Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 358

Entwicklung von Microservices mit zusammensetzbaren API-Bausteinen

Sandro Speth

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Dipl.-Inf. Johannes Wettinger
Beginn am:	1. September 2016
Beendet am:	3. März 2017
CR-Nummer:	C.2.4

Kurzfassung

Der Microservice-Architekturstil findet heute im Bereich des Service Computing häufig Anwendung. Anwendungen, die diesem Architekturstil folgen, werden als menge von „kleinen“ undabhängigen Diensten implementiert. Jeder Dienst ist für eine überschaubare und klar definierte Geschäftsfunktionalität zuständig. Dabei kummunizieren diese sogenannten Microservices nicht nur untereinander über Application Programming Interfaces (APIs), sondern stellen den Benutzern der Anwendung ausgewählte Funktionalitäten über APIs zur Verfügung. Aus diesem Grund spielen APIs im Kontext des Microservice-Architekturstils eine zentrale Rolle. Da der Microservice-Architekturstil dem Prinzip der „Smart Endpoints & Dump Pipes“ folgt, verlagert sich viel Komplexität der Anwendung in die API-Endpunkte. Hinzu kommt, dass typischerweise eine gewisse API-Vielfalt nötig ist, da eine Art von API (z.B. REST, SOAP, Messaging usw.) nicht für alle Fälle die optimale Lösung bezüglich der unterschiedlichen Verwendung darstellt. Um eine solche API-Vielfalt zu ermöglichen, müssen Entwickler viele verschiedene Technologien und API-Frameworks beherrschen. Die Zielsetzung der vorliegenden Arbeit ist diese Situation durch automatische Generierung von APIs zu verbessern. Mit Hilfe von wiederverwendbaren Adaptern kann ein Entwickler solide APIs automatisch erzeugen. Diese Adapter werden zu API-Stacks zusammengefügt, wodurch mehrere verschiedene Arten von APIs für eine einzelne Anwendung generiert werden können. Da die Adapter generisch und wiederverwendbar sind, können diese für mehrere Anwendungen genutzt werden. Ein umfassendes Framework stellt eine Sammlung an Adaptern bereit und ermöglicht damit diese neuartige Methode zur Zusammensetzung von APIs basierend auf vielfältigen Adaptern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Der Microservice-Architekturstil wird in den letzten Jahren viel diskutiert und gewinnt im Bereich des Service Computing zunehmend an Bedeutung. Herkömmliche monolithische Anwendungen sind als große einzelne zusammenhängende Blöcke entwickelte Systeme. Im Gegensatz dazu werden Microservices „als Menge an kleinen und unabhängigen Services“[FLW16] entwickelt, die einzeln einsetzbar sind. Wenn Anwendungen dem Microservice-Architekturstil folgend implementiert werden, spielen Application Programming Interfaces (APIs) eine zentrale Rolle, da die vielen einzelne Services angesteuert werden müssen. Dabei kommunizieren die Microservices nicht nur untereinander über APIs, sondern stellen auch ausgewählte Funktionalitäten den Benutzern der Anwendung über APIs zur Verfügung. Um die Kommunikation zwischen den Microservices und mit den Benutzern zu erleichtern, folgt der Microservice-Architekturstil dem Prinzip der „Smart Endpoints & Dump Pipes“[FL15]. Nach diesem Prinzip wird die Middleware zur Kommunikation der Microservices möglichst einfach gehalten. Da interne Prozeduraufrufe der Monolithen in einem Microservice-Architekturstil zu entfernten Prozeduraufrufen oder anderen komplexeren APIs, wie Messaging APIs oder REST werden, resultiert eine Verlagerung der Komplexität aus der Anwendungslogik in die API-Endpunkte.

Die Anzahl der API-Endpunkte steigt mit der Anzahl der Microservices an. Da bei größeren und komplexeren Anwendungen mehr Microservices entwickelt werden müssen, gibt es mehr API-Endpunkte, die entwickelt und gewartet werden müssen. Eine Art von API ist typischerweise nicht ausreichend für eine Anwendung, da es keine optimale Lösung gibt, die alle Fälle abdeckt. So eignet sich eine Messaging API eher für die Kommunikation zwischen den Microservices, während sich eine REST API besser für die Kommunikation mit den Benutzern der Anwendung eignet. Es empfiehlt sich die Nutzung verschiedener Arten von APIs (API-Vielfalt), was aufgrund der zeitintensiven Mehrfach-Entwicklung der API-Endpunkte oftmals kaum möglich ist.

Bestehende Methoden zur API Entwicklung besitzen verschiedene Defizite und Anforderungen, die erfüllt werden müssen. Bei diesen Methoden müssen die Entwickler oft selbst jeden API-Endpunkt in verschiedene API Typen entwickeln und warten. Um eine größere API-Vielfalt zu erhalten, können die Arten von APIs automatisch mit Hilfe von wiederverwendbaren API Adaptern generiert werden. Für die Generierung nach dieser Methode muss ein API-Endknoten lediglich in einer API-Art vorliegen, und wird durch die Adapter auf andere API-Arten abgebildet. Dazu werden die API-Endpunkte über eine Definitionssprache festgelegt.

Um Microservices und Adapter zu verpacken und auszurollen bietet sich eine Container Technologie wie Docker an[JP15]. Dies bewirkt, dass die Microservices plattformunabhängig

ausführbar sind. Zur Verwaltung der Adapter werden diese ebenfalls in Containern virtualisiert.

Diese Arbeit stellt verschiedene API Konzepte, wie Remote Procedure Call und REST vor und geht besonders auf den Microservice Architekturstil ein. Zusätzlich werden die Definitionssprachen Swagger[Tea14], für REST APIs, und Protocol Buffers 3[Goo17a], für Remote Procedure Call APIs, genauer betrachtet. Des Weiteren wird die Technologie Docker als etablierter Repräsentant der Container Virtualisierung vorgestellt, die aktuell häufig für Microservices verwendet wird. Anschließend wird in dieser Arbeit auf eine API Entwicklungsmethode eingegangen, bei der verschiedene API Typen mit API Adaptern generiert werden können. Insbesondere wird erläutert, inwiefern diese Methode die beschriebenen Anforderungen und Defizite an API Entwicklungsmethoden erfüllt. Der Entwurf und die Umsetzung eines konkreten Adapters zur Abbildung einer RPC API auf eine REST API ist ein weiterer Bestandteil dieser Arbeit. Hierfür wird gRPC als eine effiziente und Standard-basierte (HTTP2) Implementierung von RPC verwendet und REST mit dem HTTP Protokoll angewandt.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen und verwandte Arbeiten: Zunächst werden die Grundlagen dieser Arbeit beschrieben. Es wird auf verschiedene Konzepte wie Microservices, Container Virtualisierung oder unterschiedliche API Typen eingegangen. Zusätzlich werden verwandte Arbeiten vorgestellt.

Kapitel 3 – API Entwicklungsmethode und Framework beschreibt eine konkrete Methode, wie sich APIs über wiederverwendbare API Adapter erzeugen lassen. Diese Methode wird in einem konkreten Framework implementiert, welches ebenfalls in diesem Kapitel vorgestellt wird.

Kapitel 4 – Design und Implementierung beschreibt das Design und die Implementierung eines gRPC-REST API Adapter. Insbesondere wird hierbei die Abbildung der gRPC API auf eine REST API betrachtet.

Kapitel 5 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte für zukünftige Forschung vor.

2 Grundlagen und verwandte Arbeiten

Das folgende Kapitel ist in zwei Abschnitte unterteilt. Der erste Abschnitt stellt die grundlegenden Konzepte vor, die in dieser Bachelorarbeit verwendet werden. Anschließend werden verschiedene Technologien vorgestellt, die in den Konzepten Verwendung finden und im praktischen Teil der Bachelorarbeit zur Implementierung der Adapter genutzt werden.

2.1 Konzepte

Dieser Abschnitt ist in fünf Bereiche unterteilt und behandelt die für die Bachelorarbeit relevanten Grundlagen und entsprechende verwandte Arbeiten. Abschnitt 2.1.1 bietet einen Einstieg in das Thema Application Programming Interface (*API*), in dem erklärt wird, was unter einer API zu verstehen ist. Darüber hinaus wird eine kurze Einführung in verschiedene Arten von APIs gegeben, und diese werden klassifiziert. Anschließend werden in den Abschnitten 2.1.2 und 2.1.3 zwei Arten von APIs, Representational State Transfer und Remote Procedure Call, im Detail erklärt. Abschnitt 2.1.4 stellt das Konzept der Container Virtualisierung und dessen Vorteile gegenüber normalen virtuellen Maschinen vor. Durch Container Virtualisierung können verschiedene Konzepte, wie das API Deployment und Packaging erleichtert werden. Als Letztes werden in Abschnitt 2.1.5 die beiden Architekturstile Service Oriented Architecture und Microservices vorgestellt und miteinander verglichen.

2.1.1 Application Programming Interfaces (API)

Jede Server-Applikation bietet mehrere Funktionen für einen Client an, die über eine Schnittstelle verfügbar sind. Diese Schnittstelle kann in unterschiedlichen Technologien implementiert sein und verschiedene Funktionalitäten bereitstellen, zum Beispiel REST, Funktionsaufrufe, Prozeduraufrufe, synchroner Nachrichtenaustausch und viele mehr. Alle diese Schnittstellen legen die Möglichkeit der Interaktion mit der Applikation fest und werden als Applikation Programming Interface (kurz *API*) bezeichnet. Es gibt verschiedene Möglichkeiten APIs zu klassifizieren. Im Folgenden werden mehrere Arten von APIs kurz vorgestellt und klassifiziert.

Plugin/Library APIs:
Unter einer Library API versteht man eine API, die direkt als Bibliothek eingebunden werden

kann. Es gibt API Bibliotheken in verschiedenen Sprachen, z.B.: Python, die in das eigentliche Programm eingebunden werden können. Entwickler müssen Codestücke nicht mehr neu schreiben, sondern können existierende und bereits getestete Codestücke verwenden, ohne die Größe (Lines of Code) und dadurch die Komplexität des eigenen Codes zu erhöhen. Dabei wird zwischen verschiedenen Unterkategorien von Library APIs unterschieden. Es gibt APIs, die ihre Funktionalität self-contained mitliefern, wie z.B. die Util-Libraries aus Java[1]. Außerdem gibt es APIs, die im Hintergrund Endpoint APIs verwenden, wie die Google Client Libraries[2]. Die jeweilige Endpoint API selbst kann dann z.B. eine REST oder Messaging API sein. Google bietet über APIs Zugriff auf verschiedene Google Dienste, wie Google Maps oder YouTube. Um den Zugriff auf diese APIs zu erleichtern, bietet Google die sogenannten Client Libraries für einige Programmiersprachen an, bei denen die eigentlichen Endpoint APIs im Hintergrund aufgerufen werden.

Endpoint APIs:
In einer Endpoint API erfolgt die Interaktion mit dem jeweiligen Service über sogenannte Endpunkte. Diese Endpunkte können beispielsweise Verbindungspunkte zu einem Webservice oder Adressen von Ressourcen eines Services sein, welche typischerweise als Uniform Resource Identifier (*URI*) repräsentiert werden. Die Endpoint APIs können in REST APIs, RPC APIs, Messaging APIs, ereignisgesteuerten APIs und Streaming APIs unterteilt werden. Diese Arten von Endpoint APIs werden im Folgenden kurz erläutert.

REST API:
Unter einer Representational State Transfer (*REST*) API[Mas11] versteht man eine auf Ressourcen basierende API. Die URIs zeigen hierbei direkt auf die jeweiligen im Service hinterlegten Ressourcen, um diese manipulieren zu können. Oftmals implementieren REST Services HTTP als Protokoll zur Übertragung der Ressourcen. Über die HTTP Methoden[W3C17b] können diese dann verändert, gelöscht, hinzugefügt oder abgefragt werden. Der Architekturstil REST wird in Abschnitt 2.1.2 dieser Bachelorarbeit noch genauer vorgestellt.

RPC API:
In Remote Procedure Call (*RPC*) APIs werden ausgewählte Operationen des Servers bereitgestellt. Diese Operationen bilden die jeweiligen Endpunkte des Services und können über entfernte Proceduraufrufe von einem Client genutzt werden. Sowohl auf Server, als auch auf der Clientseite, existieren Stubs, die die vorhandenen Funktionen implementieren. Die eigentliche Kommunikation zwischen Client und Server findet zwischen den Stubs statt und erfordert unter Anderem definierte Nachrichtenformate, sowie Protokolle zur Nachrichtenübertragung. Solche Nachrichtendefinitionen lassen sich mit dem in Abschnitt 2.2.3 vorgestellten Protocol Buffers schreiben. Mehr Details zu RPC werden in Abschnitt 2.1.3 vorgestellt.

Messaging API:
Server und Client können aufgrund verschiedener Ursachen in unterschiedlichen Program-

[1] https://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html
[2] https://developers.google.com/api-client-library/

miersprachen implementiert sein. Zum Beispiel können beide Kommunikationspartner zu unterschiedlichen Zeiten von unterschiedlichen Entwickerteams geschrieben worden sein. Um eine passende Kommunikation beider Instanzen zu gewährleisten, kann ein sprachunabhängiger Bereich nötig sein. Messaging-Systeme können als unabhängiges Sprachrohr zwischen den Applikationen dienen. Dies erlaubt beiden Kommunikationspartnern über ein gemeinsames Nachrichtenparadigma, wie einen Nachrichtenbus, zu kommunizieren.

Messaging erlaubt einen „send and forget“ Ansatz für eine asynchrone Kommunikation. Hierbei muss der Sender nicht warten, bis der Empfänger die Nachricht empfangen und bearbeitet hat. Er kann die Nachricht „senden“ und anschließend „vergessen“, bis er eine Antwort erhält. Insbesondere muss der Sender nicht warten, bis das Messaging-System die Nachricht gesendet hat, da es für diesen genügt, sobald das Messaging-System die Nachricht erhält. Sobald dies geschehen ist, kann der Sender sich weiter seinen Aufgaben widmen, da das Messaging-System parallel dazu die Nachricht übermittelt. Der Empfänger kann nach Empfangen/Bearbeiten der Nachricht eine Antwort in Form eines Acknowledgements oder eines Ergebnisses an den Sender zurücksenden. Dies kann beispielsweise als Callback Mechanismus realisiert werden, bei dem jede Nachricht eine Antwort erfordert. Dadurch entsteht eine lose Kopplung der Kommunikationspartner. Aufgrund der asynchronen Kommunikation können Sender und Empfänger in ihrem eigenen Tempo arbeiten, weil sie jeweils nicht auf den Kommunikationspartner warten müssen. Da Sender und Empfänger nicht blockiert werden, können beide weiter arbeiten, während der Partner eine Nachricht beantwortet und dadurch einen höheren Durchsatz erzeugen.

Messaging erlaubt einen Einsatz von Applikationen, die ohne aktive Verbindung zum Netzwerk arbeiten. Sobald eine Netzwerkverbindung vorhanden ist, können die Nachrichten synchronisiert werden, da die Nachrichten in einer Warteschlange gespeichert werden. Dabei tritt das Messaging-System in der Rolle des Vermittlers zwischen Sender und Empfänger auf. Falls ein Kommunikationspartner die Verbindung verliert, muss dieser, anstatt zu jedem Kommunikationspartner eine Verbindung aufzubauen, sich nur zum Messaging-System verbinden.

Messaging-Nachrichten sind, wie HTTP Nachrichten, vergleichbar mit Briefen. Jede Nachricht besteht aus einem Header und einem Body. Der Header enthält die Serveradresse, sowie die Metadaten, wie die originale Struktur der Daten, und weitere Informationen. Der Body enthält die eigentlichen versendeten Daten.

Es exitieren jedoch Probleme bei der Nachrichtenübermittlung bei Messaging APIs. Ein Nachrichtenkanal garantiert zwar die Übermittlung, trifft jedoch keine Aussagen darüber, wann eine Nachricht übermittelt wird. Dies ist problematisch, falls Nachrichten voneinander abhängen. Dadurch werden Realtime oder Near-Realtime mit Messaging nur schwer umsetzbar. Des Weiteren gibt es einige Szenarios, in denen nicht mit „send and forget“ gearbeitet werden kann, da bei diesen der Sender erst mit der Arbeit nach der Antwort der Nachricht fortfahren kann. Für diese synchronen Szenarien muss ein Mittelweg zwischen dem synchronen und asynchronen Ansatz gefunden werden.

Streaming API:
Bei einer Streaming API bleibt die Verbindung zwischen Server und Client so lange wie möglich erhalten. Dabei gibt es verschiedene Szenarien.

1. Serverseitig: Der Client baut einen Kanal zum Server auf und sendet die Anfrage, ähnlich zu einer herkömmlichen RPC API. Anstatt eine einzelne Antwort zu erhalten, erhält der Client einen Strom an Antworten. Sobald neue Daten verfügbar sind, sendet der Server diese über die offene Verbindung an den Client. Dies hat zur Folge, dass der Client nicht explizit den Server nach neuen Informationen anfragen muss, sondern diese direkt von diesem zugesandt bekommt.
2. Clientseitig: Ähnlich zum serverseitigen Stream wird beim client-seitigen Stream ein Kanal zwischen Server und Client aufgebaut. In diesem Fall kann der Client einen Strom an Anfragen an den Server senden, die bei diesem dann gesammelt bearbeitet und mit einer einzelnen Antwort beantwortet werden.
3. Beidseitig: Beim beidseitigen Stream können sowohl Client als auch Server einen Strom an Anfragen/Antworten über den Kanal senden.

Streaming APIs bieten den Vorteil, dass aufgrund des offenen Kanals die Netzwerklatenz zum auf- und abbauen einer Verbindung reduziert werden kann. Für RPC APIs bietet das in 2.2.4 vorgestellte *gRPC*-Framework die Möglichkeit die oben beschriebenen Varianten an Streams zu Implementieren. Das auf TCP basierende WebSocket bietet einen bidirektionalen Stream zwischen einer Webanwendung und einem Webserver. Mit dem von W3C entwickelten *Server-sent Events*[Hic09] können ebenfalls kontinuierlich Daten von einem Server an einen Client über einen offenen Kanal gesendet werden.

Weitere Klassifikationsdimensionen:
Die verschiedenen Arten an APIs lassen sich außerdem durch deren Eigenschaften unterscheiden. So kann man diese in synchrone und asynchrone APIs aufteilen. Einige Arten an APIs lassen sich auch unter dem Begriff der event-driven APIs sammeln.

Synchrone API:
Synchrone APIs bieten eine Kommunikation zwischen Sender und Empfänger, bei der beide Kommunikationspartner aktiv an der Verbindung teilnehmen. Eine Anfrage des Senders wird über einen festen Kommunikationskanal übermittelt und beim Empfänger direkt nach eintreffen bearbeitet. Von den oben genannten Endpoint APIs gehören RPC und Streaming APIs zu den synchronen APIs. Bei den Library APIs hängt dies speziell von der jeweiligen Bibliothek ab. Da die Java Util Bibliotheken beispielsweise über Funktionsaufrufe direkt aufgerufen werden, zählen diese zu synchronen APIs.

Asynchrone API:
Asynchrone Kommunikation erfordert im Gegensatz zu synchronen APIs nicht die aktive Teilnahme beider Partner. Bei einer solchen API kann der Sender die Anfrage an den Empfänger senden, ohne einen speziellen Kommunikationskanal offen halten zu müssen. Der Empfänger kann die Anfrage bearbeiten, sobald dieser verfügbar ist. Wähenddessen kann der Sender

andersweitig weiterarbeiten. Zu den asynchronen APIs zählen z.B. REST oder Messaging APIs, da bei diesen keine direkte Verbindung geöffnet wird. Bei beiden APIs kann der Client weiterarbeiten bis die Antwort des Servers eintritt.

Event-driven API:
In ereignisgesteuerten APIs rücken „Ereignisse als zentrales Strukturierungskonzept“[BD10] für die Interaktion der zusammenarbeitenden Applikationen in den Fokus. Wird ein Ereignis ausgelöst, reagiert die jeweilige Applikation entsprechend. Um dies zu veranlassen muss für jedes Ereignis ein entsprechender Listener (auch Event-Handler genannt) implementiert sein, der auf das Eintreten des Ereignisses wartet und anschließend die passenden Aktionen ausführt.

