit zu sorgen. 4, 12, 17

Domäne Ein Arbeitsbereich, Anwendungsgebiet oder ein Problemfeld. 21, 22

E2E Eine Testmethode, um den Verlauf einer Applikation von Anfang bis Ende zu testen. Dadurch sollen mithilfe von simulierten aber echten Nutzerszenarien die Daten- und Systemintegration mit dessen Komponenten validiert werden. 4, 26

Load Balancer Ein Gerät, welches ankommende Netzwerkanfragen zwischen Servern verteilt, sodass kein Server mit Anfragen überlastet wird. 5

On-Premise Eine Applikation, die nur lokal ausgeführt und benutzt werden kann. 17

SOLID Prinzipien der objektorientierten Entwicklung. Diese Prinzipien sind:

- Single-Responsibility-Prinzip
- Open-closed-Prinzip
- Liskov-substitution-Prinzip
- Interface-segregation-Prinzip
- Dependency-inversion-Prinzip

Die Anfangsbuchstaben der Prinzipien bilden das Akronym *SOLID*. 11

Literaturverzeichnis

Allspaw, J. & Robbins, J. (2010). *Web Operations: Keeping the Data On Time*. O'Reilly Media, Inc. Verfügbar 19. Januar 2020 unter https://learning.oreilly.com/library/view/web-operations/9781449377465/. (Siehe S. 16)

Chelladhurai, J. S., Singh, V. & Raj, P. (2017). *Learning Docker* (2. Aufl.). Packt Publishing Ltd. Verfügbar 20. Januar 2020 unter https://learning.oreilly.com/library/view/learning-docker-/9781786462923/. (Siehe S. 19, 20)

Conway, M. E. (1968). How do Committees Invent. *Datamation*, *14*(4), 28–31. Verfügbar 15. Januar 2020 unter http://www.melconway.com/Home/Committees_Paper.html (siehe S. 12)

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall. Verfügbar 22. Dezember 2019 unter https://learning.oreilly.com/library/view/service-oriented-architecture-concepts/0131858580/. (Siehe S. 3, 6, 8, 9)

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley. (Siehe S. 21, 22, 24, 25).

Fong, J. (2018). *Are Containers Replacing Virtual Machines?* Verfügbar 20. Januar 2020 unter https://www.docker.com/blog/containers-replacing-virtual-machines/. (Siehe S. 18)

Foote, B. & Yoder, J. (1997). Big Ball of Mud. *Pattern Languages of Program Design*, *4*, 654–692. Verfügbar 3. Januar 2019 unter http://www.laputan.org/mud/ (siehe S. 5)

Fowler, M. & Lewis, J. (2014). *Microservices: a definition of this new architectural term*. ThoughtWorks. Verfügbar 14. Januar 2020 unter https://martinfowler.com/articles/microservices.html. (Siehe S. 9, 12)

Gadzinowski, K. (2017). *Creating Truly Modular Code with No Dependencies*. Verfügbar 3. Januar 2019 unter https://www.toptal.com/software/creating-modular-code-with-no-dependencies. (Siehe S. 5)

Gallipeau, D. & Kudrle, S. (2018). Microservices: Building Blocks to New Workflows and Virtualization. *SMPTE Motion Imaging Journal*, *127*(4), 21–31. https://doi.org/10.5594/JMI.2018.2811599 (siehe S. 1, 6, 9, 12, 16)

Google LLC. (2020). *Datensatz zum weltweiten Interesse zu den Suchthemen der Architekturstile Microservices und Serviceorientierter Architektur*. Verfügbar 14. Januar 2020 unter https://trends.google.de/trends/explore?date=all&q=%2Fm%2F011spz0k,%2Fm%2F0315s4. (Siehe S. 9)

Kane, S. P. & Matthias, K. (2018). *Docker: Up & Running: Shipping Reliable Containers in Production* (2. Aufl.). O'Reilly Media, Inc. Verfügbar 20. Januar 2020 unter https://learning.oreilly.com/library/view/docker-up/9781492036722/. (Siehe S. 19, 20)

