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. Einleitung

1.1. Motivation

Die Internationale Filmschule Köln (IFS) ist eine Bildungsinstitution, die sich auf die Vermittlung von künstlerisch-wissenschaftlichen Inhalten spezialisiert hat. Das Erstellen von visuellen Effekten bildet einer der Studiengänge ab, welcher von der IFS für Interessenten bereitgestellt wird. Bei diesem werden 3D-Grafiken generiert, die oftmals durch ihre Rechenintensität mehrere Stunden lang berechnet werden müssen. Während dieser Generierung kann ein Computer kaum weiter benutzt werden, da dieser all seine Ressourcen für die Berechnung der 3D-Grafiken einsetzt.

Um das Problem von langen Renderzeiten zu vermeiden, wurden Renderfarmen entwickelt. Diese verteilen eine 3D-Szene zum Rendern auf mehrere Computer. Dadurch muss jeder genutzte Computer nur einen Bruchteil der Berechnungen für die gleiche 3D-Szene durchführen, wodurch sich die gesamte Renderzeit der Szene verringert. Eine dieser Renderfarmen ist *Royal Render*.

Dieses Konzept funktioniert allerdings nur mit einer größeren Anzahl an Computern innerhalb eines Netzwerks. Die IFS besitzt jedoch zum aktuellen Zeitpunkt zu wenige Computer, um den Vorteil einer Renderfarm-Software komplett ausreizen zu können. Aus diesem Grund kollaboriert die Hochschule Hamm-Lippstadt mit der IFS. Diese besitzt deutlich mehr Computer, welche auch in einer Renderfarm zusammengeschlossen sind. Die Computer werden allerdings häufiger für den normalen Gebrauch und weniger als Renderfarm benutzt, weswegen die nicht genutzte Rechnerstärke der Hochschule Hamm-Lippstadt von der IFS mitbenutzt werden kann.

Bei der momentanen Kollaboration zwischen der IFS und der Hochschule Hamm-Lippstadt lädt die IFS ein Renderprojekt auf einen Datenserver hoch, von dem aus ein Mitarbeiter der Hochschule Hamm-Lippstadt dieses herunterladen und der Renderfarm übergeben kann. Sobald die Renderfarm mit dem Rendern der 3D-Szene fertig ist, muss der Mitarbeiter die gerenderten Daten wieder manuell auf den Datenserver hochladen. Derweilen bekommt der IFS-Student weder Kenntnisse darüber, wie lange sein Rendering noch prozessiert werden muss, noch Informationen darüber, ob sein Rendering überhaupt gerendert wird.

Diese Prozesse könnten mithilfe einer Software automatisiert werden. So würden zum einen Mitarbeiter der Hochschule Hamm-Lippstadt entlastet werden, zum anderen könnten Studenten der IFS effizienter arbeiten. Die Nutzung der Computer der Hochschule Hamm-Lippstadt würde zu einer Minimierung der Renderzeit von 3D-Szenen der IFS beitragen. Ebenfalls führen somit die Renderprozesse nicht mehr dazu, dass die komplette Computerstärke eines IFS-Studenten gebraucht wird, wodurch dieser an weiteren rechenintensiven Projekten arbeiten kann.

1.2. Zielsetzung

Ziel dieser Arbeit ist es eine erste lauffähige Version des Backends der Software unter dem Projektnamen *rrRemote* zu entwickeln. Diese soll ermöglichen, über ein extern entwickeltes Frontend Renderprojekte in die Royal Render-Instanz der Hochschule Hamm-Lippstadt automatisiert zu übermitteln und gerenderte Daten an den IFS-Studenten zurückzusenden. Die Basis dafür schafft eine in Docker-Container verpackte Microservice-Architektur, welche durch automatisierte Installations-Skripts schnell und ständig erweitert werden kann. Wie vorher erwähnt, handelt es sich hierbei um eine erste Version und damit keine vollwertige Software, die entwickelt werden soll. Diese soll vor allem Studenten der IFS ermöglichen Renderprojekte an die Hochschule Hamm-Lippstadt zu übertragen und dort rendern lassen zu können, sodass keine weitere menschliche Zwischeninstanz seitens der Hochschule Hamm-Lippstadt nötig ist. Das Frontend wird in einem eigenständigen Projekt entwickelt, weshalb es nicht als Teil dieser Arbeit betrachtet wird.

1.3. Aufbau der Arbeit

In dem ersten Kapitel wird die Motivation und Zielsetzung dieser Arbeit erläutert. Dabei wird kurz auf die aktuelle Lage der Problemstellung der IFS beim Rendern von 3D-Szenen eingegangen und ein möglicher Lösungsansatz präsentiert.

Das zweite Kapitel beschäftigt sich mit einer grundlegenden Anforderungsanalyse für das Backend der Software rrRemote. Dabei werden die Anforderungen der IFS im Vergleich mit den Funktionalitäten der Renderfarm-Software Royal Render betrachtet.

Darauffolgend werden im dritten Kapitel die Grundlagen der Softwarearchitektur besprochen, welche für die zu entwickelnde Software von Relevanz sind. Dabei wird die oftmals genutzte monolithische Softwarearchitektur mit distributierten Softwarearchitekturen verglichen. Ebenfalls werden Unterschiede von Containern und virtuellen Maschinen betrachtet. Des Weiteren beschäftigt sich diese Arbeit mit Konzepten und Komponenten der Entwurfsmethodik des Domain-Driven-Designs. Am Schluss des Kapitels werden dann übliche Microservice Entwurfsmuster kurz erläutert.

Das vierte Kapitel wird in die Bereiche der Konzeptionierung und der Implementierung aufgeteilt. Dabei wird zuerst die Softwarearchitektur des Backends von rrRemote evaluiert. Anschließend werden verwendete Technologien für die Umsetzung erläutert. Danach wird auf die letztendliche Realisierung der vorher skizzierten Softwarearchitektur eingegangen, die das Backend von rrRemote darstellt.

Zuletzt wird im fünften Kapitel diese Arbeit zusammengefasst und ein kurzer Ausblick auf zukünftige Arbeiten gegeben, die an der Software rrRemote stattfinden könnten.

2. Anforderungsanalyse

2.1. Anforderungen der Anspruchsgruppen

Das zu entwickelnde System soll es ermöglichen, dass Studenten der IFS ihre Projekte mithilfe von Computern der Hochschule Hamm-Lippstadt ohne externe Hilfe rendern können. In einem Gespräch mit Studenten, sowie mit Angestellten der IFS und mit Studenten des Computervisualistik und Design Studiengangs der Hochschule Hamm-Lippstadt, wurden die Anforderungen der zu entwickelnden Software evaluiert. Die funktionalen Anforderungen sind für eine erste Version des zu entwickelnden Systems wie folgt definiert:

- Ein *Autodesk Maya-* und *Solidangle Arnold-basiertes* Projekt muss in das System hochgeladen werden können.
- Renderjobs müssen durch das System vorbereitet und an Royal Render übergeben werden können.
- Daten von übergebenen Renderjobs, wie verbleibende Renderzeit, Renderstatus, etc., müssen vom Frontend abgefragt werden können.
- Die entstandenen gerenderten Daten müssen dem Künstler wieder bereitgestellt werden.
- Renderjobs müssen vom Künstler abgebrochen werden können.
- Renderjobs müssen vom Künstler gelöscht werden können.
- Renderjobs sollten vom Künstler aktualisiert werden können.

Dabei ist es wichtig, dass das System reaktiv bleibt und weitere nicht-funktionale Anforderungen erfüllt. Diese sind wie folgt aufgelistet:

- Das System muss den Nutzer ständig über den Status seines Renderjobs informieren können und antwortbereit sein.
- Fehler im System müssen für Angestellte der Renderfarm strukturiert und analysierbar sein.
- Fehler im System dürfen die Reaktivität des Systems nur kurzfristig beeinflussen.

(Bonér et al., 2014)

2.2. Analyse der Funktionsweise von Royal Render

In der Filmbranche werden regelmäßig 3D-Szenen erstellt, die verschiedene rechenintensive Parameter beinhalten. Diese sind zum Beispiel: Partikeleffekte, physikalische Simulationen und hochauflösende Texturen. Die Inhalte müssen für jedes Bild, das gerendert werden soll, neu berechnet werden. Solange nur ein Computer an der 3D-Szene rechnet, kann ein Rendering bis zu mehrere Tage, Wochen oder Monate andauern. Ein zehnsekündiger Film, der mit 24 Bildern die Sekunde laufen soll, muss also 240 Bilder berechnen lassen. Würde ein Bild genau einen Tag zum Rendern benötigen, so ergäbe dies 240 Tage Renderzeit.

Holger Schönberger versucht mit Royal Render das Problem von langen Renderzeiten zu minimieren, indem das Programm die Masse an zu rendernden Daten an mehrere Computer verteilt. Hätte man also 240 Computer, so würde jeder dieser Computer bei dem oben genannten Beispiel, ein einziges Bild rendern. Statt 240 Tage Renderzeit, wird die komplette Bildsequenz in einem Tag vervollständigt.

Um dieses Ziel zu erreichen, benötigt Royal Render einen Datenserver, auf dem Renderprojekte und gerenderte Daten abgespeichert werden können, Clients, die Renderings vollziehen können und einen Server, der Renderprojekte an Clients delegieren und diese orchestrieren kann. Dabei kann der Datenserver und der Server zur Delegation von Renderprojekten eine Einheit bilden. Wichtig ist hierbei nur, dass alle Clients Zugriff auf den Datenserver bekommen. Respektiv werden diese Komponenten in Royal Render als *Fileserver*, *rrClients* und *rrServer* benannt. Der Aufbau von Royal Render und dessen Komponenten in der Hochschule Hamm-Lippstadt wird durch die Abbildung 2.1 verdeutlicht (Schönberger, 2019b).

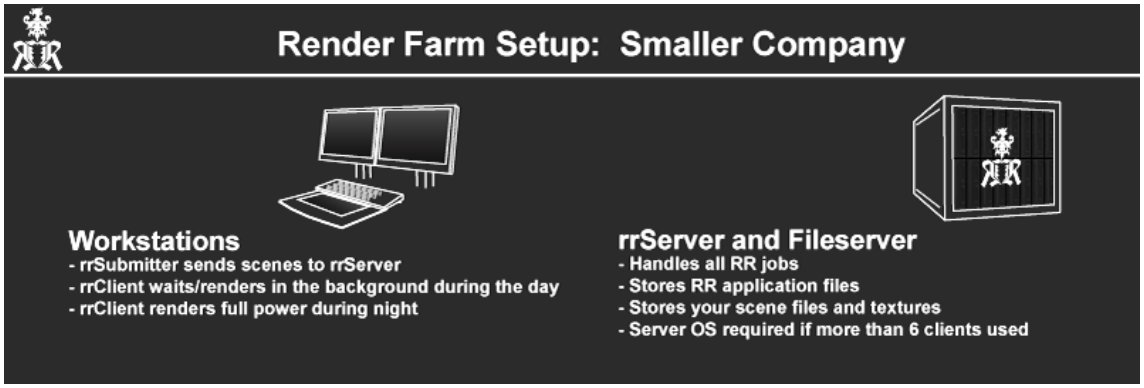

Abbildung 2.1.: Funktionsweise und Aufbau der Royal Render Komponenten in einem kleinen Unternehmen. Gleichzeitiger Aufbau von Royal Render in der Hochschule Hamm-Lippstadt (Schönberger, 2019a)

Es können jedoch nicht direkt alle Computer in einem internen Netzwerk für das Rendern von Daten benutzt werden. Mit einer Installationsdatei von Royal Render kann auf den Computern der rrClient installiert werden. Anschließend wird dieser Computer beim rrServer als potenziell nutzbare Rechenmaschine zum Ausführen von Renderings registriert. Ebenfalls muss der rrServer auf einem Windows-, oder einem Linux-Server installiert werden (Schönberger, 2019a; Schönberger, 2019b).

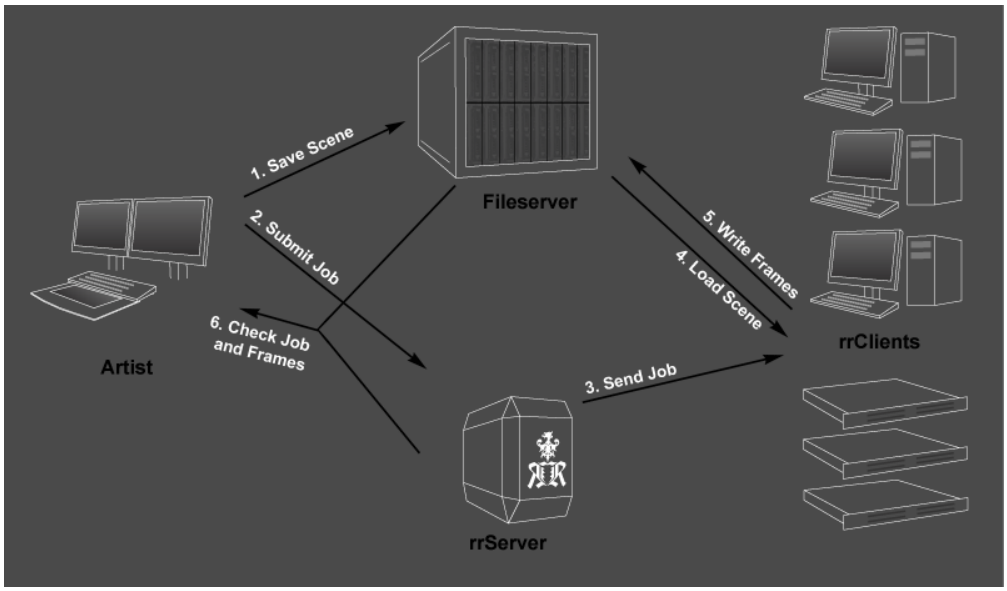

Abbildung 2.2.: Arbeitsverlauf beim Übertragen eines Renderprojektes nach Royal Render. Der rrServer ist in dieser Abbildung vom Fileserver physisch getrennt (Schönberger, 2019a)

Nachdem alle Systeme installiert wurden, kann Royal Render Renderprojekte an rrClients delegieren. Wie in Abbildung 2.2 dargestellt, muss zuerst die 3D-Szene auf dem Datenserver gespeichert werden, damit jeder rrClient auf diese Zugriff bekommt. Über ein Plugin einer 3D-Applikation wie Autodesk Maya, oder über das Royal Render Programm *rrSubmitter*, kann dann das Renderprojekt an Royal Render übergeben werden. Alternativ kann auch das Befehlszeilen-Werkzeug *rrSubmitterconsole* dafür benutzt werden. Dabei bietet der rrSubmitter eine grafische Benutzeroberfläche, während die rrSubmitterconsole eine textbasierte Eingabemethode erfordert. Demnach bietet sich die Nutzung der rrSubmitterconsole als Schnittstelle für externe Programme an, die Renderprojekte an Royal Render übergeben müssen. Nachdem das Renderprojekt auf dem Fileserver abgelegt und der rrServer informiert wurde, dass ein Rendering des Projektes starten kann, sendet der rrServer für jegliche rrClients denselben Renderjob mit verschiedenen Bildsequenzabschnitten. Danach wird die Szene für jeden rrClient in einen temporären Ordner abgespeichert. In diesen kann ein rrClient gerenderte Daten abspeichern und diese dann später an den rrServer übertragen. Dies reduziert die Auslastung des Netzwerks und sorgt für ein Caching der Szenen-Datei, da ein erneutes Übertragen dieses Renderjobs keinen weiteren Kopiervorgang benötigt, solange die 3D-Szene nicht weiter modifiziert wurde. Danach wird die 3D-Szene von den aktivierten rrClients geladen und die Anzahl der Bilder gerendert (Schönberger, 2019a; Schönberger, 2019b; Schönberger, 2020b).

Royal Render ist eine Software die viele Modifizierungsmöglichkeiten bietet. In den Applikationsdateien des rrServers findet man viele Textdateien, die Informationen zum Erweitern der Software beinhalten. Ein Renderjob kann zum Beispiel mit weiteren Python-Skripts versehen werden, die nach Erreichung eines gewissen Renderjobstatus ausgeführt werden. Falls von rrClients fehlerhafte Extended Dynamic Range (EXR)-Dateien gerendert werden, welche im Dateinamen das Prefix *_broken_* beinhalten, können diese beispielsweise durch ein Python-Skript bei einem Beenden des Renderprozesses gelöscht werden, um die Speicherkapazität des Fileservers anzuheben. Es finden sich ebenfalls die 3D-Applikations-Plugins zum Überführen eines Renderjobs nach Royal Render als modifizierbare Python-Skripts in den Applikations-Dateien des rrServers. Die Konfiguration des rrServers und der rrClients erfolgt ausschließlich über das Programm *rrConfig*, welche eine grafische Benutzeroberfläche zur Verfügung stellt. Für Renderapplikationen, wie Solidangle Arnold, müssen in Royal Render Lizenzserver und weitere Konfigurationen in textbasierten Dateien festgelegt werden, damit Royal Render funktionieren kann. Zudem müssen die Python-Skripte, welche für jene Renderjobstati aktiviert werden sollen, auch über Konfigurationsdateien in Royal Render eingeladen werden, in welchen definiert werden kann, mit welchen Parametern aus Royal Render ein Python-Skript ausgeführt werden soll.

Royal Render bietet drei Schnittstellen zum Sammeln von Daten zu bestimmten Renderjobs. So kann über eine Python-, NodeJS- und eine C++-Anbindung beispielsweise der Renderjobstatus eines Renderprojektes ermittelt werden.

Royal Render wurde nicht anhand einer speziellen Softwarearchitektur entwickelt. Trotzdem weist diese Software einige Aspekte von verschiedenen Softwarearchitekturen auf. Der Großteil von Royal Render ist als modularer Monolith anzusehen, da er durch verschiedene Konfigurationsdateien und Zusatz-Plugins erweitert werden kann. Wie der rrServer mit den rrClients agiert, ähnelt allerdings einer Service-Oriented Architecture (SOA). Die rrClients sind dabei als Services anzusehen, die vom rrServer orchestriert werden. Er vergibt Renderjobs an rrClients, welche daraufhin das jeweilige Renderprojekt kopieren, das Projekt rendern und die gerenderten Daten wieder zum Fileserver übertragen. Die Softwarearchitektur des Monolithen und SOA wird in Abschnitt 3.1 behandelt. Der rrServer beinhaltet bei jedem Renderjob durch seinen Renderjobstatus eine endliche Zustandsmaschine (perönliche Nachricht von Holger Schönberger, Januar 02, 2020).

Die Funktionalität von Royal Render lässt alle funktionalen Anforderungen aus Abschnitt 2.1 zu. Das Hochladen eines Renderprojektes oder gerenderter Daten ist nicht Teil von Royal Render. Dies kann von einem externen System übernommen werden. Dennoch können externe Python-Skripts bei bestimmten Royal Render-Events, wie das Finalisieren eines Renderjobs, ausgeführt werden, um bei dem zu entwickelnden Remote-Controller solche Funktionalitäten zu aktivieren. Das Vorbereiten von Renderjobs für Royal Render sollte hauptsächlich durch Studenten der IFS erfolgen, wobei bestimmte Parameter durch das zu entwickelnde Backend von rrRemote zwangsweise angepasst werden müssen. Renderjobinformationen, wie die verbleibende Renderzeit eines Renderjobs oder dessen Status, können mithilfe von Python-, NodeJS-, oder C++-Anbindungen von Royal Render abgefragt werden. Da der rrServer für jeden Renderjob, wie vorher beschrieben wurde, durch die Renderjobstati eine endliche Zustandsmaschine beinhaltet, können deren Renderprozesse durch Modifizieren des jeweiligen Renderjobstatus mithilfe der Royal Render-Anbindungen, geändert werden. Durch die rrSubmitterconsole kann durch verschiedene Programmiersprachen mithilfe einer Textzeile ein Renderjob nach Royal Render übergeben werden.

3. Grundlagen der Softwarearchitektur

3.1. Architekturstile

3.1.1. 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.

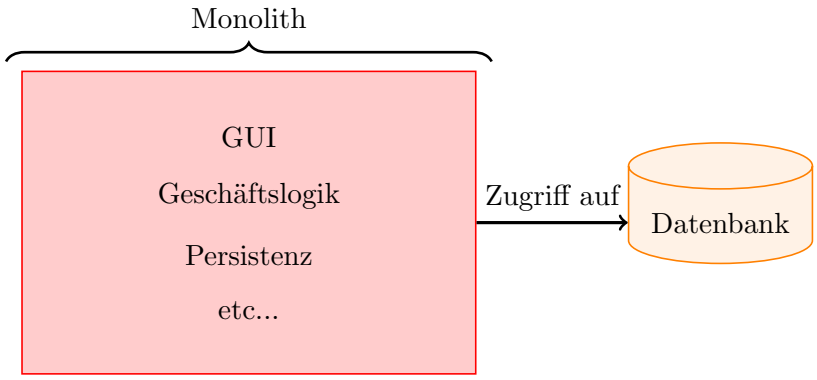

Abbildung 3.1.: Skizze eines Einzelprozess-Monolithen (Newman, 2019)

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 3.1 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 3.2 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).

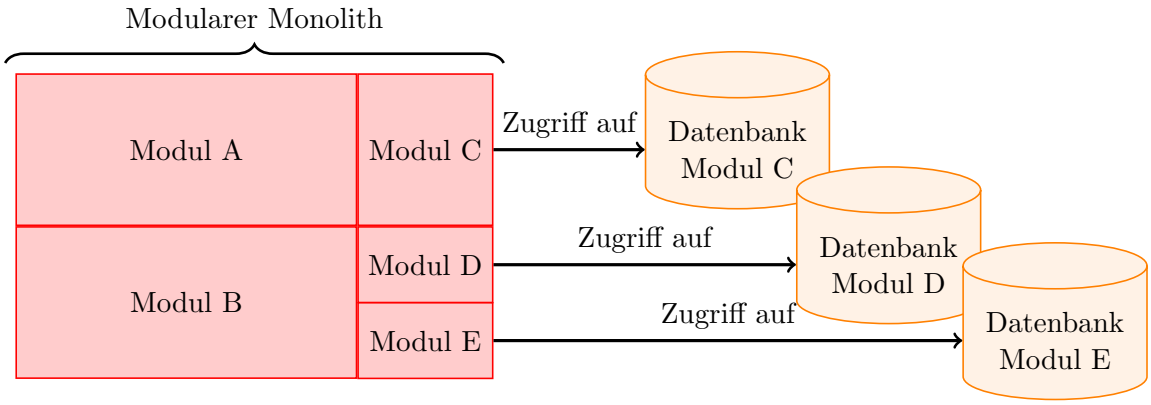

Abbildung 3.2.: Skizze eines modularen Monolithen (Newman, 2019)

Verteilter Monolith

Der verteilte Monolith ist weniger ein Architekturstil, als vielmehr ein unerwünschtes Artefakt, welches aus nicht eingehaltenen, spezifischen Entwurfsprinzipien der 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 3.1.2 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 Abschnitt 4.2 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 3.1.2 und Unterabschnitt 3.1.3 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.