Im Gegensatz zu einer Messaging API wird bei der ereignisgesteuerten API keine Nachricht direkt übermittelt. Vielmehr wird die Nachricht ohne spezielles Ziel versandt und muss durch einen Listener beachtet werden. Dies birgt das Risiko, dass eine Nachricht nicht bearbeitet wird, sofern keiner der vorhandenen Listener darauf reagiert. Da die Nachrichten nicht direkt übermittelt werden, benötigt der Sender keine Informationen über die Schnittstelle oder Implementierung des Empfängers, was zu einer losen Kopplung der Komponenten führt. Dies unterscheidet sich zu den bisher vorgestellten Anfrage-Antwort Interaktionen, wie beispielsweise RPC, da Events nur darlegen, dass etwas geschehen ist, jedoch keine Anfrage auf eine spezielle Bearbeitung geben.

Eine REST API kann durch Ereignisse unterstützt werden. Trifft eine Antwort des Servers ein, kann ein Ereignis ausgelöst werden, dass die aktuelle Arbeit pausiert und eingehende Antwort bearbeitet.

2.1.2 Representational State Transfer

Der Architekturstil **Re**presentational **S**tate **T**ransfer (kurz *REST*) findet häufig in Webservices Anwendung. REST Server sind, entgegen vieler anderen Architekturstilen, nicht Operationen-basiert, sondern Ressourcen-basiert. Das bedeutet in REST werden Repräsentationen von Ressourcen zwischen Server und Client übertragen. Diese Repräsentationen sind oft im JSON (mehr dazu in Abschnitt 2.2.1) oder XML Format gespeichert. Änderungen der Ressourcen werden daher auf den Repräsentationen ausgeführt. So kann beispielsweise eine Ressource ein Produkt aus einem Webshop sein. Das Produkt besitzt viele Eigenschaften, wie einen Namen, einen Preis, ein Gewicht und eine Größe. Die Repräsentation kann ein JSON Objekt sein, dass das Produkt beschreibt. Im Folgenden wird der Begriff Ressource als Synonym für die Ressourcen-Repräsentation genutzt.

Ein REST Server ermöglicht einem Clienten über die CRUD Operationen (*Create*, *Retrieve*, *Update*, *Delete*) das Hinzufügen, Verändern, Empfangen und Löschen der Ressourcen. Oft implementieren REST Services HTTP (siehe Abschnitt 2.2.7) als Protokoll zur Ressourcenübertragung. Es ist keine Pflicht HTTP als Protokoll zu nutzen. So kann REST auch andere

Protokolle verwenden, um die oben genannten CRUD Operationen auf Ressourcen durchzuführen. In dieser Bachelorarbeit wird hauptsächlich HTTP für REST verwendet, da dies eine oft genutzte Implementierung des Protokolls ist. Die HTTP-Repräsentation der CRUD Operationen sind *PUT*, *POST*, *UPDATE* und *DELETE* (oft auch als HTTP Verben[W3C17a] bezeichnet) und werden auf URI Pfaden der Ressourcen aufgerufen. Es gilt zu beachten, dass jede dieser HTTP Verben idempotent ist. Mehrfaches Aufrufen eines Requests mit einer bestimmten URI führt daher immer zum selben Ergebnis. So ist es beispielsweise nicht möglich zwei GET Anfragen auf die URI `\products\{id}` mit unterschiedlichen Ergebnissen auszuführen. Jedoch können mehrere URIs auf die selbe Ressource verweisen. Weitere Informationen zu HTTP finden sich in Abschnitt 2.2.7.

REST fordert bei der Implementierung eines Servers die Einhaltung von sechs Prinzipien. Nur bei Einhaltung aller sechs Prinzipien kann ein Server wirklich als *RESTful Server* bezeichnet werden. Im Folgenden betrachten wir diese sogenannten *Bedingungen* (engl. Constraints)[Mas11] im Detail.

Uniform Interface:
Das Prinzip erfordert die Definition einer einheitlichen Schnittstelle zwischen Server und Clienten um die Architektur zu vereinfachen und besteht aus vier Eigenschaften.

1. *Repräsentation der Ressourcen* - Da REST ressourcenbasiert ist, werden die einzelnen Ressourcen über entsprechende URIs identifiziert. Diese URIs werden in den jeweiligen Requests als Adresse für den Zugriff und die Manipulation der Ressourcen genutzt. Der Response-Body enthält die passende Repräsentation der (veränderten) Ressource.
2. *Ressourcenadressierbarkeit* - Ressourcen werden durch URIs adressiert. In Verbindung mit der URL des REST Services ermöglichen die URIs einen standardisierten Zugriff des Clienten auf die Ressourcen.
3. *Selbstbeschreibende Nachrichten* - Durch Verwendung von Standardmethoden, wie die HTTP Verben, lassen sich selbstbeschreibende Nachrichten erzeugen. So kann der Server dahingehen implementiert werden, dass beispielsweise ein *GET* auf der URL `/products/3` das Produkt mit der ID 3 zurückgibt.
4. *HATEOAS* - Die letzte Eigenschaft schränkt den Zugriff des Clienten auf die Ressourcen ausschließlich über die URL des Servers ein.

Stateless:
Das *Stateless*-Prinzip ist essentiell für REST. Dieses Prinzip fordert Zustandslosigkeit der API auf der Serverseite. Das bedeutet, dass jeder Request an den Server genügend Informationen beinhalten muss, um es dem Server zu ermöglichen, den Request korrekt zu bearbeiten. Der Zustand wird indirekt über die Repräsentation der Ressourcen, die bei einer Response übertragen werden, gespeichert. Falls der REST Service einen anderen Zustand besitzt, muss dieser auf der Client Seite gespeichert sein, nicht auf der Serverseite. Dieses Prinzip ermöglicht eine bessere Skalierbarkeit eines Services als in zustandbehafteten Services, da Requests auf

verschiedene Maschinen verteilt werden können, ohne auf einen speziellen Serverzustand achten zu müssen.

Cachable:
Das Prinzip fordert, dass Clienten die Antworten implizit oder explizit zwischenspeichern können. Die Einhaltung dieses Prinzips ermöglicht eine Reduzierung der Client-Server-Kommunikation, da beispielsweise mehrfach gleiche Requests reduziert werden.

Client-Server:
Die Trennung zwischen Client und Server durch die Verbindung mit einer einheitlichen Schnittstelle ermöglicht eine strikte Trennung der Aufgaben beider Seiten. Der Server ist ausschließlich für die Bereitstellung des Dienstes und die Speicherung der Ressourcen zuständig, während die Clienten die Ressourcen anfragen und die Benutzerschnittstelle beinhalten.

Layered System:
Der Client kann in einem RESTful Service keine direkte Verbindung zum Server vorraussetzen. Da RESTful Services mehrschichtig aufgebaut werden sollen, kann spezielle Software oder Hardware zwischen den Clienten und dem Server agieren. Diese verschiedenen Schichten ermöglichen eine bessere Skalierbarkeit als in Services mit direkter Verbindung zum Client, da hier beispielsweise Daten zwischengespeichert werden, oder irgendwelche Filter angewandt werden können.

Code on Demand:
Das letzte Prinzip ist die einzige optionale der sechs Einschränkungen an einen RESTful Service. Dieses Prinzip besagt, dass der Server temporär Code an den Clienten übertragen kann, der diesen ausführt.

Eine Verletzung von nur einem Prinzip (außer Code on Demand) bedeutet zwangsweise, dass der Service nicht mehr RESTful ist. Die Entwickler von RESTful Services müssen alle der obligatorischen Prinzipien einhalten. Dadurch ergeben sich eine Reihe an Vorteilen. Ein RESTful Service besitzt eine erhöhte Skalierbarkeit und Portabilität. Außerdem lässt sich die Schnittstelle mit wenig Auswirkungen auf den Rest der Services verändern.

Es existieren Requests die eine direkte Antwort nicht ermöglichen. So kann der Server beispielsweise bei einem POST Requests einer großen Ressource zur Erstellung dieser lange benötigen. Für solche Anfragen muss geklärt werden, wie sie gehandhabt werden. Es ist möglich, dass der Client wartet, bis der Server die Anfrage erfolgreich bearbeitet hat und den entsprechenden HTTP Response Code zurücksendet. Allerdings ist dies oftmals nicht gewünscht, da der Client in dieser Zeit blockiert und daher keine weiteren Aktionen durchführen kann. Im Fall eines POST Requests ist es ebenfalls keine gute Idee direkt den Response Code mit der Location der zukünftigen Ressource zurückzugeben und diese erst anschließend zu erstellen. Dabei würde der Client zwar nicht blockieren, die Anfrage könnte allerdings anschließend scheitern, was der Client nicht mitbekommen würde. Alternativ kann es dabei passieren, dass der Client auf die Ressource zugreifen möchte, obwohl diese noch nicht erzeugt wurde.

Um dieses Problem zu lösen existiert das REST Command Pattern[3], das im Folgenden vorgestellt wird. Das Command Pattern beschreibt ein Verfahren, um mit sog. „long-running jobs“ umgehen zu können. Hierfür benötigt man eine Warteschlange, in der die long-running jobs eingereiht werden. Die Warteschlange wird periodisch von einem Prozess abgearbeitet. Requests werden nach dem Eintreffen direkt mit dem HTTP Response Code `202 Accepted` (siehe 2.2.7) beantwortet. Als URL wird nun nicht mehr die Location der Ressource, die erstellt werden soll, sondern die Position der Job Ressource in der Warteschlange zurückgegeben. Die Job Ressource kann verschiedene Werte beinhalten, wie einen Status der Bearbeitung (Running, Complete, Canceled). Um den Status des Jobs zu erfragen und damit herauszufinden ob die zu erstellende Ressource bereits final verfügbar ist, kann die Job Ressource über die zuvor zurückgegebene URL über ein *GET* Request angefragt werden. Sobald die zu erstellende Ressource erzeugt wurde sollte bei einem *GET* Requests auf die Job Ressource ein `303 See Other` Response Code und die Location der erzeugten Ressource zurückgegeben werden.

Nun gibt es zwei Möglichkeiten das Löschen der Job Ressource zu implementieren. Für die erste Möglichkeit muss der API Client einen *DELETE* Requests senden. Bis dahin antwortet der Server mit 303. Nachfolgende *GET* Requests auf die Job Ressource werden mit dem Response Code `404 Not Found` beantwortet. Die zweite Möglichkeit erfolgt durch Garbage Collection. Hierbei löscht der Server die Job Ressource nach abschließen der Aufgabe und antwortet an nachfolgende *GET* Requests mit dem Response Code `410 Gone`. Durch dieses Verfahren kann der Client nach Senden der Anfrage weiter seine Arbeit fortfahren und muss nur hin und wieder den Status der Job Ressource anfragen.

2.1.3 Remote Procedure Calls

Remote Procedure Calls (kurz *RPC*) ist ein weit verbreiteter Mechanismus zum Nachrichtenaustausch von verteilten Server-Client-Anwendungen. Beim RPC können Prozesse (Clients) Prozeduren in anderen Adressräumen (Server) synchron aufrufen. Hierbei muss der Client während der gesamten Anfrage eine aktive Verbindung zum Server haben. Da der Prozeduraufruf synchron erfolgt blockiert der Client nach dem Senden der Anfrage und kann erst mit der Bearbeitung fortfahren, sobald die Antwort des Servers angekommen ist. Die vom Clienten gesendete Anfrage enthält die auszuführende Funktion mit allen benötigten Parametern. Der Server muss die angegebene Funktion implementiert haben. Ist dies der Fall, führt der Server die Funktion mit den gegebenen Parametern aus und schickt die Antwort an den Clienten zurück. Der beim RPC verwendete Nachrichtentransfer zwischen Client und Server kann z.B. in TCP oder UDP implementiert werden. Die Schnittstelle wird vom Server bereitgestellt und kann beispielsweise durch eine Interface Definition Language (*IDL*), wie das in Abschnitt 2.2.3 vorgestellte Protocol Buffers definiert und durch einen entsprechenden Compiler erzeugt werden.

[3] http://farazdagi.com/blog/2014/rest-long-running-jobs/

Sowohl Clients als auch Server müssen für den korrekten Nachrichtenaustausch die zu sendenden Datenstrukturen serialisieren und deserialisieren können. Hierbei können Sender und Empfänger in unterschiedlichen Programmiersprachen mit unterschiedlichen Datenstrukturen implementiert sein. Unter Umständen existiert eine gegebene Datenstruktur nur in einer der beiden Programmiersprachen. Für diese muss eine korrekte Serialisierung und Deserialisierung erfolgen. Ein weiteres Problem, das dadurch entsteht, ist die Konvertierung von Datentypen. Da Sender und Empfänger nicht zwangsläufig in der selben Programmiersprache implementiert sind, können diese auch verschiedene Arten von Datentypen besitzen. Dadurch ist es sinnvoll standardisierte Datenrepräsentationen zu nutzen. Hierfür bieten sich das in Abschnitt 2.2.1 vorgestellte Dateiformat JSON[Bra14] oder ähnliche Formate, wie YAML[BKEI05] oder XML[Bra+] an.

Treten beim RPC Aufruf keine Fehler auf garantiert RPC, dass eine Operation genau einmal (*exactly once*) ausgeführt wird. Es können allerdings auch unterschiedliche Fehler auftreten, wie die Kommunikationsfehler. Diese treten auf, wenn entweder die Anfrage oder das Ergebnis fehlerhaft übertragen wird, oder bei der Übertragung verloren geht. Zusätzlich werden die sogennanten Knotenfehler betrachtet. Bei Knotenfehlern kann der Server die angeforderte Prozedur nicht beenden, wenn ein Hardware- oder Softwarefehler vorliegt. Ein weiterer Knotenfehler liegt vor, wenn der Client abstürzt bevor der RPC Aufruf beendet wurde. Dies kann ebenfalls durch Hardware- oder Softwarefehler passieren.
Betrachten wir im Folgenden, zusätzlich zu der oben genannten *exactly-once* Strategie, die entsprechenden Fehlersemantiken:

- *Maybe:* Die Anfrage wird entweder überhaupt nicht, oder genau einmal ausgeführt. Es gilt zu beachten, dass der Client nicht über den Zustand des Aufrufs informiert wird. Er kann folglich nicht beurteilen, ob ein Fehler aufgetreten ist, oder ob die Prozedur noch aufgerufen wird.
- *At-least-once:* Eine Anfrage wird mindestens einmal ausgeführt. Eine mehrfache Ausführung der Anfrage ist allerdings möglich, weshalb diese Art von Fehlersemantik nur für idempotente Operationen geeignet ist. Nur bei idempotenten Operationen verändert sich das Ergebnis nicht nach mehrfachem ausführen. Bei Knotenfehlern kann die Ausführung nicht garantiert werden.
- *At-most-once:* Eine Anfrage wird höchstens einmal (ohne Knotenfehler genau einmal) ausgeführt.

Eine von Google Inc. bereitgestellte Implementierung von Remote Procedure Call wird in Abschnitt 2.2.4 vorgestellt. Es existieren noch weitere Implementierungen von Remote Procedure Call, wie *Thrift*[SAK07] oder *JSON-RPC*[Gro+12].

2.1.4 Container Virtualisierung

Wenn Anwendungen in Containern visualisiert werden, spricht man von der sogenannten Container Visualisierung. Dadurch sind die Anwendungen unabhängig vom jeweiligen Betriebssystem, auf dem der Container läuft, und müssen nur an diesen angepasst werden.

Entgegen herkömmlicher Virtualisierung entfällt der Hypervisor, die Schicht zwischen der Hardware (oder ggf. Host-Betriebsystem) und den virtuellen Betriebsystemen der normalen virtuellen Maschinen (VMs). Stattdessen teilen sich die verschiedenen virtuellen Maschinen, auch Container genannt, direkt den Kernel des Host-Betriebssystems. Aus diesem Grund entstehen einige Vorteile für die containerbasierte Virtualisierung. So muss sich entgegen herkömmlicher Virtualisierung, bei der jede virtuelle Maschine ein eigenes Betriebssystem nutzt, nur ein einziges Betriebssystem um Hardwareaufrufe kümmern. Dadurch entfällt der bei der Hypervisor Virtualisierung entstehende Overhead. Zusätzlich müssen die von mehreren Container genutzten Teile des Betriebssystems nur einmal gelesen werden, was dazu führt, dass die Container deutlich leichtgewichtiger und ressourcenschonender als normale virtuelle Maschinen sind. Dadurch entstehen jedoch auch Nachteile. Da jeder Container direkt das Host-Betriebssystem nutzt, kann dies zu Sicherheitsproblemen führen. So benötigt der Benutzer meist root-Rechte, um einen Container zu starten. Ist der Container erst einmal gestartet, so ist dieser beispielsweise in der Lage System-Ressourcen mit anderen Containern zu teilen oder weitere Container zu starten.

Container Virtualisierung hat Vor- und Nachteile für die darauf liegenden Anwendungen. So lassen sich große Softwaremonolithen in Microservices unterteilen, die jeweils in eigenen Containern virtualisiert werden. Diese Container sind leichter hochzuskalieren, als große Softwaremonolithen. Dadurch können auch einzelne Funktionalitäten größerer Applikationen skaliert werden, ohne die Skalierung der kompletten Applikation vornehmen zu müssen. Dies erhöht die Geschwindigkeit der Anwendung, spart Ressourcen und falls benötigt Lizenzen Durch die Container Virtualisierung entsteht jedoch eine geringe Isolation zwischen den Containern und deren Anwendungen. Teilen sich Anwendungen mehrerer Container Daten, so kann das Ändern des Datensatzes durch die Anwendung eines Containers die Anwendung eines anderen Containers beeinflussen. Falls eine Anwendung innerhalb eines Docker Containers root-Rechte besitzt, kann diese Anwendung diverse Änderungen auf dem Host-Betriebssystem vornehmen. Diese Nachteile sind bei normalen Virtuellen Maschinen nicht gegeben, da dort eine bessere Isolation der jeweiligen Anwendungen herrscht und eine Anwendung nur root-Rechte für das VM-Betriebssystem erhalten kann.

In Abschnitt 2.2.6 wird eines der bekanntesten Beispiele für eine containerbasierte Virtualisierung, Docker, vorgestellt.

2.1.5 SOA und Microservices

Der folgende Abschnitt stellt die zwei Architekturstile *Microservices* und *Service-Oriented Architecture* (*SOA*) vor und vergleicht diese. Betrachten wir zuerst die Definition und Werte von Microservices.

Der aufkommende und stark gehypte Microservices Architekturstil[New15][Fow17] wird aktuell häufig diskutiert. Trotzdem existiert keine klare Definition, was ein Microservice tatsächlich ist. Der Softwareentwickler, und einer der Erstunterzeichner des Agilen Manifests, Martin Fowler und sein Kollege James Lewis beschreiben den Architekturstil Microservice als ein „Ansatz für die Entwicklung einer einzigen Anwendung in Form einer Reihe kleiner Services, die jeweils in einem eigenen Prozess laufen und die durch einfache Mechanismen kommunizieren - oft durch HTTP-Ressourcen-basierte APIs“[FL15]. Die Dienste orientieren sich an Geschäftsfunktionalitäten und werden durch vollautomatisches Deployment sind diese Dienste unabhängig voneinander aktualisierbar und austauschbar. Es gibt nur eine minimale zentrale Verwaltung dieser Dienste, die unterschiedliche Programmiersprachen wie auch Datenspeicher-Technologien verwenden können. Um den Microservices Architekturstil besser zu verstehen, bietet es sich an diesen mit einem monolithischen Architekturstil zu vergleichen. Microservice Architekturen sind gegensätzlich zu monolithischen Architekturen. Sie werden nicht in einem einzelnen großen Block entwickelt und gewartet, sondern in einer Reihe an kleinen und unabhängigen Diensten. Bis heute existiert keine allgemein anerkannte oder etablierte Meinung, wie groß Microservices sein sollten[FLW16]. Nach Fowler folgt der Microservice-Architekturstil dem Prinzip der „Smart Endpoints & Dump Pipes“[Fow17], was bedeutet, dass die Middleware zur Kommunikation zwischen und mit Microservices so einfach wie möglich gehalten wird. Daraus resultiert, dass mehr Komplexität in die API-Endpunkte verlagert werden muss.