Namiot, D. & Sneps-Sneppe, M. (2014). On Micro-services Architecture. *International Journal of Open Information Technologies*, *2*(9), 24–27. Verfügbar 22. Dezember 2019 unter http://injoit.org/index.php/j1/article/view/139/104 (siehe S. 5, 6)

Newman, S. (2015). *Building Microservices: designing fine-grained systems*. O'Reilly Media, Inc. Verfügbar 15. Januar 2020 unter https://learning.oreilly.com/library/view/building-microservices/9781491950340/. (Siehe S. 12, 16)

Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Inc. Verfügbar 21. Dezember 2019 unter https://learning.oreilly.com/library/view/monolith-to-microservices/9781492047834/. (Siehe S. 1–6)

Richards, M. (2016). Microservices vs. Service-Oriented Architecture. Verfügbar 15. Januar 2020 unter https://learning.oreilly.com/library/view/microservices-vs-service-oriented/9781491975657/ (siehe S. 10, 11)

Richardson, C. (2018). *Microservices Patterns*. Manning Publications. Verfügbar 22. Dezember 2019 unter https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/. (Siehe S. 3, 5, 6)

Richardson, C. (2019a). *A pattern language for microservices*. Verfügbar 16. Januar 2020 unter https://microservices.io/patterns/index.html. (Siehe S. 35)

Richardson, C. (2019b). *Pattern: Microservice Architecture*. Verfügbar 16. Januar 2020 unter https://microservices.io/patterns/microservices.html. (Siehe S. 14)

Stürmer, M. & Gauch, C. (2018). Open Source Studie Schweiz 2018. Verfügbar 20. Januar 2020 unter https://www.oss-studie.ch/open-source-studie-2018.pdf (siehe S. 20)

Takai, D. (2017). *Architektur für Websysteme: Serviceorientierte Architektur, Microservices, Domänengetriebener Entwurf*. Carl Hanser Verlag. (Siehe S. 1, 5–12, 16, 17, 21).

Thomas, D. & Hunt, A. (2019). *The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition* (2. Aufl.). Addison-Wesley. Verfügbar 15. Januar 2020 unter https://learning.oreilly.com/library/view/the-pragmatic-programmer/9780135956977/. (Siehe S. 11)

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley. (Siehe S. 22–25).

Vernon, V. (2016). *Domain-driven design distilled*. Addison-Wesley. (Siehe S. 21, 22, 25).

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *2015 10th Computing Colombian Conference (10CCC)*, 583–590. https://doi.org/10.1109/ColumbianCC.2015.7333476 (siehe S. 10)

Westeinde, K. (2019). *Deconstructing the Monolith (Shopify Unite Track 2019)*. Shopify. Verfügbar 21. Dezember 2019 unter https://www.youtube.com/watch?v=ISYKx8sa53g&t=298. (Siehe S. 2)

Name: Robin Dürhager
Matrikel-Nr.: 2150495
Studiengang: Computervisualistik und Design

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Literatur und Hilfsmittel angefertigt habe. Wörtlich übernommene Sätze und Satzteile sind als Zitate belegt, andere Anlehnungen hinsichtlich Aussage und Umfang unter Quellenangabe kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und ist auch noch nicht veröffentlicht.