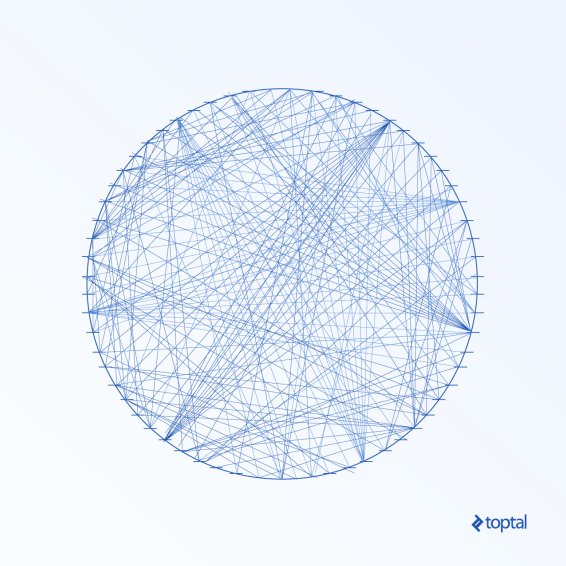

Abbildung 3.3.: Darstellung der grenzenlosen Kommunikation der Komponenten eines *Big Ball of Mud* (Gadzinowski, 2017)

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 3.3 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 3.4 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).

3.1.2. 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).

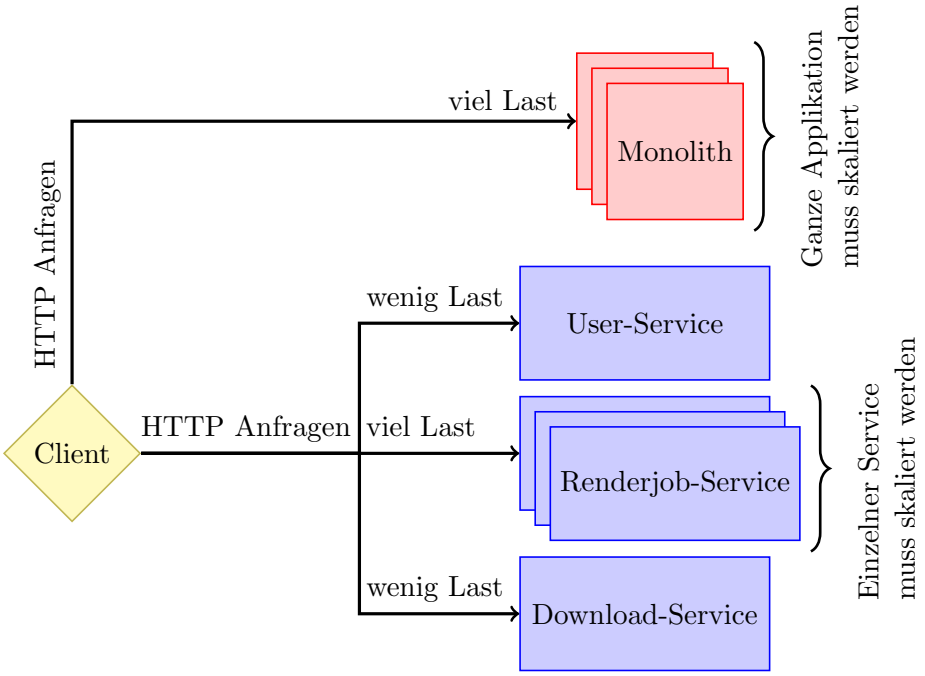

Abbildung 3.4.: Gegenüberstellung der Skalierungsform eines Monolithen und eines verteilten Systems

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 Unterabschnitt 3.4.3 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).

3.1.3. 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).

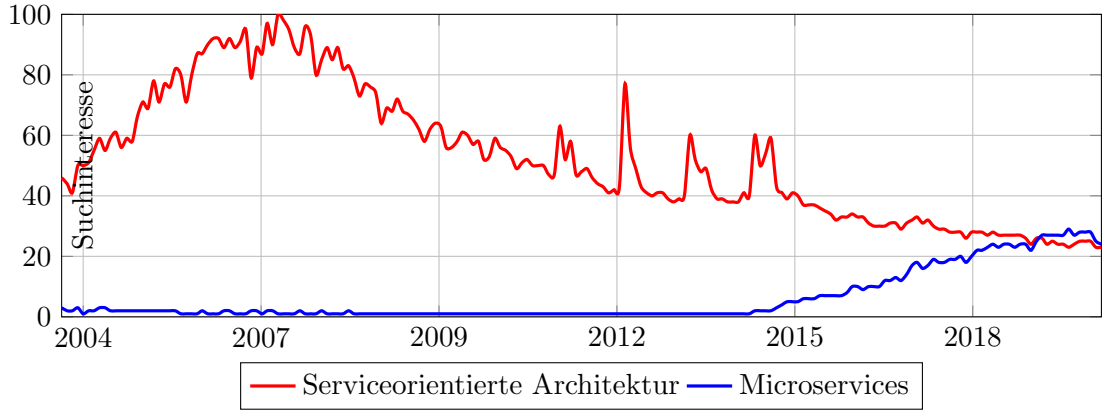

Abbildung 3.5.: Suchinteresse an den Architekturstilen Microservices und Serviceorientierte Architektur ab 2004 im weltweiten Vergleich (Google LLC, 2020)

Wie Abbildung 3.5 zeigt, erhöhte sich die Suchanfrage in Google nach SOA stetig bis zu einem Hochpunkt im Jahr 2007. Wie in Unterabschnitt 3.1.2 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).

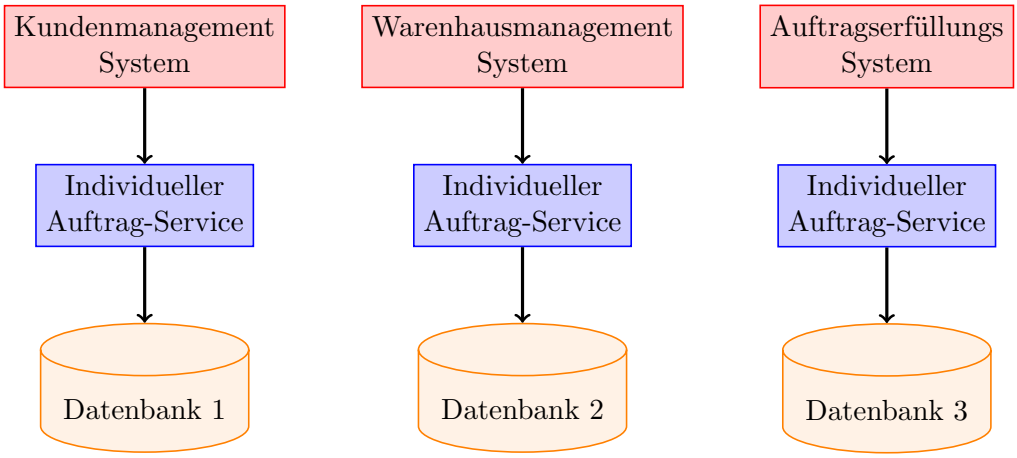

Abbildung 3.6.: Verschiedene Unternehmenssysteme, die jeweils ihre eigene Datenbank und Implementierung eines Auftrag-Service besitzen (Richards, 2016)

SOA benutzt einen *share-as-much-as-possible* Grundsatz, während sich Microservices auf einen *share-as-little-as-possible* Stil beziehen. Abbildung 3.6 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 3.7 dargestellt ist, gelöst werden (Richards, 2016; Thomas & Hunt, 2019).

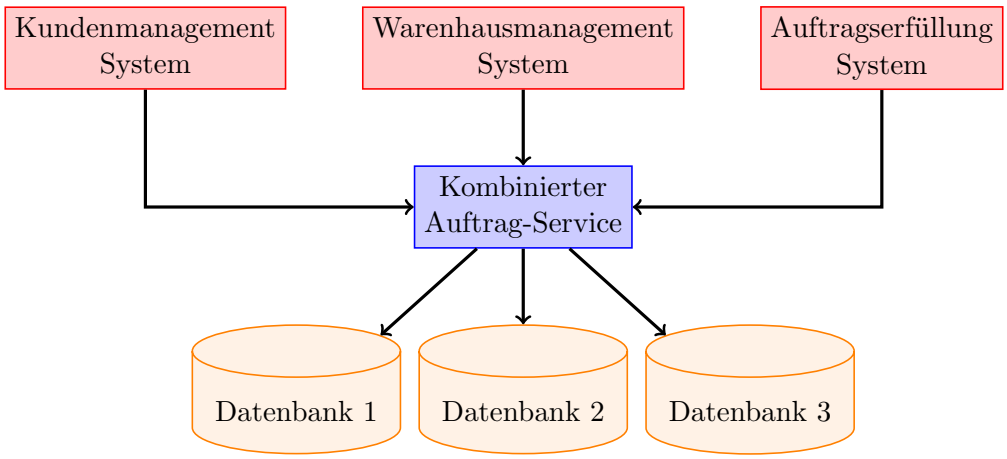

Abbildung 3.7.: Verschiedene Unternehmenssysteme, die eine Servicekomponente teilen, welche alle gebrauchten Datenbanken kombiniert (Richards, 2016)

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 3.1.2 erläutert wurde. Die Software droht damit sich, wie in Unterunterabschnitt 3.1.1 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 3.3 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 Unterabschnitt 3.4.1 kurz erläutert. Weitere Entwurfsmuster von Microservices werden in Abschnitt 3.4 behandelt (Takai, 2017, S. 18–20; Conway, 1968, S. 31; Newman, 2015; Fowler & Lewis, 2014).

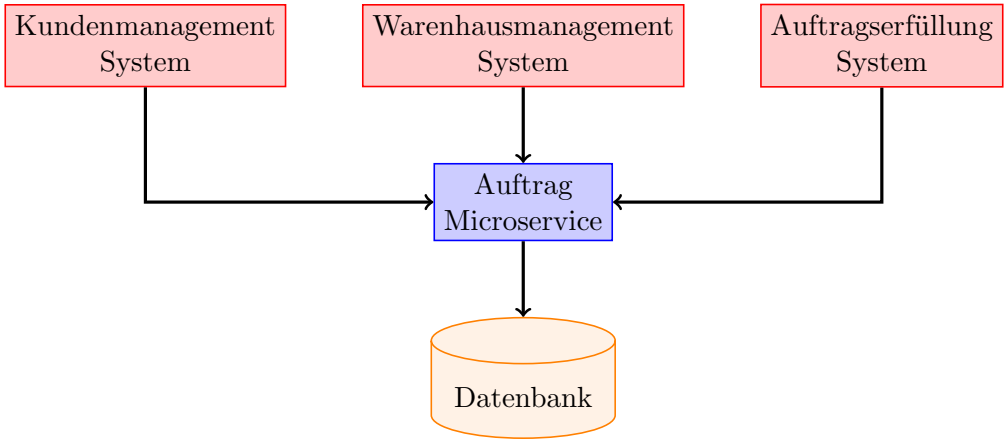

Abbildung 3.8.: Verschiedene Unternehmenssysteme, die einen Microservice teilen, welcher eine Datenbank besitzt und mithilfe von REST eine bestimmte Datenform von einkommenden Anfragen erwartet

Abbildung 3.8 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 3.9 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.

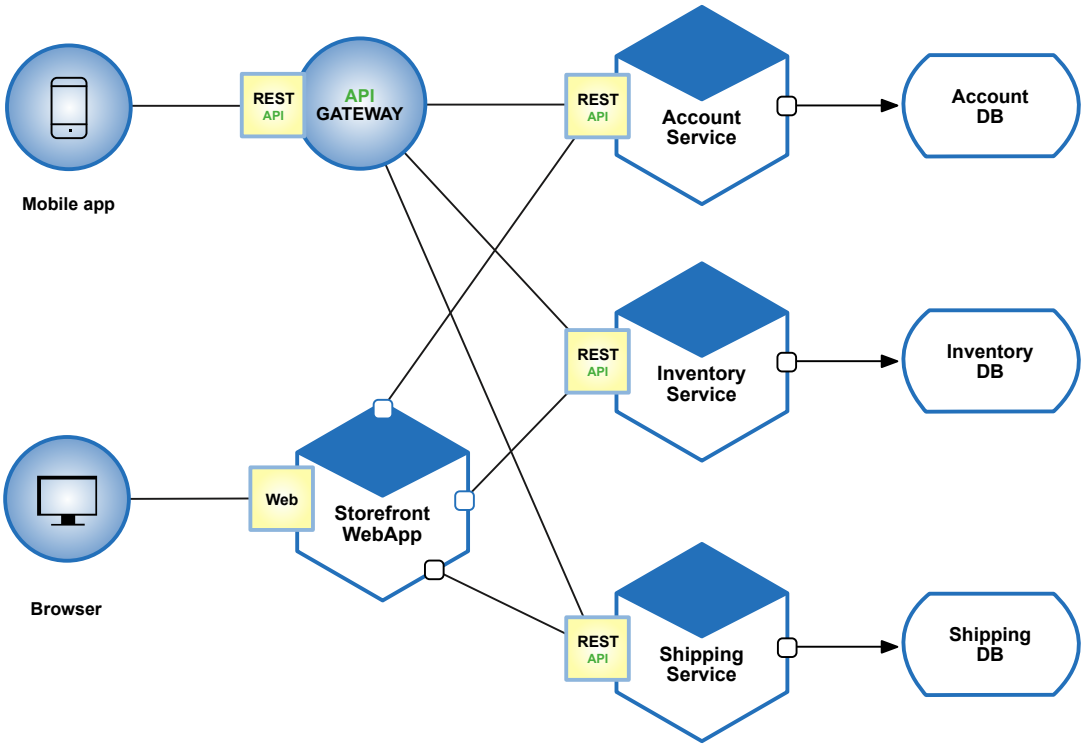

Abbildung 3.9.: Eine typische Microservice-Architektur eines fiktiven E-Commerce Systems (Richardson, 2019b)

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 3.4 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 3.2 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.

3.2. Container vs virtuelle Maschinen

Wie in Unterabschnitt 3.1.2 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 3.1.3 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 3.10 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.

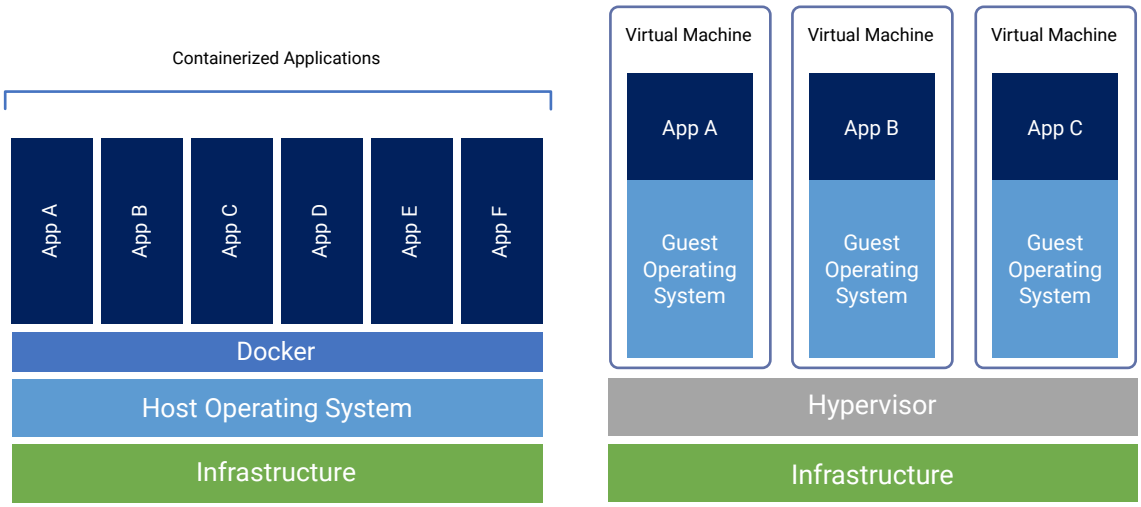

Abbildung 3.10.: Gegenüberstellung der Aufbauweise von Containern (links) und virtuellen Servern (rechts) (Fong, 2018)

Wie die Abbildung 3.10 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 3.10 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).

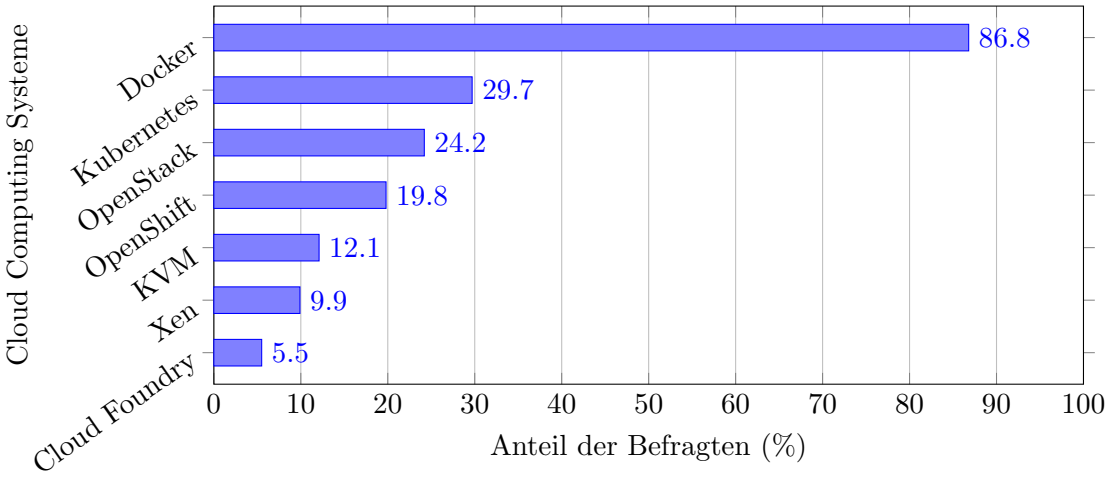

Abbildung 3.11.: Unter 91 Schweizer Unternehmen durchgeführte Umfrage zur Verwendung von Open Source Cloud Computing Systemen (Stürmer & Gauch, 2018)

Abbildung 3.11 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 3.11 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.

3.3. Domain-Driven-Design

Wie in Unterabschnitt 3.1.2 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).

3.3.1. 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).

3.3.2. 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 3.12 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).

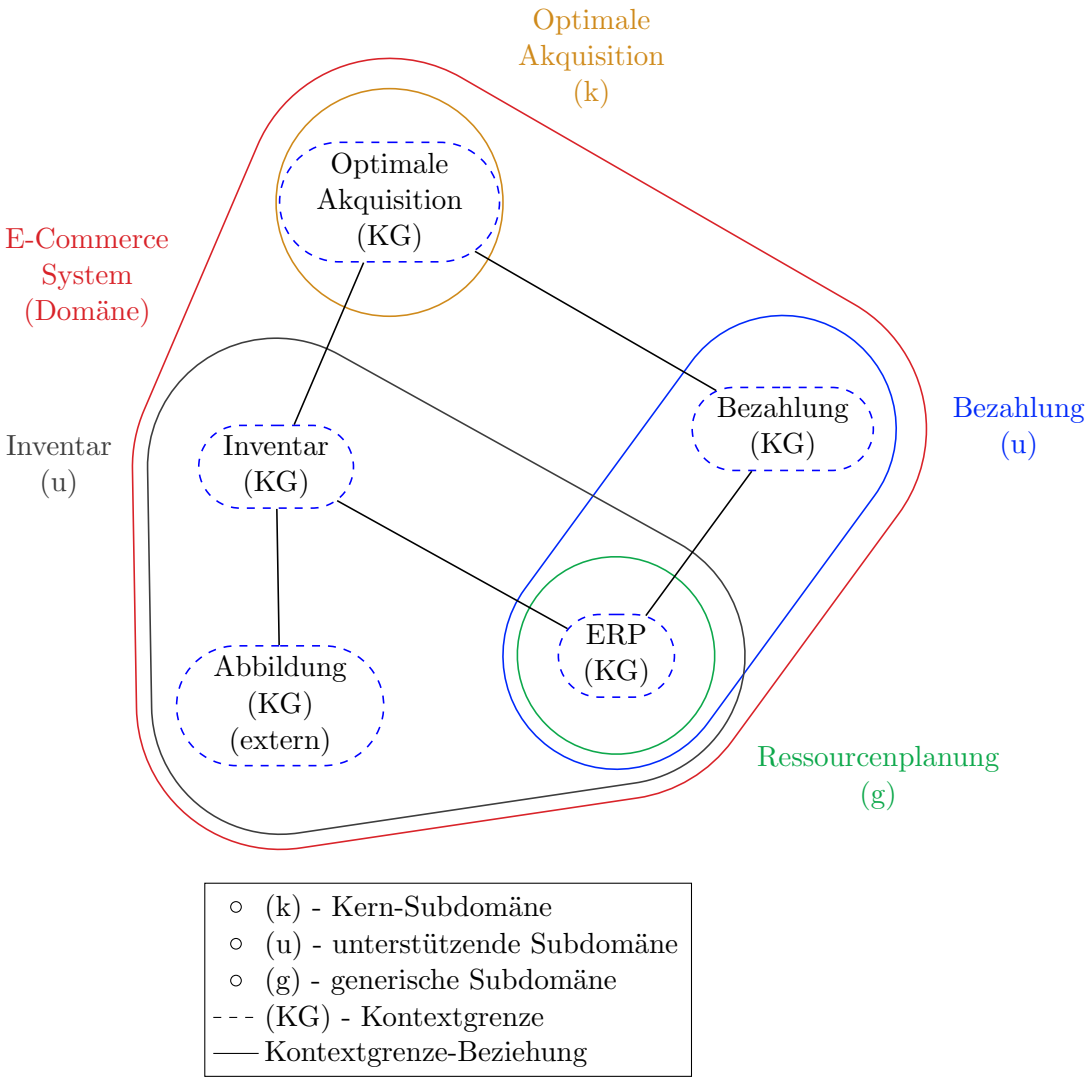

Abbildung 3.12.: Beispiel einer Kontextkarte anhand eines E-Commerce Systems mit einer Domäne, unterstützenden und generischen Subdomänen, einer Kerndomäne, Kontextgrenzen und Beziehungen zwischen den Kontextgrenzen. Die Subdomänen erstrecken sich teilweise über mehrere Kontextgrenzen (Vernon, 2013, S. 58)

Zum Überführen eines *Problemraumes* in einen *Lösungsraum* (engl. *Solution Space*) können Subdomänen durch Kontextgrenzen voneinander sprachlich abgegrenzt werden. Wie in Abbildung 3.12 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 3.12, 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 3.12 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.

3.4. Microservice Entwurfsmuster

Nachdem das Konzept und die Komponenten von DDD im vorigen Abschnitt erläutert wurden, widmet sich dieser Teil den für diese Arbeit als relevant zu betrachtenden Entwurfsmustern von Microservices. Auch wenn diese viele Vorteile erbringen, fügen Services in verschiedenen Bereichen weitere Komplexität hinzu. Diese Bereiche sind zum Beispiel

Kommunikation zwischen Services, sowie *Sicherheit* und *Auffindbarkeit* der Services in einem Netzwerk. Abbildung A.2 liefert eine Übersicht über die wichtigsten Entwurfsmuster für Microservices. In Abschnitt 4.1 werden diese Bereiche weiter destilliert und auf den praktischen Teil dieser Arbeit angewandt.