Wie jeder Architekturstil besitzen Microservices verschiedene Vor- und Nachteile.
Da Microservices kleine abgeschlossene Softwareprodukte darstellen, lassen sie sich effizient von kleinen Teams entwickeln. Durch die kleinen Entwicklerteams lassen die Microservices sich stärker Modularisieren. Dieser Vorteil schwindet meist mit zunehmender Teamgröße, wie man sie oftmals für große Monolithen benötigt. Aufgrund der verteilten Modularität steigt jedoch die Komplexität beim Programmieren an.
Unabhängige kleine Dienste sind leichter einzusetzen. Da jeder Microservice in sich abgeschlossen ist, reduziert sich die Wahrscheinlichkeit für einen kompletten Applikationsabsturz, da im Fehlerfall nur der jeweilige Dienst abstürzt. Nehmen wir als Beispiel ein Webshop. Der Webshop besitzt einen Dienst für die Produktempfehlungen, sowie einen Dienst für den Einkaufswagen. Falls der Dienst für die Produktempfehlungen abstürzt, so läuft der Dienst für den Einkaufswagen dennoch weiter und der Kunde kann seinen Einkauf abschließen. In einem Monolithen hingegen würde die komplette Applikation abstürzen, wodurch der Kunde keine Möglichkeit mehr besitzen würde, den Einkauf erfolgreich abzuschließen.

Zusätzlich empfiehlt sich eine umfassende Überwachung der Microservices. Stürzt einer der Dienste ab, meldet das Monitoring dies und der Dienst kann direkt neu gestartet werden.

Durch die vielen verteilten Services muss jedoch auf serviceübergreifende Datenkonsistenz geachtet werden. Es besteht die Gefahr, dass in Service A Daten verändert werden, was eine Änderung der Daten in Service B anstößt. Wird in der Zwischenzeit auf Daten von Service B zugegriffen, können sich diese in einem inkonsistenten Zustand befinden.
Der Microservice-Architekturstil erlaubt das Vermischen verschiedener Programmiersprachen in einer Applikation, da jeder Service in der Programmiersprache implementieren lässt, die sich am besten für die Geschäftsfunktionalität eignet. Dies ermöglicht ein effizienten Einsatz verschiedener Technologien und Frameworks, da die Technologien und Frameworks ausgewählt werden können, die die Arbeit am besten erledigen. Allerdings müssen die Entwickler viele Programmiersprachen und Frameworks beherrschen, um tatsächlich die passenden Technologien nutzen zu können.

Obwohl Microservices leichter als Monolithen zu entwickeln sind, verlagert sich die Komplexität in die Kommunikation zwischen den Diensten. Dies kann zu verschiedenen Schwierigkeiten, wie beispielsweise für das serviceübergreifenden Debugging, führen. Aufgrund vieler Microservices einer Applikation gewinnt Continuous Delivery[HF10] in diesem Architekturstil einen großen Wert, da die große Anzahl an Services sonst nur schwer zu handhaben ist.

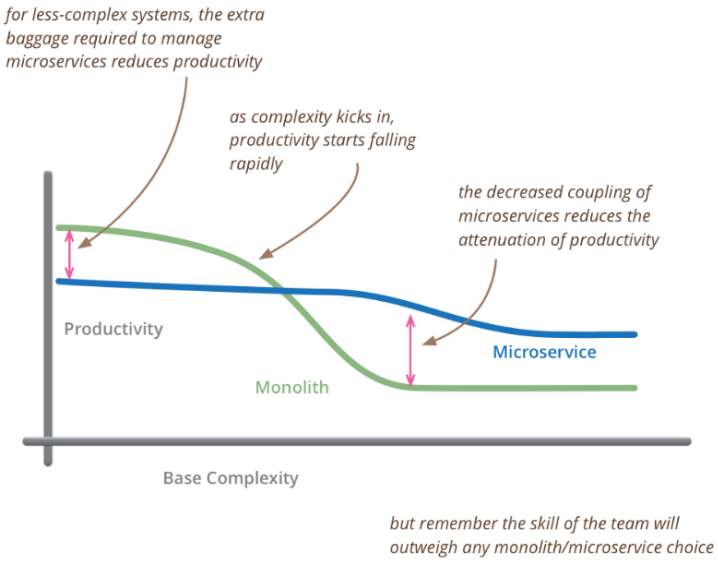

Abbildung 2.1: Microservices im Vergleich zu Monolithen im Hinblick auf Komplexität und Produktivität [Fow17]

Doch wann genau sollten Microservices eingesetzt werden? Dies hängt von der Kompexität des Systems ab. Mit steigender Komplexität sinkt, wie in Abbildung 2.1 gezeigt, die Produktivität eines Softwareproduktes, egal ob Microservice oder Monolith. Da die einzelnen Microservices jedoch von eher kleinerer Komplexität sind, sinkt die Produktivität eines aus Microservices gebauten komplexen Systems im geringeren Maße, als die eines großen Monolithen. Aufgrund des hohen Kommunikations- und Managementaufwands von Microservices besitzt ein aus solchen bestehendes System mit kleiner Komplexität eine geringere Produktivität als ein

Monolith. Microservices sollten daher erst in Systemen eingesetzt werden, deren Komplexität zu groß wird, um sie als Monolith effizient einsetzen zu können. Aus diesem Grund sollte zu Beginn der Entwicklung eines Systems genau durchdacht werden, wie komplex dieses ungefähr werden wird, da ein späterer Umstieg von einem Monolithen zu Microservices oftmals, aufgrund der verzweigten Struktur eines Monolithen, nicht effizient umsetzbar ist.

Definieren wir als nächstes den Architekturstil der Service-Oriented Architecture (kurz *SOA*). Das Problem hierbei ist, dass keine allgemein akzeptierte Definition für SOA existiert. Stattdessen gibt es viele verschiedene, teils nicht kompatible, Definitionen, was unter SOA zu verstehen ist. Aus diesem Grund betrachten wir eine im Hinblick auf den Vergleich zu Microservices passende Definition. Bei dieser wird die SOA als ein Architekturstil bezeichnet, der Systeme aus netzwerke verteilten Diensten zusammenstellt. Diese Dienste können genutzt werden um Geschäftsfunktionalitäten zu implementieren[LL09]. Die Idee ist das Entwickeln vieler wiederverwendbarer Dienste, um ein möglichst flexibles Zusammensetzen neuer Applikationen zu ermöglichen. Je mehr kleinere Services vorhanden sind, desto wahrscheinlicher ist es Teile der Services wiederverwenden zu können, ohne die Logik neu implementieren zu müssen. Ein Service in der SOA muss hierfür standardisierte Schnittstellen bereitstellen, mit denen andere Anwendungen interagieren können. Die Kommunikation erfolgt dabei über Protokolle, wie REST oder RPC.

Da es verschiedene Definitionen von SOA gibt, ist es schwer beide Stile miteinander zu vergleichen. Für die oben beschriebene Definition stellt sich allerdings die Frage nach einem Unterschied der SOA zum Architekturstil der Microservices. Viele sind der Meinung, dass es keinen Unterschied zwischen SOA und Microservices gibt. Martin Fowler hingegen bezeichnet den Architekturstil der Microservices als eine Untermenge des Einsatzes von SOA[Fow17]. Der Architekturstil der Microservices erfordert, dass jeder Service selbst einsetzbar ist. Diese Forderung ist in SOA nicht zwangsläufig gegeben. Oftmals sind die SOA Services innerhalb von einsetzbaren Monolithen implementiert.

James Lewis berichtet in einem Interview mit einem Reporter des „Software Engineering Radio“[Rad14], dass ein großes Problem der Unterscheidung von Microservices mit SOA darin liegt, dass ein großer Unterschied zwischen der Idee der Service-Oriented Architecture und der Umsetzung dessen in den Firmen existiert. Er beschreibt, dass Microservices einen deutlich inkrementelleren und iterativeren Ansatz, Systeme zu entwickeln, verfolgen, als in der SOA. Zusätzlich merkt er an, dass auch Unterschiede im Implementierungsmodell existieren. So bevorzugen Microservices eher leichtgewichtige Kommunikationskanäle, anstatt eines schwergewichtigen Enterprise Service Bus (ESB)[Cha04]. Ein ESB agiert bei der SOA als zwischengeschaltete Schicht, die die Kommunikation der verschiedenen Services ermöglicht. Services, die über einen ESB eingesetzt werden, können durch ein Konsument oder Ereignis ausgelöst werden. Dadurch werden synchrone oder asynchrone Interaktionen zwischen einem und mehreren Akteure unterstützt[Mar06]. Insgesamt hält James Lewis jedoch fest, dass Microservices stark verwandt mit der SOA sind.

Es lässt sich also zusammenfassend sagen, dass Microservices und SOA sich nur in wenigen Details unterscheiden, die sich nicht widersprechen. Microservices legen mehr Wert auf die

Bereitstellung einzelner Services, insbesondere auf unabhängige Bereitstellung der Services, als in der SOA. Dadurch kann der Architekturstil der Microservices als eine spezielle Untermenge der SOA angesehen werden.

2.2 Technologien

Nachdem Abschnitt 2.1 die grundlegenden Konzepte skizziert hat, widmet sich dieser Abschnitt nun den zugehörigen Technologien dieser Konzepte. Dieser Abschnitt teilt sich in sieben Unterabschnitte auf. Zu Beginn wird in Abschnitt 2.2.1 das in vielen APIs verwendete Datenformat JSON genauer betrachtet. Anschließend behandelt Abschnitt 2.2.2 das Framework Swagger, das zur Definition und automatischen Generierung von REST APIs genutzt wird. Abschnitt 2.2.3 stellt Protocol Buffers 3 vor. Dieses dient zur Serialisierung von JSON Objekten im Binärformat und zur Definition von RPC Schnittstellen für Services. In Verbindung mit Protocol Buffers kann mit dem in Abschnitt 2.2.4 vorgestellte gRPC, eine von Google bereitgestellte Instanz von RPC, ein Service mit RPC API implementiert werden. Anschließend behandelt Abschnitt 2.2.5 die API Middleware Kong. Als Beispiel einer Container Virtualisierung wird in Abschnitt 2.2.6 Docker vorgestellt. Abschließend wird in Abschnitt 2.2.7 die Technologie HTTP erläutert.

2.2.1 JSON

JavaScript Object Notation (kurz *JSON*) ist eine Möglichkeit kompakt verschiedene Daten in einem eigenen Format zu speichern. Für eine möglichst große Unabhängigkeit existieren in vielen verschiedenen Sprachen Parser um JSON Objekte in Strings zu parsen und umgekehrt. Entgegen vielen anderen Formaten ermöglicht JSON das Speichern von Daten in menschen- und maschinenlesbarer Form.

Im Folgenden betrachten wir den Aufbau eines JSON Dokumentes, das durch das JSON Schema [org17] definiert wird. Das kleinste JSON Objekt besteht nur aus einer öffnenden geschweiften Klammer und einer schließenden geschweiften Klammer. In den Klammern stehen optional die Eigenschaften des Objektes.

Eine Eigenschaft besteht immer einen Bezeichner und einen Datentyp. Der Name des Bezeichners ist frei wählbar, sollte aber aussagekräftig sein. Außerdem müssen Bezeichner zwingend in Anführungszeichen gehalten werden.

Als Datentyp sind Null-Werte, Bool'sche Werte, Zahlen, Strings, Arrays, sowie weitere JSON Objekte erlaubt. JSON Objekte können beliebig tief verschachtelt werden.

JSON Beispiel

Betrachten wir in Listing 2.1 ein konkretes Beispiel für ein JSON Objekt. Das in dem Beispiel gezeigte Objekt kann für einen Webshop Service als Bestellung Instanz verwendet werden.

Listing 2.1 JSON Objekt für eine Webshop Order

```
{
   "id": "8",
   "products": [
      "27",
      "2",
      "4",
      "0"
   ],
   "customer": {
      "id": "1",
      "firstname": "Donald",
      "lastname": "Duck",
      "shippingAddress": "Erpelstr. 7, 0815 Entenhausen"
   },
   "status": "NEW"
}
```

Das Objekt in Listing 2.1 besitzt vier Attribute. Das erste Attribut ist vom Typ `string` und speichert die Order ID. Das zweite Attribut ist ein `array` von Produkt IDs, die in der Bestellung enthalten sind. Zusätzlich existiert ein Attribut für die Daten des Kunden. Diese Kundendaten sind selbst wiederum ein JSON Objekt mit verschiedenen Attributen. Außerdem beinhaltet das Beispiel Objekt noch ein Attribut vom Typ `string` für den Bestellungsstatus.

Vergleich zu XML

Ähnlich zu JSON ermöglicht XML die Darstellung von hierarchisch strukturierten Daten als Textdateien und kann für Remote Procedure Calls und für den Datenaustausch zwischen Anwendungen genutzt werden. XML Objekte sind wie JSON Objekte, sowohl menschenlesbar, als auch computerlesbar. Für beide Formate existieren verschiedene Parser, JSON Objekte sind allerdings, im Gegensatz zu XML Objekten, in JavaScript direkt unterstützt. Da XML zusätzliche Bibliotheken benötigt werden, um die Daten aus einem Document Object Model zu bekommen, lässt sich JSON in JavaScript Anwendungen deutlich einfacher nutzen als XML [Nur+09]. Beide Formate unterstützen nur in Ausnahmen die Speicherung von Daten im Binärformat. So können Binärdaten z.B. in einen Base64 String gespeichert werden, was allerdings nicht optimal bezüglich der Performanz und Speicherverbrauch ist. Somit eignen sich XML und JSON nur wenig zur Repräsentierung von Binärdaten. Sollen JSON Objekte im Binärformat serialisiert werden, kann das in Abschnitt 2.2.3 vorgestellte Datenformat Protocol Buffers genutzt werden. Nuseitov et al haben an der Montana State University eine Fallstudie zum Vergleich von

JSON und XML als Datenaustauschformate durchgeführt. In dieser Studie wurden sowohl die Übermittlungszeit, als auch der Ressourcenverbrauch beider Formate in verschiedenen Testfällen ermittelt. Als Ergebnis erhielten sie, dass JSON Encoding im Allgemeinen schneller als XML Encoding ist. Außerdem fanden sie heraus, dass JSON weniger Ressourcen benötigt, als XML [Nur+09].

2.2.2 Swagger

Swagger ist ein, zu RAML[4] oder API Blueprint[5] alternatives, Framework für RESTful APIs[Tea14]. Die aktuelle Swagger-Spezifikation geht zur Zeit im Open API Standard auf, der von einigen Unternehmen, wie Google unterstützt wird. Es ist möglich mit Swagger RESTful APIs als JSON Objekte im JSON oder YAML Format zu definieren. Hierfür existieren zwei Verfahren. Mit dem ersten Verfahren lässt sich aus einer Swagger Datei mit Hilfe von Swagger-Codegen [Swa17] automatisch eine REST API in verschiedenen Sprachen erzeugen. Das zweite Verfahren kann aus einer implementierten REST API automatisch eine Swagger Datei generieren. Obwohl Swagger den JSON Standard erfüllt, existiert keine Forderung JSON Objekte für Ein- und Ausgaben in der API zu verwenden.

Die Swagger Datei muss aufgrund der Datei Namenskonvention `swagger.json` oder `swagger.yml` benannt werden. Alle Namen der API Definition sind case-sensitive.

Zuerst betrachten wir die in Swagger erlaubten Datentypen. Grundsätzlich sind alle, auf JSON basierenden, primitive Datentypen erlaubt. Zusätzlich zu den normalen primitiven Datentyp gibt es den primitiven Datentyp `file`. Objekt Modelle können ähnlich zu *JSON-Schema*[org17] als *Swagger Schema Object* definiert werden.

Im Folgenden betrachten wir den Aufbau eines Swagger Objektes, das ein Grundobjekt zur API Definition darstellt. Jede Swagger Datei muss mit der Swagger Definition beginnen. Für die aktuelle Version muss, wie im folgendem Beispiel, den Wert 2.0 sein: `"swagger": "2.0"`. Zusätzlich wird ein Informationsobjekt benötigt, um wichtige Daten über die API zu speichern. Erforderliche Daten in einem Informationsobjekt sind der Namen der Applikation, sowie die Version der aktuellen Applikations-API. Davon abgesehen kann man optional eine Reihe weiterer Informationen angeben, wie die Beschreibung der Applikation.

Anschließend werden die in der REST API erlaubten Pfade mit den jeweiligen erlaubten Operationen definiert. Als Pfade sind nicht-leere, relative Pfade auf die individuellen Ressourcen erlaubt. Die jeweiligen Pfade werden an den Basispfad (`basePath`) der Applikation hinzugefügt, um eine komplette Pfad URL zu erzeugen. Ein *Path Item Object* definiert die erlaubten HTTP Verben in sogenannten *Operation Objects* auf einem speziellen Pfad. Die Pfade müssen

[4] http://raml.org/
[5] https://apiblueprint.org/

syntaktisch mit einem \ beginnen, wie in folgendem Beispiel gezeigt: `\{path}`. Ein solches *Operation Object* besteht aus optionalen und obligatorischen Bereichen.

Betrachten wir zuerst die optionalen Bereiche. Das `summary` Feld ermöglicht eine Kurzzusammenfassung der Operation in nicht mehr als 120 Zeichen. Soll die Operation ausführlicher beschrieben werden, so ist dies im `description` Bereich möglich. Zusätzlich kann zu Dokumentationszwecken mit `externalDocs` auf externe Dokumente verwiesen werden. Die Operationen können optional Parameter übergeben bekommen. Diese lassen sich einzeln im `parameters` Bereich als *Parameter Object* definiert werden. Für die Art der Parameterübergabe existieren 5 verschiedene Parametertypen: Path, Query, Header, Body und Form. Zusätzlich zum Ort des Parameters (Parametertyp) benötigt jedes der Parameterobjekte einen Namen und Informationen, ob der jeweilige Parameter optional oder obligatorisch ist. Optional kann wieder mit `description` eine Beschreibung des Parameters angegeben werden. Je nachdem welcher Typ an Parameterübergabe vorliegt werden noch weitere Informationen benötigt. Wird der Parameter beispielsweise im body übergeben, so muss ein entsprechendes *Schema Object* angegeben werden. Für alle anderen Arten benötigt der Parameter einen Datentyp. Falls dieser der Typ `array` ist, müssen die Item Objekte im `items` Bereich beschrieben werden. Hier wird also definiert, welche Art an Elementen im Array erlaubt sind. Zusätzlich zum Datentyp können noch weitere Attribute festgelegt werden.

Obligatorisch muss für jede der Operationen mindestens eine erwartete Rückmeldung definiert werden. Das `responses` Objekt dient hierfür als Container. Es gilt zu beachten, dass das `responses` Objekt nicht zwingend alle HTTP response codes (siehe Abschnitt 2.2.7) abdecken muss. Erwartet wird jedoch die Definition für eine erfolgreiche Rückmeldung und für bekannte Fehler. Mindestens muss jedoch ein response code angegeben werden. Dieser sollte der erfolgreichen Rückmeldung/Operationsaufruf entsprechen. Alle anderen, nicht explizit angegebenen, response codes lassen sich mit in dem `default` Bereich abdecken. Jedes der response Objekte benötigt eine textuelle Beschreibung im typischen `description` Bereich. Zusätzlich sind weitere Informationen, wie ein *Schema Object* oder ein Beispiel, möglich. Die Rückgaben können Objekte beinhalten, deren Form als *Reference Object* definiert werden. Das *Reference Object* ist hierbei eine JSON Referenz auf Objekte, die am Ende der Datei im `definitions` Bereich definiert werden.

In Listing 2.2 ist ein Swagger Objekt in YAML zu sehen. Wie beschrieben wird erst die Swagger Version definiert und anschließend grundlegende Informationen, wie den Titel der Anwendung und deren Beschreibung gegeben. Der Basispfad ist in dem gegebenen Beispiel leer. Unter `paths:` wird dann beispielhaft eine *GET* Operation auf dem Pfad `/orders/{id}` definiert. Der erforderliche Parameter `id` wird im Pfad mit gegeben und ist vom Typ `integer`. Als Rückgaben werden zwei Arten definiert. Für eine erfolgreiche Ausführung der Operation soll unter dem HTTP Code 200 das entsprechende *Order* Objekt zurückgegeben werden. Hierbei wird für die Definition des Objektes auf das entsprechende *Reference Object* im `definitions` Bereich referenziert. Alle anderen Rückgaben werden im `default` Bereich mit einem Fehlerobjekt abgedeckt.