Ort, Datum	Unterschrift

A. Anhang

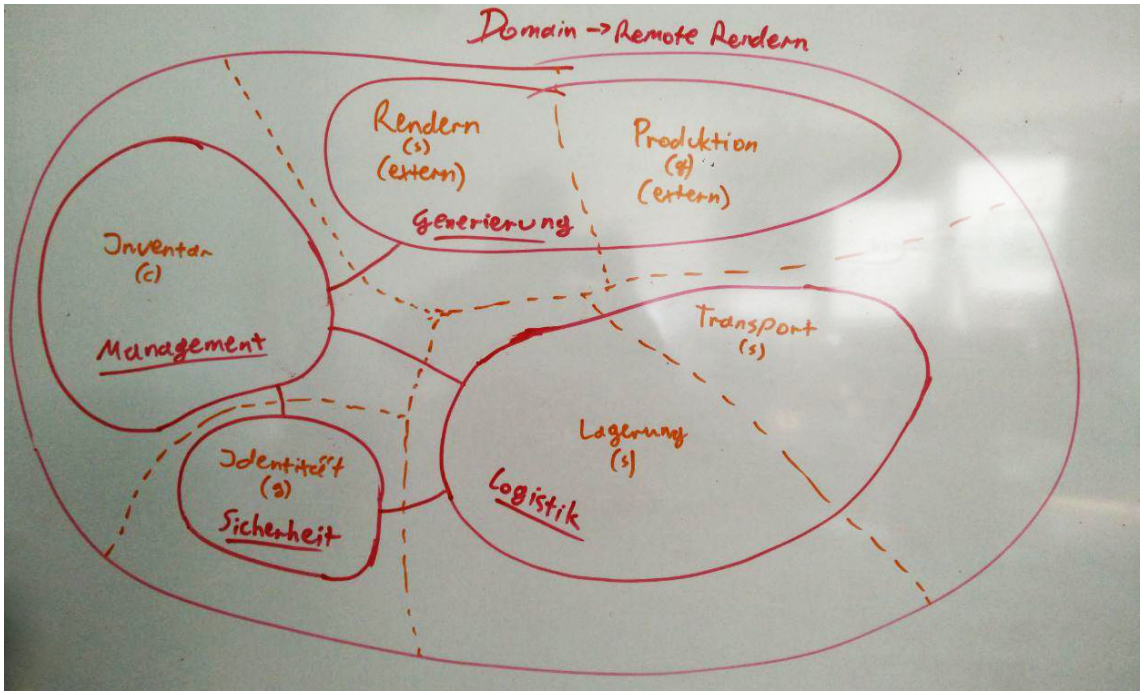
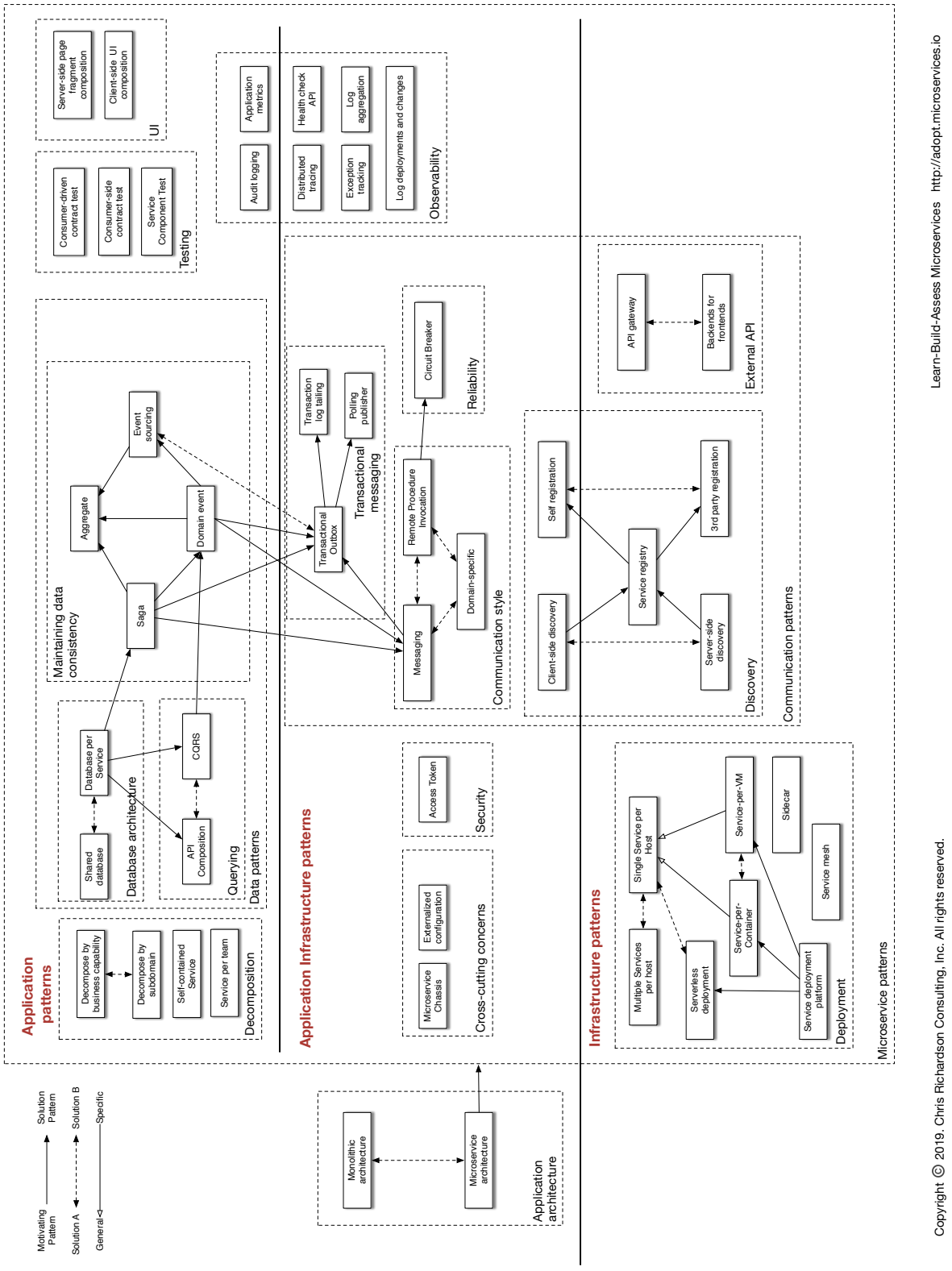