3.4.1. Kommunikation

Microservices sind einzelne Prozesse, die unabhängig voneinander agieren sollen. Trotzdem müssen einige Services miteinander kommunizieren und Daten austauschen. Diese Kommunikation kann allerdings nicht wie im Monolithen durch einfache Methodenaufrufe passieren, da Microservices in einem Netzwerk verteilt sind. Deshalb müssen eine interne und eine externe Kommunikation über verschiedene Implementierungen stattfinden. Es werden im Folgenden verschiedene Kommunikationsmöglichkeiten erläutert.

Externe Kommunikation

- **Direkte Client-zu-Service-Kommunikation:** Ein Client muss jeden Service direkt anfragen, wodurch dieser alle IPs und Ports kennen muss, die von den Services zur Kommunikation freigegeben werden. Diese Form der Kommunikation hat allerdings zur Folge, dass das *Frontend* sehr stark an das *Backend* gekoppelt ist.
- **API Gateway:** Wie in Abbildung 3.13 dargestellt, vereint ein *API Gateway* alle APIs, die von Microservices zur Nutzung des Frontends benutzt werden können. Dem Frontend-Entwickler erscheint die API des Systems monolithisch, da er nur eine URL mit Direktiven benutzen muss, sofern bei der API *REST* genutzt wurde. API Gateways werden häufig mit einem *Service-Discovery-Werkzeug* benutzt, damit es Anfragen auf die dahinterliegenden Services verteilen kann. Ein API Gateway kann auch für die interne Kommunikation genutzt werden.
- **Backends for frontends:** Hier handelt es sich um eine Variation eines API Gateways. Dabei wird für jedes Gerät, welches ein Frontend darstellen kann, ein eigenes API Gateway erstellt. Anstatt ein API Gateway zu haben, das auf alle Anfragen reagiert, hat man zum Beispiel ein Web-App API Gateway für die dazugehörige Webapplikation, ein Mobile API Gateway für Smartphone-Nutzer und ein Public API Gateway für externe Entwickler.

(Richardson, 2018; Richardson & Smith, 2016, S. 15–18)

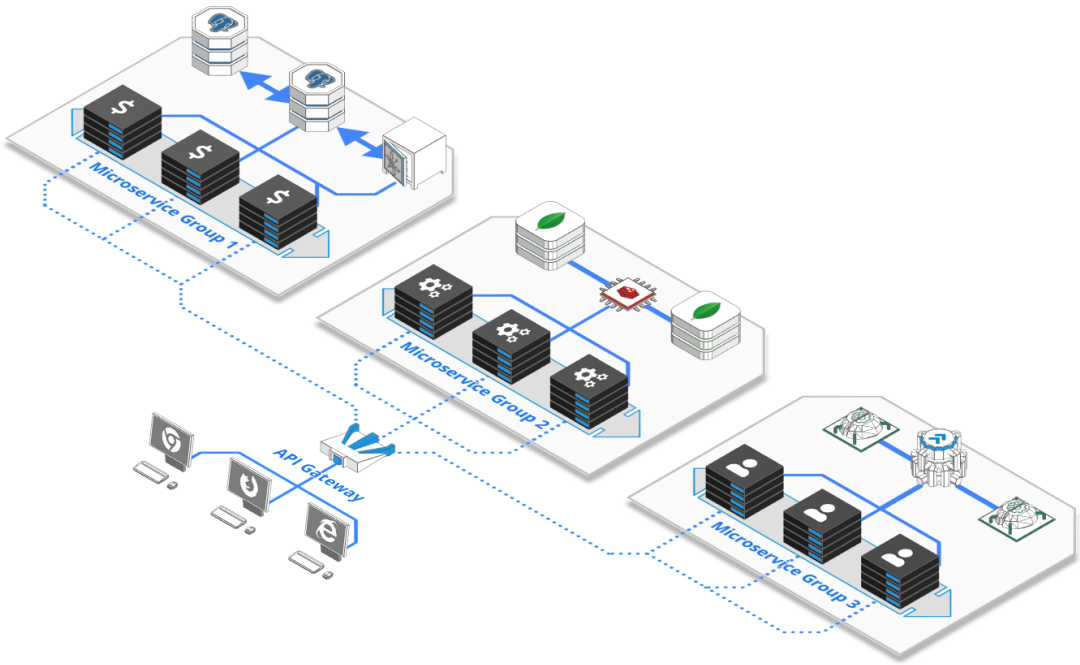

Abbildung 3.13.: Darstellung von Clients, die einen Server mit einem API Gateway anfragen, welches die Anfrage auf die jeweiligen Microservices aufteilt (Hempel, 2019)

Interne Kommunikation

- **Direkte Servie-zu-Service-Kommunikation:** Ähnlich wie zur *Client-zu-Service-Kommunikation* kommunizieren hier Services direkt miteinander. Durch das Festlegen von IP-Adresse und Ports der genutzten Services werden diese miteinander gekoppelt, weshalb eine solche Form der *Inter-Service-Kommunikation* zu vermeiden ist.
- **Message Queue:** Wie in Abbildung 3.14 gezeigt, ist eine *Message Queue* bzw. ein *Message Broker* ein Service, der Nachrichten von anderen Services über ein Netzwerkprotokoll annimmt und ähnlich wie beim *Beobachter-Muster* an abonnierte Services verteilt. Die Services senden eine Nachricht an eine bestimmte Message Queue, um Events bei anderen Microservices auszulösen. Sender und Empfänger werden dadurch voneinander getrennt.

(Richardson, 2018; Richardson & Smith, 2016, S. 50–51)

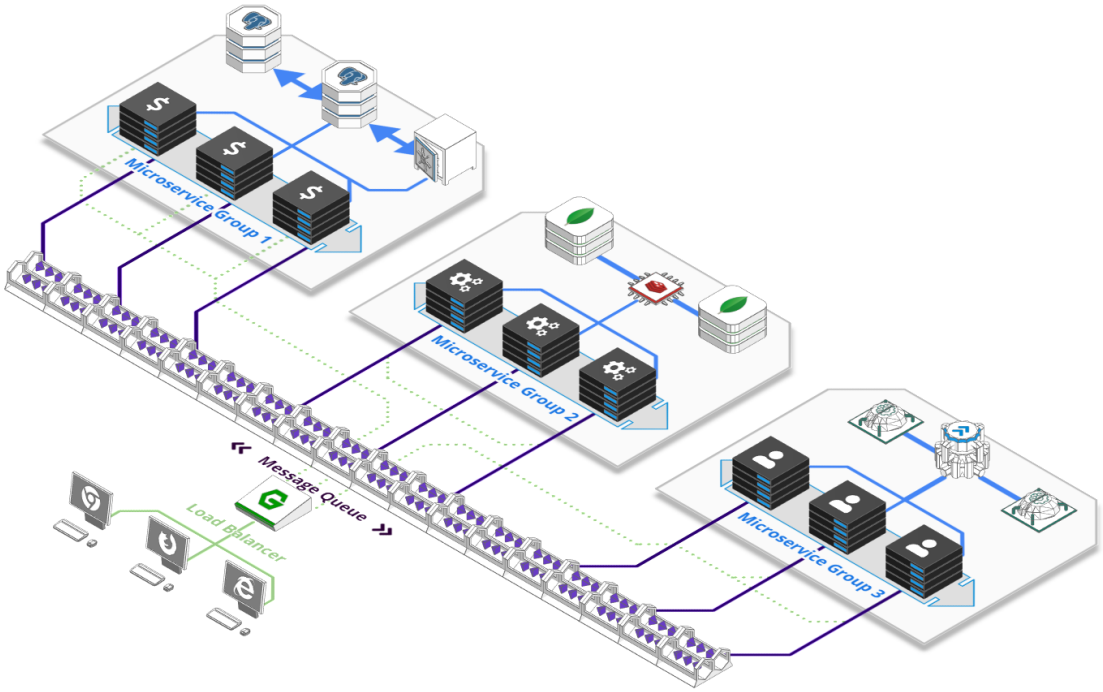

Abbildung 3.14.: Eine Message Queue, die Nachrichten von Services zu anderen leitet (Hempel, 2019)

Kommunikationsstile

- **REST:** Dies ist ein Architekturstil, der Systemen eine oftmals HTTP-basierte API hinzufügt. Die Hauptmerkmale einer REST-basierten API sind Zustandslosigkeit, eine einheitliche Schnittstelle und Addressierbarkeit. Solche APIs können mithilfe der typischen *CRUD-Operationen* über einen Uniform Resource Identifier (URI) mit verschiedenen Direktiven angesprochen werden.

- **GraphQL:** Neben REST wird von Facebook seit 2015 die *Graph Query Language* entwickelt. Anders als bei einer REST-Schnittstelle gibt es hier nur eine URI, auf der alle Anfragen auflaufen. Bei *GraphQL* werden keine gewöhnlichen HTTP-Anfragen mit CRUD-Operationen versendet, sondern eine eigene Sprache, die an die JavaScript Object Notation (JSON) erinnert. Die Versendung dieser Sprache findet in der Regel über die HTTP-Methode *POST* statt.

  Während durch *GraphQL-Queries* Daten von einem Server angefragt werden können, bieten *GraphQL-Mutations* einen Weg, um Daten zu modifizieren. GraphQL-Queries bilden somit die CRUD-Operation *Read* ab, während GraphQL-Mutations in sich die

CRUD-Operationen *Create*, *Update* und *Delete* vereinen. Diese werden wie REST-Endpunkte vom Server vordefiniert und können mit einem *GraphQL-Client* angefragt werden.

Mittels dieser Notation kann man deklarative Datenanfragen versenden. Dieses *Schema*, welches an den Server übermittelt wird, muss dann mithilfe von *Resolvern* aufgelöst werden, die anschließend Daten aus der Datenbank holen. Der Server filtert diese vor und schickt dann nur die spezifisch angefragten Daten zurück. Code 1 zeigt in Zeile 1–11 beispielhaft ein vom Server definiertes Schema, welches vom Client mit der Abfrage in Zeile 13–20 dann konsumiert wird.

Oftmals wird GraphQL als ein Ersatz für REST angesehen, weshalb beide Stile des Öfteren miteinander verglichen werden. Allerdings fehlen bei GraphQL einige Eigenschaften, die REST besitzt. GraphQL serialisiert seine Rückgabedaten in der Regel mit *JSON*, wodurch Themen wie ein Download von Binärdaten nicht möglich sind. Hierbei bleibt jedoch durch das junge Alter des Themas abzuwarten, wie sich GraphQL weiterentwickeln wird.

(Fielding, 2000, S. 76–86; Johansson, 2017, S. 20–22; Helgason, 2017, S. 7–10; Facebook, 2018)

```
1  # SERVER: Queries definition
2  type Query {
3    getUserBy(id: ID!): User!
4  }
5
6  # SERVER: User type definition
7  type User {
8    id: ID!
9    name: String!
10   email: String!
11 }
12
13 # CLIENT: Actual GraphQL Query
14 query {
15     getUserBy(id:ID!) {
16         id
17         name
18         email
19     }
20 }
```

Code 1: GraphQL-Schema eines Servers und GraphQL-Query eines Clients

3.4.2. Überwachung

Da jeder Service in seinem eigenen isolierten Kontext existiert, müssen diese eine Möglichkeit haben nach außen hin zu kommunizieren, falls etwas im System fehlschlägt. Deshalb braucht ein verteiltes System ein hohes Maß an Überwachungsmöglichkeiten.

- **Log Aggregation:** Um das System überwachen zu können, sollten Microservices über *Logs* neueste Ereignisse im System an einen zentralisierten *Logging-Service* senden. Dort werden dann die Logs zusammengefasst und können von Entwicklern nach bestimmten Kriterien durchsucht werden.
- **Distributed Tracing:** Dies ist eine automatisch generierte ID, die bei jeder externen Anfrage mitgesendet und zwischen Microservices geteilt wird. Dadurch lässt sich ermitteln, welche Anfrage zu welchem Zeitpunkt etwas im System ausgeführt hat. Die ID muss dabei in jedem Log mitgeführt werden. Dieses Entwurfsmuster kann auch Informationen speichern, wie zum Beispiel die Start- und Endzeit einer Anfrage oder einer Operation, um die Anfrage zu erfüllen.

(Richardson, 2018)

3.4.3. Auffindbarkeit

Damit Microservices sich gegenseitig nutzen können, müssen sich diese in einem dynamischen Umfeld finden können. Ausschlaggebend dafür ist ein *Service-Discovery-System*. Dabei werden Services oftmals in einer *Service-Registry* mit ihrer momentanen IP-Adresse angemeldet. Der IP-Adresse wird dann ein Domain Name System (DNS)-Eintrag zugeschrieben. Für den Fall, dass der angemeldete Service neu gestartet werden muss und eine neue IP-Adresse zugeteilt bekommt, muss diese in der Service-Registry mit dem dazugehörigen DNS-Eintrag synchronisiert werden. Dadurch wird bei der Auflösung dieses DNS-Eintrages die neue IP-Adresse zurückgegeben. Somit können andere Services diesen trotzdem über die Service-Discovery finden. Dieses Entwurfsmuster lässt sich mit einem DynDNS-System vergleichen (Richardson, 2018; Richardson & Smith, 2016, S. 34–37).

3.4.4. Sicherheit

Ein weit verbreitetes Entwurfsmuster für die Sicherheit ist die Anwendung eines *Access Token* zum Authentifizieren von Nutzern. Dabei werden beim Login eines Nutzers dessen spezifische Nutzerdaten mit einem Schlüssel in einen JSON Web Token (JWT) enkodiert. Der Client kann die enkodierten Daten zwar ohne den Schlüssel nicht auslesen, den JWT allerdings bei jeder Anfrage an den Server mitsenden, bei dem die Daten extrahiert und der Client authentifiziert werden kann (Richardson, 2018).

3.4.5. Datenbank-Architektur

Die Datenbank kann entweder in eine *Shared Database* oder *Database per Service* eingeteilt werden. Wie die Shared Database schon aussagt, gibt es hier nur eine Datenbank, die jeder Service mitbenutzt. Allerdings läuft man hierbei Gefahr, dass Microservices auf andere Teilbereiche der Datenbank zugreifen, welche eigentlich von anderen Microservices verwaltet werden. Dadurch ist eine Isolation der Services nicht mehr gegeben. Eine Database per Service ist für Microservices demnach angebrachter, um deren Isolation beizubehalten. Dabei können allerdings durch *distributierte Transaktionen* weitere Herausforderung in der Datenkonsistenz auftreten (Richardson, 2018).

4. Konzeptionierung und Implementierung

Ein Remote-Zugang bei einer Applikation involviert in seiner Grundstruktur, dass Daten vom Client zur Applikation übertragen, dort prozessiert und anschließend die modifizierten Daten zurückgesendet werden müssen. Diese Hauptaspekte der Daten-Transition und der Daten-Prozessierung sollten vom System abgedeckt werden, damit letztendlich eine komplette Funktionalität des Systems sichergestellt werden kann.

Nachdem die Grundlagen für eine Microservice-Architektur erläutert wurden, wird in den folgenden Unterabschnitten die Softwarearchitektur betrachtet, welche für das Backend von rrRemote von Relevanz ist. Dabei wird sowohl auf den Entstehungsweg als auch auf verschiedene Entscheidungsgründe für die Architektur eingegangen.

4.1. Evaluierung der Softwarearchitektur

Um herauszufinden, welche Microservices für eine erste Version der Applikation implementiert werden sollten, wurde zuerst über mehrere Iterationen eine Kontextkarte entwickelt, welche die Domäne in weitere Subdomänen und Kontextgrenzen aufteilt. Diese Kontextkarte wird anhand von Abbildung 4.1 dargestellt.

Die Domäne *Remote Rendern* beinhaltet mehrere verschiedene Kontextgrenzen. Wie der Name bereits aussagt, beschäftigt sich diese Domäne sowohl mit der Linguistik des Renderings im Sinne der Filmindustrie als auch mit der technischen Seite eines Remote-Controllers. Demnach entstehen die Kontextgrenzen *Generierung* und *Logistik*. Ersteres beschäftigt sich mit Bezeichnungen wie *Render-Layer* und *Render-Apps*, während sich zweiteres mit Terminologien wie *Streams* und *Files* auseinandersetzt. Dies bedeutet allerdings nicht, dass gewisse Wörter einzigartig sind und nur in ihrem eigenen Kontext existieren. Die Kontextgrenze der *Logistik* kann durchaus wie die Kontextgrenze der *Generierung* eine Bedeutung für Renderjobs aufweisen, welche sich allerdings in ihren Attributen unterscheiden.

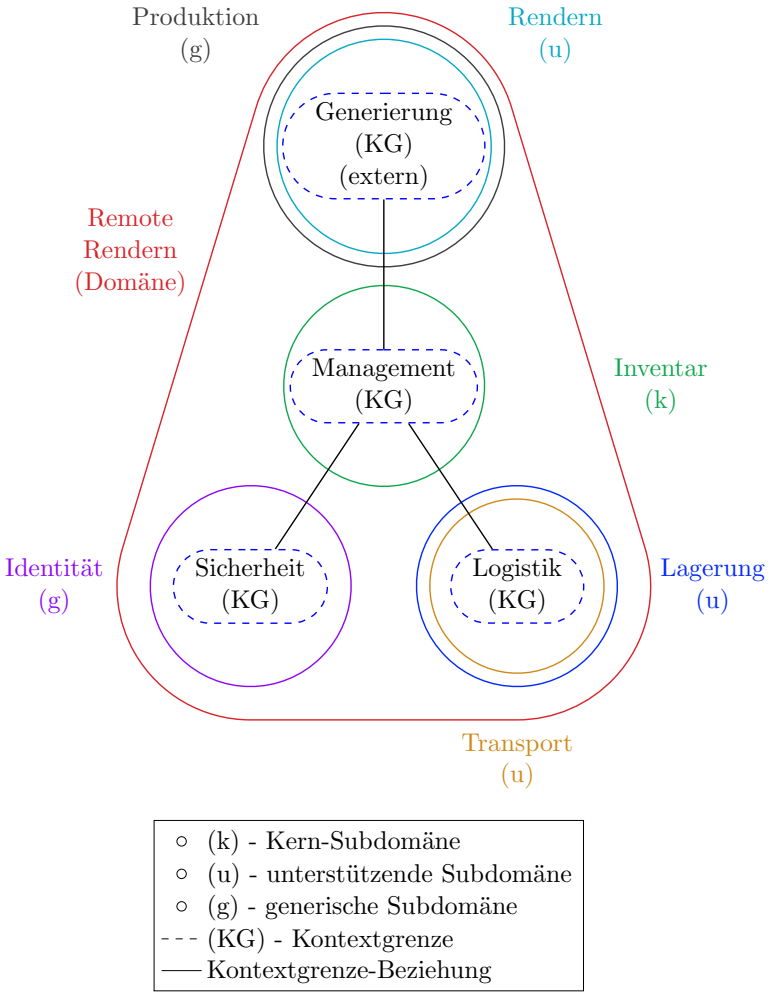

Abbildung 4.1.: Kontextkarte der zu entwickelnden Software der Domäne *Remote Rendern*

Die *Logistik* ist in der Applikation dafür verantwortlich ankommende Dateien in MinIO und dem lokalen Dateisystem zu verwalten. Sie ist allerdings nicht dazu berechtigt die Dateien im Kontext der Filmindustrie und der *Generierung* prozessieren und validieren zu lassen. Stattdessen existiert aus diesem Grund der *Management-Kontext*. Dieser bekommt für jeden Renderjob relevante Daten von der *Logistik*, um diesen validieren und weiter prozessieren zu können. Dieser ist vergleichbar mit einem Object-Relational Mapper (ORM), da die einkommenden und zuerst als generisch betrachteten Dateien zum Rendern zuerst auf die Regelungen der On-Premise Software Royal Render angeglichen werden müssen. Die Kontextgrenze *Sicherheit* ist eine zusätzliche linguistische Grenze, damit die Anzahl an Künstlern, die diese Renderfarm nutzen können, überschaubar bleibt.

Sobald die Kontextgrenzen bestimmt wurden, kann in der Kontextkarte von Abbildung 4.1 die Domäne des *Remote Renderns* in Subdomänen unterteilt werden. Dabei bildet die Subdomäne *Inventar* den Kern der Applikation. Einkommende, generische Daten für Royal Render müssen modifiziert und validiert werden, während später von Royal Render ausgehende Daten von der Applikation zusammengefasst und dem Künstler bereitgestellt werden müssen. Somit bildet die *Inventar-Subdomäne* die Kerndomäne der Applikation und muss darum, wie in Abschnitt 3.3 behauptet, am meisten ausgearbeitet werden. Die nächsten Subdomänen finden sich bei den Kontextgrenzen *Generierung* und *Logistik*.

Die linguistische Grenze *Generierung* wird von zwei Subdomänen geteilt. Die Erste ist die externe generische Subdomäne *Produktion*, welche sich mit der Erstellung von 3D-Szenen, also der Arbeit des Künstlers in einer 3D-Software, beschäftigt. Diese Subdomäne liefert zwar Ressourcen für die Domäne, ist allerdings keine Vorraussetzung für die Funktionalität der Domäne. Die unterstützende Subdomäne *Rendern* beschreibt die On-Premise Software Royal Render. Die Subdomäne ist unterstützend, da ohne das Rendern die Domäne des Remote Renderns nicht erfolgreich sein kann. Es könnten zwar Daten verschoben, jedoch nicht modifiziert werden. Die Annotation *extern* der Kontextgrenze *Generierung* existiert, da die *Produktion* und das *Rendern* als externe Anwendungen zu betrachten sind. Dennoch ist Royal Render kein Drittanbieter-Black-Box-System, wie es in Unterabschnitt 3.1.1 beschrieben wird. Wie sich in Abschnitt 2.2 zeigt, gewährt dieses System externen Entwicklern viele Einblicke in dessen Funktionsweise und bietet diverse Modifizierungsmöglichkeiten. Der Begriff des Renderns findet sich also am häufigsten in der Kontextgrenze der *Generierung* wieder.

Damit Daten gerendert werden können, müssen diese von außerhalb angenommen und in dem zu entwickelnden System persistiert werden. Aus diesem Grund wird die Kontextgrenze der *Logistik* von den beiden unterstützenden Subdomänen *Transport* und *Lagerung* umhüllt. Beide Subdomänen betrachten die einkommenden Daten als generische binäre *Streams* und klassifizieren diese nicht in weiter detaillierte Konzepte wie Renderjobs im Sinne der Filmindustrie. Die Subdomäne des *Transports* beschäftigt sich mit der Annahme und Einlagerung der ankommenden Daten in das System, während sich die *Lagerung* um die Persistenz dieser Daten kümmert.

Die Domäne des *Remote Renderns* kann mit einer *Logistik*, einem *Management* und der *Generierung* von Daten existieren und funktionieren, beinhaltet jedoch keine Restriktionen für die Künstler, die dieses System nutzen werden. Aus diesem Grund existiert die generische Subdomäne der *Identität*. In dieser finden sich die Konzepte von *Accounts* bzw. *Artists*, um das System und jeden Nutzer vor unerwünschten Nutzungsmöglichkeiten zu schützen.