Listing 2.2 Teil eines Swagger Objekts

```
swagger: '2.0'
info:
  title: Webshop REST API
  description: Webshop Microservice REST API, which allows different operations.
  version: "1.0.0"
basePath: /
paths:
  /orders/{id}:
    get:
      summary: Get order with ID id
      description: GET on order ressources. Returns a specific order with given id.
      parameters:
        - name: id
          in: path
          description: order id given in path
          required: true
          type: integer
          format: int32
      responses:
        200:
          description: An specific order
          schema:
             $ref: '#/definitions/Order'
        default:
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
```

Der letzte Teil eines Swagger Objektes ist der `definitions` Bereich. In diesem werden die für die Operationen zurückgegebenen oder übergebenen Objekte als Schema Objekte definiert. Zuerst muss der Typ des Objektes definiert werden. Im Beispiel aus Listing 2.3 ist dies `object`. Anschließend werden die Eigenschaften des Objektes im `properties` Bereich definiert. Jede der Eigenschaften benötigt einen Typ. Es können jedoch auch weitere Informationen, wie eine Beschreibung der Eigenschaft gegeben werden. In dem gegebenen Beispiel aus Listing 2.3 besitzt das Objekt `Order` eine ID, eine Beschreibung, die Liste an Produkten der Bestellung, sowie Kosten der Bestellung.

2.2.3 Protocol Buffers

Als Protocol Buffers (kurz *proto*) wird ein Toolset und Sprache zur Serialisierung von JSON Objekten im Binärformat bezeichnet. Zusätzlich zur Definition von JSON Nachrichtenobjekten und deren Serialisierung, kann Protocol Buffers auch als Interface Definition Language für Remote Procedure Call Schnittstellen genutzt werden. Die Nachrichten- und Schnittstellendefinitionen werden in einer `.proto` Datei gespeichert.

Listing 2.3 `definitions` Bereich eines Swagger Objektes

```
definitions:
 Order:
   type: object
   properties:
    order_id:
     type: integer
     format: int32
     description: Unique identifier representing a specific order for a given id.
    description:
     type: string
     description: Description of order.
    products:
     type: array
     description: List of products of the order.
     items:
      $ref: '#/definitions/Product'
    cost:
     type: number
     format: double
     description: Cost of the order.
```

Das Datenformat wurde von Google Inc. entwickelt und in der aktuellen Version, Protocol Buffers 3[Goo17a], gibt es Implementierungen für die Programmiersprachen C++, Java, C, C#, Python, Ruby, JavaScript und einige andere. Mit Hilfe des Protocol Buffer Compilers kann eine Protocol Buffers Datei in Code einer gewünschten und unterstützten Sprache kompiliert werden.

Protocol Buffers zeichnet sich besonders durch seine Einfachheit und Performanz aus. Da die Objekte in das Binärformat serialisiert werden, können diese resourcensparend über RPC übermittelt werden. Das ebenfalls von Google entwickelte gRPC (siehe Abschnitt 2.2.4) kann wiederum genutzt werden, um die im Protocol Buffers definierte RPC Schnittstelle in der Anwendung zu realisieren.

Der Aufbau einer Protocol Buffers 3 Datei beginnt mit `syntax = "proto3"`, um die Syntax von Protocol Buffers 3 zu verwenden.

Anschließend werden die Nachrichtenobjekte definiert. Jede Nachricht beginnt mit dem Schlüsselwort `message`, gefolgt von dem Nachrichtennamen. Eine Protocol Buffers Datei kann mehrere Nachrichten definieren. Im Folgenden betrachten wir den Aufbau einer Nachricht im Detail.

Jede Nachricht kann beliebig viele Felder besitzen. Jedes der Felder definiert ein Name-Value-Paar und eindeutig nummeriert, um die Position des jeweiligen Feldes der Nachricht im Binärformat identifizieren zu können. Um keine Fehler bei bereits serialisierten Objekten zu erzeugen, sollten diese tags nicht geändert werden, da die Zuordnung zu den jeweiligen Feldern ansonsten nicht mehr korrekt funktionieren kann. Die Felder können entweder als einzelnes

Element oder aus mehreren Elementen bestehen, wobei die Ordnung der Elemente eines Felds erhalten bleibt.

Als Datentyp der Felder sind sowohl skalare Typen, enums, als auch andere Nachrichten Objekte erlaubt. Tabelle 2.1 zeigt einige der in Protocol Buffers erlaubten skalaren Typen mit der entsprechenden Beschreibung und dem Standardwert im Vergleich zu den Programmiersprachen C++, Java und Python.

.proto Typ	Anmerkungen	C++ Typ	Java Typ	Python Typ
double		double	double	float
float		float	float	float
int32	Benutzt variable-length encoding. Ineffizient für die Speicherung negativer Zahlen.	int32	int	int
int64	Benutzt variable-length encoding. Ineffizient für die Speicherung negativer Zahlen.	int64	long	int/long
uint32	Benutzt variable-length encoding.	uint32	int	int/long
uint64	Benutzt variable-length encoding.	uint64	long	int/long
bool		bool	boolean	bool
string	Ein String muss immer UTF-8 codiert oder ein 7-bit ASCII Text sein.	string	String	str/unicode
bytes	Kann jegliche willkürliche Folge von Bytes beinhalten.	string	ByteString	str

Tabelle 2.1: Ausschnitt an in Protocol Buffers 3 erlaubten Datentypen [Goo17a]

Falls eine Nachricht geparsed wird und eines der Felder keinen speziellen Wert beinhaltet, wird der entsprechende in der Tabelle genannte Standardwert gesetzt. Für Nachrichtenfelder wird das Feld nicht gesetzt. Falls ein Feld `repeated` ist, ist der Standardwert leer, vergleichbar zu leeren Listen. Sobald die Nachricht geparsed wird, werden alle nicht gesetzten Felder standardinitialisiert. Anschließend ist es nicht mehr unterscheidbar, ob ein Feld nicht gesetzt war, oder der gesetzte Wert dem Standardwert entspricht. Dies sollte bei der Implementierung eines Servers beachten, sodass möglichst keine Werte genutzt werden, die den Standardwerten entspricht.

Zu den normalen skalaren Typen und Nachrichten Objekten können auch Maps definiert werden. Diese lassen sich durch `map<key, value> m = id;` definieren. Als Schlüssel sind hierbei alle skalaren Typen, außer Floatingpoint Typen erlaubt, während als Werte jeder Typ gespeichert sein kann. Möchte man zum Beispiel deutsche KFZ-Kennzeichen den Städten zuordnen, kann man dies zum Beispiel durch `map<string, string> kfz_mapping = 1;` implementieren.

Listing 2.4 Protocol Buffers Nachrichtendefinition mit Any-Typ

```
import "google/protobuf/any.proto";

message ErrorStatus {
  string message = 1;
  repeated google.protobuf.Any details = 2;
}
```

Zusätzlich können Aufzählungen (Enumerations) definiert werden. Dadurch können Felder dieser Enums erzeugt werden um gewährleisten zu können, dass das Feld nur einen aus einer begrenzten Anzahl an Werten besitzt. Der Aufbau eines Enums ist ähnlich zum Aufbau einer Nachricht. Es können Werte festgelegt werden, die durch einen eindeutigen Identifizierer gekennzeichnet werden. Allerdings muss es einen ID Wert geben, der 0 ist. Die Enumerationen können sowohl innerhalb einer Nachricht, als auch außerhalb definiert werden, wobei außerhalb von Nachrichten definierte Enumerationen in allen Nachrichtendefinitionen verwendet werden können, anstatt nur in einer Nachricht. Enumerationen, die innerhalb einer Methode definiert wurden können in anderen Methoden nur verwendet werden, indem diese dort über den Nachrichtentyp aufgerufen werden.

Zusätzlich zu Aufzählungen können auch weitere Nachrichten innerhalb einer Nachricht definiert werden. Diese sogenannten Nested Types können beliebig tief verschachtelt werden.

Um nicht alle Nachrichten in einer Protocol Buffers Datei definieren zu müssen, lassen sich andere Protocol Buffers Dateien mithilfe des `import` Statements einbinden. Hierfür kann man, wenn man beispielsweise die Definitionen der Datei „outsource.proto“ in der Datei „samples.proto“ einbinden möchte, in der Datei „samples.proto“ zu Beginn folgende Zeile schreiben: `import "/outsource.proto";`.

Optional können die Protocol Buffers Dateien in verschiedene Pakete eingeordnet werden. Hierfür kann man, ähnlich zu Java, mit dem Schlüsselwort `package` ein Paket folgendermaßen zu Beginn einer Datei definieren: `package samples;`. Möchte man nun auf Nachrichten innerhalb eines anderen Packetes zugreifen, so muss dies über den Paketnamen erfolgen. So kann man zum Beispiel mit `samples.Foo` auf die Nachricht `Foo` im Paket `samples` zugreifen.

Möchte man Nachrichten definieren, die Felder eines zur Definition unbekannten Nachrichtentyps beinhalten sollen, kann man diese durch das Schlüsselwort Any definieren. Um den Any-Typ nutzen zu können, muss zuvor folgender Import getätigt werden: `google/protobuf/any.proto`. Listing 2.4 zeigt eine Nachrichtendefinition unter Nutzung des Any-Typs.

Protocol Buffers kann JSON Objekte im Binärformat kodieren. Dies ist deutlich effizienter als eine normale Verwendung von JSON Objekte, da die in binär kodierten Objekte deutlich platzsparender und dadurch schneller zu versenden sind.

Falls die Anwendung beispielsweise eine RPC API bereitstellen soll, kann dies ebenfalls in der Protocol Buffers Datei definiert werden. Hierfür lassen sich Service Schnittstellen schreiben, die

durch den Compiler zu Schnittstellen Code kompiliert wird. In einem solchen Service können dann die RPC Methoden definiert werden. Das Beispiel in Listing 2.5 zeigt exemplarisch die Definition einer RPC Webshop Service API.

Listing 2.5 Protocol Buffers Definition einer Service Schnittstelle für RPC

```
service WebShop {
  rpc listProducts(ListProductsParams) returns (stream Product) {}
  rpc checkAvailability(ProductId) returns (Availability) {}
  rpc storeOrderDetails(Order) returns (OrderId) {}
  rpc getOrderDetails(OrderId) returns (Order) {}
  rpc cancelOrder(OrderId) returns (Order) {}
  rpc calcTransactionCosts(OrderId) returns (Costs) {}
}
```

Der durch die RPC Schnittstelle generierte Code lässt sich dann mit dem ebenfalls von Google entwickelten gRPC implementieren, das wir im folgenden Abschnitt näher betrachten.

2.2.4 gRPC

Das Framework gRPC wurde von Google entwickelt, um RPC Dienste entwickeln zu können[Goo17b]. Mit gRPC lassen sich Services implementieren, die über einen Remote-Zugriff Service-Methoden mit entsprechenden Parametern und Rückgabewerten aufrufen können. Standardmäßig wird für die Service Definition von gRPC Diensten das bereits vorgestellte Protocol Buffers als Interface Definition Language verwendet. Dabei werden sowohl die Service Schnittstelle, als auch das Nachrichtenformat (Objektformat) definiert. gRPC ist nicht auf Protocol Buffers als IDL beschränkt. Es kann auch eine andere IDL verwendet werden, falls dies gewünscht ist. Ebenfalls müssen über Prozeduraufrufe zusätzlich keine Protocol Buffers Objekte versandt werden. So lassen sich beispielsweise auch normale JSON Objekte versenden.

gRPC unterstützt eine Vielzahl an Programmiersprachen, wie Java, Python, JavaScript. Aufgrund dessen können gRPC Server und Clients auf verschiedenen Umgebungen laufen und miteinander interagieren. Auf der Serverseite wird dann jeweils die in der IDL definierte Schnittstelle implementiert, damit der gRPC Server nach dem Start die Client Anfragen bearbeiten zu können. Mit gRPC lassen sich vier verschiedene Arten an Service Methoden definieren und implementieren:

1. Die einfachste Art ist eine einzelne Anfrage mit einzelner Antwort. Hier sendet der Client eine einzelne Anfrage an den Server. Dieser bearbeitet die Anfrage vollständig und sendet anschließend ebenfalls eine einzelne Antwort an den Client zurück. Das ist vergleichbar mit einem normalen lokalen Funktionsaufruf.
2. Die zweite Art ist sehr ähnlich zur ersten. Hier sendet der Client ebenfalls eine einzelne Anfrage an den Server. Dieser gibt allerdings nicht eine einzelne Antwort zurück, sondern

eine Streamsequenz. Der Client kann anschließend die Antworten aus diesem Stream lesen, solange dieser Nachrichten enthält.

3. Auch der Client kann nicht nur einzelne Anfragen senden, sondern Streams. In einem solchen Fall liest der Server den Stream an Nachrichten komplett ein und bearbeitet anschließend die Anfrage. Erst nach vollständiger Bearbeitung der Anfrage sendet der Server in der dritten Methodenart eine Antwort an den Client zurück.
4. Zusätzlich zu den bereits beschriebenen Arten lässt sich auch ein bidirektionaler Stream implementieren. Hier sendet der Client einen Stream an Nachrichten an den Server. Dieser kann einen Stream an Antworten an den Client zurücksenden. Da beide Streams unabhängig voneinander arbeiten, ist es nicht vorgeschrieben, ob der Server die Nachrichten gesammelt bearbeitet (die Bearbeitung beginnt erst, nachdem alle Nachrichten gelesen wurden), oder ob er nach jeder einzelnen Anfrage eine Antwort sendet.

Betrachten wir nun das Vorgehen bei der Erzeugung eines gRPC Services genauer. Zuerst sollte mit einer Interface Definition Language, wie Protocol Buffers, ein Service definiert werden. Diese Definition kann mit einem Compiler sowohl den Client, als auch den Server Code der Schnittstellen erzeugen.
Auf der Serverseite können die Methoden, die in der Schnittstelle definiert wurden, implementiert werden. Empfängt der Server eine Anfrage, enkodiert das gRPC Framework die Anfrage, führt die entsprechend hinterlegte Service Methode aus und kodiert anschließend wieder die Antwort zum Versenden an den Client.
Auf der Clientseite existiert ein sogenannter Client Stub. Dieser implementiert die gleichen Methoden wie der Service. Bei einer gewünschten Anfrage kann dann der Client diese Methoden im Client Stub aufrufen. Hierbei sendet das gRPC Framework die Anfrage an den Server und gibt nach Empfangen der Antwort des Servers diese zurück.

gRPC bietet viele Arten an Authentifizierungsmechanismen an, um Sicherheitsmechanismen unterstützen zu können. Die vorimplementierten Mechanismen sind SSL/TLS, die tokenbasierende Authentifikation mit Google und die gRPC eigene Authentifizierungs-API. Letztere stellt alle notwendigen Informationen als Credentials bereit, sobald ein Aufruf erfolgt. Darüber hinaus lassen sich auch eigene Systeme einbinden, um möglichst flexibele Authentifizierungen zu ermöglichen.

2.2.5 Kong

Kong ist eine skalierbare open-source API Middleware, die auf NGINX ausgeführt wird. Sie „erlaubt den Entwicklern die Komplexität und Entwicklungszeiten zu reduzieren“[Mas17]. Das Kong Gateway kann auf jede RESTful API gesetzt werden und diese durch Plugins (Kong Plugins) erweitern, die zusätzliche Funktionalitäten und Dienste anbieten. Insbesondere bildet das API Gateway einen einzelnen Einstiegspunkt aller Clients für mehrere Services.[Ric17]
Drei Eigenschaften lassen sich besonders hervorheben:

- *Skalierbarkeit:* Durch Hinzufügen von weiteren Maschinen kann Kong das System leicht horizontal skalieren. Dadurch lassen sich viele Anfragen mit kleiner Latenz bearbeiten.
- *Modularität:* Da Kong sehr modular aufgebaut ist können beliebig weitere Kong Plugins hinzugefügt und konfiguriert werden.
- *Systemunabhängig:* Kong kann auf beliebigen Infrastrukturen ausgeführt werden.

Im Folgenden wird die Architektur eines Kong Systems mit anderen Architekturen verglichen. Abbildung 2.2 zeigt schematisch den Aufbau eines Systems von mehreren Services. Dabei sind viele verschiedene Funktionalitäten, z.B.: Caching, über mehrere Services verteilt mehrfach vorhanden. Diese Redundanzen sorgen dafür, dass normale Systeme dazu neigen große Monolithen zu werden. Aufgrund des Zusammenhangs können einzelne Services nur schwer erweitert werden, ohne die anderen Dienste zu beinflussen.

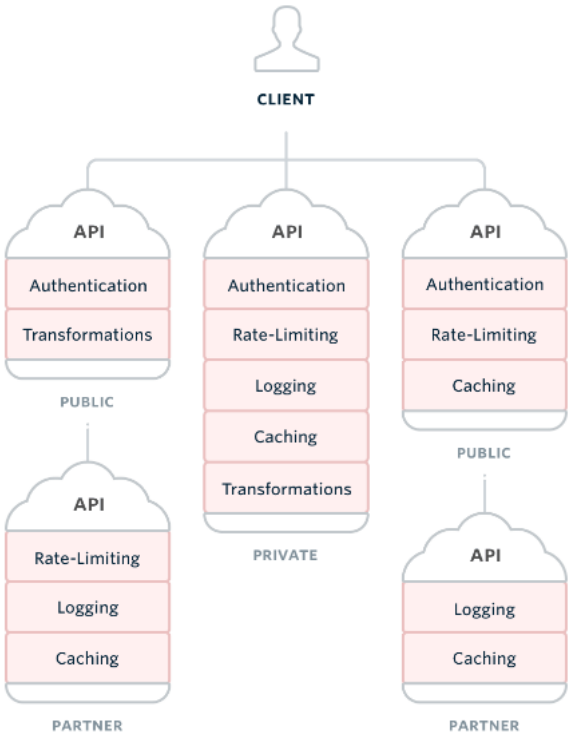

Abbildung 2.2: Schematischer Aufbau bisheriger Architekturen [Mas17]

Im Gegensatz dazu zeigt Abbildung 2.3 schematisch den Aufbau eines Kong Systems. Hier werden alle Funktionalitäten gesammelt und an einem Platz vereint. Das reduziert die oben genannten Redundanzen und ermöglicht ein einfaches Hinzufügen weiterer Funktionalitäten über die Kong Plugins. Aufgrund dieser Architektur lassen sich die einzelnen Dienste effizient erweitern, ohne starke Seiteneffekte auf den Rest des Systems zu bewirken.

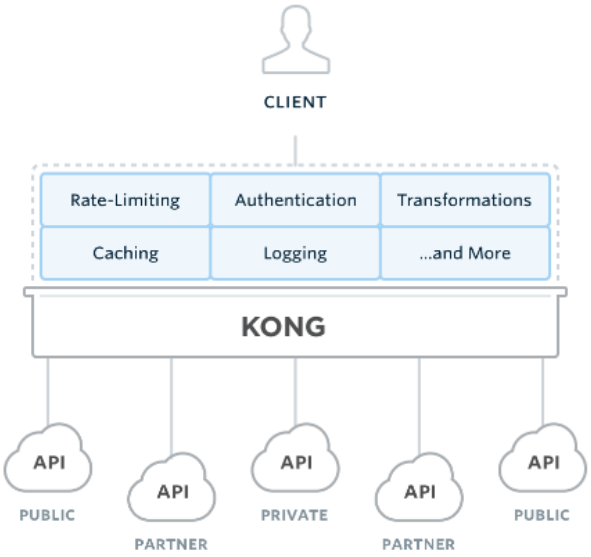

Abbildung 2.3: Schematischer Aufbau einer Kong Architektur [Mas17]

Falls mehrere Kong Knoten auf den selben Datenspeicher zugreifen, lassen diese sich zu Cluster zusammenfügen. Dabei kommunizieren die Knoten über TCP und UDP miteinander. Hierzu ist es wichtig, dass ein Kong Cluster nicht zwingend komplett im selben Datencenter laufen muss. Das Hinzufügen und Verlassen der Knoten eines Clusters wird von Kong automatisch durchgeführt. Dabei wird jedem Knoten automatisch eine IPv4 Adresse zugewiesen. Falls eine der automatisch erzeugten IPv4 Adressen ungültig ist kann der Entwickler diese von Hand überschreiben. Das Kong Cluster erlaubt durch einfaches Hinzufügen weiterer Maschinen ein horizontales Skalieren des Systems. Trotz des Clusters können aufgrund des verteilten Zugriffs auf die Daten weiterhin Konsitenzprobleme existieren.

Um eine korrekte Interaktion der Services zu gewährleisten, müssen einige Netzwerkanforderungen erfüllt werden. Zum Beispiel müssen alle Server sowohl TCP, als auch UDP nutzen können. Davon abgesehen müssen die verschiedenen Datencenter über Tunnelmechanismen, wie VPN, miteinander verbunden sein. Dies wird nicht von Kong übernommen und muss daher vom Entwickler selbst realisiert werden.