Abbildung A.1.: Skizze der Kontextkarte der Domäne *Remote Rendern*

```
class AuthenticatedDataSource extends RemoteGraphQLDataSource {

  // Diese Methode wird für jede Anfrage über das Apollo Gateway aufgerufen
  async willSendRequest({ request, context }) {

    // "token" ist ein vom Apollo Server weitergeleiteter Wert
    // Dieser kann ein String oder null sein
    // "auth" ist eine Firebase Authentication Instanz
    const { token, auth }: Context = context

    // Beim ersten Poll des GraphQL-Schemas wird diese Funktion ausgeführt
    // der Token ist dann null, da die Anfrage vom Apollo Gateway kam
    if (!token) return

    // Falls ein Token vom Apollo Server gesetzt wurde, der nicht null ist
    // versuche den Token über Firebase in eine ID zu konvertieren
    // Falls dies nicht funktioniert, führe den catch block aus.
    const result = await auth
      .verifyIdToken(token)
      .catch(err => console.error(err))

    if (!result) throw new Error('Artist is not Authorized')

    // Speicher den konvertierten Token in den "authorization" header
    // Für den Fall, dass es Queries gibt, die keinen Authorisierten Nutzer brauchen:
    // Speicher ebenfalls die "artistid" ab, falls diese extrahiert werden konnte
    request.http.headers.append('authorization', token)
    request.http.headers.append('artistid', result.uid)

    return
  }
}
```

Code 1: Apollo Gateway Implementierung eines Sicherheitsmechanismus zur Absicherung gegen unautorisierte Zugriffe



Learn-Build-Assess Microservices http://adopt.microservices.io

Abbildung A.2.: Microservice Entwurfsmuster im Überblick (Richardson, 2019a)