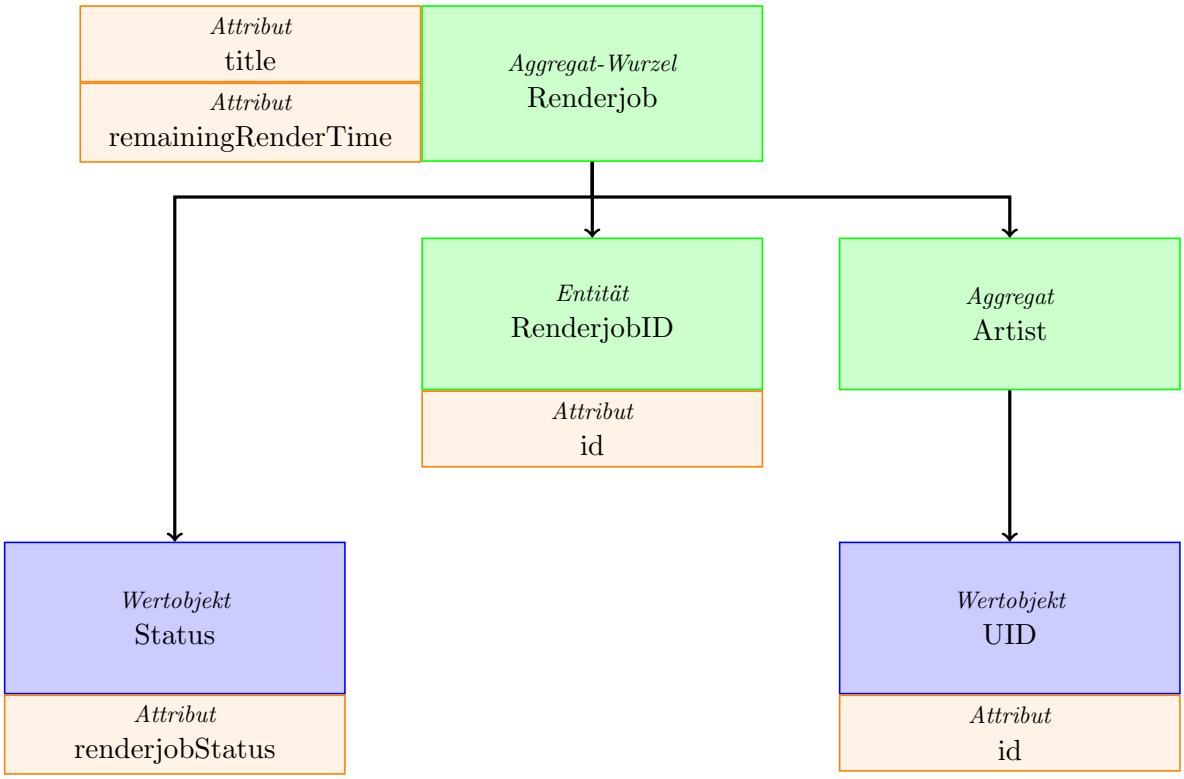

Abbildung 4.2.: Aufbau eines Renderjobs in der *Management-Kontextgrenze* der Domäne *Remote Rendern*

Bei Betrachtung der Abbildung 4.1 fällt auf, dass jegliche Beziehungen von Kontextgrenzen bei der *Management-Kontextgrenze* zusammenführen. Man beachte hierbei, dass Konzepte wie ein *Anti-Corruption Layer*, sowie *Upstream* und *Downstream* nicht in der Abbildung 4.1 verwendet wurden, um diese nicht unnötig zu verkomplizieren. Diese Beziehungslinien bedeuten, dass Konzepte aus anderen Kontextgrenzen in der verknüpften Kontextgrenze wiederzufinden sind. Beispielsweise besteht ein *Renderjob-Aggregat* im *Management* aus den Entitäten *RenderjobID* und *Artist*, dem Wertobjekt *Status* und den Attributen *title* und *remainingRenderTime*. In der *Logistik* besteht ein Renderjob aus den Entitäten *RenderjobID*, *Artist* und mehreren *RenderjobPath-Wertobjekten*. In den verschiedenen Kontexten treten somit die gleichen Bezeichnungen auf, jedoch mit unterschiedlichen Bedeutungen. Abbildung 4.2 und Abbildung 4.3 stellen die unterschiedlichen Bedeutungen des Domänenobjektes *Renderjob* in den Kontexten *Management* und *Logistik* dar. Dabei werden *Aggregate* und *Entitäten* in derselben Farbe dargestellt. Wie in Abschnitt 3.3 beschrieben, ist ein Aggregat nichts weiteres als eine Entität, die weitere Entitäten und Wertobjekte in sich vereint. *Attribute* sind primitive Typen, die zu den jeweiligen Wertobjekten oder Entitäten gehören und keine weitere Abstraktion benötigen.

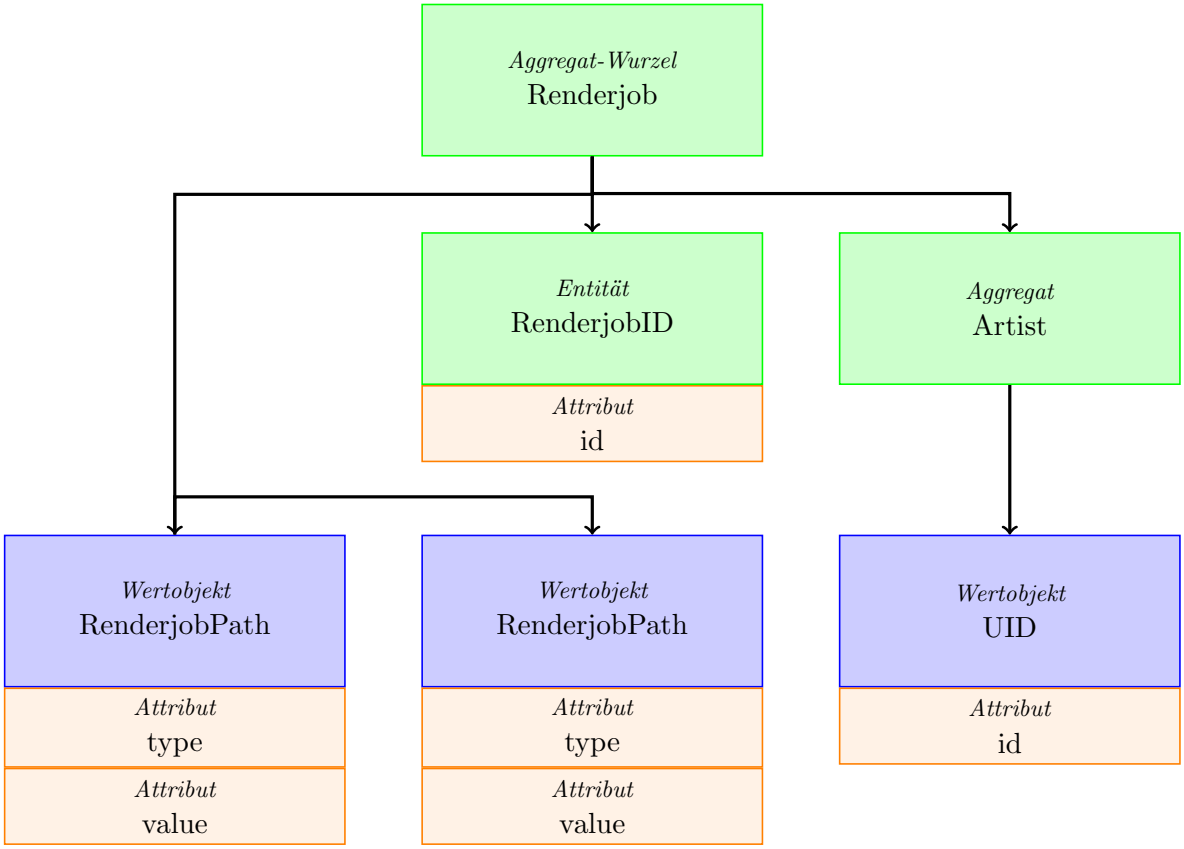

Abbildung 4.3.: Aufbau eines Renderjobs in der *Logistik-Kontextgrenze* der Domäne *Remote Rendern*

Sowohl in Abbildung 4.2 als auch in Abbildung 4.3 finden sich Gemeinsamkeiten der beiden Renderjob-Domänenobjekte. In beiden Varianten gibt es eine *RenderjobID*, mit der das Aggregat identifiziert werden kann. Ebenfalls findet sich bei beiden eine Repräsentation des *Artist-Aggregat* wieder, wodurch *Artists* aus einer anderen Kontextgrenze referenziert werden können. Allerdings beinhaltet der Renderjob der *Management-Kontextgrenze*, anders als in der *Logistik-Kontextgrenze*, das Attribut *remainingRenderTime*, welches über die verbleibende Renderzeit des Renderjobs informieren soll. Ein Renderjob der *Logistik* hingegen definiert zwei *RenderjobPath-Wertobjekte*. Zum Einen wird dort der Speicherpfad zum Renderjob auf dem lokalen Dateisystem festgehalten, während im anderen Wertobjekt der Pfad des Renderjobs im Objektspeicher von MinIO gespeichert wird. Der S3-Objektspeicher *MinIO* wird in Unterabschnitt 4.2.4 kurz erläutert.

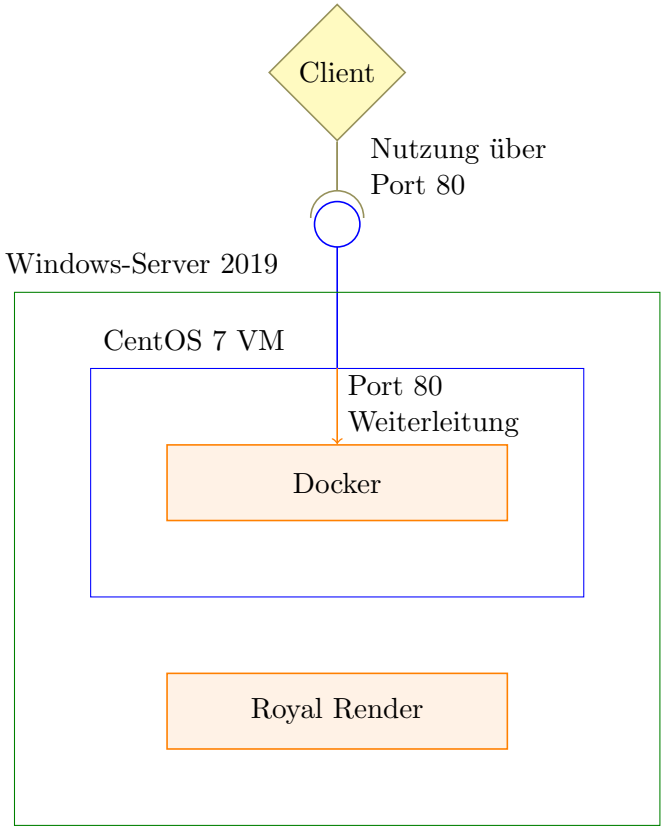

Abbildung 4.4.: Skizze vom Aufbau der gegebenen Serverstruktur

Für das mit dieser Arbeit verknüpfte Softwareprojekt wurde ein Windows Server 2019 mit einer installierten Royal Render Version zur Verfügung gestellt. Der Server wird hauptsächlich als Sammelstelle für studentische Projekte der 3D-Visualisierung inklusive rrServer und Fileserver für Royal Render genutzt. Windows Server können keine linux-basierten Container nutzen, wie in Abschnitt 3.2 besprochen. Da Docker das Host-System nutzt und die für dieses Projekt gebrauchten Technologien auf Linux basieren, muss ein weiteres linux-basiertes System vorrausgesetzt werden. Aus diesem Grund wurde über *Hyper-V*, ein Verwaltungswerkzeug für virtuelle Maschinen, eine VM mit CentOS 7 installiert, auf der die Docker Community-Edition laufen kann. Es wurde sich für CentOS 7 entschieden, da auf der offiziellen Installationsanleitung von Docker, für Docker Engine unter Linux CentOS, explizit auf CentOS 7 verwiesen wird. Es lässt sich somit vermuten, dass Docker unter CentOS 7 stabil läuft. Ebenfalls wurde diese Linux-VM mit den üblichen HTTP-Ports 80 und 443 für das Internet geöffnet, sodass externe Applikationen mit dem Backend von rrRemote kommunizieren und interagieren können. Abbildung 4.4 skizziert diesen Aufbau (Docker, 2019).

Bei der Abbildung 4.4 ist darauf zu achten, dass nur die Linux-VM mit dem Port 80 für das Internet geöffnet wurde. Externe Applikationen können dann über die öffentliche IP-Adresse und den Port 80 der Linux-VM auf ein Backend zugreifen, das auf diesem Port wartet. Beispielsweise stellt Docker in der Grafik ein Backend bereit, welches diesen Port nutzt. Demnach wird eine ankommende Frage über diesen Port an Docker auf denselben Port weitergeleitet. Dadurch, dass nur die Linux-VM für das Internet bereitgestellt wird und diese nur mit dem Windows-Server über HTTP-Anfragen oder Netzwerkfreigaben interagieren kann, ist der Windows-Server weiterhin vor externen Zugriffen geschützt.

Betrachtet man die Kontextkarte aus Abbildung 4.1, so lassen sich aus dieser verschiedene Microservices ableiten. Die über mehrere Iterationen entwickelte Softwarearchitektur des Backends von rrRemote findet sich in Abbildung 4.5. Da eine Kontextgrenze mehrere Subdomänen in sich vereinen kann, ist es für größere Projekte nicht zu empfehlen einen Microservice pro Kontextgrenze zu entwickeln. Laut Chris Richardson empfiehlt sich die Projektion einer Kontextkarte auf Microservices mithilfe der *Decompose by Subdomain* Methode. Dabei wird pro Subdomäne ein Microservice entwickelt, der sich auf die ihm zugeteilte Subdomäne spezialisiert (Richardson, 2018).

Auffällig ist bei der Abbildung 4.5, dass die Kontextgrenze der *Generierung* die einzige externe linguistische Grenze ist. Für die Subdomäne *Produktion* können keine automatisierten Systeme, sondern nur Künstler eingesetzt werden, die das zu entwickelnde System nutzen wollen. Die *Rendern* Subdomäne besteht aus der On-Premise Software Royal Render und ist somit ein schon implementierter Service, der durch weitere Services orchestriert werden kann. Somit muss der Kontext der *Generierung* technisch nicht weiter abgedeckt werden.

Die Subdomäne *Identität* spezialisiert sich auf die Eingrenzung der Nutzungsmöglichkeiten der Künstler, die das zu entwickelnde System nutzen wollen, sodass jene Künstler nur Einfluss auf ihre eigenen Renderprojekte haben. Somit kann diese Subdomäne auf einen *Artist-Service* projiziert werden.

Die *Transport-Subdomäne* beschäftigt sich mit der Annahme und Persistierung von ankommenden Daten aus der Produktion. Die tatsächliche Persistenz dieser Daten wird allerdings durch die *Lagerung-Subdomäne* erzielt. Die *Transport-Subdomäne* kann somit zum *File-Service* gehören, während die *Lagerung* auf den Objektspeicher *MinIO* zugewiesen werden kann, wie in Unterabschnitt 4.2.4 behandelt wird. In Abbildung 4.5 ist MinIO, wie auch *Firebase Auth*, *Firebase Firestore*, Royal Render und das lokale Dateisystem als Datenbank dargestellt, da in der entwickelten Softwarearchitektur diese Komponenten als reine Datenhaltungscontainer betrachtet werden.

Zuletzt steht die *Inventar-Subdomäne* und die damit einhergehende *Management-Kontextgrenze* im Mittelpunkt der Kontextkarte von Abbildung 4.1. Da es sich hierbei um die Formung der vorher generisch betrachteten Dateien zu Renderjobs handelt und diese in die *Generierung* zu transportieren sind, kann diese Subdomäne auf einen *Renderjob-Service* projiziert werden.

Zusammengefasst lassen sich die folgenden drei zu implementierenden Microservices aus der Kontextkarte von Abbildung 4.1 ableiten: *File-Service*, *Renderjob-Service* und *Artist-Service*. Diese Services werden in der Abbildung 4.5 in Blau dargestellt.

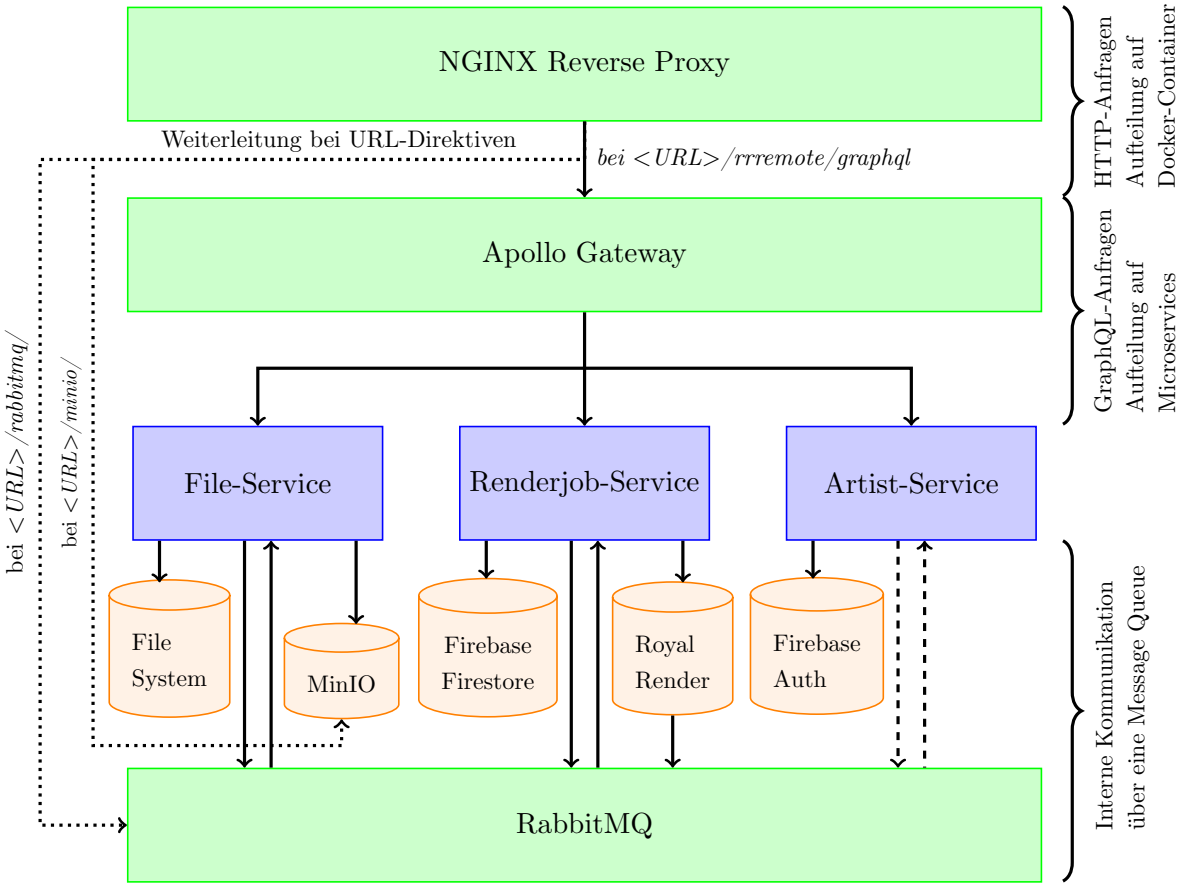

Abbildung 4.5.: Softwarearchitektur des zu entwickelnden Backends für rrRemote

Eine deklarative Datenakquise ist von Vorteil, wenn unter anderem mit aufwändigen Anfragen gearbeitet wird. Da in früheren Projekten mit GraphQL gearbeitet wurde und diese Technologie die Möglichkeit bietet von Datengruppen gewisse Datenfelder und damit größere Berechnungen auf dem Server auszuschließen, implementieren die Microservices eine GraphQL-API auf Basis von *Apollo Federation*. Diese APIs werden mithilfe eines *Apol-*

lo Gateways akkumuliert, sodass das Frontend eine monolithische API des abgebildeten Systems angezeigt bekommt. Die GraphQL-Anfrage kann dann auf den jeweiligen Service aufgeteilt werden. Da Apollo Federation benutzt wird, muss ein Apollo Gateway zur Zusammenführung von Apollo Federation-GraphQL-Schemata vorrausgesetzt werden. Die Funktionsweise und Verwendung von Apollo Federation in rrRemote, wird in Unterabschnitt 4.2.1 behandelt.

Da unter anderem von außerhalb auf verschiedene Docker-Container zugegriffen werden muss und das Apollo Gateway nur GraphQL-Anfragen interpretieren kann, existiert vor dem Apollo Gateway ein *NGINX Reverse Proxy* und somit ein weiteres API Gateway. Dieses kann wie bei einer REST-Architektur, je nach URL-Direktiv, einkommende HTTP-Anfragen auf verschiedene Services aufteilen. Beispielsweise ist in der Abbildung 4.5 mit der punktierten Linie dargestellt, dass eine Anfrage auf der URL *www.test.de/minio/* den Nutzer zu der im System vorhandenen MinIO-Instanz weiterleitet, während die URL *www.test.de/rabbitmq/* zu der RabbitMQ-Instanz führt.

Der NGINX Reverse Proxy, das Apollo Gateway, die RabbitMQ-Instanz, als auch MinIO sind hierbei vorbereitete und veröffentlichte Docker-Container, die auf der vorgegebenen Serverstruktur bereitgestellt werden können. In der Abbildung 4.5 sind solche Container grün markiert, mit Ausnahme von MinIO, da dieser Service für die Datenerhaltung von Renderjobs zuständig ist.

Zuletzt fällt auf, dass die Microservices untereinander nicht direkt verbunden werden. Dies würde eine Kopplung der Services vorraussetzen. Stattdessen nutzen die Services die RabbitMQ-Instanz, welche eine *Message Queue* ist. RabbitMQ ermöglicht die asynchrone Kommunikation zwischen Microservices, da diese eine Nachricht in die Message Queue versenden, aber nicht zwingend auf eine Antwort des angefragten Services warten müssen. Die Message Queue sorgt dann dafür, dass jener angefragte Service diese Nachricht erhält. Bei einer Inter-Service-Kommunikation über das API Gateway und dem HTTP-Protokoll, muss ein Service auf die Antwort des angefragten Services warten. Derweilen ist der anfragende Service nicht zustandslos und somit für weitere Anfragen nicht erreichbar. Sind weitere Microservices abhängig von einem gerade unerreichbaren Service, so kann sich diese Problematik der Unerreichbarkeit auf die weiteren abhängigen Services kaskadieren.

Da damit gerechnet werden muss, dass unter anderem große Datenmengen heruntergeladen und prozessiert werden müssen, was eine längere Zustandsbehaftung eines Microservice zur Folge haben kann, ist eine asynchrone Kommunikation zwischen den Services von Relevanz. Demnach agiert die Softwarearchitektur im Sinne des reaktiven Manifests *nachrichtenorientiert* und *antwortbereit* (Bonér et al., 2014).

4.2. Verwendete Technologien

4.2.1. Apollo

Apollo ist ein Projekt der *Meteor Development Group*. In diesem werden Werkzeuge entwickelt, die das Arbeiten mit GraphQL simplifizieren sollen. Unter anderem liefert Apollo somit einen GraphQL-fähigen Server und einen dazugehörigen GraphQL-fähigen Client, welche respektiv in Backends und Frontends integriert werden können (Blades, Rayzis et al., 2019).