Jeder Kong Knoten besitzt einen Status. Dabei stehen vier Status Werte zur Wahl:

1. *active*: Der Knoten ist aktiv und Teil des Clusters.
2. *failed*: Das Cluster kann den Knoten nicht erreichen. In diesem Fall versucht Kong zunächst automatisch den Knoten wieder zu verbinden. Ist dies nicht möglich, muss der Knoten manuell aus dem Cluster entfernt werden. Es gibt mehrere Gründe für ein *failure*. Zum Beispiel kann der Knoten Netzwerkprobleme haben oder abgestürzt sein.
3. *leaving*: Während der Knoten das Cluster verlässt befindet er sich im Zustand *leaving*.

4. *left*: Sobald der Knoten erfolgreich das Cluster verlassen hat wechselt er in den Zustand *left*.

Für administrative Zwecke bietet Kong eine interne RESTful API an, die standardmäßig auf den Port 8001 hört. Jeder Knoten eines Clusters ist in der Lage die API Befehle der Admin API auszuführen. Dabei hält Kong selbstständig die Konfiguration aller Knoten konsistent.

Die Nutzung von API Gateways, wie Kong, bietet einige Vorteile. Sie isolieren die Clients von der tatsächlichen Unterteilung der Anwendung in Services. Zusätzlich kann die für jeden Client optimale API angeboten werden. Darüber hinaus wird der Client vereinfacht, da die Anwendungslogik mehrere Services anzusprechen vom Client in das API Gateway verlagert wird. Es entstehen allerdings Nachteile bei der Verwendung von API Gateways. So wird die Komplexität erhöht, da das API Gateway entwickelt und verwaltet werden muss. Außerdem erhöht sich die Reaktionszeit der Anwendung aufgrund der zusätzlichen Netzwerkübertragungen durch das API Gateway[Ric17].

2.2.6 Docker und Docker Compose

Docker Compose, als spezielles Beispiel für eine Container Virtualisierung, erlaubt das Paketieren einer Anwendung, inklusive aller für die Ausführung der Applikation notwendigen Abhängigkeiten, in einem standardisierten Container. Durch die Paketierung der Applikation in einem kompletten Ordnersystem, das alle notwendigen Abhändigkeiten enthält, kann sichergestellt werden, dass die Applikation immer gleich ausgeführt wird, ganz gleich auf welcher Plattform der Docker Container läuft.

Docker Container sind leichtgewichtig, plattformunabhängig und ermöglichen durch die Aufteilung in verschiedene Schichten den Schutz der Anwendung nach außen. Durch die Containervirtualisierung mehrerer Anwendungen, werden diese zu einander und zur jeweiligen Infrastruktur getrennt. Dennoch können mehrere Container verschiedene Dateien untereinander teilen, sowie Festplattenschreibzugriffe durchführen.

Insgesamt betrachtet ähneln Docker Container sehr normalen Virtuellen Maschinen. Es lassen sich aber dennoch ein paar Unterschiede feststellen. Obwohl auf Docker Container zwar eine ähnliche Ressourcen Isolation, wie virtuelle Maschinen existiert, sind diese aufgrund eines anderen architekturellen Aufbaus deutlich portabler und effizienter, als normale virtuelle Maschinen.

Abbildung 2.4 zeigt die verschiedenen Schichten bei Verwendung von virtuellen Maschinen. Zwischen den virtuellen Maschinen und dem Betriebssystem muss noch eine Zwischenschicht, der Hypervisor, zwischengelegt werden. Die virtuelle Maschine selbst beinhaltet dann die „Applikation mit allen benötigten Binärdateien und Bibliotheken und ein gesamtes virtuelles Betriebssystem“[Inc17]. Die Größe der virtuellen Maschine kann deshalb mehrere tausend Megabyte groß sein, was nicht sehr effizient ist.

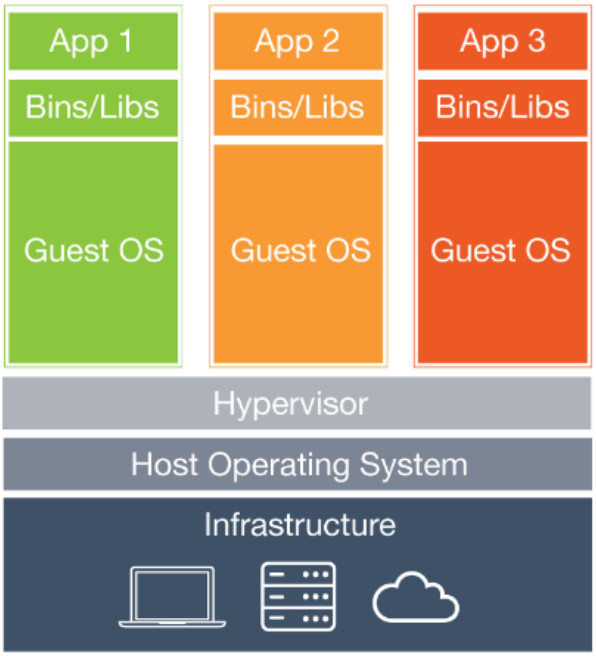

Abbildung 2.4: Verwendung von Virtuellen Maschinen mit je einer Applikation[Inc17]

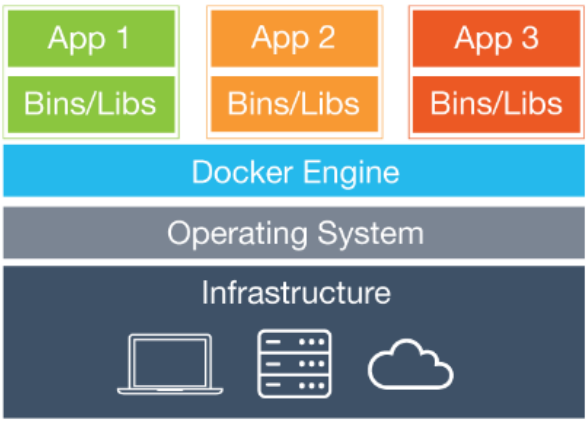

Abbildung 2.5: Verwendung eines Docker Containers mit 3 Applikationen [Inc17]

Dem entgegengestellt zeigt Abbildung 2.5 die Funktionsweise eines Docker Containers. Der Docker Container liegt hierbei als Zwischenschicht zwischen dem Betriebssystem und den Applikationen. Es wird kein Hypervisor benötigt. Auf dem gemeinsamen Docker Container befinden sich dann die Applikationen mit den jeweiligen benötigten Bibliotheken. Da Docker Container nicht an eine spezielle Infrastruktur gebunden sind, sind die Applikationen nur vom Container abhängig. Da Docker Container viel leichtgewichtiger sind, als virtuelle Maschinen, wird hier deutlich weniger Speicher benötigt, als bei der Verwendung letzterer.

Ein weiterer Vorteil, der durch die Container Virtualisierung mit Docker Compose entsteht, ist dass die Entwickler sich weniger Sorgen um die jeweilige Zielplattform machen müssen. Sie können sich die Zeit sparen, die Anwendung regelmäßig auf der Zielplattform testen zu müssen. Dadurch können Entwickler sich mehr auf die Entwicklung der Applikation kümmern, welche anschließend in einem Docker Container aufgesetzt wird. Der Docker Container kann dann abschließend als Ganzes verschifft werden. Darüber hinaus gilt es zu beachten, dass Docker Container schnell die enthaltenen Anwendungen hochskalieren können. Dadurch ist es möglich, flexibel auf einen höheren Workload zu reagieren.

2.2.7 HTTP

Der folgende Abschnitt stellt das Protokoll Hypertext Transfer Protocol (kurz *HTTP*) vor[W3C17b]. HTTP ist ein, auf TCP basierendes, zustandsloses Dateiübertragungsprotokoll beliebiger Daten, d.h. Informationen früherer Übertragungen werden nicht gespeichert und gehen verloren. Sollen Übertragungsinformationen gespeichert werden, kann dies über eine weitere Anwendung realisiert werden. In der Regel findet eine unverschlüsselte Übertragung der Daten statt. Sollen die Daten verschlüsselt versendet werden, lässt sich dies über HTTPS ermöglichen.

In HTTP sendet ein Client eine Anfrage an den Server, welcher nach Bearbeitung der Anfrage eine Antwort an den Client zurücksendet. Anfrage und Antwort werden dabei als Nachrichten bezeichnet und bestehen aus einem Nachrichtenkopf (Header), sowie dem Nachrichtrumpf (Body). Dabei beinhalter der Header die Informationen über den Body, wie Inhaltstyp oder Codierung. Die eigentlichen Daten werden im Body übertragen.

HTTP bietet eine Reihe an Operationen, die sogenannten HTTP Verben, an, um Daten löschen, hinzufügen oder verändern zu können[W3C17a]. Sie werden immer auf einem Ressourcenpfad ausgeführt. Die wichtigsten und meistgenutzten HTTP Verben sind *GET*, *PUT*, *POST* und *DELETE*.

- *GET*: Eine *GET* Anfrage auf einem Pfad, um die hinterlegten Daten zu lesen, ohne diese zu verändern. Die Antwort der Anfrage enthält dann die entsprechenden Daten aus dem Datensatz, sowie den HTTP Status 200.
- *PUT*: Mit *PUT* Anfrage lassen sich Daten ersetzen oder verändern. *PUT* liefert den Status Code 200 zurück, falls die Ressource erfolgreich verändert wurde, oder 204, falls die Antwort Daten beinhaltet. Die Operation kann auch genutzt werden, um Ressourcen zu erzeugen. In diesem Fall liefert die Antwort den Status Code 201 für die erfolgreiche Erzeugung.
- *POST*: Die *POST* Anfrage wird dazu verwendet, neue Daten einem Datensatz hinzuzufügen. Die Operation gibt den HTTP Status 201 und einen Link zu dem erstellten Datum, falls dies erfolgreich hinzugefügt werden konnte.

- *DELETE*: Sollen auf einer URI hinterlegte Daten gelöscht werden, kann dies in einer *DELETE* Anfrage geschehen. Beim erfolgreichen Löschen der Ressource liefert die Antwort den HTTP Status 200, falls im Body der Antwort Daten, wie das gelöschte Element, gesendet werden, oder 204, falls die Antwort keinen Inhalt im Body besitzt.

Alle genannten HTTP Verben, außer POST, sind idempotent. Das heißt mehrmaliges Ausführen der Operation auf der gleichen URI bewirkt immer das gleiche Ergebnis. Ebenfalls strittig ist die Idempotenz von *DELETE*. Ist eine Ressource einmal gelöscht, so wird *DELETE* immer den HTTP response code 404 zurückgeben. Ein erneutes Löschen ist allerdings nicht möglich.

Betrachten wir nun die HTTP Statuscodes genauer. Jede HTTP Anfrage wird mit einem der folgenden Statuscodes und einer im Header befindlichen kurzen Beschreibung des Fehlers, falls vorhanden, beantwortet. Die verschiedenen Codes haben eine unterschiedliche semantische Bedeutung, wie zum Beispiel die Bestätigung einer erfolgreich bearbeiteten Anfrage.

- **Informationen:** Im Falle einer länger andauernden Bearbeitung der Anfrage, kann eine Antwort zur Information dessen gesendet werden. Dies ist notwendig, da viele Clients sonst nach einiger Zeit von einem Fehler ausgehen und die Anfrage abbrechen würden. Die HTTP Statuscodes für Informationen beginnen alle mit einer 1.
- **Erfolgreiche Operation:** Die response codes für eine erfolgreiche Bearbeitung einer Anfrage beginnen alle mit einer 2.
- **Umleitung:** Wurden die angeforderten Daten verschoben, so kann der Server dem Client einen entsprechenden Antwort zusenden, die dann die neue Position der gewünschten Daten und einen mit 3 beginnenden Statuscode enthält.
- **Client-Fehler:** Die mit 4 beginnenden Statuscodes zeigen einen Fehler auf der Clientseite an. Statuscode 404 kann zum Beispiel verwendet werden, wenn der Client eine Ressource verändern möchte, die im Datensatz des Servers nicht existiert.
- **Server-Fehler:** Für Fehler auf der Serverseite sind die Statuscodes reserviert, die mit einer 5 beginnen.

3 API Entwicklungsmethode und Framework

Nachdem in Kapitel 2 verschiedene API Konzepte und Technologien vorgestellt wurden, betrachten wir in diesem Kapitel die Entwicklung von Microservice APIs näher. Hierzu werden in Abschnitt 3.1 zunächst die Defizite von bestehenden Ansätzen zur Entwicklung von APIs aufgezeigt. Anschließend wird in Abschnitt 3.2 eine Methode zur Entwicklung zusammensetzbarer API Bausteine vorgestellt. Abschnitt 3.3 stellt zum Schluss dieses Kapitels ein Framework für zusammensetzbare API Bausteine vor.

3.1 Defizite und Anforderungen

In Abschnitt 2.1 wurden bereits einige Konzepte für APIs vorgestellt, darunter REST und RPC. Heutzutage werden die meisten APIs individuell entwickelt. Um eine wohldefinierte API zu erhalten muss diese erst genau von Grund auf geplant werden. Anschließend kann der Entwickler sich für eine Art von API entscheiden. Diese Entscheidung hängt von den Designentscheidungen der Planung ab. So eignet sich für eine REST API eher, wenn direkt auf Ressourcen zugegriffen werden sollen, als eine RPC API. Ist die API von Grund auf geplant und eine Entscheidung für eine Art von API getroffen, muss der Entwickler oft noch kleine Anpassungen im Design der API vornehmen. Insgesamt muss hier erst einmal viel Zeit investiert werden, bevor die Entwickler mit der Implementierung der API beginnen können.

Obwohl viele Entwickler erfahren damit sind Executables zu erzeugen und diese über die Kommandozeile auszuführen, fehlt es den Entwicklern oft an Expertise die Funktionalitäten über eine gute und geeignete API bereitzustellen. Ist die Entscheidung für eine Art von API gefallen, muss ein Entwickler sich erst eine gute Expertise bezüglich dieser Art von API, wie zum Beispiel in REST, aneignen, sofern er nicht bereits Erfahrungen damit gemacht hat. Diese Einarbeitung kostet viel Zeit, die der Entwickler wiederum nicht für die Implementierung der eigentlichen Anwendungslogik investieren kann. Die aus dem Prinzip der „Smart Endpoints & Dump Pipes“[FL15] resultierende größere Komplexität der API-Endpunkte führt dazu, dass oftmals ein großer Teil der Entwicklung eines Microservices dafür verwendet werden muss.

Der Ansatz der individuellen Entwicklung erfordert oft zur Unterstützung eine Vielzahl an Frameworks und Bibliotheken, in die sich die Entwickler einarbeiten müssen. REST APIs können beispielsweise mit dem in Abschnitt 2.2.2 vorgestellten Framework Swagger definiert

werden. Um damit effizient eine REST API zu entwerfen, muss die API jedoch zuvor einmal konzipiert sein. Swagger erleichtert dem Entwickler nur die Arbeit der Definition der REST API, nicht die Konzeption derselben. Dennoch bietet die Verwendung von Swagger Vorteile bei der Entwicklung einer API, da aus der geschriebenen Swagger Datei über das Framework Swagger Codegen[Swa17] automatisch ein Code-Skelett der definierten REST API generiert werden kann. Ein entsprechender Ansatz kann bei der Entwicklung von RPC APIs verfolgt werden. Hier lassen sich die RPC Aufrufe zusätzlich zu den erlaubten Nachrichtenformaten in einer Protocol Buffers Datei definieren und ebenfalls automatisch über entsprechende Compiler erzeugen.

Durch herkömmliche Ansätze zur Entwicklung von APIs entsteht oft eine inhärente Verkopplung von API und Anwendungslogik. Eine Trennung beider ist daher kaum möglich, was die Erweiterbarkeit der Anwendung mit anderen Arten von APIs stark begrenzt. Zusätzlich kann die Erweiterung der bereits bestehenden API mit neuer Anwendungslogik erschwert sein. Typischerweise sind eine Anwendung und deren API im selben Technologie-Stack implementiert. Für die Entwicklung einer API müssen in diesen Fällen die Defizite des jeweiligen Technologie-Stacks in Kauf genommen werden.

Bei der Entwicklung einer Anwendung ist eine sehr frühe Entscheidung nötig, welche Art von API unterstützt werden soll, da die Implementierung dementsprechend angepasst werden sollte. Daraus resultieren einige Probleme. Werden bei der Implementierung einige Defizite bezüglich der Art von API der Anwendung festgestellt, oder wird gar eine weitere Art von API benötigt, muss eine weitere API zusätzlich entwickelt werden. Da für diese weitere Expertise und zusätzliche Zeit benötigt wird, ist der Umstieg auf eine andere Art von API meist sehr komplex. So könnte ein Microservice noch zusätzlich eine Messaging API benötigen, weil die bereits implementierte REST + HTTP API für die interne Kommunikation mit anderen Microservices zu ineffizient ist. Zusätzlich steigt die Größe des Codes (Lines of Code) an und der Service wird schwerer zu warten. Aufgrund der frühen Entscheidung, der inhärenten Verkopplung von API und Anwendungslogik und des schweren Umstiegs auf andere Arten von APIs kommunizieren die meisten Services nur über eine Art von API an. Dies grenzt die Anwendung des Services ein. So kann ein Service, der mit REST kommuniziert nicht mit einem Service verbunden werden, der über RPC kommuniziert. Ebenfalls schränkt es die Entwicklung eines Clients ein. Dieser muss zwangsläufig in der gleichen Art der API, wie der Service, mit dem Service kommunizieren können.

Zusammenfassend lässt sich sagen, dass die bestehenden Ansätze zur Entwicklung und Betrieb von APIs sehr zeit- und kostenintensiv sind. Außerdem lassen sich die entstehenden Services nur schwer oder gar nicht mit anderen Arten von APIs erweitern. Durch die Entwicklung einer API für ein Service wächst die Größe des Codes (Lines of Code), ohne dabei einen Mehrwert für die Kernfunktionalitäten zu erzeugen. Dadurch wird die Wartung des Service komplizierter, insbesondere, wenn die API Entwicklung keine Kernkompetenz des gegebenen Umfeldes ist. Folgende Aufzählung fasst die oben genannten Anforderungen zur Entwicklung einer guten API zusammen.

1. Entkopplung von Anwendungslogik und API
2. leicht änderbare Entscheidung für eine Art von API
3. Spezifikation der Schnittstelle mit Hilfe einer IDL
4. Erweiterbarkeit der API
5. Möglichkeit weitere Arten an APIs bereitzustellen

3.2 CLARA Methode

APIs eignen sich besonders, Funktionalitäten programmatisch zu nutzen. Besonders sprachunabhängige APIs, wie REST, vereinfachen die Einbindung von Applikationen. Dadurch können verschiedenste Technologien hinter den APIs verborgen und genutzt werden, ohne dass sich ein API Konsument Gedanken über die Implementierung im Hintergrund machen muss.

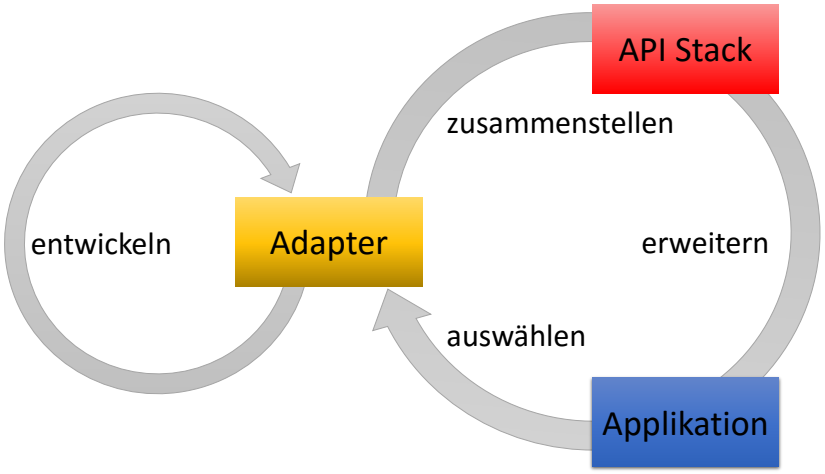

Abbildung 3.1: Überblick der CLARA Methode bestehend aus zwei Zyklen

In diesem Abschnitt betrachten wir eine Methode, um dynamisch spezifische Arten an APIs (zum Beispiel REST oder Messaging APIs), wie in Abbildung 3.1 gezeigt, zu entwickeln. Die Methode folgt einem abstrakten und generischen Ansatz, bei dem die API automatisch generiert werden soll und wird *CLARA* (Compose Lovely APIs based on Reusable API-Adapters) genannt. Auf einer abstrakten Ebene besteht die Methode aus zwei Phasen, die beliebig durchlaufen werden können. Die Schnittstelle beider Phasen und das Herz der Methode bilden Adapter, die den Prozess zur Erstellung vielfältiger APIs vereinfachen sollen. In der Entwicklungsphase (links) werden diese generischen und wiederverwendbaren Adapter entwickelt oder erweitert. Um die zweite Phase durchlaufen zu können, müssen bereits Adapter existieren. Die zweite

Phase beschäftigt sich mit der Erweiterung von Applikationen mit verschiedenen spezifischen Arten von APIs. Ist eine Applikation entwickelt, können beliebig viele Adapter ausgewählt werden, um eine API bereitzustellen. Befinden sich in dem Pool an API Bausteinen nicht die gewünschten Adapter, kann entsprechend Phase 1 durchlaufen und die benötigten generischen und wiederverwendbaren Adapter entwickelt werden. Diese Adapter können anschließend wie die anderen Adapter für unterschiedliche Anwendungen verwendet werden. Sind alle notwendigen Adapter ausgewählt, können sie den Anforderungen entsprechend zu einem API Stack zusammengestellt werden. Dieser erweitert die Applikation und stellt die Funktionalitäten der Applikation über ausgewählte Arten an APIs bereit. Entstehen neue Anforderungen an die API der Applikation können die Zyklen neu durchlaufen werden. Dadurch lassen sich einfach vielfältige APIs erzeugen. Existierende Adapter können beliebig für andere Applikationen wiederverwendet werden.

Im Folgenden wird die Methode im mehr Detail vorgestellt. Anstatt herkömmliche API Entwicklungsframeworks zu nutzen, um spezifische Arten an APIs von Grund auf zu entwickeln, kann mit CLARA der Prozess zur Erstellung vielfältiger APIs vereinfacht werden. Ein spezifischer Adapter, beispielsweise ein REST Adapter, kann genutzt werden, um die Funktionalitäten einer Applikation über eine API bereitzustellen. Abbildung 3.2 zeigt schematisch die Verbindung von einer Applikation mit einem einzelnen Adapter. Ein solcher API Adapter kann wie oben beschrieben ein REST+HTTP API Adapter sein. Es sind allerdings auch andere API Adapter wie RabbitMQ Adapter, JSON-RPC Adapter, etc. möglich.

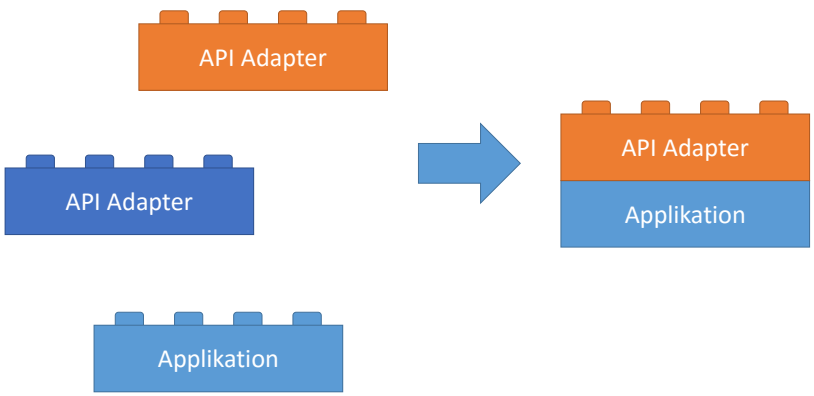

Abbildung 3.2: Minimale Erweiterung der Applikation über einen einzelnen API Adapter

Zwischenliegende Adapter (*Intermediate Adapters*) können optional weitere Funktionalitäten, wie die Authentifikation, der bereitgestellten API hinzufügen. Zusätzlich kann ein weiterer Adapter dafür verwendet werden, um die gleiche Funktionalität in einer anderen Art verfügbar zu machen, zum Beispiel über eine Messaging API, was eine API Vielfalt fördert. Da keine immer optimale Art von API exisitert, ist diese API Vielfalt entscheidend. Obwohl REST

in gewissen Fällen eine gute Wahl darstellt, arbeiten Messaging, RPC oder Streaming APIs manchmal viel besser, z.B. aus Effizienz- und Performance-Gründen. Aus diesem Grund bietet es sich an einfach, ohne großen Aufwand, weitere APIs erzeugen und zu einer bestehenden Applikation, wie einem Microservice, hinzufügen zu können.

Die API Bausteine sollen generisch, wiederverwendbar und konfigurierbar sein. Durch Implementierung von bewährten Praktiken, wie bestimmte Arten an APIs gebaut werden, können gewöhnliche und wiederkehrende Probleme vermieden werden, die bei der API Entwicklung von Grund auf auftreten können. Solche bewährte Praktiken können beispielsweise zeigen, wie langläufige Aufgaben mit REST zu implementieren sind, oder wie ein synchroner Aufruf mit einer Messaging API nachgebildet werden kann.

Damit die Adapter mit einer Applikation kommunizieren können, muss die Applikation selbst eine API, z.B. RPC, bereitstellen. Über die API stellt die Applikation spezielle Funktionalitäten über einen Endpunkt bereit. Eine solche Applikation kann von Grund auf entwickelt werden. Obwohl die von der Applikation bereitgestellten Endpunkte auch eine Art von API darstellen, die von anderen Applikationen genutzt werden können, sind im Sinne der API Vielfalt andere Arten von APIs besser für bestimmte Anwendungsszenarios geeignet. Aus diesem Grund sind die API Adapter (REST, JSON-RPC, Messaging, etc.) mit den Applikationen über deren Endpunkte verbunden, um verschiedene APIs anzubieten. Die Applikationen können allerdings weiterhin untereinander über die ursprüngliche Art an API (z.B. RPC) kommunizieren. Die Kommunikation von Applikation und Adapter ermöglicht eine Entkopplung von Anwendungslogik und API. Dadurch wird die in Abschnitt 3.1 vorgestellte Anforderung 1 erfüllt. Anforderung 2 (leicht änderbare Entscheidung für eine Art von API) wird ebenfalls erfüllt. Soll auf eine andere Art von API umgestiegen werden, so kann leicht ein anderer Adapter ausgewählt werden. Dabei muss der vorherige Adapter nicht ausgetauscht werden. Es können auch beide Adapter mit der Applikation kommunizieren, wodurch diese zwei Arten an API bereitstellt. Dies erfüllt Anforderung 5, da dadurch die API durch mehrere Arten der Kommunikation erweitert wird.

Ein API Adapter muss jedoch nicht zwangsläufig die API in andere Arten von APIs überführen, wie ein RPC-REST Adapter oder ein RPC-RabbitMQ Adapter, etc. Es existieren auch Zwischenadapter, die sogenannten *Intermediate Adapter*, die den Endpunkt einer Art von API bekommen und einen anderen Endpunkt derselben API Art bereitstellen. Solche Intermediate Adapter können verschiedene Middlewarefunktionalitäten, wie zum Beispiel die Umformung von Endpunkten (z.B. Filteroperationen, Authentifizierung, Anfragelimitierung, Monitoring, Analyse, Inhaltstransformationen, etc.) implementieren, was die API durch weitere Funktionen erweitert (Anforderung 4).

Ein Intermediate Adapter kann in Verbindung mit jeder Anwendung und jedem anderen API Adapter, einschließlich weiteren Intermediate Adapter, genutzt werden, sofern diese über dieselbe Art von API wie der Intermediate Adapter kommunizieren. Dadurch können viele Adapter, wie in dem in Abbildung 3.3 gezeigten Beispiel gestapelt werden.

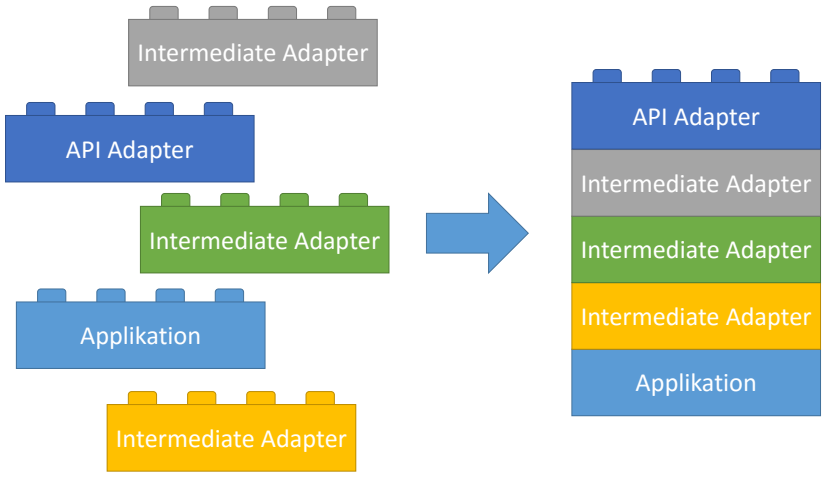

Abbildung 3.3: Schematische Darstellung der Verbindung von mehreren API Bausteinen und einer Applikation

Die API Bausteine können wie Legosteine kombiniert und gestapelt werden, um einen beliebigen API Stack zu bilden und zu bündeln. Um diese Zusammensetzung der Applikationen und API Adapter zu erleichtern, ist es äußerst empfehlenswert die Applikation in einem Container laufen zu lassen. Alle API Bausteine (API Adapter und Intermediate Adapter) sollten über die gleiche Art an API, z.B. RPC, miteinander kommunizieren können, um eine möglichst flexible Einsetzbarkeit der Adapter zu ermöglichen. Es kann sein, dass manche Adapter weitere Informationen benötigen, um die entsprechenden Endpunkte sauber auf eine andere Art an API abzubilden. Ein weiteres Beispiel für das Zusammensetzen von API Stacks über Adapter findet sich in Abbildung 3.4.

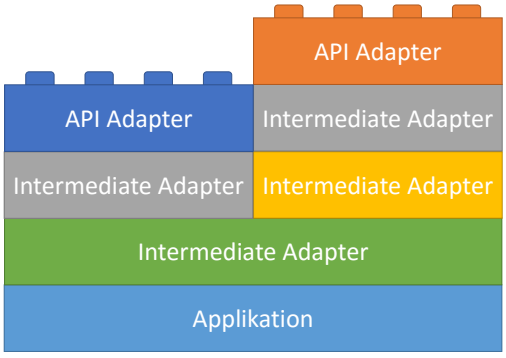

Abbildung 3.4: Parallele API Stacks einer Applikation

Die Adapter sollten so klein und gebündelt sein, wie nur möglich. Im Fall von komplexeren Adaptern ist es äußerst empfehlenswert diese über Plugins modular und erweiterbar zu halten. Die spezifische Schnittstelle zwischen den Adaptern und Plugins hängt dabei von der jeweiligen Adapterimplementierung ab. Optional können Adapter die Schnittstelle mit Hilfe einer Interface Definition Language spezifizieren, was Anforderung 3 erfüllt. Der folgende Abschnitt stellt ein Framework vor, dass in dem eine Sammlung von Adaptern vorliegt und die Zusammensetzung dieser zu API Stacks erleichtert.

3.3 CLARA Framework

Um die oben beschriebene Methode zu nutzen, bietet es sich an ein Framework, wie *CLARA*[Spe17] zu verwenden. CLARA bezweckt die Einrichtung eines offenen Ökosystems aus zusammensetzbaren API Bausteinen, wie wiederverwendbare Wrapper und Adapter. Die API Bausteine erleichtern bei Benutzung den Prozess der Erstellung von vielfältiger APIs, wodurch keine herkömmlichen API Entwicklungsframeworks mehr genutzt werden müssen, um spezifische Arten von APIs von Grund auf zu bauen. Wie oben beschrieben sind die API Bausteine generisch, wiederverwendbar und konfigurierbar.

Im CLARA Ökosystem geht es darum API Bausteine zusammenzusetzen, um APIs zu generieren. Daher muss jeder Baustein spezifische Schnittstellen implementieren und respektieren. Um technische Vielfalt zu fördern und die Implementierung der zusammensetzbaren API Bausteine durch verschiedene Technologien zu erlauben, sind die technischen Grundvoraussetzungen des Ökosystems Docker Container und gRPC:

- API Bausteine sind in Docker Containern verpackt. Daher können bewährte Container Entwicklungs- und Besetzungstolls, sowie Plattformen wie Kubernetes und Docker Compose genutzt werden.
- API Bausteine kommunizieren über Protocol Buffers (proto3)[Goo17a] und gRPC[Goo17b], da dies effizient, performant und aufgrund von HTTP2[BTP15] auch Standard-basiert ist. Es existieren gRPC Bibliotheken für verschiedene Programmiersprachen. Da diverse Programmiersprachen unterschiedlich gut für verschiedene Probleme geeignet sind, lassen sich die API Bausteine in der am besten geeigneten Programmiersprache implementieren. Dieses Verfahren ist dadurch technologie-agnostisch.

Das Framework beinhaltet verschiedene Arten von API Bausteinen, hauptsächlich Intermediate Adapter und API Adapter. Diese werden, ähnlich zu Legobausteinen, zu API Stacks kombiniert und gestapelt. Ein API Stack kombiniert mehrere Adapter mit einer bestimmten Applikation, wie es schematisch in Abbildung 3.3 dargestellt ist.

Für die Kommunikation mit anderen Adaptern und der gRPC Anwendung muss ein Adapter alle Protocol Buffers und gRPC Kernbestandteile verstehen und die Umgebungsvariablen, wie die IP Adresse und den Hostnamen des unterliegenden gRPC Endpunktes (entweder durch

eine gRPC Anwendung oder einen weiteren Adapter bereitgestellt) lesen können, um sich mit diesem zu verbinden. Ist der Adapter ein Intermediate Adapter, so muss dieser eine gRPC bereitstellen, um es weiteren Adaptern zu ermöglichen darauf aufbauend zu arbeiten.

Durch einen gRPC Endpunkt, der über eine Protocol Buffers Datei charakterisiert wird, bietet eine gRPC Applikation konkrete Funktionalitäten (Anwendungungs-/Geschäftslogik) an. Um die Zusammensetzung der gRPC Applikationen, wie Microservices, die Funktionen eines Webshops implementieren, und API Adapter zu erleichtern, wird von dem Framework empfohlen die Applikation in einem Docker Container auszuführen. Die Schnittstelle eines gRPC Applikationscontainers sieht dabei wie folgt aus:

- Ein gRPC Applikationscontainer muss das Label `org.clara.kind="app"` besitzen.
- Eine gRPC Applikation muss eine `/api/main.proto` Datei bereitstellen, um die gRPC Schnittstelle zu spezifizieren.
- Der `/api` Ordner muss als Shared Container Volume von dem gRPC Applikationscontainer bereitgestellt werden. Dieser Ordner wird von den API Adaptern verwendet. Dabei muss das `Dockerfile` der gRPC Applikation die Anweisung `VOLUME /api` beinhalten.
- Der `/api` Ordner kann optional weitere Dateien beinhalten, um zusätzliche Metadaten festzulegen. Diese Metadaten können von den API Adaptern gelesen und berücksichtigt werden.
  Zum Beispiel könnte ein REST API Adapter zusätzliche Informationen verwerten, wie die jeweiligen RPC Operationen auf die Ressourcen und deren dazugehörigen CRUD Operationen abzubilden sind.
- Standardmäßig legt eine gRPC Applikation den gRPC Endpunkt dar, der über die `/api/main.proto` Datei beschrieben wird und auf den TCP Port `50051` hört. Falls die gRPC Applikation seine gRPC API über einen anderen Port preisgibt, muss der Docker Container das Label `org.clara.api-port="..."` mit der entsprechenden Port Nummer als Wert beinhalten.

gRPC Applikationen können von Grund auf in einer beliebigen Programmiersprache geschrieben sein, die von dem gRPC Framework unterstützt wird. Alternativ können Wrapper genutzt werden, um gRPC Endpunkte für ein existierenden Code dynamisch und automatisiert bereitzustellen.

Betrachten wir nun die Adapter genauer. Wie beschrieben kann eine gRPC Applikation über verschiedene Wege erzeugt werden. Die Adapter kommunizieren mit einer gRPC Applikation über die jeweiligen gRPC Endpunkte und stellen entweder selbst wieder gRPC Endpunkte bereit, wie die Intermediate Adapter, oder andere Arten von APIs, wie REST über HTTP. Die Intermediate Adapter können, wie in Abbildung 3.5 schematisch dargestellt und in Abschnitt 3.2 beschrieben, mit jeder beliebigen gRPC Applikation und weiteren API Adaptern verbunden werden. In diesem Beispiel wird eine gRPC Applikation durch verschiedene Intermediate Adapter erweitert, um Funktionalitäten, wie Monitoring, ein Anfragelimit und Authentifikation einzubinden. Zusätzlich wird ein REST API Adapter darauf aufbauend eingebunden, um

zusätzlich zur gRPC API beispielsweise eine REST + HTTP API nach außen anbieten zu können.

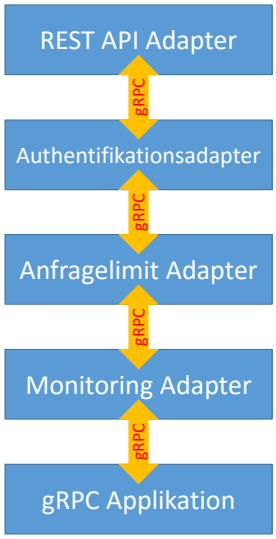

Abbildung 3.5: Beispiel eines Stacks aus API Adaptern

Um diese Komposition von gRPC Applikationen und API Adaptern zu erleichtern, empfiehlt das Framework einen Adaper ebenfalls in einem Docker Container zu virtualisieren. Die Schnittstelle eines Adapter Containers ist wie folgt:

- Ein Adapter Container muss das `Dockerfile` des Adapters das Label `org.clara.kind="adapter"...` besitzen. Im Falle eines Intermediate Adapters muss das Label `org.clara.kind="intermediate-adapter"` lauten. Falls der Adapter selbst einen API Endpunkt zur Verfügung stellt, muss der Docker Container das Label `org.clara.api-port` Beschriftung mit der entsprechenden Port Nummer beinhalten.
- Ein Adapter kann das `org.clara.keywords` Label besitzen, um die Eigenschaften der bereitgestellten API zu spezifizieren. Die Keywords sind dabei in einer durch Kommata getrennten Liste von Schlüsselwörtern aufgeführt.
  - Schlüsselwörter für Protokolle und Datenformate: `http`, `grpc`, `xml`, `json`, `json-rpc`,...
  - Schlüsselwörter für Fähigkeiten: `sync`, `async`, `rest`, `rpc`, `event-driven`, `messaging`, `streaming`,...
- Für API Adapter Container muss das `/api` Verzeichnis ein Shared Container Volume sein. Dieses wird typischerweise von der unterliegenden gRPC Applikation (oder dem Intermediate Adapter) bereitgestellt und kann von dem API Adapter Container genutzt werden. Die Umgebungsvariable `API_DIR` kann optional dem Adapter bereitgestellt

werden, um ein anderes Verzeichnis als das `/api` Verzeichnis zu nutzen. Dabei muss der Adapter diese Variable verarbeiten, sofern diese gesetzt ist.

- Der Adapter benötigt als einzige Datei die `/api/main.proto` (eventuell auch aus dem geänderten Verzeichnis) und muss auf Basis dieser ohne zusätzliche Metadaten geeignet arbeiten.
- Zusätzlich kann der API Adapter, wie oben beschrieben, Metadaten über die `/api/meta.yml` Datei erhalten, um eine bessere Abbildung auf die jeweilige Art von API zu erzeugen.
- Um korrekt mit der unterliegenden Applikation/Adapter kommunizieren zu können, muss der Adapter alle Protocol Buffers 3 und gRPC Features verstehen.
- Ein Adapter muss die Umgebungsvariablen `GRPC_HOST` (IP Adresse und Hostname) und `GRPC_PORT` (Port Nummer) kennen, um mit dem unterliegenden gRPC Endpunkt kommunizieren zu können.
- Ein Adapter muss seinen Start verzögern, bis der unterliegende gRPC Endpunkt verfügbar ist. Dieses Verhalten kann Implementiert werden, indem in dem `Dockerfile` der Befehl `CMD dockerize -wait tcp://$GRPC_HOST:$GRPC_PORT ...`[Wil17] eingetragen wird. Dies ist erforderlich, da der Adapter erst nach Verfügbarkeit des unterliegenden gRPC Endpunktes sicher sein kann, dass die erforderliche `main.proto` Datei exisitiert.