Da in früheren Projekten mit *Apollo Server* als GraphQL-Server gearbeitet wurde und diese Technologie eine große Nutzeranzahl und damit einhergehende Unterstützung mit sich führt, fiel die Entscheidung auf das GraphQL-Server-Framework *Apollo-Server-Express*. Dieses benutzt im Hintergrund das beliebte *REST-API* Framework *Express* und das GraphQL-Server-Framework *Apollo Server 2*, welches stark von *GraphQL-Yoga* inspiriert wurde (Hauser, 2018).

Im Jahr 2019 wurde *Apollo Federation* veröffentlicht. Diese Technologie bietet eine Erweiterung zu dem vorher üblichen *Schema-Stitching-Workflow*, mit welchem man pro Microservice ein eigenes GraphQL-Schema implementieren konnte, was später in einem Service zusammengefügt wurde. Schema-Stitching entstand allerdings aus der Notwendigkeit GraphQL in einer Microservice-Architektur zu implementieren, da es bis Apollo Federation keine Technologie gab, die sich auf diesen Aspekt fokussiert hat. Durch mehrere Problematiken mit Schema-Stitching entstand dann Apollo Federation. Dadurch lässt sich auch pro Microservice ein GraphQL-Schema implementieren, welches später durch ein Apollo Gateway zusammengefügt wird. Mithilfe einer speziellen Syntax kann man nun pro Service GraphQL-Typen implementieren und diese dann von externen Services erweitern lassen. Apollo benennt dies als *Separation of concerns*, da nun spezielle Typen in ihren respektiven, semantisch korrekten Microservices verweilen können.

Ein Beispiel bei dem Projekt rrRemote wäre, dass der *Artist-Service* den Typ *Artist* mit einer *E-mail* und einer *ID* enthält. Der *Renderjob-Service* könnte dann den *Artist-Typ* mit *Renderjobs* erweitern. Abbildung 4.6 zeigt ein typisches GraphQL-Schema, wie es in mehreren GraphQL-Projekten zu finden ist, während Abbildung 4.7 das neue Apollo Federation Äquivalent davon darstellt (Baxley III, 2019).

```
type User {
  name: String
  reviews: [Review]
  recentPurchases: [Product]
}
```

```
type Product {
  name: String
  price: String
  reviews: [Review]
}
```

```
type Review {
  body: String
  author: User
  product: Product
}
```

Abbildung 4.6.: Typisches GraphQL-Schema mit den Typen *User*, *Product* und *Review*. Jedes dieser Typen besitzt eine direkte Referenz zu einem anderen Typ (Baxley III et al., 2020).

Damit das Apollo Gateway die zu nutzenden Microservices immer finden kann, wird im Zusammenhang mit Apollo Federation der *Apollo Graph Manager* benutzt. Dies ist ein Service-Discovery System, welches sich auf GraphQL-spezifische Applikationen spezialisiert hat. Durch diese Technologie benötigt das Apollo Gateway keine vordefinierte statische Liste mit IP-Adressen und Ports der zu betrachtenden Microservices. Stattdessen wird das Apollo Gateway von dem Apollo Graph Manager zur Laufzeit dynamisch über Veränderungen in dem GraphQL-Schema informiert, woraufhin sich dieses aktualisieren kann (Blades, Zionts et al., 2019).

```
type User {
  name: String
}
```

```
type Product {
  name: String
  price: String
}

extend type User {
  recentPurchases: [Product]
}
```

```
type Review {
  body: String
  author: User
  product: Product
}

extend type User {
  reviews: [Review]
}

extend type Product {
  reviews: [Review]
}
```

Abbildung 4.7.: Apollo Federation Äquivalent zur Abbildung 4.6 (Baxley III et al., 2020).

4.2.2. Firebase

In einigen Fällen ist es sinnvoll externe Technologien für ein Projekt einzukaufen. Im Sinne von Domain-Driven-Design ist dies öfter der Fall bei generischen Subdomänen. *Firebase* ist ein von Google entwickeltes Backend as a Service (BaaS)-System, welches viele verschiedene Werkzeuge für Entwickler bereitstellt, die Technologien wie Datenbanken, Authentifizierung, etc. nicht selbst aufsetzen möchten, da es nicht Hauptbestandteil der zu entwickelnden Software ist. Hinzu kommt, dass Firebase ein Preismodell besitzt, welches

Hobbyisten und freien Entwicklern ermöglicht, deren Werkzeuge kostenfrei zu nutzen, solange sich die Anzahl von Anfragen an Firebase in Grenzen halten. Beispielsweise erlaubt *Firebase Firestore* insgesamt 20.000 Schreiboperationen pro Tag. Eine Schreiboperation wäre im Backend von rrRemote beispielsweise ein Statusupdate eines Renderjobs. Daraus ergeben sich bei 100 aktiven Nutzern und 5 Schreiboperationen pro Renderjob 40 Renderjobs je Nutzer pro Tag. Bei nur einem aktiven Nutzer sind 2.000 Renderjobs pro Tag möglich.

Authentifizierung ist eine Technologie, die in vielen Projekten genutzt wird. Da, wie in Abschnitt 4.1 beschrieben, die *Identität* eine generische Subdomäne ist und es viel Zeit in Anspruch nehmen würde eine ausgereifte Authentifizierungsmöglichkeit zu implementieren, wird in der zu entwickelnden Applikation die Technologie namens *Firebase Authentication* verwendet. Dieser Service benutzt einen Token-basierten Authentifizierungsweg mithilfe der Standards *OpenID Connect* und *OAuth 2.0*.

Zuzüglich einer funktionierenden Authentifizierung müssen Daten pro Nutzer persistiert werden, sodass für jeden Nutzer erkennbar ist, welche Renderjobs ihm gehören. Für diesen Fall wird *Firestore* genutzt, welches eine MongoDB-ähnliche Dokumenten-basierte Datenbank zur Verfügung stellt. Da eine solche Datenbank eine objektorientierte Struktur aufweist, ist eine Nutzung dieser mit einer objektorientierten Programmiersprache weniger aufwändig und benötigt kaum Einarbeitungszeit. Hinzu kommt, dass Dokumente in einem solchen Speicher eine JSON-ähnliche Struktur aufweisen. Da, wie in Abschnitt 4.3 noch erklärt wird, *NodeJS* für das Backend von rrRemote benutzt wird, müssen die angefragten Werte später nicht mehr mit viel Aufwand auf Objekte in der Applikation projiziert werden.

4.2.3. Logging

Zur Überwachung eines Microservice Systems ist es wichtig einen konstanten Strom an Informationen der implementierten Services zu bekommen. Dafür eignet sich *structured logging*, also das Speichern von Applikations-logs in einem Datenformat, wie zum Beispiel JSON. Der Vorteil davon ist es, dass später die Logs mit einem Werkzeug wie *Kibana* auf bestimmte Metriken gezielt durchsucht werden können. Aggregiert man die Logs in einer festen Struktur, dann lassen sich daraus auch Datensätze ermitteln, die bei der Analyse der Nutzung der Software helfen können.

Um das Potenzial von structured logging ausnutzen zu können, sollten die Logs zentralisiert festgehalten werden. Für diesen Fall nutzt die Applikation *Logz.io*. Ausschlaggebend für diese Entscheidung war das Preismodell des Anbieters, da dieser pro Tag 1 Gigabyte an Logs akzeptiert, welche für einen Tag persistiert werden. Logz.io bietet ebenfalls eine Integration der Applikation *Kibana*, mit welcher diese generierten Logs dann nach bestimmten Metadaten durchsucht werden können.

Damit in den Microservices Logs erstellt werden können, benutzen diese die Bibliothek *Winston*. Mit dieser lassen sich relativ schnell strukturierte Logs erstellen, die dann in Logz.io hochgeladen werden können. Es wird Winston genutzt, da *Logz.io* eine modifizierte Version dieser Bibliothek namens *winston-logzio* bietet, mit der die generierten Logs ohne Umwege in Logz.io hochgeladen werden können.

Ebenfalls ist Winston, im Vergleich zu dessen Konkurrenten *bunyan*, die wöchentlich am meisten heruntergeladene Logging-Bibliothek, was auf eine hohe Nutzeranzahl und damit Unterstützung schließen lässt. Bunyan hingegen wurde zum aktuellen Zeitpunkt seit drei Jahren nicht mehr aktualisiert. Bei *Winston* liegt das letzte Update ein Jahr zurück.

Da Logs nicht in jeder Situation zu einem Service wie *Logz.io* hochgeladen werden können, kann Winston auch Logs auf dem lokalen Dateisystem festhalten. Damit diese trotzdem nach Logz.io transportiert werden, kann in Docker ein Container mit der Applikation *Filebeat* erstellt werden. Dies ist ein Container, der verschiedene Logging-Dateien beobachtet und bei Veränderungen dieser die veränderten Daten in einem Anbieter wie *Logz.io* aktualisiert.

4.2.4. Speicher

Da die Applikation größere binäre Daten persistieren muss, benutzt diese den Objektspeicher namens *MinIO*. Dieser wirbt damit der weltweit schnellste Objektspeicher zu sein, der bei Lese- und Schreiboperationen Geschwindigkeiten bis zu 183 Gigabyte pro Sekunde erreichen kann. MinIO setzt bei seiner Nutzungsweise auf eine Amazon S3-kompatible Variante. Dabei können sogenannte *Buckets* erstellt werden, in denen Daten gespeichert werden können. Beispielsweise könnte ein jeder Nutzer sein eigenes Bucket besitzen, in dem Projektdaten und deren später erstellten Renderings gespeichert werden. Die Kompatibilität von MinIO mit der Amazon-Web-Service S3 SDK, sowie die von MinIO versprochene Performance, als auch dessen Open-Source-Basis, führten zu der Entscheidung diese Applikation als Persistenzschicht für binäre-Daten in das Backend zu integrieren (Minio, Inc., 2020).

4.3. Umsetzung

Für die Umsetzung des Backends der Applikation rrRemote wird *NodeJS* benutzt, eine in C und C++ entwickelte und auf der Google V8 Engine basierende JavaScript-Laufzeitumgebung. Mithilfe von NodeJS können langlebige Server-Prozesse entwickelt werden, die in JavaScript geschrieben sind. Dabei setzt NodeJS auf einen Einzelprozessdienst mit einem asynchronen *I/O eventing-model*. NodeJS selber kann durch dessen Einzelprozess-Architektur Methoden nur synchron behandeln. Da diese Technologie allerdings auch auf C++ basiert, können als asynchron markierte Operationen an C++-Prozesse abgegeben werden, die dann neben dem JavaScript-Prozess behandelt werden können. Sobald die asynchronen Methoden ausgeführt wurden, wird eine vorher als Parameter definierte *Callback-Funktion* in den Einzelprozess überliefert. Daraufhin kann diese von JavaScript aufgelöst und der Rückgabewert des C++-Prozesses extrahiert werden. Da abzusehen ist, dass die zu entwickelnde Applikation mehr auf I/O-Operationen basiert, JavaScript eine populäre, web-basierte Sprache ist, NodeJS mit Docker einen hohen Grad an Unterstützung bietet und in früheren Projekten mit NodeJS entwickelt wurde, fiel die Entscheidung auf die Anwendung *NodeJS* (Tilkov & Vinoski, 2010, S. 80; Trott et al., 2020; Schonning, Wine et al., 2020; Schonning, Do et al., 2020).

Dabei wird allerdings nicht direkt JavaScript als Programmiersprache genutzt. Stattdessen wurde sich hier für die von Microsoft entwickelte JavaScript Erweiterung *TypeScript* entschieden. JavaScript nutzt einen dynamischen Typisierungsworkflow, weshalb man sich in jenen deklarierten Methoden nicht sicher sein kann, welche Parametertypen der Methode zur Laufzeit übergeben werden. Dies kann zu fehlerhaftem Verhalten und im schlimmsten Fall zu möglichen Angriffspunkten des Systems führen. TypeScript hingegen fügt JavaScript Konzepte wie *Klassen*, *Interfaces* und *Enumerations* hinzu und sorgt damit in der Applikation zur Laufzeit und Entwicklungszeit für eine Typsicherheit. Damit wird eine generell stärkere Sicherheit für das komplette zu entwickelnde System erzeugt. Zuzüglich kann die Verwendung von TypeScript dafür sorgen, dass IDEs wie Visual Studio Code Autovervollständigung von JavaScript-Funktionalitäten anbieten.

Des Weiteren wird zur Komposition der zu entwickelnden Microservices das Werkzeug *Docker Compose* genutzt. Hier kann man mit einer YAML-Datei, wie sie im Anhang unter Code 3 zu finden ist, die Services und deren Startoptionen festlegen, welche für die Software vorhanden sein müssen. Mit einem Befehl wie *docker-compose up* lassen sich dann ausgewählte oder alle definierten Services simultan starten. Anhand von Automatisierungen mit CI/CD müssen durch Docker Compose keine weiteren komplexen Docker-Skripts aufgesetzt werden, damit die Services gestartet werden können.

Dank der, wie in Unterabschnitt 3.1.3 erklärt wurde, losen Kopplung von Microservices, können diese unabhängig voneinander als einzelne Prozesse installiert werden. Für diesen Fall wird das CI/CD-Werkzeug *GitLab CI* genutzt. Diese Entscheidung fiel, da der Code des Projektes in einem GitLab-Repository gespeichert wird. GitLab stellt dabei viele DevOps-Werkzeuge zur Verfügung, um den Entwicklern viele redundante Arbeitsschritte zu automatisieren und damit zu ersparen. Ebenfalls lassen sich mit GitLab CI sogenannte *GitLab Runner* auf dem eigenen Server installieren. Dadurch ist man nicht auf einen SSH-Zugriff auf den Server angewiesen, um dort einen Service neu installieren zu können. GitLab Runner observieren das Projekt auf neue CI/CD-Jobs, woraufhin diese die notwendigen Schritte ausführen, wie sie in einer gitlab-ci.yml-Datei festgehalten werden und im Anhang in Code 4 zu betrachten sind. Dabei werden Build-Prozesse in einem eigenen Docker-Container ausgeführt, damit das eigentliche System durch Erstellung und Test dieser Container keinen Schaden nehmen kann. Alle CI/CD-Skripte funktionieren für alle Microservices von rrRemote ähnlich. Zusammengefasst werden folgende Schritte ausgeführt, damit ein Microservice neu aufgesetzt werden kann:

- Die NodeJS-Applikation wird in einem NodeJS-Container kompiliert.
- Kompilierte Dateien, die im *build/* Ordner sind, werden als *Artefakte* für die nächsten GitLab CI-Arbeitsschritte hochgeladen.
- Durch die *Dockerfile* wird die Applikation in einen Docker-Container verpackt. Um Dateigrößen klein zu halten, werden dafür die vorher kompilierten Artefakte genutzt.
- Der kompilierte Docker-Container wird in der GitLab Container-Registry gespeichert.
- der neue Docker-Container wird aus der Container-Registry auf den Server heruntergeladen. Er wird mit vordefinierten Startoptionen durch die Docker Compose YAML-Datei gestartet.

In dem im Anhang enthaltenen Code 4 fällt auf, dass die *Testphase* auskommentiert wurde. Aus zeitlichen Gründen muss in der CI/CD-Pipeline auf Unit Tests verzichtet werden. Trotzdem sollte in der Umsetzung von Microservices mit mehreren Entwicklern immer Testing verwendet werden, damit die Integrität des aktualisierten Microservice im System sichergestellt werden kann. Das Dockerfile des rrRemote *Artist-Service* kann als Beispiel im Anhang unter Code 2 betrachtet werden (Evans, 2004, S. 342).

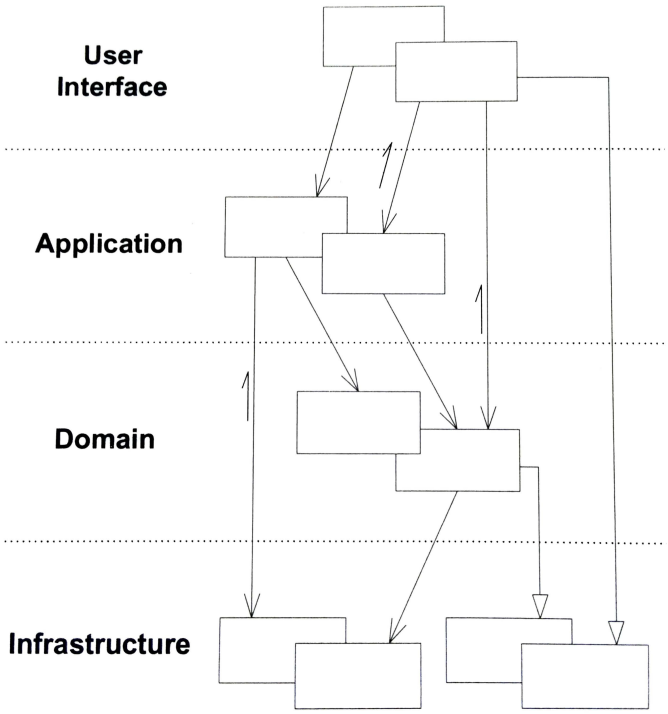

Abbildung 4.8.: Typische Gliederung einer Schichtenarchitektur (Evans, 2004, S. 68)

Die folgenden Microservices werden anhand einer *Schichtenarchitektur* implementiert, die durch Abbildung 4.8 inspiriert ist. Dabei besitzen die Microservices kein *User Interface*, sondern lediglich Kommunikationsschnittstellen für GraphQL zur Client-Server-Kommunikation und RabbitMQ für die Inter-Service-Kommunikation.

4.3.1. API Gateways

Für die Erreichbarkeit aller Microservices im System werden für den Client zwei API Gateways zur Verfügung gestellt. Der Erste agiert als HTTP-Reverse Proxy Server, der ankommende HTTP-Anfragen anhand von URL-Direktiven auf andere Ressourcen weiterleitet. Diese können weitere Verzeichnisse mit statischen Websiten, Docker-Container oder weitere auf dem System installierte Applikationen sein. Der von einem API Gateway vertretene Grundgedanke ist, dass ein Server mehrere Schnittstellen bietet, auf die über einen einzigen Anlaufpunkt zugegriffen werden kann. Ein Beispiel hierfür liefert die Abbildung 4.5 (Montesi & Weber, 2016, S. 6; Richardson, 2018).

Das erste API Gateway namens *NGINX Reverse Proxy* existiert, um externe HTTP-Anfragen auf weitere Docker-Container zu leiten. Dies hat den Vorteil, dass man ohne weitere Komplikationen direkt über ein URL-Direktiv auf Container zugreifen kann. Somit lässt sich die zu entwickelnde Software kontrollierter testen. Gleichzeitig können somit auf demselben Server verschiedene Applikationen installiert werden, die für sich stehen oder rrRemote erweitern. Es könnte also für die *Log-Aggregation* ein eigener Container aufgesetzt werden, damit die Zentralisierung von Logs nicht extern stattfinden muss. Der im Anhang enthaltene Code 5 zeigt die Konfiguration des ersten API Gateways, damit die URL-Direktiv-basierte Weiterleitung von externen HTTP-Anfragen funktionieren kann. Dieser definiert die Konfiguration für die Direktive, die respektiv auf die MinIO-Instanz, das Apollo Gateway und die RabbitMQ-Instanz verweisen.

Während der NGINX Reverse Proxy einen voll funktionsfähigen Webserver abbildet, der als Apache-Konkurrent anzusehen ist, implementiert ein Apollo Server und damit auch ein Apollo Gateway einen leichtgewichtigen NodeJS-Server, der eine spezifische Aufgabe besitzt. Da ein API Gateway unter anderem die Qualitäten der *Sicherheit* implementiert und dies noch nicht im vorherigen API Gateway passiert ist, muss das Apollo Gateway ein Sicherheits-Modul für dessen GraphQL-API implementieren. Weitere Services wie MinIO und RabbitMQ werden durch einen eigenen Sicherheitsmechanismus geschützt, weshalb diese nicht weiter von einem API Gateway abgesichert werden müssen. Dieser Mechanismus ist im Anhang als Code 6 dargestellt, in welchem versucht wird, den ankommenden OAuth 2.0 Token über Firebase Authentication in ein *Artist-Objekt* zu konvertieren und aus diesem die ID des anfragenden Nutzers zu extrahieren (Richardson, 2018; Montesi & Weber, 2016, S. 6).

Durch *Apollo Federation* können Services nun, wie in Abbildung 4.7 dargestellt, GraphQL-Typen referenzieren, ohne dass diese im eigenen Service vorhanden sind. Aus diesem Grund muss das Apollo Gateway auch die Anforderung erfüllen, diese Referenzen und die damit verbundenen Informationen in anderen Microservices zu finden. Code 7, welcher im Anhang zu finden ist, beschreibt eine simple GraphQL-Query, mit der von dem Backend Informationen über einen Renderjob angefragt werden können. In dieser wird die ID des Renderjobs und dessen zugehöriger Artist mit seiner E-Mail-Adresse angefragt. Der Prozessablauf des Apollo Gateways findet sich in Abbildung 4.9 wieder. Die interne Auflösung eines Renderjobs und eines *Artists* wurde aus Gründen der Übersichtlichkeit aus dem Sequenzdiagramm abstrahiert.

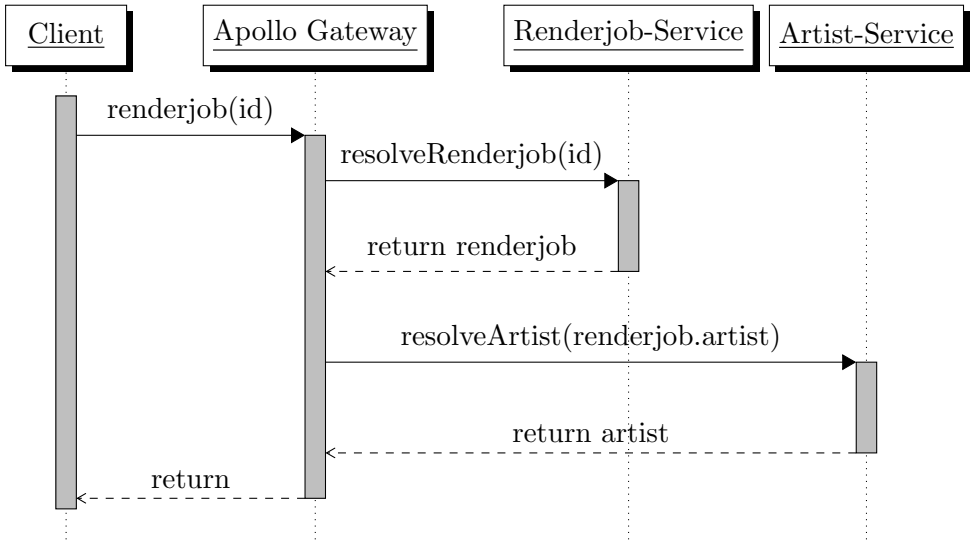

Abbildung 4.9.: Sequenzdiagramm zur beispielhaften Darstellung der Funktionsweise eines Apollo Gateways

In Abbildung 4.9 ist der Ablauf eines Apollo Gateways bei der im Anhang unter Code 7 zu findenden GraphQL-Query dargestellt. Dabei wird ein Renderjob und dessen Artist anhand der übergebenen RenderjobID gefunden. Zuerst betrachtet das Apollo Gateway den *Renderjob-Service* und fordert von diesem ein Renderjob-Objekt an, welches unter anderem eine Referenz des dazugehörigen Artist enthält. Nachdem das Apollo Gateway die vom Client angefragten Informationen über den Renderjob bekommen hat, fragt das Apollo Gateway den *Artist-Service* nach einem Artist-Objekt, welches der in dem Renderjob enthaltenen Artist-Referenz ähnelt. Nachdem alle angefragten Objekte dem Apollo Gateway übermittelt wurden, wird dem Client ein Objekt zurückgegeben, welches der angefragten Struktur gleicht.