Die Schnittstelle eines Intermediate Adapters muss zusätzlich folgende Regeln einhalten:

- Ein gRPC-gRPC Intermediate Adapter muss des Weiteren die Schnittstelle eines gRPC Applikationscontainers implementieren, um es anderen Adaptern zu ermöglichen darauf aufbauend zu laufen.
- gRPC-gRPC Intermediate Adapter verändern möglicherweise die die urpsrüngliche gRPC Schnittstelle. Daher müssen die Veränderungen auch in den Dateien des `/api` Verzeichnisses (oder dem geänderten Verzeichnis), wie die `main.proto` oder Dateien für Metadaten, berücksichtigt werden. Standardmäßig werden die existierenden Dateien unmittelbar verändert und überschrieben.
  Falls die Umgebungsvariable `UPDATED_API_DIR` dem Adapter Container mit einem anderen Pfad als `/api` oder `API_DIR` bereitgestellt wird, muss der Adapter das existierende Verzeichnis duplizieren und alle Veränderungen dem kopierten Verzeichnis hinzufügen. Der Inhalt des originalen Verzeichnisses bleibt unverändert. Der durch `UPDATED_API_DIR` definierte Pfad sollte ein Unterverzeichnis des `/api` oder `API_DIR` Verzeichnisses sein, um in einem Shared Container Volume zu sein.

4 Design und Implementierung

Im folgenden Kapitel beschäftigen wir uns mit dem Design eines API Adapters. Als Beispiel eines solchen API Adapters betrachten wir die Funktionsweise eines gRPC-REST Adapters und dessen Implementierung in NodeJS genauer. Der gRPC-REST Adapter wird dabei im Folgenden nur noch als Adapter bezeichnet, sofern es aus dem Kontext hervor geht, dass es sich dabei um diesen handelt. Dieser API Adapter soll den von einem Microservice bereitgestellten, und eventuell von diversen Intermediate Adapter veränderten, gRPC Endpunkt in einen äquivalenten REST Endpunkt umwandeln und eine API Anbindung an den Microservice herstellen. Der Adapter folgt dem Konzept der in Kapitel 3.2 vorgestellten Methode CLARA und ist Teil des in Kapitel 3.3 vorgestellten Frameworks.

Die Funktionsweise des Adapters lässt sich in drei Abschnitte untergliedern. Im ersten Abschnitt wird der von der unterliegenden Schicht bereitgestellte gRPC Endpunkt über die Protocol Buffers 3 Datei eingelesen und zur Weiterverarbeitung aufbereitet. Anschließend werden im zweiten Abschnitt des Adapters die aufbereiteten Daten auf eine Swagger-Definition übertragen, welches dann eine Definition der zur erzeugenden REST Schnittstelle darstellt die äquivalent zur gRPC Schnittstelle ist. Zuletzt werden im dritten Abschnitt die in der Swagger Datei definierten URL-Pfade implementiert. Während Abschnitt 1 für alle API Adapter möglich ist, müssen Abschnitt 2 und 3 an den entsprechenden API Adapter angepasst werden. So benötigt ein gRPC-Messaging Adapter eine Abbildung auf eine Messaging Schnittstellendefinition und die Erzeugung derer in Abschnitt 3. Betrachten wir zunächst den ersten Abschnitt.

4.1 Schritt 1: Analyse der gRPC Schnittstelle

In diesem Abschnitt betrachten wir wie aus der Protocol Buffers-Datei die Informationen gewonnen werden können, die für eine Swagger Datei benötigt werden. Die Implementierung bezieht sich dabei speziell auf ein in NodeJS implementierter Adapter. Es gibt sicherlich äquivalente Möglichkeiten in anderen Programmiersprachen, die hier allerdings nicht weiter betrachtet werden.

Grundlegend muss der Adapter zu Beginn die Protocol Buffers-Datei einlesen, welche die gRPC Schnittstelle definiert. Diese Datei befindet sich wie in Kapitel 3.3 vorgestellt im Shared Container Volume des Applikationscontainers (bspw. `/api/main.proto`). Der Adapter bindet hierfür

die gRPC Bibliothek für NodeJS (`grpc`)[grp15] ein. Mit dem Befehl `grpc.load(PROTO_PATH)`[1] und dem übergebenen Pfad zur `main.proto` wird die Protocol Buffers-Datei eingelesen und in ein JSON Objekt gespeichert. Dieses Objekt enthält alle benötigten Informationen und ist wie folgt aufgebaut: In den äußeren Hierarchien befinden sich die Protocol Buffers (Unter-) Pakete als Properties des Objektes. Der Wert einer jeden Property ist entsprechend ein Unterpaket oder die Message und Service Objekte, wobei die direkten Message Objekte nur die entsprechenden encode und decode Funktionen beinhalten. Die tatsächlichen Strukturen der Message Objekte befinden sich in den Service Objekten, welche die für den Adapter einzigen verwendbaren Objekte darstellen, da diese alle benötigten Informationen beinhalten. Ein kleines Beispiel eines solchen Service Objektes ist in Listing 4.1 dargestellt. Die zugehörige Protocol Buffers-Datei befindet sich in Listing 4.2.

Es ist zu beachten, dass das JSON Objekt der Protocol Buffers-Datei zirkular aufgebaut ist. Jedes Element enthält selbst immer das Elternobjekt. Betrachten wir ein Service Objekt an dem gegebenen Beispiel genauer. Jeder Service enthält ein Feld mit dem Servicenamen, sowie die zugehörigen RPC Operationen im `children` Array, die in der Protocol Buffers-Datei definiert wurden. Zusätzlich beinhaltet das Objekt noch weitere Objekte, die für den Adapter allerdings irrelevant sind und deshalb nicht näher betrachtet werden. Jedes Objekt einer RPC Operation enthält alle benötigten Informationen über die Operation, wie den Namen der Operation oder die Namen der Request und Response Objekte. Die Message Definitionen der Request und Response Objekte befinden sich in den `resolvedRequestType` und `resolvedResponseType` Feldern und sind ebenfalls eigene JSON Objekte. Zusätzlich enthalten die Operationsobjekte noch Informationen, ob die Request oder Response Objekte als Stream übergeben werden.

Der Adapter iteriert über die RPC-Operationen und extrahiert rekursiv die relevanten Daten aus den Objekten. Dabei werden die Request und Response Objekte genauer betrachtet, die wie alle anderen Message Objekte aufgebaut sind. In Listing 4.3 ist ein verkürztes Beispiel einer Message Definition angegeben. Dabei wurden alle für den Adapter unwichtigen Elemente zur besseren Übersichtlichkeit aus dem Objekt entfernt. Die Message Objekte besitzen einen Namen, einen Klassennamen und im `children` Array die Felder als JSON Objekte. Jedes Feld Objekt wiederum enthält alle in Protocol Buffers möglichen Informationen, wie `required` oder `repeated`, sowie den Typ des Feldes. Ist ein Feld selbst von einem Message Typ, wie es in dem gegebenen Beispiel bei dem Feld `available` der Fall ist, kann auf die entsprechende Message Definition über die Property `resolvedType` des Feldes zugegriffen werden.

Der Adapter iteriert über die Felder (`children` Arrays) der Message-Definitionen. Liegt ein Message Typ als Feld vor, extrahiert der Adapter die für das proto-Swagger Mapping relevanten Informationen. Dabei wird rekursiv auf Message Definition über die `resolvedType` Property des Nachrichtenfeldes zugegriffen. Die aufbereiteten Message Definitionen werden dann in einer HashMap zwischengespeichert. Dadurch kann sichergestellt werden, dass keine Message mehrmals analysiert wird. Submessages bekommen den Namen der Parent-Message als Präfix angehängt, um Namenskonflikte zu vermeiden. Primitive Datentypen werden direkt

[1] http://www.grpc.io/docs/tutorials/basic/node.html

Listing 4.1 Kleines Beispiel eine Service Objekt eines als geladenes JSON Objekt

```
service: {
   builder: { Object }, # Some builder object
   parent: { Object }, # The parent object
   name: 'WebShop',
   className: 'Service',
   children: [ # Service operations
   {
     builder: [Object],
     parent: [Circular],
     name: 'listProducts',
     className: 'Service.RPCMethod',
     options: {},
     requestName: 'ProductId',
     responseName: 'Product',
     requestStream: true,
     responseStream: true,
     resolvedRequestType: [Object],
     resolvedResponseType: [Object] },
    {
     builder: [Object],
     parent: [Circular],
     name: 'checkAvailability',
     className: 'Service.RPCMethod',
     options: {},
     requestName: 'ProductId',
     responseName: 'Availability',
     requestStream: false,
     responseStream: false,
     resolvedRequestType: [Object],
     resolvedResponseType: [Object] }
   ],
   # Other unrelevant informations
}
```

auf entsprechende JSON Datentypen abgebildet. Dies ist nötig, da Protocol Buffers mehr primitive Datentypen zulässt als JSON, wodurch das Swagger Objekt ansonsten fehlerhaft wäre. Die gesammelten Informationen werden als JSON Objekte in ein Array gespeichert. Eines der Objekte enthält dabei folgende Informationen: Servicename, Paketname, Operationen, Messages. Im Falle von mehreren Services oder Pakete werden dadurch mehrere Objekte dem Array hinzugefügt. Das Array enthält dadurch alle für die Abbildung auf eine Swagger-Definition benötigten Informationen und kann hierfür in Schritt 2 weiterverwendet werden.

Listing 4.2 Die zu Listing 4.1 gehörende Protocol Buffers-Datei

```
syntax = "proto3";

message Product {
  string id = 1; // product ID should be a valid UUID
  string name = 2;
  string producer = 4; // producer name
  float weight = 5; // in kilogram
  float price = 6; // in EUR
  Availability available = 7;
}

message ProductId {
  string id = 1;
}

message Availability {
  bool available = 1; // true if product is available
}

service WebShop {
  rpc listProducts(stream ProductId) returns (stream Product) {}
  rpc checkAvailability(ProductId) returns (Availability) {}
}
```

4.2 Schritt 2: Abbildung auf eine Swagger-Definition

In Abschnitt 4.1 wurde beschrieben, wie der Adapter die Informationen der gRPC Schnittstellendefinition erhält. Der Adapter soll aus einer gRPC API eine, wie in Kapitel 2.1.2 beschrieben, saubere REST API generieren. Damit die generierten REST APIs immer nach dem gleichen Schema aufgebaut sind, werden die aus der Protocol Buffers Datei extrahierten Informationen über die gRPC API auf eine Schnittstellendefinition für REST abgebildet. Hierfür nutzt der Adapter die in Kapitel 2.2.2 vorgestellte Definitionssprache Swagger. Der Grundbaustein für die Abbildung auf eine REST API bildet eine Schablone für eine Swagger-Definition, die die Struktur der generierten REST API wiedergibt und im Folgenden vorgestellt wird. Die generierte Swagger-Definition wird nach Abschluss der Abbildung sowohl in JSON, als auch in YAML gespeichert, wodurch der Anwender die für ihn ansprechendere Form betrachten kann.

Betrachten wir die Swagger Schablone, die zur besseren Lesbarkeit in YAML dargestellt ist und auf die die gRPC Schnittstelle abgebildet wird. Die RPC Operationen werden mit Hilfe des in Kapitel 2.1.2 beschriebenen REST Command Pattern in REST dargestellt. Listing 4.4 zeigt die Definition der Schablone für den Teil des Path Objektes, der die grundlegende HTTP POST Anfrage eines Operationsaufrufs darstellt. Der Adapter erzeugt für jede der RPC Operationen den gegebenen Swagger Teil, wobei die Platzhalter im Pfad entsprechend durch den Paketnamen, sowie den Operationsnamen ersetzt werden. Im Body der POST Anfrage wird in der REST API die entsprechende Request Message der Operation als Parameter übergeben. Hierfür

Listing 4.3 Beispiel des JSON Objektes einer Message Definition

```
{
 # builder and parent
 name: 'Product',
 className: 'Message',
 children:
  [ { # builder and parent
     name: 'id',
     className: 'Message.Field',
     required: false,
     repeated: false,
     # ...
     type: [Object],
     resolvedType: null,
     id: 1,
     # ...
   },
    # ...
    { # builder and parent
      name: 'available',
      className: 'Message.Field',
      required: false,
      repeated: false,
      # ...
      type: [Object],
      resolvedType: [Object], # Object of 'Availability' message definition
      id: 7,
      # ...
    } ],
   # ...
}
```

muss in der Schablone jeweils der Verweis auf das entsprechende Message Objekt im Swagger Definitions Teil angepasst werden.

Als Response der Anfrage wird eine Instanz Ressource zurückgegeben. Diese Instanz Ressource folgt dem in Kapitel 2.1.2 vorgestellten REST Command Pattern und enthält die wichtigen und aktuellen Informationen über den Status der Operation enthält. Diese Instanz Ressource kann dann mit einem HTTP GET Request angefragt werden und ermöglicht dadurch dem Benutzer jederzeit Informationen über die Operation zu erhalten. Wurde die Operation beendet, enthält diese Instanz Ressource einen Link zu den Ressourcen, die eigentlich als Ergebnis der Operation zurückgegeben wurden. Diese Ressourcen können dann wiederum mit einem HTTP GET Request angefragt werden.

Für den HTTP GET Request der Instanz Ressource erzeugt der Adapter für jede Operation den in Listing 4.5 angegebenen Teil des Path Objektes. Zusätzlich zu den GET Requests werden für jede Operation PATCH Requests der Instanz Ressourcen erzeugt. Diese ermöglichen eine Veränderung der Instanz Ressource. Zur besseren Übersicht wurde dieser Teil in Listing 4.5

Listing 4.4 Swagger Schablone des Adapters: Grundlegender Path Objekt Teil

```
paths:
  /<PLACEHOLDER_PKG_SERVICE_NAME>/<PLACEHOLDER_OPERATION_NAME>:
    post:
      summary: PLACEHOLDER
      consumes:
        - application/json
      parameters:
        - name: start
          in: query
          type: boolean
          default: true
        - name: input
          in: body
          schema:
            $ref: '#/definitions/<PLACEHOLDER_OPERATION_REQUEST_MESSAGE>'
      responses:
        202:
          description: Instance resource
          headers:
            Content-Location:
              description: Path to created instance resource
              type: string
          schema:
            $ref: '#/definitions/Instance'
```

durch einen Kommentar abgekürzt. Besitzt die RPC Operation einen Stream als Response, passt der Adapter die Definition der „responses“ des GET Requests entsprechend auf eine alternative Darstellung für Streams dar.

Außer den bereits genannten Pfaden werden noch eine Reihe an verschiedenen Pfaden für die jeweiligen Felder der Messages definiert. Dabei wird unterschieden, ob Response oder Request als Stream vorliegt, oder nicht. Betrachten wir in Listing 4.6 das Pfad Objekt für Felder von einer Response Message. Dieser Pfad wird nur erzeugt, sofern die gRPC Operation keinen Stream an Response Messages enthält. Listing 4.6 zeigt die Definition des GET Requests auf die Felder der Response Message der gRPC Operation. Als Parameter wird im Pfad die ID der Instanz übergeben, die durch den oben beschriebenen POST Request zurückgegeben wurde. Außerdem ersetzt der Adapter den Platzhalter für den Feldnamen durch den entsprechenden Feldnamen. Für jedes einzelne Feld der Message generiert der Adapter daher den kompletten Block und fügt diesen den Pfad-Objekten hinzu. Der GET Request liefert als Antwort den Wert des gewünschten Feldes. Beim Generieren der Pfade für die Felder muss der Adapter jeweils den Typ des Feldes dem Pfadobjekt hinterlegen. Um eine möglichst spezifische Definition zu erhalten, hinterlegt der Adapter dem Typ des Feldes noch ein Format Feld. So können die JSON Datentypen, wie „number“ oder „integer“ zu Swagger Datentypen spezifiziert werden, wie beispielsweise zu „int32“.

Listing 4.5 Swagger Schablone des Adapters: GET Request der Instanz Ressource

```
/<PLACEHOLDER_PKG_SERVICE_NAME>/<PLACEHOLDER_OPERATION_NAME>/instances/{id}:
  patch: # Path part of swagger file
  get:
    summary: Get instance resource
    produces:
      - application/json
    parameters:
      - name: id
        in: path
        description: Unique identifier of instance
        required: true
        type: string
      - name: excludeOutput # HINT: add this param only if operation does NOT have out
          stream
        in: query
        description: Omit instance output in response
        type: boolean
        default: false
    tags:
      - Instances
    responses:
      200:
        description: Instance resource
        schema:
          type: object
          allOf:
          - $ref: '#/definitions/Instance'
          - type: object
            properties:
              out:
                $ref: '#/definitions/<PLACEHOLDER_OPERATION_RESPONSE_MESSAGE>'
```

Entsprechend angepasste Pfad Objekte für gRPC Operationen mit Streams als Response oder Request werden ebenfalls erzeugt. Jedem Pfad Objekt werden sogenannte „tags“ hinzugefügt, nach denen gefiltert werden kann. Dies ist sinnvoll, da bei der Abbildung eine große Anzahl an Pfaden entstehen und oftmals viele Operationen der gRPC API abgebildet werden. Zusätzlich zu den oben dargestellten „tags“ werden noch Paketname, Servicename und Operationsname hinzugefügt, wobei Servicename und der Operationsname jeweils den Paketnamen als Präfix (für den Operationsnamen auch zusätzlich der Servicename) erhalten. Dadurch entstehen auch beim Filtern keine Namenskonflikte.

Als letztes erzeugt der Adapter den Defintions Teil des Swagger Objektes. Hierbei werden immer die Objektdefinitionen, wie in Listing 4.7 dargestellt, für die beiden Instanz Ressourcen generiert. Die Instanz Ressourcen wird durch den oben beschriebenen POST Request zurückgegeben und enthält Informationen darüber, ob und wann die Operation gestartet bzw. beendet wurde, das Objekt erzeugt wurde, sowie Fehlermeldungen, falls die gRPC Operation nicht fehlerfrei abgelaufen ist. Darüber hinaus enthält die Instanz Ressource nach Abschluss der

Listing 4.6 Swagger Schablone des Adapters: GET Request auf Felder der Response Message

```
/<PLACEHOLDER_PKG_SERVICE_NAME>/<PLACEHOLDER_OPERATION_NAME>/instances/{id}/out/
  fields/<PLACEHOLDER_FIELD_NAME>:
    get:
      summary: Get output field
      parameters:
        - name: id
          in: path
          description: Unique identifier of instance
          required: true
          type: string
      tags:
        - Instances
        - Fields
      responses:
        200:
          description: Field value
          schema:
            type: <PLACEHOLDER_FIELD_TYPE>
```

gRPC Operation die URI der Ressourcen der gRPC Response Message. Zusätzlich werden alle in Schritt 1 des Adapters gespeicherten Message Definitionen entsprechend der Swagger Spezifikation im Definitions Teil abgebildet. Enthält eine Protocol Buffers Message eine Aufzählung (Enumeration), so wird diese in einer für Swagger validen Darstellung abgebildet. Besitzt die Protocol Buffers-Datei ein Paket, wird dieses als Präfix vor den jeweiligen Message Objekten angefügt.

Protocol Buffers-Dateien können andere Protocol Buffers-Dateien importieren. Enthält eine Message ein Feld vom Typ einer importierten Message, erkennt der Adapter nicht, dass die importierte Message bereits durch eine andere Protocol Buffers-Datei vorliegt und speichert diese zusätzlich als neue Message Definition mit dem aktuellen Paketnamen als Präfix ein. Diese Redundanz wird bei der Abbildung auf Swagger mit übertragen, weshalb die Message in der Swagger Definition mehrmals aufgeführt wird. Aufgrund der unterschiedlichen Paketnamen als Präfix der Messages entstehen jedoch keine Namenskonflikte, die sich negativ auf die Funktionalität des Adapters auswirken könnten. Ist die Swagger Datei generiert, kann der Adapter mit Schritt 3 fortfahren.