4.3.2. Artist-Service

Wie in Abschnitt 4.1 erwähnt, beschäftigt sich der *Artist-Service* ausschließlich mit *Nutzer-Accounts* bzw. *Artists*. Wenn man das Backend von rrRemote auf eine generische Microservice-Architektur projiziert, übernimmt der *Artist-Service* die Rolle der Authentifizierung.

In Abbildung 4.5 ist zu sehen, dass der *Artist-Service*, anders als bei dem *File-Service* und *Renderjob-Service*, mit der RabbitMQ-Instanz gestrichelte Pfeile teilt. Dies bedeutet, dass im *Artist-Service* bisher keine RabbitMQ-Schnittstelle implementiert wird, da dieser noch keine Operationen besitzt, welche andere Services des Systems involvieren. Demnach ist eine RabbitMQ-Schnittstelle für diesen Service zwar angedacht, wird jedoch für die

erste Version des Backends nicht benötigt und somit nicht implementiert. Die Artists von rrRemote werden in Firebase Authentication persistiert, weshalb dieser Service außer der GraphQL- und der RabbitMQ-Schnittstelle noch eine Anbindung an Firebase Authentication besitzt. Diese Verbindung ist im Sinne von Domain-Driven-Design (DDD) in der *Infrastruktur-Ebene* durch ein *Artist-Repo* abstrahiert. So müssen sich Domänenobjekte und höhere Ebenen des Systems nicht mit technischen Spezifikationen von Firebase auseinandersetzen, sondern bleiben in ihrer linguistischen Grenze bestehen. Der *Artist-Service* definiert für GraphQL und somit für das Frontend die Query *currentArtist* und den Typen *Artist*. Dabei kann ein Artist durch dessen *UID* in dem Service gefunden werden.

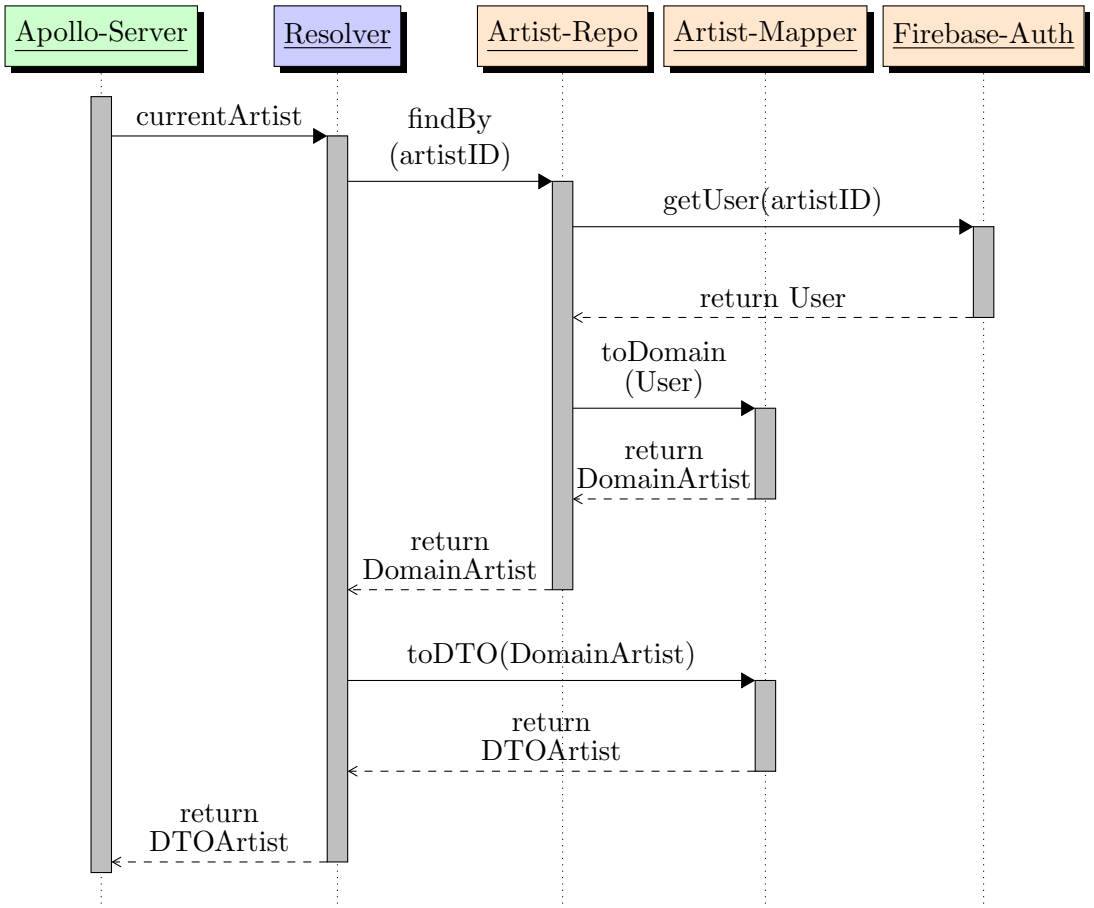

Abbildung 4.10.: Funktionsweise des GraphQL-Endpunktes *currentArtist*

Abbildung 4.10 bildet die Implementierung der Query *currentArtist* ab. Mit *Data Transfer Object (DTO)* ist in dem Sequenzdiagramm ein GraphQL-Objekt gemeint, also ein Objekt, das der angefragten GraphQL-Struktur entspricht. Die Architekturebenen *Interfaces*, *Application* und *Infrastructure* sind in den folgenden Sequenzdiagrammen respektiv als Instanzen mit den Farben Grün, Blau und Orange gekennzeichnet.

Wenn das Apollo Gateway den momentan eingeloggten Artist ermitteln soll, sendet dieses, wie es in Abbildung 4.10 zu sehen ist, die Anfrage weiter an den *Artist-Service*, da dieser der Service ist, welcher den Artist-Typen implementiert. Daraufhin wird der *currentArtist-Resolver*, wie er in der Abbildung 4.10 als *Resolver* dargestellt ist, ausgeführt. Aufgrund der Schichtenarchitektur aus Abbildung 4.8 lässt sich ableiten, dass jegliche Funktionalitäten aus der Interfaces-Ebene Zugriff auf Konzepte aus darunterliegenden Ebenen bekommen können. Aus diesem Grund nutzt der Resolver das Artist-Repo, um einen Artist aus der Persistenzschicht zu ermitteln. Da ein Repository in DDD der Domäne dient, gibt dieses immer Domänenobjekte zurück. Da allerdings die GraphQL-Struktur eines Artists von der Darstellung eines Domänen-Artists abweichen kann, muss der Domänen-Artist in einen GraphQL-Artist konvertiert werden. Nach der Konvertierung kann nun der Artist an das Frontend übermittelt werden.

Der *Artist-Service* implementiert ebenfalls eine Apollo Federation spezifische, bereitgestellte *__resolveReference-Methode*, mit welcher das Apollo Gateway ermitteln kann, wie ein Artist anhand eines Referenzobjektes aufgelöst werden kann. Diese Methode muss für jene GraphQL-Typen implementiert werden, die von einem anderen Service referenziert werden. Da die Implementierung dieser Methode der *currentArtist* Implementierung ähnelt, wird sie in dieser Arbeit nicht weiter behandelt.

4.3.3. File-Service

Der *File-Service* ist die Implementierung der unterstützenden Subdomäne *Transport*. Dieser besitzt die alleinige Aufgabe Renderjobs, welche in diesem Kontext als generische Dateien und Datenströme anzusehen sind, für die *Generierung* zu modifizieren und in der *Lagerung* zu verwalten. Diese besteht aus einem remote-basierten und einem lokalen Lager, zwischen welchen Daten verschoben werden müssen, damit die *Generierung* erfolgen kann. Aus diesem Grund ist der *File-Service* für folgende Aufgaben verantwortlich:

- Die Übertragung von Renderjobs aus dem MinIO-Speicher auf das Lokale Dateisystem

- Eine Vorbereitung der mitgeführten Dateien, die zum Starten der Generierung eines Renderjobs benötigt werden
- Die Verpackung und der Transport von gerenderten Dateien vom lokalen Speicher in den remote-basierten Speicher

Ein Nutzer des Systems sollte nicht darüber entscheiden können, wann welche Dateien transportiert werden sollen. Daher wird der *File-Service* nur vom *Renderjob-Service* delegiert. Aus diesem Grund besitzt der *File-Service* auch keine GraphQL-Mutations oder GraphQL-Queries, sondern ausschließlich RabbitMQ-Endpunkte, mit denen der Service gesteuert werden kann. Trotzdem implementiert dieser den GraphQL-Typen *Storage*, mit welchem vom Client aus herausgefunden werden kann, in welchem Speicherplatz von MinIO der Renderjob zu finden ist. Dieser kann allerdings nur über den *Renderjob-Service* und einen dazugehörigen Renderjob ermittelt werden.

Wie in Unterabschnitt 4.2.4 beschrieben, nutzt MinIO eine Amazon S3-kompatible Umsetzung in welcher mit sogenannten *Buckets* und *Objects* gearbeitet wird. Dabei ist ein Bucket mit einer Partition eines Computers zu vergleichen, auf dem Daten persistiert werden können, wie in S3 als Objects beschrieben. Ein Bucket ist dabei der Eingangspunkt eines S3-Speichers und kann diesen anhand von Direktiven nach abgespeicherten Objekten untersuchen. Der Pfad eines im S3-Speicher abgelegten Objektes lässt sich also mit folgendem Muster beschreiben: *<Bucket>/...Optionale, weitere Direktive.../<Objektname>*. Dabei muss der Objektname Dateiendungen enthalten, je nachdem welche Art von Datei extrahiert oder gespeichert werden soll. Renderjobs, die mithilfe von rrRemote generiert werden sollen, werden ähnlich wie das vorherige Muster wie folgt abgespeichert: *<ArtistID>/<RenderjobID>.zip*. Dabei ist die Variable *ArtistID* das Bucket und die Variable *RenderjobID* der Objektname des Renderjobs. Da Renderjobs aus größeren Ordnerstrukturen bestehen können, ist es von Vorteil diese in eine ZIP-Datei zu archivieren und in MinIO zu laden. Die ArtistID entsteht durch den *Artist-Service* und dadurch zwingend durch Firebase Authentication, während die RenderjobID vom *Renderjob-Service* und dadurch mithilfe von Firebase Firestore generiert wird. So wird sichergegangen, dass jeder Renderjob einzigartig im System existiert und in jeglichem Speicher gefunden werden kann. Beim Transport der Renderjobs zwischen dem remote-basierten und dem lokalen Lager wird diese Konvention beibehalten. Das vorher genannte S3-Muster ist somit auch im lokalen Lager als *...weitere Direktive.../<ArtistID>/<RenderjobID>/<Dateien des Renderjobs>* zu finden. So wird eine Datenkonsistenz zwischen MinIO und dem lokalen Dateisystem hergestellt.

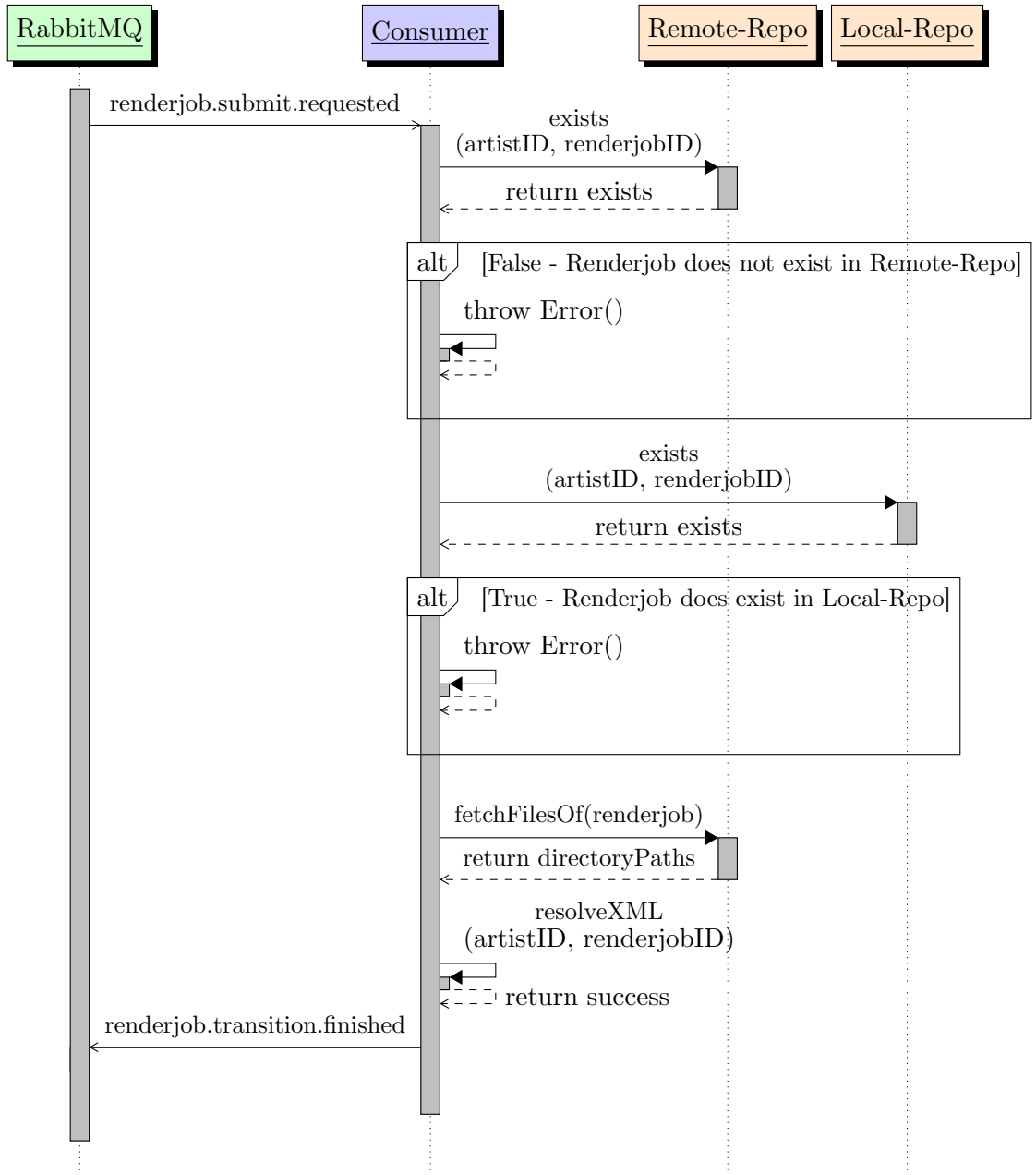

Abbildung 4.11.: Funktionsweise des *File-Service* bei einer einkommenden RabbitMQ-Nachricht auf der Route *renderjob.submit.requested*

In Abbildung 4.11 ist der grundlegende Ablauf zu sehen, wenn aus dem RabbitMQ-Service eine Nachricht von der Route *renderjob.submit.requested* übermittelt und vom *File-Service* konsumiert wird. Im Diagramm ist RabbitMQ der externe Service, wie er in Abbildung 4.5 zu betrachten ist, während die anderen Instanzen innerhalb vom *File-Service* existieren. Der *Consumer* ist dabei ein *RabbitMQ-Consumer*, der von der Bibliothek *amqplib* bereitge-

stellt wird. Dieser reagiert, sobald ihn eine Nachricht über eine bestimmte Route erreicht, wie es im Diagramm dargestellt ist. Der Download und die Vorbereitung eines Renderjobs wird nur dann eingeleitet, wenn dieser im Remote-Repo und nicht auf dem lokalen Dateisystem vorhanden ist. Falls dieser im lokalen Dateisystem existiert, so ist das Risiko geboten, dass Royal Render diesen Renderjob schon rendert. Demnach würde versucht werden die schon vorhandenen und gerade benutzten Dateien zu überschreiben, was zu fehlerhaftem Verhalten in Royal Render und in rrRemote führen kann. Demnach quittiert der Service den Prozess, falls eine dieser Konditionen zutrifft. Falls die Konditionen allerdings ein weiteres Vorgehen ermöglichen, beginnt der *File-Service* mit dem Download und der Extraktion der Renderjob-Daten mit der Methode *fetchFilesOf(renderjob)*. Der Parameter *renderjob* ist dabei ein Renderjob-Domänenobjekt der *Logistik-Kontextgrenze*.

Die Erstellung des Domänenobjektes wurde für die Übersichtlichkeit aus dem Diagramm ausgelassen. Nachdem der Renderjob aus MinIO auf das lokale Dateisystem übertragen und entpackt wurde, wird die Methode *resolveXML* ausgeführt. Im Normalfall enthält eine XML zum Übergeben eines Renderjobs an Royal Render diverse absolute Pfade. Diese müssen im Falle einer Benutzung von rrRemote mit den neuen Pfaden angeglichen werden, die nun auf den Renderjob auf dem Server verweisen. Um diese Operation zu vereinfachen, wurde das von Royal Render bereitgestellte Autodesk Maya-Script namens *rrSubmit_Maya_2017+.py* modifiziert. Zum Überführen eines Renderjobs nach Royal Render kann dieses nun auch eine für rrRemote spezifische XML Version in dem respektiven Arbeitsordner abspeichern. Diese spezielle Version enthält dann Muster, welche von rrRemote gefunden und ersetzt werden können. Beispielhaft kann eine solche XML im Anhang unter Code 8 betrachtet werden. Nach erfolgreicher Extraktion des Renderjobs und Modifizierung der XML wird eine Nachricht nach RabbitMQ auf der Route *renderjob.transition.finished* gesendet, sodass weitere Systeme zum Überführen des Renderjobs nach Royal Render aktiviert werden.

Das Verpacken der gerenderten Daten geschieht, sobald Royal Render mit dem Rendern des letzten Renderjobs fertig ist. Daraufhin wird ein Python-Skript ausgeführt, dass eine Nachricht an RabbitMQ über die Route *renderjob.status.update* sendet, welche unter anderem dazu führt, dass der Ordner *images* in eine ZIP-Datei komprimiert wird. Der ZIP-Datei wird eine Bezeichnung nach dem Muster *<RenderjobID>_RENDERINGS.zip* vergeben und anschließend in das MinIO-Bucket des jeweiligen Künstlers hochgeladen.

4.3.4. Renderjob-Service

Die Services namens *Artist-Service* und *File-Service* werden oft vom *Renderjob-Service* orchestriert um gewisse Operationen durchführen zu können. Dies spiegelt auch Abbildung 4.1 wider, da die Kontextgrenze *Management* im Mittelpunkt des Domänenmodells steht. Das bedeutet auch, dass der *Renderjob-Service* die meisten Referenzen zu Domänenobjekten aus anderen Kontextgrenzen besitzt. Demnach bildet dieser Service den Kern des Backends von rrRemote und deckt damit die Kerndomäne *Inventar* ab. Gleichzeitig bedeutet dies, dass der *Renderjob-Service* für das Frontend der Eintrittspunkt für die meisten vom Apollo Gateway delegierten GraphQL-Anfragen ist.

Wie der Name *Renderjob-Service* aussagt, ist dieser für alle Tätigkeiten zuständig, die unmittelbar mit Renderjobs im Kontext des *Managements* zu tun haben. Somit kann, wie im Anhang unter Code 9 zu sehen ist, ein Artist über diesen Renderjobeinträge in der Datenbank registrieren, solche an Royal Render übergeben, Renderjobs abbrechen und deren Titel aktualisieren lassen. Anschließend bekommt das Frontend eine neue RenderjobID, mit welcher im Backend von rrRemote spezifische Prozesse eingeleitet werden können.

Ebenfalls ist der Service für das Verwalten und Synchronisieren der Daten zwischen Royal Render und der eigenen Datenbank verantwortlich. Beispielsweise ist im Anhang in Code 9 zu sehen, dass ein Renderjob die Attribute *remainingTime* und *status* besitzt. Erstere Information wird direkt aus Royal Render extrahiert, während Zweiteres eine Selektion aus dem in Royal Render gespeicherten Renderjob-Status und dem in der eigenen Datenbank abgespeicherten Status ist. Royal Render besitzt, wie in Abschnitt 2.2 beschrieben wurde, verschiedene Werte, die den momentanen Status eines Renderjobs in Royal Render beschreiben. Da rrRemote diese Stati mit eigenen Stati erweitert, muss aus beiden Systemen der jeweils in dem Renderprozess als später zu betrachtende Status dem anfragenden Künstler übermittelt werden. Beispielsweise speichert Royal Render am Ende eines Renderjobs einen *Finished* Status ab, während rrRemote nach dem Hochladen der Renderings einen *Downloadable* Status besitzt. Der Rückgabewert einer Anfrage des *status* Attributes eines Renderjobs wäre somit *Downloadable*, da dieser im gesamten Prozess nach einem *Finished* Status erfolgt.

Wie in Abschnitt 2.2 erwähnt, besitzt dieser Service mithilfe der NodeJS-C++-Addons für Royal Render eine Anbindung an dieses On-Premise System. Dadurch lassen sich zum Beispiel Daten wie die noch verbleibende Renderzeit eines Renderjobs und dessen Status extrahieren und weiter verarbeiten. Ebenfalls können dadurch verschiedene renderjobspezifische Operationen in Royal Render aktiviert werden wie zum Beispiel das Abbrechen oder Löschen eines Renderjobs.

Auch wenn eine Übergabe eines Renderjobs durch diese Anbindungen stattfinden kann, wird in der Dokumentation in Royal Render darauf verwiesen, dass besser für die jeweilige Renderjob-Szene eine XML mit den nötigen Übergabe- und Renderoptionen erstellt wird. Durch das Programm *rrSubmitterconsole* wird diese anschließend an Royal Render übermittelt. Da die NodeJS-Docker-Container mit *musl libc* statt mit dem sonst öfters genutzten *glibc* kompiliert werden, ist eine Nutzung der rrSubmitterconsole aus einem NodeJS-Docker-Container heraus nicht möglich. Stattdessen existiert auf dem Windows-Server 2019 ein *Proxy-Service*, der bestimmte Tätigkeiten, die eine spezifische Umgebung benötigen, von Docker-Containern über RabbitMQ annimmt und nach Royal Render weiterleitet (Schönberger, 2020a).

Letztendlich ist der *Renderjob-Service* auch für die grundlegende Validierung von Renderjobs verantwortlich. Da sich keine Applikationen wie Autodesk Maya oder SideFX Houdini in den Docker-Containern befinden, kann ein Renderjob nicht in dessen Inhalt validiert werden. Dennoch sollte sichergestellt werden, dass bestimmte Dateien und Pfade im Renderjob vorhanden sind. So bekommt der *Renderjob-Service* aus einer RabbitMQ-Nachricht eine Liste an Direktiven, die relativ zu dem jeweiligen Renderjob existieren und vom *Renderjob-Service* nach bestimmten Vorraussetzungen validiert werden. So wird sichergestellt, dass der *Renderjob-Service* keinen Zugang zum Dateisystem benötigt, jedoch eine Vollständigkeit der benötigten Daten überprüfen kann. Ein Zugang zum Dateisystem sollte nach dem Single-Responsibility-Prinzip im *Renderjob-Service* vermieden werden, da der *File-Service* diese Verantwortung schon übernimmt. Der *Renderjob-Service* beinhaltet Regelsätze, mit denen die Liste an relativen Pfaden kontrolliert wird. Da das System sich auf Renderjobs aus der Applikation Autodesk Maya spezialisiert, besitzt der Service einen Maya-Regelsatz, der folgende Regeln zum Validieren der Pfade nutzt:

- Eine Datei namens *workspace.mel* muss im Wurzelverzeichnis des Renderjobs vorhanden sein.
- Für entstehende Renderings einer Maya-Szene muss das Renderjobverzeichnis einen *images* Ordner enthalten.
- Eine Datei mit der Endung *.ma* oder *.mb* muss im Wurzelverzeichnis oder in einem Unterverzeichnis vorhanden sein.
- Es muss eine XML im Wurzelverzeichnis existieren, die zur Übergabe des Renderjobs nach Royal Render dient.

Falls all diese Regeln in den ankommenden relativen Pfaden zutreffen, kann eine Übertragung des Renderjobs nach Royal Render eingeleitet werden. Nach dem Validieren der Pfade werden für den *Proxy-Service* Royal Render Optionen konkateniert, sodass nur bestimmte Computer der Renderfarm für den Renderjob genutzt werden und eine Wiederfindbarkeit dieser mithilfe der Royal Render-Anbindungen simplifiziert wird. Da ein Künstler und ein Renderjob jeweils eine ID durch Firebase zugeordnet bekommen, die garantiert einzigartig ist, können diese in Royal Render als *UserName* und *CompanyProjectName* respektiv eingesetzt werden. Dabei ist ein CompanyProjectName im Sinne von rrRemote ein einziger Renderjob und kein Projekt, in dem mehrere Renderjobs vereint werden. Mithilfe einer Funktion der Royal Render-Anbindungen können dann nach diesen Variablen angefragte Renderjobs in Royal Render gefiltert werden.

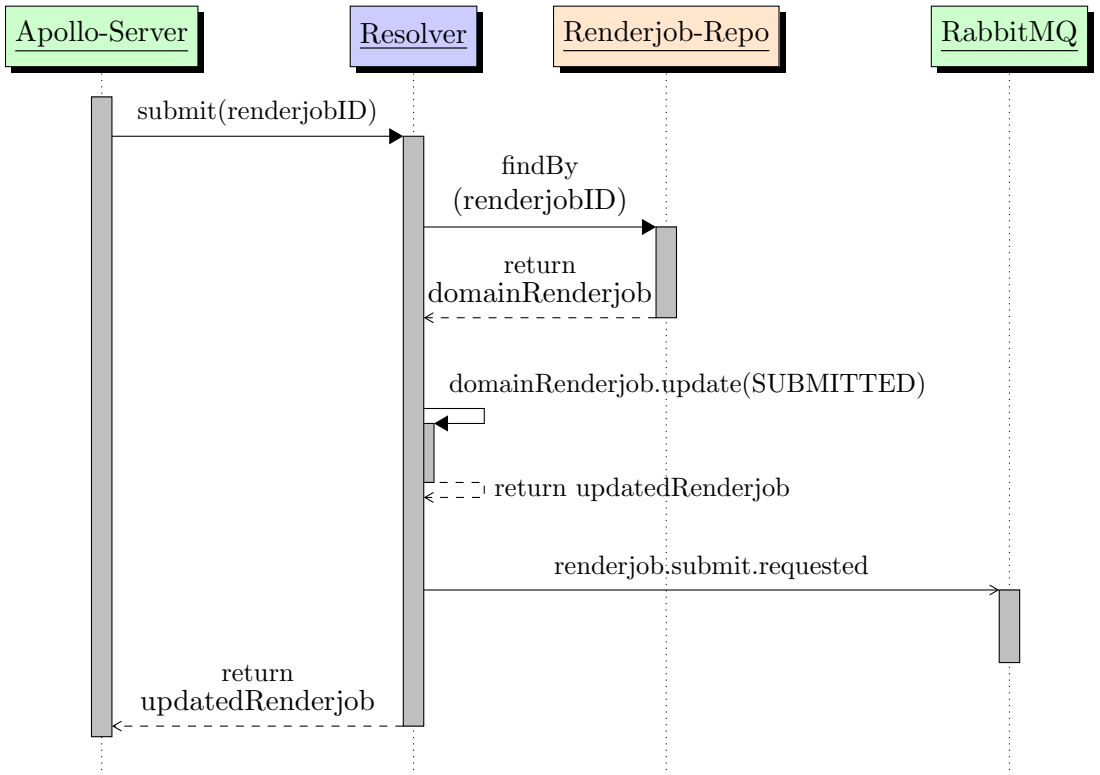

Abbildung 4.12.: Darstellung des durch eine GraphQL-Anfrage gestarteten Prozesses zur Übertragung eines Renderjobs nach Royal Render

Die Abbildung 4.12 beschreibt die Sequenz, wenn ein Künstler die GraphQL-Mutation namens *submit(RenderjobID)* nutzt. Dabei ist zu sehen, dass der *Renderjob-Service* nur den Status des Renderjobs mit der dazugehörigen RenderjobID in der Datenbank auf den Wert *SUBMITTED* ändert. Danach wird eine Nachricht an RabbitMQ auf der Route *renderjob.submit.requested* gesendet. Diese Route ist ebenfalls in der Abbildung 4.11 als Auslöser für dessen Sequenz zu finden. Mithilfe von RabbitMQ lässt sich auch ein sogenannter Remote Procedure Call (RPC) auslösen. Ähnlich wie bei gewöhnlichen asynchronen Nachrichten sendet der anfragende Service eine Nachricht auf einer Route an den RabbitMQ-Service und wartet dann allerdings auf einen Rückgabewert. Es wurde sich hierbei bewusst gegen eine RPC entschieden, da das später im *File-Service* erfolgende Herunterladen und Entpacken eines Renderjobs mehrere Minuten dauern kann. Während dieser Zeit muss der *Renderjob-Service* seinen Zustand bewahren und ist für weitere Anfragen blockiert.

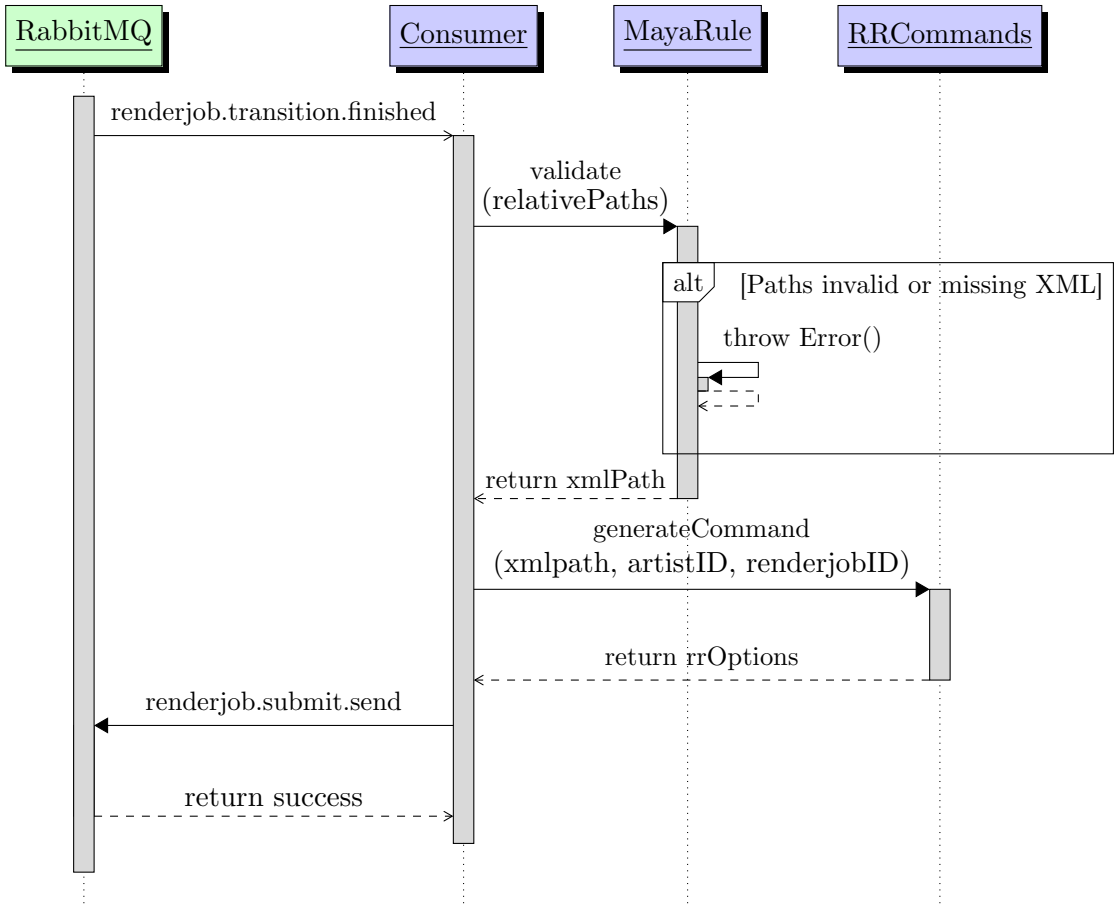

Abbildung 4.13.: Sequenzdiagramm zum Prozess des Erstellens der Royal Render Übertragungsoptionen nach Beendigung des Downloads und Entpackens eines Renderjobs durch den *File-Service*

Nachdem, wie in Abbildung 4.11 dargestellt ist, eine Nachricht auf der Route *renderjob.transition.finished* vom *Renderjob-Service* konsumiert werden kann, beginnt dieser mit dem Validieren der relativen Pfade des jeweiligen Renderjobs und mit der Generierung der Übertragungsoptionen für Royal Render. Sobald die Optionen generiert wurden, müssen diese an den *Proxy-Service* gesendet werden. Dies erfolgt im *Renderjob-Service* über einen RPC. Falls der *Proxy-Service* beim Übertragen des Renderjobs von Royal Render einen Fehler bekommt, wird dieser an den *Renderjob-Service* zurückgesendet, woraufhin eine weitere Fehlerbehandlung durchgeführt wird. Falls der RPC allerdings keinen Error enthält, wurde der Renderjob erfolgreich an Royal Render übergeben und wird anschließend gerendert.

5. Fazit und Ausblick

Ziel dieser Arbeit war es einen Remote-Controller zu entwickeln, sodass grundlegende Funktionalitäten der On-Premise Software *Royal Render* von außerhalb des Hochschulnetzwerks der Hochschule Hamm-Lippstadt genutzt werden können.

Zum Beginn dieser Arbeit wurden grundlegende Anforderungen der Anspruchsgruppen für das Projekt rrRemote ermittelt, sodass diese bei der Konzeptionsphase als Richtlinien in Betracht gezogen werden konnten. Dazu wurde mit Studenten und Angestellten der Internationalen Filmschule Köln gesprochen, die hauptsächlich angehende *Visual Effects Artists* sind und somit einen Teil der Domänenexperten und die Zielgruppe für rrRemote darstellen. Mithilfe einer Analyse der Funktionsweise von Royal Render wurden die Anforderungen der Anspruchsgruppen eingegrenzt, um eine mögliche Implementierung dieser aus der Sichtweise von Royal Render gewährleisten zu können. Diese Anforderungen sind für eine erste Version wie folgt definiert worden:

- Ein Autodesk Maya- und Solidangle Arnold-basiertes Projekt muss von rrRemote angenommen werden.
- rrRemote muss ohne weitere manuelle Eingriffe das angenommene Renderprojekt für rrRemote vorbereiten und anschließend Royal Render übergeben.
- rrRemote muss die entstandenen Renderings dem Künstler bereitstellen.
- Daten von übergebenen Renderjobs wie beispielsweise verbleibende Renderzeit und Renderstatus müssen vom Frontend abgefragt werden können.
- Renderjobs sollten vom Künstler abgebrochen werden können.
- Renderjobs sollten vom Künstler gelöscht werden können.
- Renderjobs sollten vom Künstler aktualisiert werden können.

Nach einer Erläuterung der Grundlagen der Softwarearchitektur konnten durch die Entwurfsmethodik Domain-Driven-Design der Domäne *Remote Rendern* linguistische Kontextgrenzen zugeordnet werden. Anschließend konnte die Domäne in Subdomänen unterteilt und darauffolgend in Microservices überführt werden. Mithilfe von Microservice-Entwurfsmustern und empfohlenen Vorgehensweisen konnte die Softwarearchitektur für eine erste Version des Remote-Controllers von rrRemote finalisiert werden. Die grundlegenden Funktionalitäten, welche durch das System für den externen Gebrauch bereitgestellt werden, lauten wie folgt:

- Hochladen eines Renderjobs in rrRemote
- Überführung eines hochgeladenen Renderjobs nach Royal Render in der Hochschule Hamm-Lippstadt
- Abbrechen eines Renderjobs
- Aktualisierung des Titels eines Renderjobs
- Anfrage von Daten eines jeden Renderjobs pro Künstler wie zum Beispiel dessen Titel, die verbleibende Renderzeit und dessen Renderstatus

Dabei wurde besonders darauf geachtet, dass jeder Service nach dem Single-Responsibility-Prinzip und den typischen Charakteristiken von Microservices entworfen wurde. Deshalb verfügt jeder Microservice über eine GraphQL-Schnittstelle als Kommunikationsportal zum Frontend und eine RabbitMQ-Schnittstelle für die Inter-Service-Kommunikation. GraphQL bietet dabei die Möglichkeit Daten deklarativ von einem Server zu beantragen, was zu einer Performanceerhöhung führen kann, da nur die angefragten Daten ermittelt werden müssen. Durch die Integration von RabbitMQ kann die Inter-Service-Kommunikation asynchron verlaufen, wodurch jeder Service im Sinne des reaktiven Manifests eine Zustandslosigkeit behält, nachrichtenorientiert agiert und antwortbereit bleibt. Eine Datenbank existiert pro Service, sodass im Falle eines Absturzes der Datenbank nur der jeweilige Service negativ beeinflusst wird. Die Microservices wurden jeweils in einer leichtgewichtigen Linux Alpine Version mit NodeJS in Docker-Containern installiert, sodass Arbeitsspeicher- und Speicherkapazitäten geringfügig beeinflusst werden. Des Weiteren wurde für jeden Service eine Continous Integration / Continous Deployment-Pipeline aufgesetzt, um komplexe Installationsprozesse der Services zu automatisieren. Das System spezialisert sich in dessen erster Version auf das Rendering mit der Anwendung Autodesk Maya und der Renderapplikation Solidangle Arnold, weshalb das von Royal Render bereitgestellte *rrSubmit_Maya_2017+.py* Skript so modifiziert wurde, dass dieses eine für rrRemote angepasste Szenen-XML-Datei direkt aus Maya heraus exportiert.

Somit wurde ein Remote-Controller entwickelt, der die grundlegende Nutzung der On-Premise Software Royal Render erlaubt und bis auf die Löschung eines Renderjobs jede Anforderung einer ersten Version erfüllt. Das Backend von rrRemote erweitert somit die Ressourcen zum Rendern von 3D-Szenen der Internationalen Filmschule Köln mit denen der Hochschule Hamm-Lippstadt. Angestellte der Hochschule Hamm-Lippstadt müssen demnach nicht mehr manuell Renderprojekte annehmen, vorbereiten, Royal Render übergeben und die Renderings hochladen. Der Remote-Controller agiert dabei reaktiv und wurde mit Automatisierungen entwickelt, um die mögliche Weiterentwicklung und die Wartbarkeit des Systems zu gewährleisten.

Das Löschen eines Renderjobs wurde in rrRemote aus Komplexitätsgründen der Softwarearchitektur verschoben. Diese Operation beinhaltet eine distributierte Transaktion, was bedeutet, dass verschiedene Services eine Löschung durchführen müssen. Falls beim Löschen in einem Service Fehler auftreten, müssen bestimmte Maßnahmen wie zum Beispiel das Wiederherstellen gelöschter Daten ergriffen werden. Für diese Operation würde es sich demnach lohnen das *SAGA-Entwurfsmuster* zu integrieren. Bei diesem wird eine Liste von Transaktionen gespeichert, die im System ausgeführt wurden. Falls eine Transaktion fehlschlägt, können die vorherigen Transaktion rückgängig gemacht werden, sofern der jeweilige Service für dieses Ereignis die zugehörige Geschäftslogik bereitstellt.

Ebenfalls wurde in dem Backend von rrRemote aus zeitlichen Gründen kein Unit-Testing-Werkzeug genutzt. *Continous Integration* beschäftigt sich mit der Frage, ob eine Aktualisierung eines Service dieselbe Integrität aufweist wie der ursprüngliche Service. Fehler, die sich während der Entwicklung in den Service integriert haben, können so in einer *Testphase* von dem Unit-Testing-Werkzeug abgefangen werden.

Des Weiteren werden Renderings in eine ZIP-Datei verpackt und anschließend in MinIO wieder hochgeladen, was dazu führt, dass je nach Renderjob mit einem Mal große Datenmengen hochgeladen werden müssen. Dies bedeutet ebenfalls, dass keine Teile eines Renderings betrachtet werden können, da erst bei einem Abschluss des gesamten Renderjobs die Renderings hochgeladen und anschließend heruntergeladen werden können. Um diesem Problem vorzubeugen, wäre es für eine zweite Version des Backends von rrRemote von Vorteil, wenn gerenderte Einzelbilder nacheinander in MinIO hochgeladen werden würden. So könnte eine gezielte Auswahl dieser vom Frontend heruntergeladen und angezeigt werden, ohne alle Renderings des Renderjobs herunterladen zu müssen.

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. 24)

Baxley III, J. (2019). *Apollo Federation: A revolutionary architecture for building a distributed graph*. Verfügbar 22. Februar 2020 unter https://blog.apollographql.com/apollo-federation-f260cf525d21. (Siehe S. 49)

Baxley III, J., Rosenberger, J., Scheer, T., Debergalis, M., Crosby, B. & Zionts, A. (2020). *Apollo Federation overview: Implement a single data graph across multiple services*. Verfügbar 22. Februar 2020 unter https://www.apollographql.com/docs/apollo-server/federation/introduction/. (Siehe S. 50)

Blades, T., Rayzis, P., Man, D., Zionts, A., itsjessmarina & gagansaini1212. (2019). *The Apollo GraphQL platform: How Apollo helps you go from zero to production with GraphQL*. Verfügbar 22. Februar 2020 unter https://www.apollographql.com/docs/intro/platform/. (Siehe S. 49)

Blades, T., Zionts, A., Rosenberger, J. & Barlow, S. (2019). *Managing a federated graph: How to run, manage, and deploy a federated graph*. Verfügbar 22. Februar 2020 unter https://www.apollographql.com/docs/graph-manager/federation/. (Siehe S. 50)

Bonér, J., Farley, D., Kuhn, R. & Thompson, M. (2014). *The Reactive Manifesto*. Verfügbar 9. März 2020 unter https://www.reactivemanifesto.org/. (Siehe S. 5, 48)

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. 27, 28)

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. 20)

Docker. (2019). *Get Docker Engine: Community for CentOS*. Verfügbar 20. Februar 2020 unter https://docs.docker.com/install/linux/docker-ce/centos/. (Siehe S. 45)

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. 11, 14, 16, 17)

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley. (Siehe S. 29, 30, 32, 33, 54, 55).

Facebook. (2018). *GraphQL: June 2018 Edition*. Verfügbar 26. Januar 2020 unter https://spec.graphql.org/June2018/. (Siehe S. 37)

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Diss.). University of California. Verfügbar 25. Januar 2020 unter https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. (Siehe S. 37)

Fong, J. (2018). *Are Containers Replacing Virtual Machines?* Verfügbar 20. Januar 2020 unter https://www.docker.com/blog/containers-replacing-virtual-machines/. (Siehe S. 26)

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. 13)

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. 17, 20)

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. 13)

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. 9, 14, 17, 20, 24)

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. 17)

Hauser, E. (2018). *Announcing Apollo Server 2: Simplifying API development with production best practices*. Verfügbar 22. Februar 2020 unter https://blog.apollographql.com/announcing-apollo-server-2-2b69fb4702ce. (Siehe S. 49)

Helgason, A. F. (2017). *Performance analysis of Web Services: Comparison between RESTful & GraphQL web services* (Bachelorarb.). University of Skövde. Verfügbar 25. Januar 2020 unter http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1107850&dswid=-4124. (Siehe S. 37)

Hempel, W. (2019). *Service Mesh VS API Gateway VS Message Queue: when to use what?* Verfügbar 23. Januar 2020 unter https://arcentry.com/blog/api-gateway-vs-service-mesh-vs-message-queue/. (Siehe S. 35, 36)

Johansson, P. (2017). *Efficient Communication With Microservices* (Magisterarb.). Umea University, Sweden. Verfügbar 25. Januar 2020 unter http://www8.cs.umu.se/education/examina/Rapporter/PetterJohansson2017.pdf. (Siehe S. 37)

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. 27, 28)

Minio, Inc. (2020). *High Performance, Kubernetes-Friendly Object Storage: Build high performance, cloud native data infrastructure for machine learning, analytics and application data workloads with MinIO.* Verfügbar 23. Februar 2020 unter https://min.io/. (Siehe S. 52)

Montesi, F. & Weber, J. (2016). Circuit breakers, discovery, and API gateways in microservices. *arXiv preprint arXiv:1609.05830*. Verfügbar 24. Februar 2020 unter https://arxiv.org/abs/1609.05830 (siehe S. 55, 56)

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. 13, 14)

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. 20, 24)

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. 9–14)

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. 18, 19)

Richardson, C. (2018). *Microservices Patterns.* Manning Publications. Verfügbar 22. Dezember 2019 unter https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/. (Siehe S. 11, 13, 14, 34, 35, 38, 39, 46, 55, 56)

Richardson, C. (2019a). *A pattern language for microservices.* Verfügbar 16. Januar 2020 unter https://microservices.io/patterns/index.html. (Siehe S. 84)

Richardson, C. (2019b). *Pattern: Microservice Architecture.* Verfügbar 16. Januar 2020 unter https://microservices.io/patterns/microservices.html. (Siehe S. 22)

Richardson, C. & Smith, F. (2016). *Microservices: From Design to Deployment*. Nginx, Inc. Verfügbar 24. Januar 2020 unter https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/. (Siehe S. 34, 35, 38)

Schönberger, H. (2019a). *How does it work? - Royal Render: Workflow*. Verfügbar 12. März 2020 unter https://www.royalrender.de/how-does-it-work.html. (Siehe S. 6, 7)

Schönberger, H. (2019b). *Requirements - Royal Render: Hardware: For Royal Render*. Verfügbar 12. März 2020 unter https://www.royalrender.de/requirements.html. (Siehe S. 5–7)

Schönberger, H. (2020a). *module libpyRR2*. Verfügbar 7. März 2020 unter https://www.royalrender.de/help8/index.html?modulelibpyRR2.html. (Siehe S. 64)

Schönberger, H. (2020b). *rrHelp: Local Data Folder*. Verfügbar 12. März 2020 unter https://www.royalrender.de/help8/index.html?LocalDataFolder.html. (Siehe S. 7)

Schonning, N., Do, T., Darwish, O., Borins, M., Zhigunov, S., Hwang, K., Maledong, Mozhet Byt, V., mattc41190, XhmikosR, Hakerh400, bughit, Lucas, E., McGhan, D. & Harrington, B. (2020). *The Node.js Event Loop, Timers, and process.nextTick()*. Verfügbar 7. März 2020 unter https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/. (Siehe S. 53)

Schonning, N., Wine, M., Busby, T., Nießen, T., Li, M., Hemberger, F., Noordhuis, B., Genc, H., Maiyo, E., Davis, j., XhmikosR, Maledong, objectisundefined, Murienik, A. & Delgado Diaz, E. (2020). *Don't Block the Event Loop (or the Worker Pool)*. Verfügbar 7. März 2020 unter https://nodejs.org/en/docs/guides/dont-block-the-event-loop/. (Siehe S. 53)

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. 28)

Takai, D. (2017). *Architektur für Websysteme: Serviceorientierte Architektur, Microservices, Domänengetriebener Entwurf*. Carl Hanser Verlag. (Siehe S. 9, 13–20, 24, 25, 29).

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. 19)

Tilkov, S. & Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, *14*(6), 80–83. https://doi.org/10.1109/MIC.2010.145 (siehe S. 53)

Trott, R., Rogers, M., Hemberger, F., Senkpiel, J., You, C., Gallacher, T., Nießen, T., McCandlish, S., Quadri, S., Gorniak, J., Piper, C., Kadhom, N., Pinca, L., Meiert, J. O., Noordhuis, B., Sabne, H., Devinsuit, XhmikosR & Naud-Dulude, C. (2020).

About Node.js®. Verfügbar 23. Februar 2020 unter https://nodejs.org/en/about/. (Siehe S. 53)

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley. (Siehe S. 30–33).

Vernon, V. (2016). *Domain-driven design distilled*. Addison-Wesley. (Siehe S. 29, 30, 33).

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. 18)

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. 10)

Eidesstattliche Versicherung

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.

Lippstadt, 24.03.2020 Robin Dürhager

Ort, Datum Unterschrift

A. Anhang

```
FROM node:alpine
WORKDIR /usr/src
COPY /package*.json ./
COPY /build ./build
COPY fbc.json /home/rrremote/firebase-account-file.json
RUN npm install --production
CMD ["npm", "start"]
```

Code 2: Dockerfile am Beispiel des *Artist-Service* von rrRemote

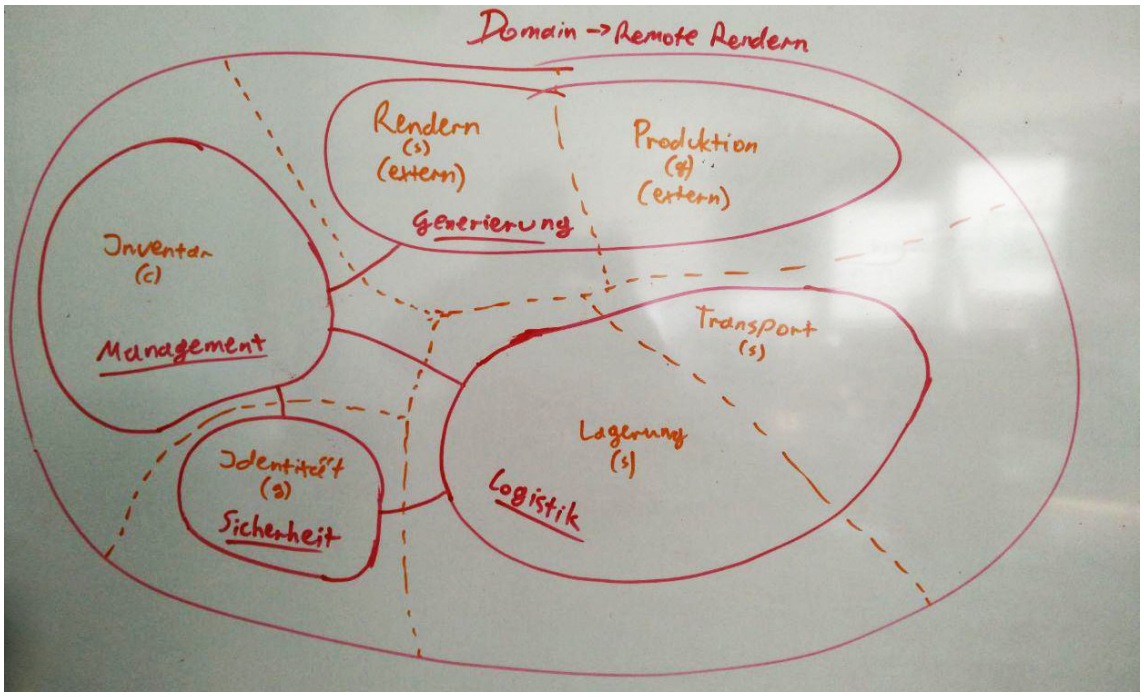

Abbildung A.1.: Skizze der Kontextkarte der Domäne *Remote Rendern*

```yaml
version: '3.7'
services:
#################### rrRemote Container ####################
  proxy:
    container_name: proxygateway
    image: registry.gitlab.com/rrremote/rrremote_proxy_gateway:latest
    restart: always
    ports:
      - 80:80

  graphqlgateway:
    container_name: graphqlgateway
    image: registry.gitlab.com/rrremote/rrremote_apollo_gateway:latest
    restart: always
    environment:
      - ENGINE_API_KEY
      - GOOGLE_APPLICATION_CREDENTIALS
    expose:
      - '8080'

#################### rrRemote Microservices ####################
  artistservice:
    container_name: rrremoteartist
    image: registry.gitlab.com/rrremote/rrremote_artist_service:latest
    restart: always
    environment:
      - GOOGLE_APPLICATION_CREDENTIALS
    expose:
      - '8080'

  projecttransitionservice:
    container_name: rrremoteprojecttransition
    image: registry.gitlab.com/rrremote/rrremote_file_transition_service:latest
    restart: always
    environment:
      - RABBITMQ_DEFAULT_USER
      - RABBITMQ_DEFAULT_PASS
      - MINIO_ACCESS_KEY
      - MINIO_SECRET_KEY
      - NODE_ENV=production
    expose:
      - '8080'
    volumes:
      - /mnt/RRPROJECTS:/mnt/RRPROJECTS

  renderjobservice:
    container_name: rrremoterenderjob
    image: registry.gitlab.com/rrremote/rrremote_renderjob_service:latest
    restart: always
    environment:
      - GOOGLE_APPLICATION_CREDENTIALS
      - RABBITMQ_DEFAULT_USER
      - RABBITMQ_DEFAULT_PASS
      - RRREMOTE_RR_USERNAME
      - RRREMOTE_RR_SECRET
      - NODE_ENV=production
    expose:
      - '8080'
    volumes:
      - /mnt/RR:/mnt/RR
```

```
      - /mnt/RRPROJECTS:/mnt/RRPROJECTS

#################### 3rd Party Services ####################
  minio1:
    container_name: minio
    image: minio/minio:RELEASE.2020-02-07T23-28-16Z
    volumes:
      - /mnt/RRPROJECTS/MINIO/data1-1:/data1
      - /mnt/RRPROJECTS/MINIO/data1-2:/data2
    expose:
      - '9000'
    environment:
      - MINIO_ACCESS_KEY
      - MINIO_SECRET_KEY
    command: server http://minio{1...4}/data{1...2}
    healthcheck:
      test: ['CMD', 'curl', '-f', 'http://localhost:9000/minio/health/live']
      interval: 30s
      timeout: 20s
      retries: 3
    restart: always

  minio2:
    image: minio/minio:RELEASE.2020-02-07T23-28-16Z
    volumes:
      - /mnt/RRPROJECTS/MINIO/data2-1:/data1
      - /mnt/RRPROJECTS/MINIO/data2-2:/data2
    expose:
      - '9000'
    environment:
      - MINIO_ACCESS_KEY
      - MINIO_SECRET_KEY
    command: server http://minio{1...4}/data{1...2}
    healthcheck:
      test: ['CMD', 'curl', '-f', 'http://localhost:9000/minio/health/live']
      interval: 30s
      timeout: 20s
      retries: 3
    restart: always

  minio3:
    image: minio/minio:RELEASE.2020-02-07T23-28-16Z
    volumes:
      - /mnt/RRPROJECTS/MINIO/data3-1:/data1#
      - /mnt/RRPROJECTS/MINIO/data3-2:/data2
    expose:
      - '9000'
    environment:
      - MINIO_ACCESS_KEY
      - MINIO_SECRET_KEY
    command: server http://minio{1...4}/data{1...2}
    healthcheck:
      test: ['CMD', 'curl', '-f', 'http://localhost:9000/minio/health/live']
      interval: 30s
      timeout: 20s
      retries: 3
    restart: always

  minio4:
    image: minio/minio:RELEASE.2020-02-07T23-28-16Z
    volumes:
      - /mnt/RRPROJECTS/MINIO/data4-1:/data1
      - /mnt/RRPROJECTS/MINIO/data4-2:/data2
    expose:
```

```
      - '9000'
    environment:
      - MINIO_ACCESS_KEY
      - MINIO_SECRET_KEY
    command: server http://minio{1...4}/data{1...2}
    healthcheck:
      test: ['CMD', 'curl', '-f', 'http://localhost:9000/minio/health/live']
      interval: 30s
      timeout: 20s
      retries: 3
    restart: always

  rabbitmq:
    container_name: rabbitmq
    image: rabbitmq:3-management
    restart: always
    ports:
      - '15672:15672'
      - '5672:5672'
    environment:
      - RABBITMQ_DEFAULT_USER
      - RABBITMQ_DEFAULT_PASS
```

Code 3: Docker Compose YAML-Datei für die Servicekomposition von rrRemote

```
stages:
    - build
    #- test
    - prepare
    - deploy
    - update

build:application:
    stage: build
    only:
        - master
    tags:
        - docker
    image: node:$NODE_VERSION-alpine
    script:
        - echo $RRREMOTE_FIREBASE_CREDS > fbc.json
        - npm install
        - npm run build
    artifacts:
        paths:
            - build/
            - fbc.json
        expire_in: 20 minutes

prepare:dockerimage:
    stage: prepare
    only:
        - master
    tags:
        - docker
    image: docker:stable
    dependencies:
        - build:application
    services:
        - docker:dind
```

```
    before_script:
        - docker info
        - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
    script:
        - docker build -t $CI_REGISTRY_IMAGE .
        - docker push $CI_REGISTRY_IMAGE

deploy:artistservice:
    stage: deploy
    tags:
        - internal-shell
    before_script:
        - docker info
        - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
    script:
        - cd /home/rrremote
        - docker-compose pull artistservice
        - docker-compose up -d --no-deps artistservice

update:schema:
    stage: update
    only:
        - master
    tags:
        - docker
    image: node:$NODE_VERSION-alpine
    script:
        - apk add git
        - npm install -g apollo
        - apollo service:push --key $ENGINE_API_KEY
↪   --localSchemaFile=./src/interfaces/graphql/schema.graphql
↪   --serviceName=rrRemoteArtist --serviceURL=http://rrremoteartist:8080/graphql
```

Code 4: GitLab CI YAML-Datei, die Arbeitsschritte für einen GitLab Runner festhält, damit Docker-Container neu gebaut, deren builds in einer Container Registry abgespeichert und dann auf dem Server neu aufgesetzt werden können

```
server {
    listen 80;
    listen [::]:80;

    server_name localhost;

    # Erlaube spezielle zeichen im Header
    ignore_invalid_headers off;

    # Filesize restriktion, im moment auf 1GB gesetzt, was nie überschritten werden
↪   sollte, da ankommende Stream-chunks definitiv unter 1 GB an Größe sein sollten
    client_max_body_size 1000m;

    # API Gateway buffern kann zu Problemen führen beim Streaming von Daten
    proxy_buffering off;

    # MinIO Direktiv
    # Muss auf / erfolgen, da minio anscheinend Direktive an die URL konkateniert, was
↪   zu internen Problemen bei MinIO führt.
    location / {
    # Annahme und Weiterleitung von Headern, wie Host, IP-Adresse, etc.
        proxy_set_header X-Real-IP $remote_addr;
```

```
21          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
22          proxy_set_header X-Forwarded-Proto $scheme;
23          proxy_set_header Host $http_host;
24
25          # Weiterleitung der Anfrage auf den Docker-Container mit dem im
    ↪  docker-compose.yml definierten container-name "minio".
26          proxy_pass http://minio:9000;
27          proxy_connect_timeout  300;
28
29          # HTTP 1.1 implementiert keepalive, wodurch die Verbindung zu einem Server nach
    ↪  dessen Anfrage nicht direkt geschlossen wird. Server-Verbindungen aufrecht zu
    ↪  erhalten statt diese immer wieder neu zu initialisieren ist kosteneffizienter
30          proxy_http_version 1.1;
31
32          # durch HTTP 1.1 muss der Connection Header bereinigt werden
33          # Quelle: http://nginx.org/en/docs/http/ngx_http_upstream_module.html#keepalive
34          proxy_set_header Connection "";
35      }
36
37      # Apollo Gateway Direktiv
38      location /rrremote/graphql {
39          # Annahme und Weiterleitung von Headern, wie Host, IP-Adresse, etc.
40          proxy_set_header Host $host;
41          proxy_set_header X-Real-IP $remote_addr;
42          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
43          proxy_set_header X-Forwarded-Host $host;
44
45          # Weiterleitung der Anfrage auf den Docker-Container mit dem im
    ↪  docker-compose.yml definierten container-name "graphqlgateway".
46          proxy_pass http://graphqlgateway:8080/graphql;
47
48          # GraphQL anfragen sollen nicht zu einem redirect führen, da es sich hierbei
    ↪  nicht um eine Änderung der UI handelt
49          proxy_redirect off;
50      }
51
52      # RabbitMQ Direktiv
53      # Quelle:
    ↪  https://stackoverflow.com/questions/49742269/rabbitmq-management-over-https-and-nginx
54      location ~* /rabbitmq/api/(.*?)/(.*) {
55          proxy_pass http://rabbitmq:15672/api/$1/%2F/$2?$query_string;
56          proxy_buffering                    off;
57          proxy_set_header Host              $http_host;
58          proxy_set_header X-Real-IP         $remote_addr;
59          proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
60          proxy_set_header X-Forwarded-Proto $scheme;
61      }
62
63      location ~* /rabbitmq/(.*) {
64          rewrite ^/rabbitmq/(.*)$ /$1 break;
65          proxy_pass http://rabbitmq:15672;
66          proxy_buffering                    off;
67          proxy_set_header Host              $http_host;
68          proxy_set_header X-Real-IP         $remote_addr;
69          proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
70          proxy_set_header X-Forwarded-Proto $scheme;
71      }
72  }
```

Code 5: Konfiguration des NGINX Reverse Proxy für rrRemote

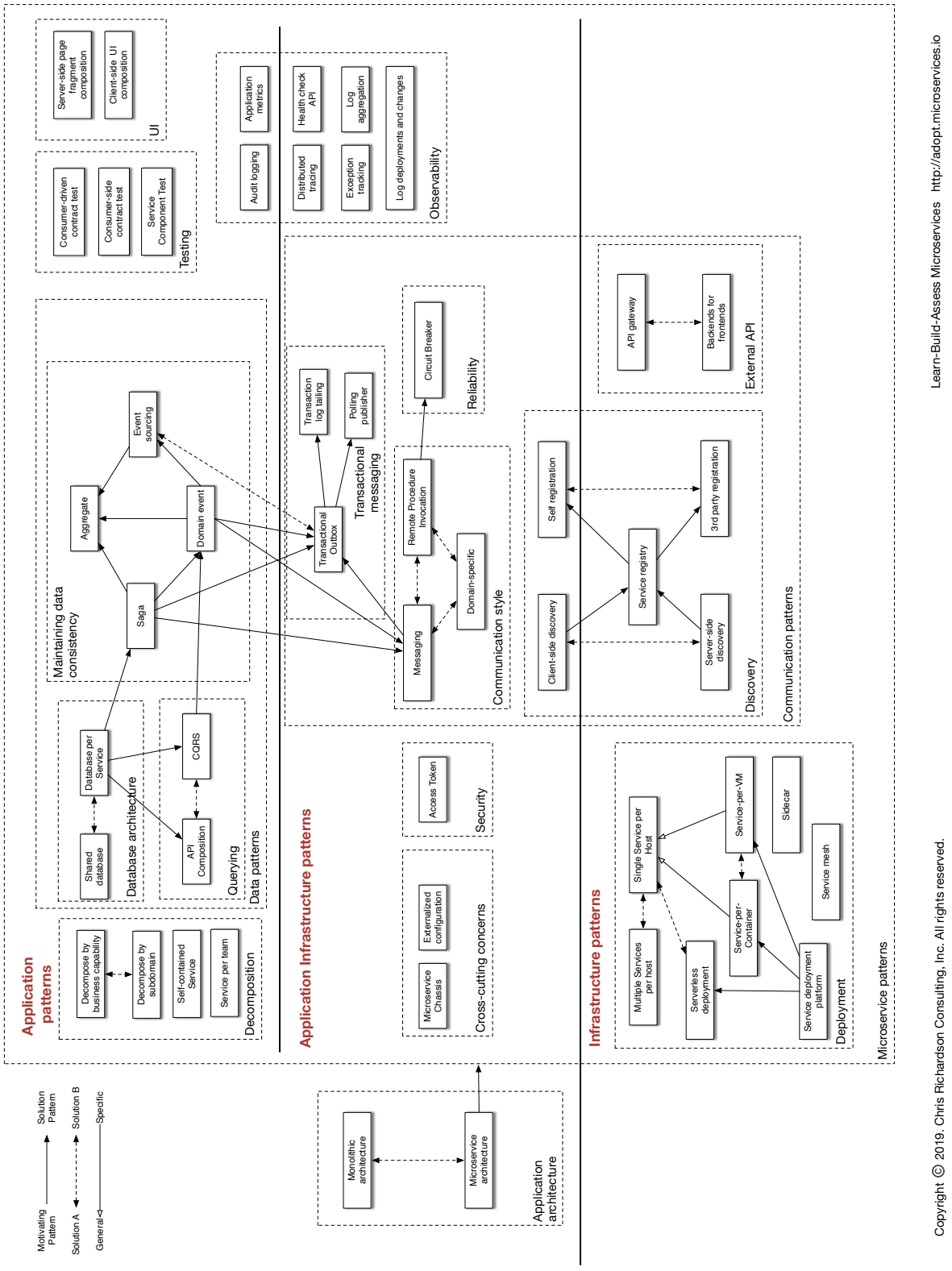

Abbildung A.2.: Microservice Entwurfsmuster im Überblick (Richardson, 2019a)

```
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 6: Apollo Gateway Implementierung eines Sicherheitsmechanismus zur Absicherung gegen unautorisierte Zugriffe

```
query {
    renderjob(id: ID!){
        id
        artist {
          email
        }
    }
}
```

Code 7: Einfache GraphQL-Query zum Anfragen eines spezifischen Renderjobs

```xml
<?xml version='1.0' encoding='UTF-8'?>
<rrJob_submitFile syntax_version="6.0">
 <DeleteXML>0</DeleteXML>
 <SubmitterParameter />
 <Job>
  <rrSubmitterPluginVersion>v8.2.24-2017</rrSubmitterPluginVersion>
  <SceneOS>win</SceneOS>
  <Software>Maya</Software>
  <Version>2018.0300</Version>
  <SceneName>[WORKSPACE_PATH]/scenes/test.mb</SceneName>
  <SceneDatabaseDir>[WORKSPACE_PATH]/</SceneDatabaseDir>
  <Renderer>arnold</Renderer>
  <RequiredLicenses />
  <Camera>camera1</Camera>
  <Layer>masterLayer</Layer>
  <Channel>beauty</Channel>
  <IsActive>True</IsActive>
  <SeqStart>1</SeqStart>
  <SeqEnd>48</SeqEnd>
  <SeqStep>1</SeqStep>
  <SeqFileOffset>0</SeqFileOffset>
  <SeqFrameSet />
  <ImageWidth>960</ImageWidth>
  <ImageHeight>540</ImageHeight>
  <ImageDir>[WORKSPACE_PATH]/images/</ImageDir>
  <ImgFilename>&lt;Layer&gt;/&lt;Channel&gt;/&lt;SceneFile&gt;.</ImgFilename>
  <ImageExtension>.exr</ImageExtension>
  <ImagePreNumberLetter>.</ImagePreNumberLetter>
  <ImageFramePadding>4</ImageFramePadding>
  <ImageSingleOutputFile>False</ImageSingleOutputFile>
  <LocalTexturesFile />
  <rendererVersion>3.0.0.2</rendererVersion>
 </Job>
 <Job>
  <rrSubmitterPluginVersion>v8.2.24-2017</rrSubmitterPluginVersion>
  <SceneOS>win</SceneOS>
  <Software>Maya</Software>
  <Version>2018.0300</Version>
  <SceneName>[WORKSPACE_PATH]/scenes/test.mb</SceneName>
  <SceneDatabaseDir>[WORKSPACE_PATH]/</SceneDatabaseDir>
  <Renderer>arnold</Renderer>
  <RequiredLicenses />
  <Camera>camera1</Camera>
  <Layer>AO</Layer>
  <Channel>beauty</Channel>
  <IsActive>True</IsActive>
  <SeqStart>1</SeqStart>
  <SeqEnd>48</SeqEnd>
  <SeqStep>1</SeqStep>
  <SeqFileOffset>0</SeqFileOffset>
  <SeqFrameSet />
  <ImageWidth>960</ImageWidth>
  <ImageHeight>540</ImageHeight>
  <ImageDir>[WORKSPACE_PATH]/images/</ImageDir>
  <ImgFilename>&lt;Layer&gt;/&lt;Channel&gt;/&lt;SceneFile&gt;.</ImgFilename>
  <ImageExtension>.exr</ImageExtension>
  <ImagePreNumberLetter>.</ImagePreNumberLetter>
  <ImageFramePadding>4</ImageFramePadding>
  <ImageSingleOutputFile>False</ImageSingleOutputFile>
  <LocalTexturesFile />
  <rendererVersion>3.0.0.2</rendererVersion>
 </Job>
</rrJob_submitFile>
```

Code 8: Aufbau einer für rrRemote exportierten XML-Datei zur Überführung eines Renderjobs nach Royal Render

```graphql
extend type Mutation {
  createRenderjob(title: String!): Storage!
  submit(renderjobID: ID!): Renderjob!
  abort(renderjobID: ID!): Renderjob!
  update(renderjobID: ID!, title: String!): Renderjob!
}

extend type Query {
  renderjob(id: ID!): Renderjob!
}

type Renderjob @key(fields: id) {
  id: ID!
  title: String!
  remainingTime: Int!
  artist: Artist!
  status: String!
  storage: Storage! @provides(fields: bucketName fileName)
}

extend type Storage @key(fields: bucketName fileName) {
  bucketName: String! @external
  fileName: String! @external
  renderjobID: ID!
}

extend type Artist @key(fields: uid) {
  uid: String! @external
  renderjobs: [Renderjob!]!
}
```

Code 9: GraphQL-Schema des *Renderjob-Service* für das Backend von rrRemote