4.3 Schritt 3: Generierung der REST Schnittstelle

Dieser Abschnitt beschreibt die dynamische Generierung der REST Schnittstelle des Adapters. Aus den Schritten 1 und 2 geht die Definition der REST Schnittstelle in Form einer Swagger Datei hervor. Diese enthält alle erlaubten Pfade mit den entsprechenden HTTP Verben. In Schritt 3 muss der Adapter mit Hilfe der Swagger Datei die HTTP Verben auf die erlaubten Pfade abbilden. Hierfür bindet der Adapter das NodeJS Express Framework mit `express =`

Listing 4.7 Swagger Schablone des Adapters: Definition der Instanz Ressource

```
definitions:
 Instance:
  type: object
  properties:
   id:
    type: string
    description: Unique identifier of instance
   started:
    type: boolean
   done:
    type: boolean
   createdAt:
    type: string
   startedAt:
    type: string
   doneAt:
    type: string
   error:
    type: string
    description: Error message if instance failed
   links:
    type: object
    description: Links to related resources such as output
```

`require(express)` ein und erzeugt eine Instanz der Klasse express mit `app = express()`. Das Routing wird auf der Instanz `app` ausgeführt. In Listing 4.8 zeigt, wie das Routing der POST Requests dynamisch durchgeführt werden könnte. Hierfür muss der Adapter zuerst die Pfad Objekte aus dem Swagger Objekt auslesen und nach den HTTP Verben unterteilen. Anschließend kann der Adapter auf die Pfad Objekte des jeweiligen HTTP Verbs das entsprechende Routing durchführen. Im Express Routing muss die entsprechende gRPC Operation ebenfalls dynamisch aufgerufen werden.

Listing 4.8 Dynamisches Routing der POST Requests

```
postPathObjects.forEach(function (pathObject) {
      app.post(pathObject.pathName, function(res, req) {
            // Invoke gRPC Operation and return instance resource
      });
});
```

Bei einem POST Request wird die entsprechende gRPC Operation aufgerufen und eine Instanz Ressource zurückgegeben. Die Instanz Ressourcen müssen von dem Adapter gespeichert werden. Hierfür bietet sich eine In-Memory-Datenbank an, welche die Instanz-Objekte speichert. Diese Datenbank kann dann entsprechend abgerufen werden und die Instanz-Objekte zurückgeben.

Betrachten wir als nächstes in Listing 4.9 den Aufruf einer gRPC Operation ohne Stream näher. Das Objekt `client` ist der CLient-Stub, der mit der unter dem Adapter liegenden Schicht kommuniziert und deren gRPC Operationen aufrufen kann. In diesem Beispiel wird die in Listing 4.2 definierte Operation `checkAvailability` aufgerufen. Der Operation wird ein der `Availability` Message entsprechenden JSON Objekt mit der ID 1 übergeben.

Listing 4.9 Aufruf einer gRPC Operation

```
client.checkAvailability({id:'1'}, function(err, response) {
 if (err) {
  // Show error message
 } else {
  if (response.available) {
   // Do something
  } else {
   // Do something
  }
 }
});
```

Für gRPC Operationen mit Streams erfolgt der Aufruf etwas anders. Ein solcher Aufruf wird, für die in Listing 4.2 definierte Operation `listProducts`, in Listing 4.10 dargestellt. Der Operationsaufruf wird einem Objekt (`call`) zugewiesen. Dieses Objekt besitzt die Methode `on`, mit der Daten empfangen werden können. Im gegebenen Beispiel werden 10 Produkte aufgelistet. Solange das Call Objekt neue Daten erhält, wird das `'data'` Event aufgerufen und das zurückgegebene Produkt ausgegeben. Das `'end'` Event wird aufgerufen, sobald alle Daten gelesen wurden.

Listing 4.10 Aufruf einer gRPC Operation mit Response als Stream

```
var call = client.listProducts({limit: 10});
call.on('data', function(product) {
 console.log('Received product: ' + product.id + ' ' + product.name);
});
call.on('end', function() {});
```

Das Routing für andere HTTP Verben erfolgt äquivalent zu Listing 4.8. Listing 4.11 zeigt entsprechend das mögliche Routing der GET Requests. Bis auf das HTTP Verb bleibt das Routing gleich wie beim POST Request. Es ändert sich allerdings die Anwendungslogik. Anstatt eine gRPC Operation aufzurufen, wird bei einem GET Request die der Operation entsprechende Instanz Ressource zurückgegeben.

Zusätzlich zum Routing muss der Adapter mit der darunter liegenden Schicht (Intermediate Adapter oder gRPC Applikation) verbinden. Hierfür müssen die in Kapitel 3.3 beschriebenen Anforderungen eingehalten werden. Insbesondere muss der Adapter über die Umgebungsvariablen `GRPC_HOST` und `GRPC_PORT` mit dem unterliegenden gRPC Endpunkt kommunizieren.

Listing 4.11 Dynamisches Routing der GET Requests

```
getPathObjects.forEach(function (pathObject) {
      app.get(pathObject.pathName, function(res, req) {
            // Return instance resource
      });
});
```

Abbildung 4.1 zeigt zusammenfassend den Überblick über die Funktionalität des gRPC-REST API Adapter. Im Docker Container der Applikation liegt die Protocol Buffers Datei, die die gRPC Schnittstelle der Applikation definiert. Diese wird wie beschrieben von dem API Adapter konsumiert auf in eine REST Schnittstellendefinition in Swagger abgebildet. Zusätzlich bietet der API Adapter die abgebildete REST API an. Ein Client kann mit der Applikation über REST kommunizieren, in dem der REST Request an den Adapter gesendet wird. Dieser bearbeitet den Request wie beschrieben und sendet eventuell gRPC Operationsaufrufe an die Applikation.

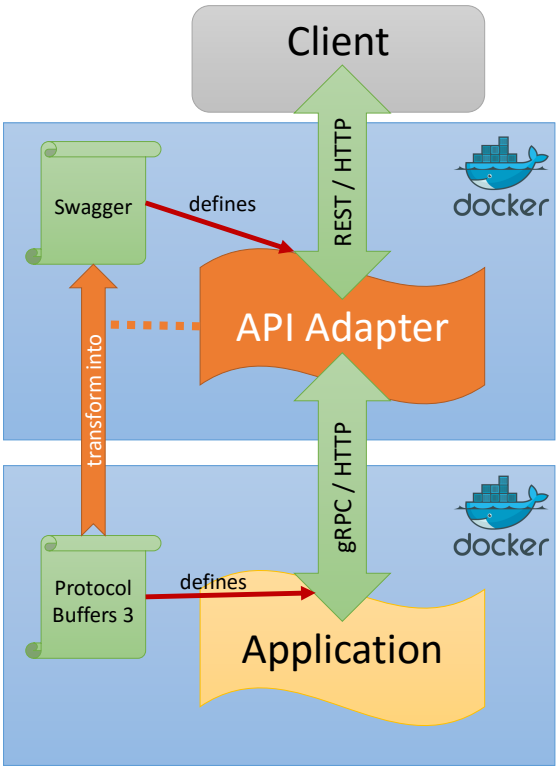

Abbildung 4.1: Überblick über die Funktionsweise des gRPC-REST API Adapters

In Kapitel 2.2.5 wurde das API Gateway Framework Kong vorgestellt. Dieses ermöglicht die Erweiterung einer REST API durch Plugins, die Funktionalitäten und Dienste anbieten. Es ist möglich den API Adapter mit Hilfe von Kong Plugins zu ergänzen. Dabei bleiben Skalierbarkeit, Mudularität und Systemunabhängigkeit erhalten. Die Erweiterung des Adapters durch Kong ermöglicht der Anwendung Funktionalitäten wie Logging, Authentifikation und Caching auch ohne Intermediate Adapter bereitzustellen.

5 Zusammenfassung und Ausblick

Diese Arbeit betrachtet verschiedene API-Konzepte und deren Entwicklung. Dabei wird zwischen Plugin/Library APIs sowie Endpunkt APIs unterschieden. Plugin/Library APIs sind Bibliotheken, die direkt in den Source Code eingebunden werden können. Der Entwickler einer Anwendung kann auf diese Bibliotheken über die bibliothekseigene API zugreifen und die verfügbaren Funktionalitäten verwenden. Einige Library APIs, wie z.B. die Java Util Library[1], liefern die Bibliotheken self-contained mit. Andere Bibliotheken verwenden für die Bereitstellung im Hintergrund dagegen selbst Endpunkt APIs. Bei einer Endpunkt API werden spezifische Endpunkte addressiert.

Im Besonderen wird in dieser Arbeit auf Remote Procedure Calls (RPC) und REST als Vertreter der Endpunkt APIs eingegangen. Eine REST API ist eine API, bei der als Endpunkte die Ressourcen einer Anwendung addressiert werden. Als Ressourcen werden oft JSON Objekte verwendet, die meist über HTTP als Protokoll vom Client angefragt, gelöscht oder verändert werden. REST besitzt insgesamt sechs Prinzipien, die bei der Implementierung der API erfüllt werden sollten: Uniform Interface, Stateless, Cachable, Client-Server, Layered System und Code on Demand. Diese sechs Prinzipien werden in Kapitel 2.1.2 genauer beschrieben.
In einer RPC API hingegen werden statt der Ressourcen die Methoden der Anwendung als Endpunkte zugänglich gemacht. Durch synchrone Aufrufe kann der Client auf die RPC Methoden der Anwendung zugreifen und über Nachrichten mit dieser kommunizieren. Die Nachrichten können in unterschiedlichen Formaten, wie Protocol Buffers und JSON repräsentiert sein.

Außer synchronen und asynchronen APIs existieren auch noch ereignisgesteuerte APIs. Hierbei werden durch verschiedenen Aktionen des Benutzers Ereignisse ausgelöst. Die Anwendung reagiert auf diese Ereignisse und führt die entsprechende Funktionalität aus.

Neben den verschiedenen API Konzepten wird auch der Microservice Architekturstil genauer betrachtet. Bei diesem bestehen die Anwendungen im Unterschied zu monolithischen Anwendungen nicht aus großen zusammenhängenden Blöcken, sondern aus mehreren kleinen Services. Sie kommunizieren untereinander über verschiedenene API Arten und stellen dem Anwender außerdem ausgewählte Funktionalitäten über eine eigene API bereit. Da jeder Service meist nur eine Business Capability implementiert, sind sie sehr leichtgewichtig einsetzbar und hochgradig skalierbar. Im Gegensatz zum Service Oriented Architecture (SOA) Stil müssen beim Microservice Architekturstil die einzelnen Services autark einsetzbar sein.

[1] https://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html

Eine Virtualisierung mittels Containern, wie Docker, ermöglicht unabhängig von der verwendeten Plattform eine vollautomatische Bereitstellung der Microservices. Durch die Verwendung von Docker Containern entfällt die bei herkömmlichen virtuellen Maschinen (VM) notwendige Schicht zwischen Host-Betriebsystem und der Anwendung. Sie greifen dabei gemeinsam auf den Kernel des Host-Betriebsystems zu. Das ist wesentlich ressourcensparender als VMs, da die von mehreren Docker Containern genutzten Teile des Betriebsystems nur einmal gelesen werden müssen. Die Lösung mit Containern hat jedoch einige Sicherheitsrisiken, da Container beim Starten Root Rechte benötigen. Mit diesen sind sie in der Lage, System Ressourcen mit anderen Containern zu teilen und weitere Container zu starten.

Bei der Entwicklung von Microservices muss besonders auf die APIs der Services geachtet werden, da sie die einzige Zugriffsmöglichkeit für den Nutzer oder andere Services bieten. Um den unterschiedlichen Anforderungen der Kommunikationspartner gerecht zu werden, sollte ein Adapter verschiedene APIs bereitstellen. Dadurch können die Services effizient über Messaging oder RPC kommunizieren, während der Benutzer des Systems beispielsweise die REST API des Services nutzt.

Durch die automatische Generierung von APIs lässt sich der Arbeitsaufwand von Entwicklern verringern. Dabei bilden wiederverwendbare API Adapter APIs auf äquivalente APIs anderer Arten ab. Zusätzliche Intermediate Adapter bilden eine optionale Zwischenschicht zwischen Services und API Adaptern. Ein Intermediate Adapter bekommt die Schnittstelle der unterliegenden Schicht (Service oder weiterer Intermediate Adapter) und modifiziert diese vor der Abbildung durch den API Adapter.

Das Framework CLARA (Compose Lovely APIs based on Reusable API Adapters)[Spe17] stellt API Adapter zur Verfügung. Damit CLARA mit Services verwendet werden kann, müssen diese einige Rahmenbedingungen erfüllen. Sie müssen mit gRPC kommunizieren und in Docker Container virtualisiert werden. gRPC ist eine performante und Standard-basierte RPC Implementierung, die in verschiedenen Programmiersprachen zur Verfügung steht. Neben den üblichen RPC Funktionen ermöglicht gRPC die Übertragung von Request und Response Nachrichten als Streams. Die Intermediate Adapter kommunizieren entsprechend alle über gRPC APIs mit den Services und den API Adaptern. Um die gRPC APIs einlesen zu können, muss ein Service seine Schnittstelle mit der Schnittstellendefinitionssprache Protocol Buffers definieren und den Adaptern bereitstellen. In einer Protocol Buffers Datei werden die erlaubten Messages und die Definitionen der RPC Operationen festgelegt. Die Adapter lesen die Protocol Buffers Dateien ein und können anschließend eine modifizierte bzw. abgebildete API erzeugen.

Ein spezifischer gRPC-REST API Adapter aus dem CLARA Framework wird in dieser Arbeit genauer betrachtet. Beim Starten des Adapters bindet dieser dynamisch zur Laufzeit eine Protocol Buffers Datei ein und extrahiert die für die Abbildung auf eine REST API notwendigen Daten. Anschließend generiert der Adapter daraus gemäß einem festgelegten Schema eine Schnittstellendefinition in Swagger. Dies ist eine Definitionssprache für REST über HTTP APIs, die auf JSON basiert. In der Definition der REST API werden die erlaubten Pfade, Parameter und erwarteten Rückgaben der HTTP Verben festgelegt. Auch die Ressourcendefinition sind in der

Swagger Datei enthalten. Mithilfe der Swagger Datei wird durch den Adapter die entsprechende REST API generiert.

Ausblick

Das CLARA Framework kann zukünftig mit zusätzlichen API Adapter erweitert werden, um eine noch größere API Vielfalt zu erhalten. Außerdem sollten weitere Funktionalitäten, wie Logging oder Authentifizierung als Intermediate Adapter bereitgestellt werden.

Neben der in dieser Arbeit vorgestellten Abbildung von gRPC APIs auf andere API Typen, wie REST, könnte das Framework auch dahingehend erweitert werden, dass es die Umwandlung anderer API Arten unterstützt. Möglich wäre ein Adapter, der eine Swagger Schnittstellendefinition konsumiert und eine gRPC API erzeugt. Dies ermöglicht den Entwicklern die Bereitstellung eines Services in mehreren möglichen API Arten, basierend auf einer einzelnen beliebigen Implementierung.

Literaturverzeichnis

[BD10] R. Bruns, J. Dunkel. *Event-driven architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Springer-Verlag, 2010 (zitiert auf S. 19).

[BKEI05] O. Ben-Kiki, C. Evans, B. Ingerson. „YAML Ain't Markup Language (YAML™) Version 1.1". In: *yaml.org, Tech. Rep* (2005) (zitiert auf S. 23).

[BTP15] M. Belshe, M. Thomson, R. Peon. „Hypertext transfer protocol version 2 (http/2)". In: (2015) (zitiert auf S. 51).

[Bra+] T Bray, J Paoli, C. Sperberg-McQueen, Y Mailer, F Yergeau. *Extensible Markup Language (XML) 1.0 5th Edition, W3C recommendation, November 2008* (zitiert auf S. 23).

[Bra14] T. Bray. „The javascript object notation (json) data interchange format". In: (2014) (zitiert auf S. 23).

[Cha04] D. Chappell. *Enterprise service bus*. "O'Reilly Media, Inc.", 2004 (zitiert auf S. 27).

[FL15] M. Fowler, J. Lewis. „Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr". In: *Objektspektrum* 1.2015 (2015), S. 14–20 (zitiert auf S. 13, 25, 45).

[FLW16] S. W. Frank Leymann Christoph Fehling, J. Wettinger. „Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point!" In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*. SciTePress, 2016, S. 7–15 (zitiert auf S. 13, 25).

[Fow17] M. Fowler. *Microservices Resource Guide*. 2017. URL: http://martinfowler.com/microservices (zitiert auf S. 25–27).

[Goo17a] Google. *Protocol Buffers 3*. 2017. URL: https://developers.google.com/protocol-buffers/docs/proto3 (zitiert auf S. 14, 33, 34, 51).

[Goo17b] Google. *gRPC*. 2017. URL: http://www.grpc.io/docs/ (zitiert auf S. 36, 51).

[Gro+12] J.-R. W. Group et al. *Json-rpc 2.0 specification*. 2012. URL: http://www.jsonrpc.org/specification (zitiert auf S. 23).

[HF10] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010 (zitiert auf S. 26).

[Hic09] I. Hickson. „Server-sent events“. In: *W3C Working Draft WD-eventsource-20091222, latest version available at< http://www. w3. org/TR/eventsource* (2009) (zitiert auf S. 18).

[Inc17] D. Inc. *Docker*. 2017. URL: https://www.docker.com/what-docker (zitiert auf S. 40, 41).

[JP15] O. T. Jerry Preissler. *Docker - perfekte Verpackung von Microservices*. 2015. URL: https://www.sigs-datacom.de/uploads/tx_dmjournals/preissler_tigges_OTS_Architekturen_15.pdf (zitiert auf S. 13).

[LL09] K. B. Laskey, K. Laskey. „Service oriented architecture“. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 1.1 (2009), S. 101–105 (zitiert auf S. 27).

[Mar06] J.-L. Maréchaux. „Combining service-oriented architecture and event-driven architecture using an enterprise service bus“. In: *IBM Developer Works* (2006), S. 1269–1275 (zitiert auf S. 27).

[Mas11] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc.", 2011 (zitiert auf S. 16, 20).

[Mas17] Mashape. *Kong*. 2017. URL: https://getkong.org/ (zitiert auf S. 37–39).

[New15] S. Newman. *Building microservices*. "O'Reilly Media, Inc.", 2015 (zitiert auf S. 25).

[Nur+09] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta. „Comparison of JSON and XML Data Interchange Formats: A Case Study.“ In: *Caine* 2009 (2009), S. 157–162 (zitiert auf S. 29, 30).

[Rad14] S. E. Radio. *Episode 213: James Lewis on Microservices*. 2014. URL: http://www.se-radio.net/2014/10/episode-213-james-lewis-on-microservices/ (zitiert auf S. 27).

[Ric17] C. Richardson. *Pattern: API Gateway / Backend for Front-End*. 2017. URL: http://microservices.io/patterns/apigateway.html (zitiert auf S. 37, 40).

[SAK07] M. Slee, A. Agarwal, M. Kwiatkowski. „Thrift: Scalable cross-language services implementation“. In: *Facebook White Paper* 5.8 (2007) (zitiert auf S. 23).

[Spe17] S. Speth. *CLARA*. 2017. URL: https://github.com/spethso/CLARA (zitiert auf S. 51, 68).

[Swa17] Swagger. *Swagger Codgen*. 2017. URL: https://github.com/swagger-api/swagger-codegen (zitiert auf S. 30, 46).

[Tea14] S. Team. *Swagger restful api documentation specification 1.2*. Techn. Ber. Technical report, Wordnik, 2014. URL: https://github.com/wordnik/swagger-spec/blob/master/versions/1.2.md (zitiert auf S. 14, 30).

[W3C17a] W3C. *HTTP Specification - Method Definitions*. 2017. URL: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html (zitiert auf S. 20, 42).

[W3C17b] W3C. *HTTP Specification*. 2017. URL: https://www.w3.org/Protocols/rfc2616/rfc2616.txt (zitiert auf S. 16, 42).

[Wil17] J. Wilder. *Dockerize Projekt*. 2017. URL: https://github.com/jwilder/dockerize (zitiert auf S. 54).

[grp15] grpc.io. *NodeJS gRPC*. 2015. URL: https://github.com/grpc/grpc/tree/master/src/node (zitiert auf S. 56).

[org17] json-schema org. *JSON Schema*. 2017. URL: http://json-schema.org/ (zitiert auf S. 28, 30).

Alle URLs wurden zuletzt am 01. 03. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift