HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Event-Driven Microservices – eine Antwort auf gestiegene Integrationsbedarfe fortschreitend verteilter Systeme?

Manuel Ottlik

Master-Arbeit im Studiengang „Angewandte Informatik“

25. Oktober 2021

Autor Manuel Ottlik
1579674
manuel.ottlik@web.de

Erstprüfer: Prof. Dr. Ralf Bruns
Abteilung Informatik, Fakultät IV
Hochschule Hannover
ralf.bruns@hs-hannover.de

Zweitprüfer: Prof. Dr. Jürgen Dunkel
Abteilung Informatik, Fakultät IV
Hochschule Hannover
juergen.dunkel@hs-hannover.de

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Master-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 25. Oktober 2021 Unterschrift

Abstract

Zusammen mit der Microservice-Bewegung werden immer häufiger synchrone Request-Response-Schnittstellen nach dem REST-Paradigma entwickelt, um Service-Landschaften zu integrieren. Die Einfachheit des Paradigmas verleitet viele Organisationen, nahezu die komplette Interprozesskommunikation ihres Ökosystems über diese Art von Schnittstelle abzuwickeln – nicht ohne Konsequenzen.

Diese Arbeit entwickelt Ansätze, wie die Integrationsprobleme, die bei übermäßiger Verwendung von REST entstehen, mithilfe von Event-Driven Architecture gelöst werden können, ohne den Status quo dieser Organisationen außer Acht zu lassen. Dafür werden der gegenwärtige Zustand der Integrationsmuster und eingesetzten Infrastruktur von Event-Driven Architecture kritisiert und Kriterien erarbeitet, die pragmatische und zugängliche Integrationsansätze erfüllen müssen. Um die Einführungskosten gering zu halten, wird eine Middleware entwickelt, die in bestehende REST-Schnittstellen eingesetzt werden kann und auf Basis der API-Aufrufe Events generiert. Darauf aufbauend werden vier Integrationsmuster entwickelt, die eine schrittweise Transformation zu Event-Driven Microservices ermöglichen. Um die Zugänglichkeit der Eventing-Infrastruktur zu erhöhen, wird außerdem wird die Standardisierung der Event-Struktur durch die CloudEvents-Spezifikation vorgeschlagen. Um die Zugänglichkeit weiter zu erhöhen, erfolgt die Kommunikation der Services nicht direkt mit dem Event-Broker, sondern über Proxies, die die Events per HTTP annehmen oder ausspielen. Um die Transparenz über den Datenfluss im System zu wahren, werden alle Produzenten und Konsumenten werden mitsamt ihrer Events durch den Beschreibungsstandard AsyncAPI dokumentiert.

Nach einer Validierung dieser Ansätze mithilfe eines Prototyps kommt diese Arbeit zu der Erkenntnis, dass der Einsatz der entwickelten Middleware für alle Organisationen sinnvoll ist, die bereits viele REST-Schnittstellen im Einsatz haben. Die Standardisierung der Event-Struktur und des Event-Protokolls mittels CloudEvents und HTTP-Proxies sowie die Dokumentation durch AsyncAPI empfiehlt sich auch unabhängig des Status quo für alle Organisationen, die Event-Driven Microservices entwickeln möchten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Programmcodeverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
AMQP	Advanced Messaging Queueing Protocol
CDC-Pattern	Change-Data-Capture-Pattern
CNCF	Cloud Native Computing Foundation
CQRS	Command-Query-Responsibility-Segregation
CRUD	Create, Read, Update, Delete
EDA	Event-Driven Architecture
FaaS	Functions-as-a-Service
HTTP	Hypertext Transfer Protocol
IDL	Interface Description Language
JSON	JavaScript Object Notation
MQTT	Message Queueing Telemetry Transport
ORM	Object-Relational Mapping
REST	Representational State Transfer
URL	Unique Resource Locator
VM	virtuelle Maschine
YAML	Yet Another Markup Language

1 Einleitung

> Synchronous communication is the crystal meth of distributed programming. It's easy and feels good, but it's very bad for you in the long run.
>
> *Martin Thompson*
> *Todd Montgommery*
> [TM21]

1.1 Motivation

In den meisten Unternehmen gibt es bei Geschäftsprozessen nur eine Richtung: mehr digitale Prozesse, weniger Papier. Das führt dazu, dass Software nicht nur mehr bei Prozessen unterstützt, sondern diese komplett abbildet. Dafür braucht es Software, die flexibel auf Änderungen reagieren kann, um die Agilität der Geschäftswelt zu wahren. Software wird deshalb in verteilten Systemen konzipiert, in denen Geschäftsprozesse durch das Zusammenspiel einzelner, austauschbarer Services abgebildet werden. Dabei nimmt der Grad der Verteilung dieser Systeme zu und die Services werden immer kleiner. So wurden in den letzten Jahrzehnten Software-Monolithen in serviceorienterte Architekturen überführt, aus welchen im Zuge der Provisionierung von Software in Cloud-Umgebungen Microservice-Landschaften gebildet werden. Wenn ein Geschäftsprozess nicht mehr innerhalb eines Service abgewickelt werden kann, sondern nur über mehrere Services hinweg, steigt der Bedarf an Interprozesskommunikation – der Kommunikation zwischen Services. Das gibt der Service-Integration auf Infrastrukturebene eine höhere Bedeutung als in Systemen, in denen ein Geschäftsprozess durch einen Service abgebildet wird und schafft somit den Bedarf für standardisierte Service-Schnittstellen.

Diese Transformation findet auch in großen Unternehmen statt, die eine historisch gewachsene und heterogene Systemlandschaft zu bewältigen haben. Sie beginnen, einige Bestandsanwendungen schrittweise in Ansammlungen von Microservices zu transformieren – diese Microservices kommunizieren dann über die standardisierten Schnittstellen

miteinander, um Geschäftsprozesse abzubilden. Für diese Schnittstellen werden oft synchrone Schnittstellen auf Basis von Hypertext Transfer Protocol (HTTP) gewählt, die dem ressourcenorientierten Representational State Transfer (REST)-Paradigma folgen, weil es sehr verbreitet und einfach zu implementieren ist.

Mit der fortschreitenden Transformation ihrer Systemlandschaft zu Microservices sehen sich diese Unternehmen mit einer Schnittstellenlandschaft konfrontiert, die von ressourcenorientierten, synchronen REST-Schnittstellen dominiert wird. Wenn die Integration von Services jedoch ausschließlich über REST erfolgt, entstehen Probleme.

1.2 Ziel der Arbeit

Diese Integrationsprobleme sollen in dieser Arbeit aufgezeigt und gelöst werden. Dafür werden unterschiedliche Lösungsansätze betrachtet, die in der Literatur Erwähnung finden. Unter anderem wird ein komplett gegenteiliger Interaktionsstil zu REST in Betracht gezogen: Event-Driven Architectures (EDAs) setzen statt synchroner Verarbeitung auf Asynchronität und auf Ereignisorientierung statt ressourcenorientierter Schnittstellen.

Dabei soll in dieser Arbeit insbesondere der Status quo der zuvor beschriebenen Unternehmen berücksichtigt werden – die Interaktionsstile werden dementsprechend nicht allein basierend auf dem Einsatz für Neuentwicklungen bewertet. Deshalb werden pragmatische und realistisch umsetzbare Ansätze für durch REST dominierte Systemlandschaften erarbeitet, in die EDA als zusätzliche Möglichkeit der Integration etabliert werden soll, um die entstandenen Integrationsprobleme zu lösen. Die Ansätze bieten Möglichkeiten, wie REST und EDA zusammenwirken können, um eine schrittweise Transformation zu ermöglichen, ohne durch den Einsatz von EDA den ausschlaggebenden Vorteil von REST zu gefährden: Einfachheit.

Mithilfe dieser Ansätze wird in dieser Arbeit die Frage beantwortet, wie EDA sinnvoll in diese Ökosysteme eingeführt werden kann und inwiefern die Erweiterung von Microservices durch EDA zu Event-Driven Microservices die Antwort auf die entstandenen Integrationsprobleme dieser Unternehmen und ihrer Schnittstellenlandschaften sind.

1.3 Inhaltsüberblick

Als Grundlage für die Beantwortung dieser Frage werden in Kapitel 2 die Gründe für die Transformation zu Microservices erläutert und dargelegt, welche Eigenschaften Microservices ausmachen. Außerdem werden die theoretischen Grundlagen zum REST-Paradigma sowie der Container-Technologie gelegt – die Prämissen für die Systemlandschaften, wie sie in dieser Arbeit behandelt werden.

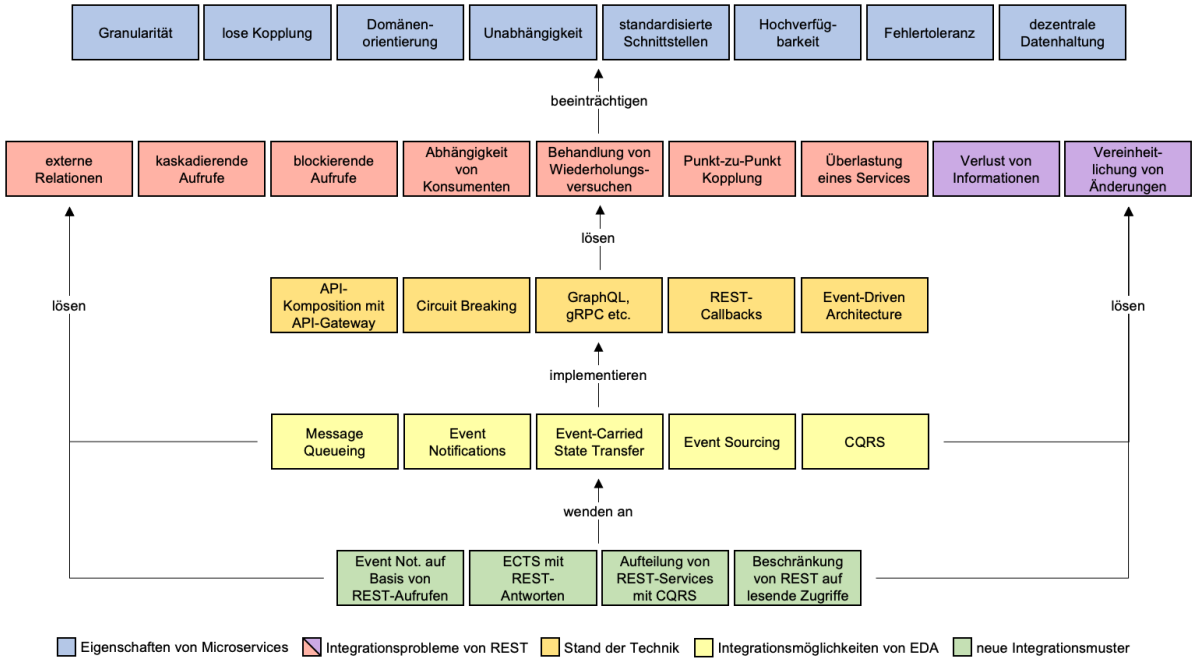

Abbildung 1.1: Strukturierung der Arbeit in fünf Ebenen.

In Kapitel 3 werden dann die Integrationsprobleme erläutert, die bei der Dominanz von synchronen REST-Schnittstellen entstehen. Abbildung 1.1 visualisiert den Aufbau der Arbeit und die Zusammenhänge ihrer Kapitel: Die rot bzw. lila hinterlegten Integrationsprobleme von REST beeinträchtigen die blau hinterlegten Eigenschaften von Microservices. Im weiteren Verlauf des dritten Kapitels wird deshalb der Stand der Technik für diese Probleme vorgestellt, in Abbildung 1.1 tiefgelb hinterlegt.

Aus der Gegenüberstellung dieser Lösungsmöglichkeiten stellt sich EDA als besonders aussichtsreich heraus. In Kapitel 4 werden deshalb die theoretische Grundlage für EDA als Architekturstil gelegt und Integrationsmöglichkeiten präsentiert, die Konzepte von EDA implementieren. Diese Integrationsmöglichkeiten sind in Abbildung 1.1 hellgelb hinterlegt und lösen einzelne Integrationsprobleme, wodurch indirekt der Beeinträchtigung der blau hinterlegten Eigenschaften entgegengewirkt wird. Gegen Ende von Kapitel 4 folgt dann eine kritische Bewertung dieser Integrationsmöglichkeiten. Auf Basis dieser Bewertung werden in Kapitel 5 Kriterien für Integrationsansätze erarbeitet, deren Bestandteil auch neue Integrationsmuster (in Abbildung 1.1 grün hinterlegt) sind, welche die zuvor eingeführten Integrationsmöglichkeiten von EDA anwenden.

Über Abbildung 1.1 hinaus werden in der kritischen Bewertung und den darauf aufbauenden Kriterien für Integrationsansätze auch Anforderungen an die Eventing-Infrastruktur gestellt. In Kapitel 6 werden deshalb Ansätze vorgestellt, die diesen Anforderungen ge-

recht werden sollen. Um die Wirksamkeit der Ansätze aus Kapitel 5 und 6 zu bewerten, werden diese in Kapitel 7 in einem Prototyp zu einer zusammenhängenden Lösung kombiniert. Anhand eines beispielhaften Szenarios und konkreter Implementierungen wird dann eine Beurteilung der Ansätze anhand der zuvor eingeführten Kriterien vorgenommen. Abschließend werden in Kapitel 8 das Ergebnis dieser Arbeit zusammengefasst und der resultierende Handlungsbedarf ermittelt, um schlussendlich einen Ausblick über weitergehende Handlungsfelder zu geben.

2 Konzeptionelle Grundlagen von Microservices

Dieses Kapitel legt die theoretischen Grundlagen für die Ausgangssituation dieser Arbeit und beschreibt die Gründe für eine Transformation zu Microservices sowie die damit verbundene Container-Technologie. Ebenso werden die Grundlagen des REST-Paradigmas als Schnittstellentechnologie und außerdem die Eigenschaften von Microservices eingeführt (Abbildung 1.1, blaue Ebene). Sie sind die Grundlage für diese Arbeit, denn sie werden durch die Integrationsprobleme beeinträchtigt und sollen durch die Ansätze, die in dieser Arbeit vorgestellt werden, wieder zum Tragen kommen.

2.1 Transformation monolitischer Strukturen zu Microservices

Microservices gelten als Industriestandard moderner, verteilter Software-Architektur und beschreiben ein Architekturmuster, in dem eine Applikation nicht in einem Stück, sondern als Summe mehrerer kleiner Services implementiert wird. Ein Service wird dabei als einzeln ausführbare und auslieferbare Einheit bezeichnet [Ric19]. Die Services werden nach diesem Muster so klein gewählt, dass sie sich auf nur eine Verantwortlichkeit fokussieren [CS16]. Geschäftsprozesse werden dann nicht mehr innerhalb eines einzelnen Services, sondern nur über mehrere Microservices abgebildet. Sie bilden in der Entwicklung der Modularisierung von Software damit nach (modularisierten) Monolithen und Web Services aus der Service-oriented Architecture eine der granularsten Aufteilungen in den verteilten Systemen. Nur Functions-as-a-Service (FaaS), also die Aufteilung von Geschäftslogik in Auslieferungseinheiten, die jeweils nur eine einzelne Funktion enthalten, setzt auf eine granularere Aufteilung von Geschäftslogik [CS16]. Auf sie soll deshalb im Ausblick in Kapitel 8.3 erneut eingegangen werden.

Welche Beweggründe dafür sorgen, dass Organisationen ihre Systeme zunehmend auf kleinere Services verteilen und monolithisch geprägte Services wie in Abbildung 2.1 in mehrere Microservices aufteilen, wird im folgenden Kapitel erläutert.

2.1.1 Gründe für die Transformation

Monolithisch geprägte Applikationen, also Services, die als eine Einheit designt, entwickelt und ausgeliefert werden, bringen zentrale Vorteile mit sich. Weil sich die komplette Logik eines Geschäftsprozesses in einem Service abspielt, sind größer geschnittene Services wesentlich einfacher und schneller zu entwickeln. Es ist wesentlich weniger Interprozesskommunikation zwischen Services nötig, was einen geringeren Bedarf an Kommunikationskomponenten in der Infrastruktur nach sich zieht und die Auslieferung vereinfacht. Monolithisch geprägte Services können somit mit einer geringeren Komplexität auf Infrastrukturebene punkten [ZRV+19]. Die Gründe für die Transformation zu Microservices, wie sie in Abbildung 2.1 dargestellt ist, sind jedoch häufig externe, sich verändernde Anforderungen, denen Landschaften, die keine Microservices einsetzen, nicht mehr gerecht werden können [Fam15].

Eine der bedeutendsten externen Veränderungen der Anforderungen an Service-Landschaften ist die hohe und variable Last, denen Systeme ausgesetzt sein können. Da mehr Geschäftsprozesse und größere Anteile dieser Prozesse mithilfe von Software umgesetzt werden und die Geschäftsprozesse je nach Fachdomäne zu unterschiedlichen Tages- oder Jahreszeiten unterschiedlich stark verwendet werden, müssen Services in der Lage sein, unerwartete Lastspitzen dynamisch abzufangen, ohne in Ruhezeiten große Betriebskosten zu verursachen. Diese dynamische Skalierbarkeit ist mit Monolithen schwer umzusetzen. Diese sind oft nur vertikal skalierbar und unterliegen somit der natürlichen Begrenzung, wie viel Rechenkapazität auf der Maschine bereitgestellt werden kann. Ist der Service auch horizontal skalierbar, muss die die gesamte Anwendung skaliert werden – mitsamt der Bereiche, die keiner Lastspitze ausgesetzt sind. Die Bereitstellung muss außerdem innerhalb weniger Sekunden erfolgen, um die Verfügbarkeit des Geschäftsprozesses nicht zu gefährden [CS16]. Für die Skalierbarkeit stellt die Abbildung der Geschäftslogik in einem einzelnen Prozess also eine Schwäche dar. Eine weitere Folge dieser Schwäche ist der eingeschränkte Technologie-Stack: Wenn der bestehende Service erweitert wird, müssen neue Funktionen mit dem Stack umgesetzt werden, der in der Entwurfsphase ausgewählt wurde. Ab einem gewissen Punkt im Lebenszyklus wird neue Funktionalität dementsprechend mit veralteter und evtl. für den Applikationsfall ungeeigneter Technologie umgesetzt. Upgrades auf eine neue Version der Technologie stellen abhängig von der Größe der Applikation meist einen erheblichen Aufwand dar [SI18]. Auch das Testen und die Auslieferung gestaltet sich bei großen Applikationen komplexer, weil für jede kleine Änderung alle Tests durchlaufen werden und die komplette Applikation neu ausgeliefert wird. Tests beanspruchen aufgrund des Umfangs mehr Zeit und sind unter Umständen von unerwarteten Seiteneinflüssen betroffen. Nichtsdestotrotz ist Testen in grob geschnittenen Services unverzichtbar: Weil die gesamte Applikation in einem Prozess läuft, kann bereits ein Speicherleck zur Nichtverfügbarkeit mehrerer Funktionen führen [Ric19].

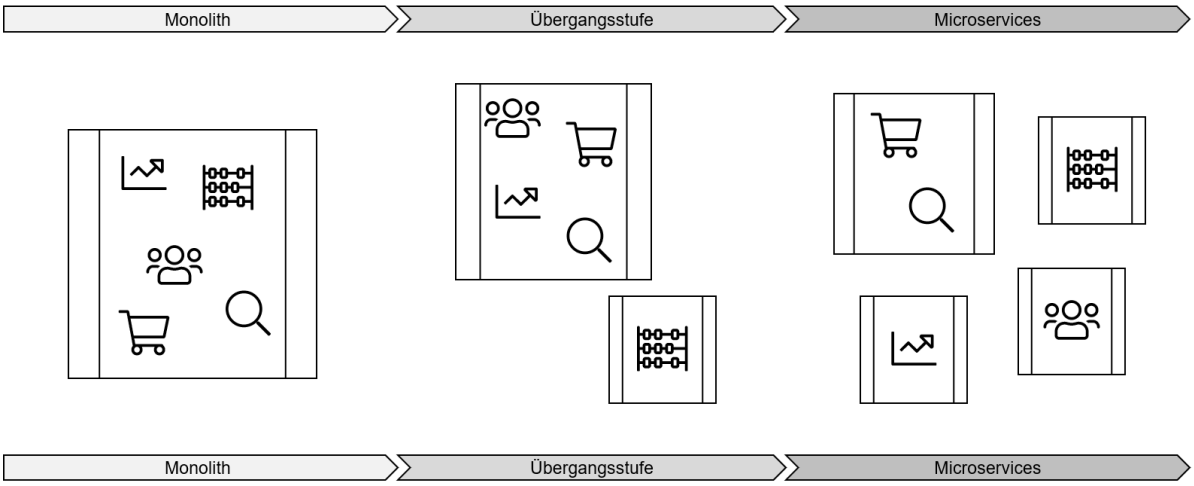

Abbildung 2.1: Schrittweise Transformation monolithischer Anwendungen zu Microservice-Landschaften.

Über die Beweggründe, die hauptsächlich auf die Größe des Services zurückzuführen sind, werden in der Literatur zusätzlich Gründe angeführt, die besonders dann ins Gewicht fallen, wenn umfangreiche Anwendungen nach ihrer initialen Entwicklung unsauber erweitert werden und die Prinzipien guter Software-Entwicklung verletzen. Da diese Beweggründe in Organisationen mit historisch gewachsenen Systemlandschaften ebenso von Bedeutung sind, werden sie in den folgenden Sätzen kurz vorgestellt.

So kann die schrittweise Erweiterung zu einer hohen Komplexität führen, in der ein einzelner Entwickelnder nicht mehr in der Lage ist, die komplette Applikation zu überblicken. Daraus ergeben sich mehrere Hindernisse, z.B. dass diese Person die Konsequenzen einer Änderung nicht mehr vollständig überblicken kann. Die Komplexität wirkt sich aber auch auf die Größe des / der Teams aus, die an der Applikation arbeiten [Ric19]. Wenn Module außerdem nicht mit klaren Schnittstellen voneinander getrennt sind, können einzelne Teams Änderungen nicht eigenständig vornehmen, sondern müssen immer im Austausch mit den üblichen Teams stehen – das verringert die Entwicklungsgeschwindigkeit. Ist die Applikation zusätzlich nach einer N-Schichten-Architektur designt und verwendet deshalb zentral verwaltete Datenbanken und Infrastruktur, bestehen außerdem Abhängigkeiten zu anderen Einheiten der Organisation [CS16].

Je nach Organisation und dem zugrundeliegendem Ökosystem fallen die vorgestellten Beweggründe unterschiedlich ins Gewicht. Auch Microservices können keine gute Softwarearchitektur erzwingen – sie bringen jedoch Eigenschaften mit sich, die auf konzeptueller Ebene besser auf die externen Anforderungen zugeschnitten sind. Diese Eigenschaften sollen im folgenden Kapitel vorgestellt werden.

2.1.2 Definierende Eigenschaften von Microservices

Der Gartner Glossar definiert die Eigenschaften eines Microservices wie folgt: „A microservice is a service-oriented application component that is tightly scoped, strongly encapsulated, loosely coupled, independently deployable and independently scalable“ [Gar21]. Über diese genannten Eigenschaften hinaus finden sich in der Literatur weitere Eigenschaften, von denen auf den folgenden Seiten die herausgestellt werden, die am häufigsten und prominentesten erwähnt werden. Alle vorgestellten Eigenschaften werden ab diesem Kapitel immer dann kursiv geschrieben, wenn sie direkt auf die folgenden Erläuterungen verweisen.

Granularität

Eine der offensichtlichsten und gleichzeitig unschärfsten Eigenschaften von Microservices ist die Größe eines einzelnen Services. Neben der berühmten „Zwei-Pizza-Regel“ lässt sich die Größe am besten im Kontrast zu monolithischen Applikationen definieren: Eine einzelne Person sollte in der Lage sein, die komplette Geschäftslogik des Services zu verstehen [Ric19]. Darüber hinaus sollte sich dieser Service auf eine Verantwortlichkeit fokussieren, also wie von Gartner definiert „tightly scoped“ sein. Bei der Zerlegung von Monolithen in mehrere Microservices werden dem „Seperation of Concerns“-Prinzip folgend stark verwandte Aufgaben in einem Service gehalten, um eine hohe Kohäsion zu gewährleisten. Je besser diese Aufteilung gelingt, desto einfacher kann *lose Kopplung* erreicht werden [New15].

Lose Kopplung

Als ewiger Antagonist zur hohen Kohäsion des Softwaredesigns steht die von Gartner genannte lose Kopplung von modularisierter Software. Kopplung kann bei Microservices laut Sam Newman in „Monolith to Microservices“ [New19] insbesondere in vier Formen auftreten: in der Implementierung, der Domäne, der Auslieferung und zeitlich.

Zeitliche Kopplung entsteht immer dann, wenn ein Service A auf einen anderen Service B warten muss, um eine Aufgabe zu erledigen. Kopplung bei der Auslieferung beschreibt einen Zustand, in dem mehrere Services nur noch gemeinsam ausgeliefert werden können, zum Beispiel, weil ein Service A die Adresse von Service B kennen muss, um mit ihm Daten auszutauschen. Kopplung in der Implementierung hingegen beschreibt den Zustand, dass sich das Verhalten von einem Service A ändert, wenn sich das Verhalten von Service B ändert. Das kann bspw. durch geteilte Datenbanken passieren. Diese Kopplung soll insbesondere durch *standardisierte Schnittstellen* vermieden werden.

Die natürlichste Form der Kopplung findet auf Ebene der Domäne statt: Wenn mehrere Services einen Prozess abbilden, sind sie durch die Domäne des Prozesses gekoppelt. Die Art der Integration, die *Domänenorientierung* und *Fokussierung* des Services können diese Kopplung jedoch reduzieren.

Domänenorientierung

Aufbauend auf der *Granularität* ist das Ziel eines jeden Services, einen fachlichen Zweck zu erfüllen – keinen technischen. Architektur- und Design-Entscheidungen müssen sich an den Anforderungen der Domäne orientieren, die Implementierung der Geschäftslogik fachlich statt technisch getrieben sein. Um die Größe eines Services zu begrenzen, sollte ein Service außerdem möglichst wenig Logik enthalten, die nicht zu seinem ursprünglichen, domänenorientierten Zweck beiträgt [FL14].

Unabhängigkeit

Neben der organisatorischen Unabhängigkeit des Teams, das für einen Microservices verantwortlich ist, müssen Services auch in mehreren technischen Aspekten unabhängig voneinander sein. So sollte ein Service unabhängig und isoliert testbar, auslieferbar und skalierbar sein (siehe Definition von Gartner). Dafür sind Eigenschaften wie *dezentrale Datenhaltung* und *standardisierte Schnittstellen* nötig. Die *Unabhängigkeit* hängt immer mit dem Grad an *loser Kopplung* und somit schlussendlich auch der *Granularität* zusammen [Fam15]. Um die *Unabhängigkeit* und *Skalierbarkeit* zu gewährleisten, dürfen Services außerdem keinen Zustand in ihrer Prozessinstanz speichern. Das bezieht sich einerseits auf die Geschäftslogik, aber auch auf eher technische Aspekte des Services wie bspw. die Authentifizierung und Autorisierung. Als Ergebnis dieser Zustandslosigkeit können die Services beliebig wiederverwendet werden [Chr19].

Standardisierte Schnittstellen

Um Daten zwischen Microservices auszutauschen, sollen ausschließlich standardisierte, klar definierte Schnittstellen verwendet werden. Diese Schnittstellen sind unabhängig von der Implementierung und gewähren dementsprechend Technologieunabängigkeit durch die freie Wahl der Programmiersprache, bspw. abhängig von der Eignung für den Anwendungsfall. Durch die definierte Schnittstelle wird der Service außerdem austauschbar [SI18]. Darüber hinaus sollten sie dem Prinzip „Smart Endpoints and Dumb Pipes“ (intelligente Endpunkte und dumme Verbindungen) folgen, um die Geschäftslogik innerhalb des Services zu halten und um die hohe Kohäsion sowie *Fokussierung der Verantwortlichkeit* des Services zu gewährleisten [FL14].

Hochverfügbarkeit durch horizontale Skalierbarkeit

Als Resultat der *Unabhängigkeit* und der *Zustandslosigkeit* können Microservices nicht nur vertikal durch die Erhöhung der Rechenkapazität, sondern auch horizontal, also durch das Betreiben des Services in mehreren Instanzen skaliert werden. Diese Skalierbarkeit über mehrere dedizierte Maschinen ermöglicht die hohe Verfügbarkeit des Services, weil der Ausfall einer einzelnen Maschine nur eine der vielen Instanzen des Services betrifft [Ric19].

Fehlertoleranz

Auch bei einer hohen Verfügbarkeit eines Gesamtsystems kann bei Tausenden Instanzen von Microservices davon ausgegangen werden, dass immer mindestens eine Instanz fehlerhaft oder ausgefallen ist. Der Ausfall einer Instanz ist in Microservice-Umgebungen somit ein erwartbares Verhalten und Teil der Auslieferung und des Betriebs von Services. Damit sich ein Fehler nicht auf das gesamte System ausbreitet, müssen einzelne Services tolerant mit diesem Fehlverhalten umgehen [Ric19].

Dezentrale Datenhaltung

Um die *Unabhängigkeit* von Services zu stärken, sollten die Daten, die von einem Service persistiert werden, nicht in einer gemeinsamen, sondern in einer ausschließlich für den Service zugänglichen Datenbank gespeichert werden. Die Datenobjekte, die ein Service verwaltet, sind dann nur noch über die *standardisierten Schnittstellen* des Services verfügbar. Daraus ergibt sich eine freie Wahl beim Speichermedium, was flexiblen und dem Anwendungsfall angemessenen Einsatz von Technologien, also Technologieunabhängigkeit, erlaubt [SI18].

Diese Eigenschaften stellen die Basis für die folgende Arbeit dar. Bevor in Kapitel 3 die Probleme eingeführt werden, die diese Eigenschaften bedrohen, soll auf den nächsten Seiten die theoretische Grundlage für den Betrieb von Microservices in Containern und das Schnittstellenparadigma REST gelegt werden.

2.2 Betrieb von Services in Container-Infrastruktur

Mit der Transformation von bestehenden Applikationen zu kleiner geschnittenen Microservices muss auch eine Transformation des Betriebs dieser Services stattfinden, da die Anzahl der zu betreibenden Services mit der Aufspaltung signifikant steigt. Als Industriestandard hat sich deshalb eine neue Form der Virtualisierung in sogenannten Containern etabliert, die gemeinsam mit der Orchestrierung dieser Container zu einer Systemlandschaft in den beiden folgenden Kapiteln eingeführt wird.

2.2.1 Container-Technologie

Der klassische und herkömmliche Weg, Applikationen zu betreiben, findet direkt auf einer physischen Maschine statt. Diese ist mit bestimmter Hardware ausgestattet und hat ein Betriebssystem installiert, auf dem zusätzlich Abhängigkeiten wie bspw. Java installiert werden, um anschließend die Applikation zu installieren und betreiben zu können. Sobald diese Maschine skaliert werden soll oder eine zusätzliche Applikation auf ihr betrieben werden soll, die aber eine andere Version derselben Abhängigkeit benötigt, gerät diese Betriebsart an ihre Grenzen. Um diese Probleme zu lösen, werden virtuelle Maschinen (VMs) verwendet. Auf einer Maschine wird dann eine Virtualisierungssoftware installiert, mithilfe der mehrere VMs betrieben werden können, die jeweils ihr eigenes Betriebssystem mitbringen. In jeder VM wird dann eine Applikation mit ihren eigenen Abhängigkeiten betrieben. Dieses Betriebsmodell gerät jedoch ebenfalls an seine Grenzen, sobald Bestandssoftware in mehrere Microservices transformiert wird, weil jede VM mit ihrem Betriebssystem installiert, gewartet und aktualisiert werden muss [HI17].

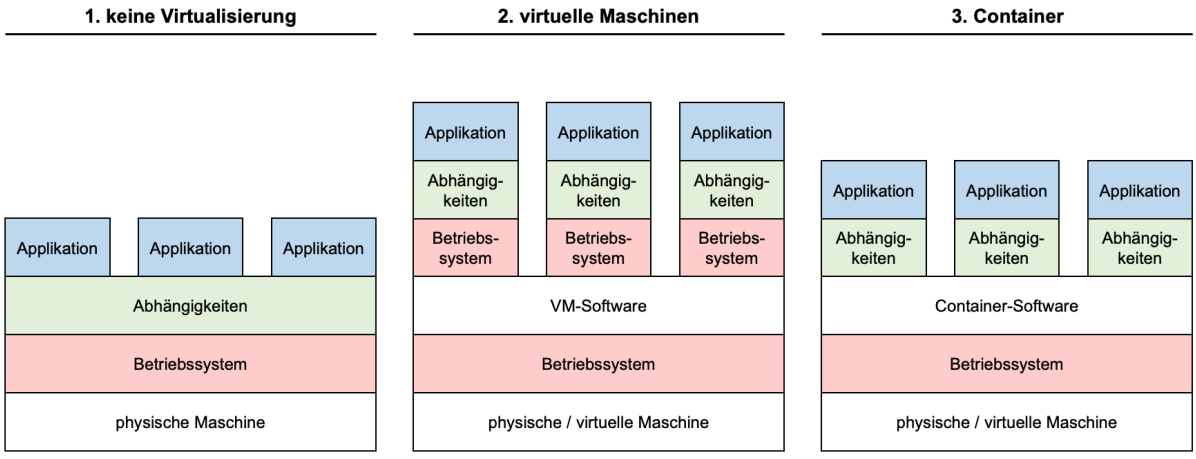

Abbildung 2.2: Evolutionsschritte der Virtualisierung von Betriebsumgebungen [HI17].

Das kostet Zeit, Rechenkapazitäten und verlängert den Startprozess einer Maschine. Diese fixen Betriebskosten je VM stellen bei der steigenden Anzahl von Services einen wesentlichen Kostenpunkt dar. Im dritten Schritt der Virtualisierung wird deshalb, wie in Abbildung 2.2 deutlich, auf das Betriebssystem für die einzelnen Virtualisierungen verzichtet. Stattdessen setzen die Container auf dem Kernel der physischen Maschine auf und können deshalb flexibler und ressourcensparender betrieben werden. Weil kein eigenes Betriebssystem gestartet werden muss, kann die Startzeit im Vergleich zu VMs deutlich verkürzt werden. Außerdem sinkt die Größe eines ausgelieferten Artefakts, wenn kein Betriebssystem mitgeliefert wird. Aufgrund dieser Eigenschaften wird im Betrieb anders mit Containern umgegangen: Statt Aktualisierungen innerhalb eines Containers zu machen, wird eine neue Version ausgeliefert und gestartet. Sobald dieser Container verfügbar ist, wird der alte Container gestoppt – somit können Ausfallzeiten reduziert werden. Im Gegensatz zu VMs existiert eine Umgebung deshalb niemals länger als ihr Container [SI18].

```
docker run mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw
```

Listing 2.1: Befehl für Bereitstellung eines MySQL-Servers mittels Docker.

Vorreiter für diese Technologie war Docker[1]. Die Laufzeitumgebung für Container ist für den Einsatz auf privaten Maschinen nach wie vor sehr verbreitet, für den Betrieb auf Server-Infrastruktur existieren bereits einige Alternativen, wie bspw. Podman[2] – die Benennung der Komponenten ist aber nach wie vor von Docker geprägt. So etwa das `Dockerfile`, das Rezept, nach dem ein Container gebaut wird. Grundlage eines jeden `Dockerfile`s ist ein Basis-Image, also eine grundlegende Konfiguration und Umgebung, auf der aufgebaut wird, etwa eine bestimmte Ubuntu-Version mit einer vorinstallierten Java-Version. Darauf aufbauend können dann unterschiedliche Befehle ausgeführt werden. In einem typischen `Dockerfile` werden erst weitere nötige Abhängigkeiten installiert, dann ein Ordner angelegt, in den der Programmcode kopiert wird und abschließend das Programm gestartet. Wird dieses Rezept ausgeführt, entsteht ein Image. Dieses Image kann für andere Container erneut als Basis-Image dienen. Ähnlich wie für Code-Artefakte existieren für Container-Images Registries, in denen die Images hochgeladen und versioniert werden. Die bekannteste Container Registry ist DockerHub[3], in der ein Großteil der öffentlich verfügbaren Basis-Images liegt. So entsteht ein großer Open-Source-Markt, in dem bspw. fertige Images für eine MySQL-Datenbank liegen. Der Endnutzer muss keinerlei Abhängigkeiten auf seiner Maschine installieren oder ein

[1] `https://www.docker.com/`
[2] `https://podman.io/`
[3] `https://hub.docker.com/`

Installationskript ausführen, sondern kann mit dem einfachen Befehl aus Listing 2.1 eine Instanz mit Standardkonfiguration hochfahren. Weil der Container gekapselt läuft, funktioniert der Betrieb unabhängig vom Betriebssystem [Voh17].

Mit dem Betrieb eines einzelnen Containers ist einem System, das in eine Vielzahl von Microservices aufgeteilt wurde, jedoch noch nicht viel geholfen. Deshalb soll im folgenden Kapitel darauf eingegangen werden, wie mehrere Container gemeinsam orchestriert werden können, um eine gesamte Systemlandschaft mithilfe von Containern zu betreiben.

2.2.2 Orchestrierung mittels Kubernetes

Als Industriestandard für diese Orchestrierung hat sich „Kubernetes“[4] etabliert. Das Projekt wurde 2014 von Google gestartet und ist eines der aktuell 16 Projekte, die im höchsten „Graduated“-Status der Cloud Native Computing Foundation (CNCF) sind. Kubernetes existiert in unterschiedlichen Distributionen, die jeweils eigene, zusätzliche Routinen und Kubernetes-Objekte bereitstellen, die über die Kernkomponenten hinaus installiert werden. Je nach Einsatzszenario von Kubernetes existieren dementsprechend unterschiedlich dimensionierte Distributionen, wie bspw. GKE[5] von Google, EKS[6] von Amazon oder das leichtgewichtige K3S[7] von Rancher Labs. Diese Distributionen werden auf physischen oder virtuellen Maschinen installiert, welche zu einem Kubernetes-Cluster zusammengeschlossen werden können, das durch mindestens einen „Master-Node“ verwaltet wird. Dieser verteilt die zu betreibenden Container auf die übrigen „Worker-Nodes“. Das Cluster kann jederzeit durch zusätzliche Worker-Nodes und somit zusätzliche verfügbare Rechenkapazität erweitert werden.

Um Infrastruktur auf dem Cluster zu betreiben, verfolgt Kubernetes einen deskriptiven Ansatz. Statt Installationsskripten werden Infrastrukturkomponenten also mittels Ressourcen beschrieben. Ändert sich die Beschreibung einer Infrastrukturkomponente, stellt Kubernetes automatisch den beschriebenen Zustand her. Diese Ressourcen werden in Yet Another Markup Language (YAML) verfasst und folgen einer vordefinierten Struktur, einer sogenannten `ResourceDefinition`. Diese Definition ist dabei wie eine Klasse in der objektorientierten Programmierung zu verstehen, von der beliebig viele Instanzen, im Falle von Kubernetes `ResourceObjects`, existieren können. Kubernetes bringt bereits eine Vielzahl von `ResourceDefinitions` mit, allerdings können durch zusätzliche Pakete auch eigene, sogenannte `CustomResourceDefinitions` angelegt werden, um Kubernetes zu erweitern. Listing 2.2 zeigt, wie auf diese Weise mit wenigen Zeilen ein hochverfügbares MySQL-Cluster betrieben werden kann, dessen Beschreibung auf einer `CustomResourceDefinition` für MySQL basiert [Clo21f].

[4] `https://kubernetes.io/`
[5] `https://cloud.google.com/kubernetes-engine/`
[6] `https://aws.amazon.com/de/eks/`
[7] `https://k3s.io/`

```
apiVersion: "mysql.oracle.com/v1"
kind: MySQLCluster
metadata:
name: ha-mysql-cluster
spec:
replicas: 3
```

Listing 2.2: Kubernetes-Objekt auf Basis einer `CustomResourceDefinition` von MySQL.

Abseits dieser vorgefertigten Definitionen wird für den Betrieb eigens entwickelter Software in Containern jedoch vornehmlich auf die durch Kubernetes nativ bereitgestellten `ResourceDefinitions` zurückgegriffen. Einen Überblick über alle existierenden Definitionen zu geben wäre an dieser Stelle nicht zielführend, deshalb soll im Folgenden beispielhaft erläutert werden, wie der Betrieb einer statischen Website funktionieren könnte. Der Aufbau der einzelnen Objekte ist zusätzlich in Abbildung 2.3 visualisiert. Als kleinste auslieferbare Einheit existiert dafür in Kubernetes der Pod. Er enthält einen oder mehrere Container und kann mehrfach nebeneinander betrieben werden – alle Container, die sich innerhalb eines Pods befinden, werden dementsprechend gemeinsam skaliert. In einem solchen Pod würde der Container liegen, in dem sich die statische Website befindet. Ein verbreitetes Muster für Pods ist das sogenannte „Sidecar-Pattern“, bei dem ein weiterer Container in den Pod gelegt wird, welcher als Proxy agiert und allen Verkehr an den eigentlich Service-Container leitet. Dieser Proxy-Container kann dann zusätzliche Logik übernehmen oder zu Monitoring-Zwecken dienen [IH19].

Um den Pod werden zwei Hüllen gelegt: ein ReplicaSet und ein Deployment. Beide gemeinsam bieten Funktionen, die das Betreiben der Pods komfortabel machen. Durch das ReplicaSet können die Instanzen des Pods verändert werden, mithilfe des Deployments können rollierende Aktualisierungen oder Rollbacks vorheriger Aktualisierungen umgesetzt werden. Weil Pods ständig neu gestartet und beliebig skaliert werden können, eignen sich ihre Adressen nicht, um die statische Website aufzurufen. Stattdessen wird ein weiteres Kubernetes-Objekt hinzugefügt: Ein Service macht eine Gruppe von Pods unter einer stabilen Adresse verfügbar, sodass die Website innerhalb des Clusters verfügbar ist. Dabei agiert der Service auch als Load Balancer und verteilt eingehenden Verkehr gleichmäßig auf die verfügbaren Pods. Um die Website schlussendlich auch außerhalb des Clusters verfügbar zu machen, wird ein weiteres Kubernetes-Objekt benötigt, eine sogenannte IngressRoute. Diese leitet eingehenden Verkehr auf Basis des Protokolls, einer Unique Resource Locator (URL) und einem Port auf einen definierten Service innerhalb des Clusters um [Voh16].

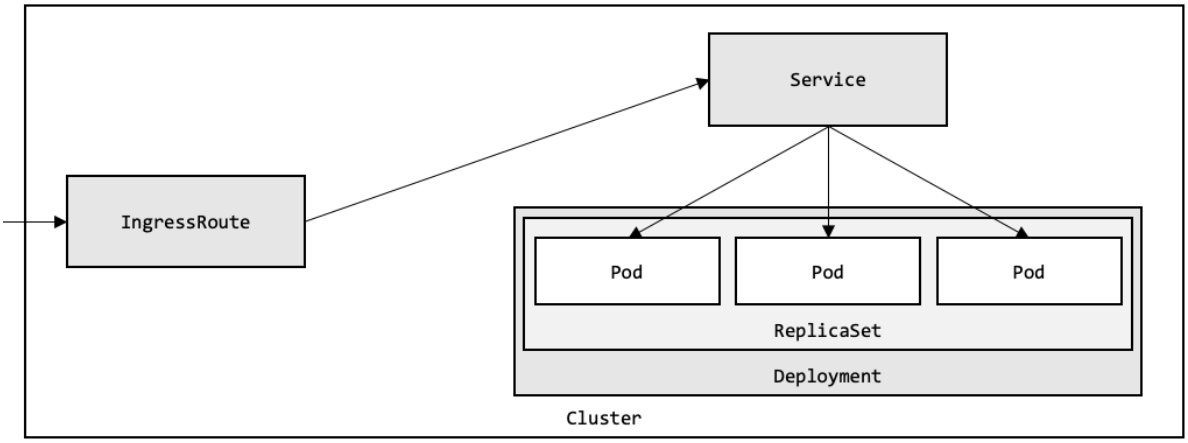

Abbildung 2.3: Notwendige Kubernetes-Objekte zur Bereitstellung einer statischen Website.

Mithilfe dieser fünf Kubernetes-Objekte ist die Auslieferung einer einfachen statischen Website möglich. Um realistische Betriebsszenarien darstellen zu können, existieren darüber hinaus jedoch einige weitere Kubernetes-Objekte, wie etwa für die Zuweisung von Speicherplatz, die Ausführung einmaliger oder wiederkehrender Aufgaben in Jobs oder das Monitoring installierter Infrastruktur – einige davon werden im Prototyp in Kapitel 7 eingeführt.

2.3 Standardisierung der Schnittstellen nach dem REST-Paradigma

Wie bereits in der Einleitung erwähnt, wird mit der Aufspaltung von Applikationen in kleinere Microservices die Standardisierung von Schnittstellen zu einem notwendigen Kriterium, um Geschäftsprozesse über Service-Grenzen hinaus abbilden zu können. Als besonders verbreitete Schnittstelle hat sich dabei das REST-Paradigma durchgesetzt, welches dem „Smart Endpoints and Dumb Pipes“-Prinzip aus Kapitel 2.1.2 folgt und gemeinsam mit dem Standard für die Beschreibung von REST-Schnittstellen, OpenAPI, in den nächsten beiden Kapiteln vorgestellt werden soll.

2.3.1 Grundlagen des REST-Paradigmas

Representational State Transfer ist ein Paradigma, nach dem Schnittstellen über das HTTP-Protokoll angeboten werden. Dabei entspricht das Paradigma keinem festen Regelwerk, sondern einer Ansammlung von Industriestandards, Spezifikationen und Best Practices – der Großteil von ihnen wird von „Standards.REST“[8] gesammelt. Weil REST auf dem zustandslosen HTTP-Protokoll aufbaut, liegt den Schnittstellen das synchrone Request-Response-Pattern zugrunde. Das bedeutet, dass Requests, also Anfragen an die Schnittstelle synchron abgearbeitet werden, der aufrufende Client also so lange blockiert, bis die Response, also die Antwort des Servers, erfolgt ist.

Das Paradigma sieht vor, dass die Geschäftslogik von Services, die eine REST-Schnittstelle anbieten, in Ressourcen modelliert wird. Eine solche Ressource wird immer im Plural geschrieben und unter einem Endpunkt, also einer bestimmten URL, verfügbar gemacht. Auf diese Ressource werden dann die typischen Operationen Create, Read, Update, Delete (CRUD) angewendet. Diese Operationen werden mithilfe der HTTP-Verben umgesetzt – Tabelle 2.1 zeigt, mit welchem HTTP-Verb unter welchem Endpunkt welche Aktion umgesetzt werden kann. Ob eine Aktion erfolgreich war, wird von der Application Programming Interface (API) mithilfe eines HTTP-Statuscodes mitgeteilt. Die Standard-Statuscodes für erfolgreiche Operationen sind ebenfalls der Tabelle zu entnehmen. Dort wird außerdem deutlich, dass unter einer Ressource verschachtelte Ressourcen existieren können. Diese dürfen nur als Sub-Ressource modelliert werden, wenn die Ressource ausschließlich unter ihrer übergeordneten Ressource existieren kann. Zwar können Ressourcen auch als Relation in anderen Endpunkten dargestellt werden, allerdings dürfen keine koexistierenden Endpunkte für dieselbe Ressource existieren [RR07].

Über die HTTP-Verben und -Statuscodes macht REST auch von den Pfad-Parametern und HTTP-Headern des Protokolls Gebrauch: Über die Pfad-Parameter, die an die URL angehängt werden, kann mit bspw. `?name=Peter&sort=asc` eine Filterung und Sortierung der angefragten Ressource vorgenommen werden. HTTP-Header werden wiederum genutzt, um sich bspw. mittels des `Authorization`-Headers gegenüber der API zu authentifizieren oder mithilfe eines `Accept: application/json` eine Antwort der API in JavaScript Object Notation (JSON) verlangt werden. JSON ist das am weitesten verbreitete Format in REST-APIs, weil es platzsparend ist und leicht von Frontends interpretiert werden kann, die oft in JavaScript geschrieben sind [Pat17].

Die geringe Komplexität dieses Paradigmas zur Erstellung standardisierter Schnittstellen hat zu weiter Verbreitung geführt: REST ist leicht zu lernen – schließlich ist die wesentliche Funktionsweise in nur einer Tabelle erklärt. Außerdem kann fast jede Sprache und fast jedes System HTTP-Aufrufe tätigen oder entgegennehmen. Es gibt jedoch auch Herausforderungen, die bei intensiver Auseinandersetzung mit dem Paradigma entstehen – begonnen mit der Modellierung von Operationen auf Ressourcen, die über die

[8] `https://standards.rest/`

Aktion	**HTTP-Verb**	**Pfad**	**HTTP-Statuscode**
alle Objekte lesen	`GET`	`/offices`	`200 OK`
Objekt erstellen	`POST`	`/offices`	`201 Created`
Objekt lesen	`GET`	`/offices/:id`	`200 OK`
Objekt komplett aktualisieren	`PUT`	`/offices/:id`	`200 OK`
Objekt teilweise aktualisieren	`PATCH`	`/offices/:id`	`200 OK`
Objekt löschen	`DELETE`	`/offices/:id`	`204 No Content`
alle Sub-Objekte auslesen	`GET`	`/offices/:id/desks`	`200 OK`
Sub-Objekt erstellen	`POST`	`/offices/:id/desks`	`201 Created`
...			

Tabelle 2.1: Darstellung von Geschäftslogik in Ressourcen-Endpunkten nach dem REST-Paradigma.

typischen CRUD-Befehle hinaus gehen. Soll etwa eine Bestellung storniert werden, kann dies nach dem Paradigma nicht über einen `/order/1/cancel` Endpunkt geschehen, da keine Verben im Pfad stehen dürfen. Stattdessen empfiehlt REST, in diesen Fällen eine Sub-Ressource wie `order/1/cancellation` anzulegen oder die Ressource mit einem booleschen Wert wie `is_cancelled` auszustatten, der dann mit `PATCH` über eine partielle Aktualisierung gesetzt wird. Die Verwendung des HTTP-Verbs `PATCH` zur partiellen Aktualisierung von Ressourcen stellt eine weitere Herausforderung dar: Wenn nur Teile einer Ressource aktualisiert werden, kann es im Falle von parallelen Zugriffen zu Race Conditions kommen oder Seiteneffekte auf andere Attribute der Ressource auslösen. Deswegen wird oft auf „JSON PATCH“[9] oder „JSON Merge PATCH“[10] verwiesen – beides Muster, die jedoch wesentlich komplexer sind. Sowohl die eingeschränkten Operationen als auch die soeben erwähnten Einschränkungen mit dem HTTP-Verb `PATCH` verdeutlichen, dass REST für komplexe Anwendungsfälle seine Stärken in lesenden Zugriffen hat [RR07].

Über diese Herausforderungen hinaus treten auf Ebene eines Ökosystems Integrationsprobleme auf, wenn REST als vorherrschende Schnittstellen-Art verwendet wird, um Services zu integrieren. Bevor diese Integrationsprobleme erläutert werden, soll jedoch ein Blick auf den Standard für die Schnittstellenbeschreibung von REST-APIs geworfen werden.

[9] `https://datatracker.ietf.org/doc/html/rfc6902/`
[10] `https://datatracker.ietf.org/doc/html/rfc7386/`

2.3.2 Beschreibung der Schnittstellen via OpenAPI

Wie auch für andersartige Schnittstellen zum Austausch von Daten existieren für REST-Schnittstellen Beschreibungsformate, die Struktur der Schnittstelle sowie ein- und ausgehende Daten beschreiben. Diese Beschreibungsformate werden Interface Description Language (IDL) genannt. Für REST-APIs ist „OpenAPI“[11] am weitesten verbreitet, der Standard hat sich gegen Alternativen wie „RAML“[12] oder „API Blueprint“[13] durchgesetzt. Das Projekt von der Firma SmartBear ist unter dem Namen „Swagger“ bekannt geworden – diese Bezeichnung findet sich auch heute noch in so mancher Dokumentation wieder. Als Swagger sich als Industriestandard durchgesetzt hat, wurde es aus Gründen der Neutralität mit der Version 3 an die OpenAPI Foundation unter dem Dach der Linux Foundation übergeben und dort als Open-Source-Projekt von über 30 Firmen unterstützt, unter anderem Google, Microsoft, IBM, Oracle oder SAP [Ope21].

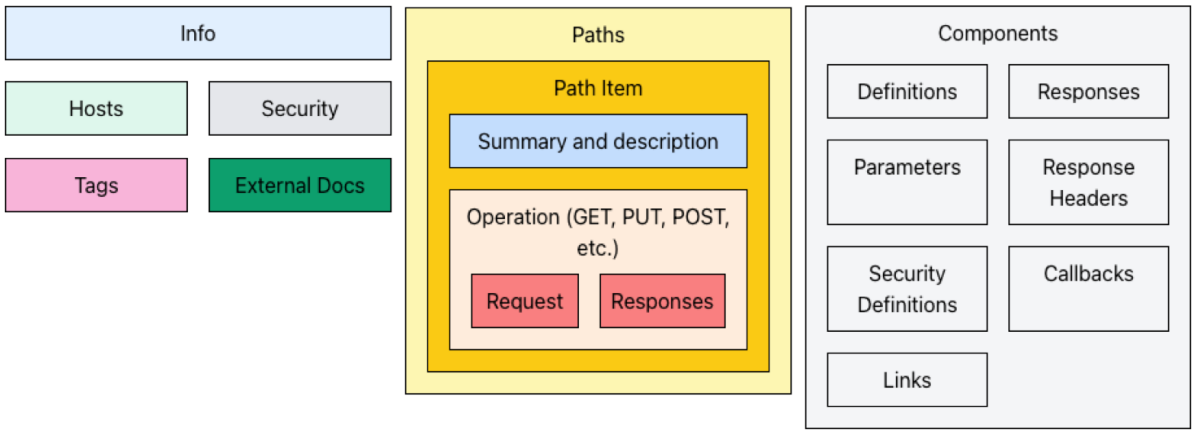

Abbildung 2.4: Aufbau einer Schnittstellenbeschreibung nach der OpenAPI-Spezifikation [Asy21b].

OpenAPI dokumentiert die Endpunkte mitsamt Pfad-Parameter, Anfrage- und Antwortstruktur sowie die HTTP-Header von REST-APIs in einem maschinenlesbaren Format. Diese Informationen werden mitsamt Meta-Informationen zu der Schnittstelle in einem Dokument abgelegt. Der Aufbau eines solchen Dokuments ist in Abbildung 2.4 dargestellt. Das Format, in dem die Schnittstellen beschrieben werden, basiert auf JSON Schema[14] – es ist somit programmiersprachenunabhängig und kann in JSON oder YAML verfasst werden. Aufgrund der Maschinenlesbarkeit können auf Basis der IDL

[11] https://www.openapis.org/
[12] https://raml.org/
[13] https://apiblueprint.org/
[14] https://json-schema.org/

Dokumentation sowie Client-Artefakte generiert, Mock-Server konfiguriert oder automatische Tests erstellt werden. Die Tooling-Welt um die OpenAPI-Spezifikation steigt deshalb beständig – ein Überblick kann sich bei „OpenAPI.Tools“[15] verschafft werden. Aufgrund dieser verfügbaren Werkzeuge gibt es mehrere Herangehensweisen, die Schnittstellenbeschreibung zu erstellen: API-First und Code-First. Im API-First-Ansatz wird die Schnittstelle per Hand mittels OpenAPI beschrieben, bevor eine Implementierung stattfindet. Geschieht dies mit allen, die an der API beteiligt sind, kann das Dokument als Vertrag zwischen Konsument und Produzent wirken. Der Code-First-Ansatz hingegen basiert auf Annotationen, die im Programmcode der Schnittstelle gemacht werden. Mithilfe dieser Annotationen kann der Generator eine Beschreibung der implementierten Schnittstelle erstellen. Dies hat wiederum den Vorteil, das die Beschreibung und die implementierte Schnittstelle nicht voneinander abweichen können [Var16].

Je nach verwendetem Ansatz können OpenAPI-Spezifikationen somit im Unternehmenskontext eine Rolle in organisatorischen Prozessen spielen. Darüber hinaus eignen sich die IDLs, um einen Katalog aller verfügbaren Schnittstellen einer Organisation mitsamt Beschreibung der Struktur zu erstellen. Bestrebungen wie diese werden gemeinsam mit den zuvor erwähnten organisatorischen Prozessen als „API-Management“ bezeichnet – OpenAPI legt als neutrales, beschreibendes Dokument, den Grundstein für diese Art von verwaltenden Tätigkeiten [De17].

Mithilfe von REST können mit wenig Aufwand Microservices entwickelt werden, die über standardisierte und mittels OpenAPI beschriebene Schnittstellen kommunizieren. Der Betrieb in Containern und deren Orchestrierung via Kubernetes ermöglicht es Organisationen, Bestandssysteme durch kleiner geschnittene und flexiblere Services abzulösen. Welche Probleme jedoch entstehen, wenn bei der Integration der neuen Services ausschließlich REST verwendet wird, wird im nächsten Kapitel erläutert.

[15] `https://openapi.tools/`

3 Entstehende Integrationsprobleme und verfügbare Technologien

Nachdem die theoretischen Grundlagen für die Funktionsweise von Microservices und REST-Schnittstellen gelegt wurden, wird in diesem Kapitel der Fokus auf die Integrationsprobleme von REST gelegt, die die zuvor vorgestellten Eigenschaften von Microservices beeinträchtigen. Deren konkrete Beeinträchtigung wird in Kapitel 3.2 analysiert, um anschließend den Stand der Technik in Bezug auf diese Probleme darzulegen.

3.1 Grenzen und Probleme des REST-Paradigmas

Das REST-Paradigma ist die verbreitetste Art und Weise, standardisierte Schnittstellen für Services bereitzustellen und oft der erste Schritt, den Unternehmen gehen, deren Anwendungslandschaft historisch gewachsen und monolithisch geprägt ist. Der Grund dafür sind die Vorteile, die bereits in Kapitel 2.3.1 angesprochen wurden – allen voran die Einfachheit und Zugänglichkeit der Schnittstelle für eine Vielzahl unterschiedlichster Clients. Sowohl ein ähnlicher, moderner Microservice als auch Anwendungen, die auf einem Großrechner laufen oder SAP-Module können HTTP-Anfragen absetzen und verarbeiten und so mit dem Service kommunizieren. Während REST in den Anfängen der Transformation zu Microservices durch seine Klarheit punkten kann und der Großteil der Geschäftslogik über die Bestandssysteme abgewickelt wird, werden mit fortschreitender Verschiebung dieser Geschäftslogik in die Microservices Herausforderungen deutlich, die es in Bezug auf REST als Integrationsstil zu bewältigen gilt [RR07].

In diesem Kapitel sollen deshalb Probleme und Grenzen des REST-Paradigmas in Bezug auf die Integration von Services in einer Microservice-Landschaft herausgestellt werden. Die Probleme beziehen sich hauptsächlich auf die Kommunikation zwischen den Services – Themen abseits der Integration wie Client-Server Kommunikation sowie Herausforderungen des Paradigmas selbst wurden in Kapitel 2.3.1 angerissen und sollen hier keine Rolle spielen. Stattdessen wird das Augenmerk insbesondere auf das Request-Response-Pattern gelegt, auf dem REST basiert sowie die Auswirkungen des ressourcenorientierten Ansatzes von REST auf die Modellierung der Domäne.

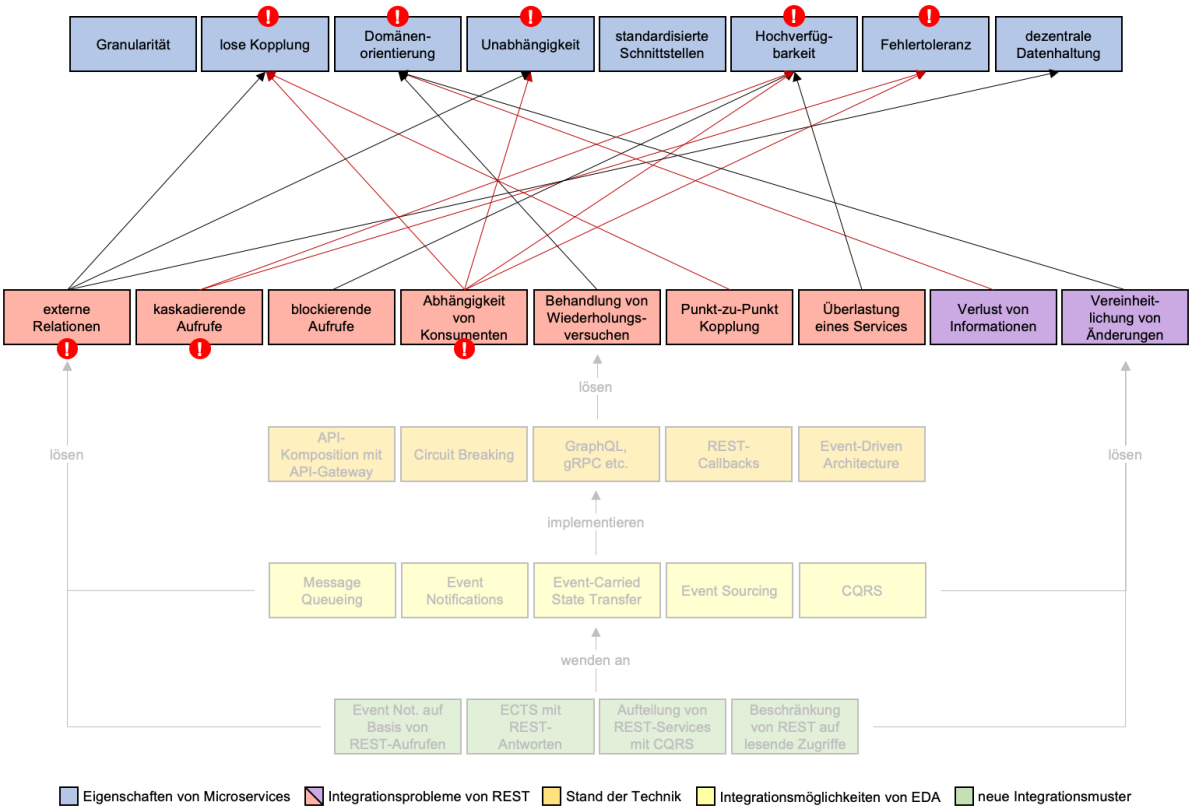

Abbildung 3.1: Auswirkungen der Integrationsprobleme auf die Eigenschaften von Microservices.

Abbildung 3.1 erweitert dafür den Überblick über den Aufbau der Arbeit aus Abbildung 1.1 und fokussiert sich auf die Beeinträchtigung der einzelnen Integrationsprobleme auf die Eigenschaften von Microservices, wie sie auf den folgenden Seiten detailliert erläutert werden. Die rot hinterlegten Probleme beziehen sich auf das Request-Response-Pattern (Kapitel 3.1.1), die lila hinterlegten Probleme auf die Modellierungsmöglichkeiten von Ereignissen (Kapitel 3.1.2). Wie auch die Eigenschaften von Microservices werden die Integrationsprobleme im Laufe der Arbeit kursiv geschrieben, wenn direkt auf sie verwiesen wird.

3.1.1 Request-Response-Pattern

Chris Richardson definiert in seinem Buch „Microservices Patterns“ [Ric19] zwei Achsen, auf denen er Integrationsstile einordnet: One-to-One versus One-to-Many und synchron versus asynchron. Tabelle 3.1 zeigt, welcher Integrationsstil welchen Achsen zugeordnet werden kann.

	One-to-One	**One-to-Many**
synchron	Request-Response	
asynchron	asynchrones Request-Response One-Way-Notifications	Publish-Subscribe Publish-async.-Response

Tabelle 3.1: Dimensionen von Integrationsstilen nach Chris Richardson [Ric19].

Das REST-Paradigma ist nach diesem Schema als synchroner Integrationsstil definiert, der mit dem Request-Response-Pattern Interaktionen zwischen genau einem anfragenden und genau einem empfangenden Service durchführt. Während dieser Stil viele sinnvolle Einsatzzwecke hat, treten Probleme auf, wenn er als einzige Art der Integration zwischen Services verwendet wird. Darüber hinaus sollten Aufrufe nur dann synchron ausgeführt werden, wenn die Synchronität fachlich notwendig ist, die Antwort also direkt weiterverarbeitet werden muss. Die entstehenden Probleme werden im Folgenden aufgelistet. Sie entstammen unterschiedlicher Literatur und verweisen jeweils auf Eigenschaften, die in Kapitel 2.1.2 eingeführt wurden, aufgrund des jeweiligen Problems jedoch nicht zum Tragen kommen, wenn für das Problem keine Lösung gefunden wird.

Externe Relationen

Ein einzelner Microservice A ist nur für einen Teil der Entitäten verantwortlich, die für einen Geschäftsprozess relevant sind. Deshalb stehen diese Entitäten oft in Relation zu Entitäten, die von anderen Services B&C verantwortet werden. Um das Prinzip der *dezentralen Datenhaltung* zu erfüllen, müssen diese externen Relationen innerhalb des Services A gespeichert und aktuell gehalten werden, um sie bspw. in Antworten von Client-Anfragen mitzusenden oder um die Erstellung von eigenen Entitäten zu validieren.

Wenn der einzige Zugang zu den Entitäten der anderen Services B&C eine REST-Schnittstelle ist und der Abruf zur Laufzeit einer Client-Anfrage oder Validierung erfolgen soll, muss der angefragte Service B/C verfügbar sein. Diese zeitliche Kopplung verletzt die Prinzip der *losen Kopplung*, *Unabhängigkeit* und *Fehlertoleranz*.

Sollen die Daten nicht zum Zeitpunkt der Client-Anfrage oder Validierung abgerufen werden, entsteht ein Konsistenzproblem, weil Service A nicht weiß, wann sich eine Ressource in Service B/C ändert. Regelmäßige Abfragen wären nicht nur inperformant, sondern außerdem nicht ausreichend. In einer Schnittstellenarchitektur, in der nur REST zur Verfügung steht, muss dementsprechend mit Abfragen zur Laufzeit gearbeitet werden [New15].

Kaskadierende Aufrufe

Das eben vorgestellte Szenario führt außerdem zu einem weiteren Problem: Auch die Services B&C werden nicht alle relevanten Daten in ihrer eigenen Verantwortung haben. Deshalb müssen sie weitere Services D-G anfragen, um eine Antwort senden zu können, die den anfragenden Client zufriedenstellt. Ist in diesem Szenario auch nur ein einziger Service (B-G) nicht verfügbar, ist die gesamte Kette von Services für die Anfrage an Service A nicht verfügbar. Das Prinzip der *Fehlertoleranz* wird somit verletzt, insbesondere aber die *Hochverfügbarkeit*, weil sich ein Fehler in einem einzelnen Service auf das gesamte Umfeld auswirken kann [Ric19].

Blockierende Aufrufe

Ein weiterer Nebeneffekt dieses Integrationsstils ist die schlechte Performance aufgrund von blockierenden Aufrufen. REST-Schnittstellen basieren auf dem HTTP-Protokoll, dessen Aufrufe synchron ausgeführt werden, also auf die Antwort gewartet wird, bis die Ausführung des Programms fortgesetzt werden kann. Diese Wartezeit wirkt sich mit wachsenden Aufrufkaskaden auf die Performance und damit schlussendlich auch auf die *Hochverfügbarkeit* aus [BHW11].

Abhängigkeit von Konsumenten

Als Alternative zu dem zuvor vorgestellten Szenario kann ein Service A Änderungen an den von ihm verantworteten Entitäten allen Services B-D mitteilen, die diese Entitäten als Relationen verwenden. Wenn Service A Kundenadressen verwaltet, würde diese Änderung dann über die REST-APIs den Services B-D bekannt gemacht werden. Doch auch dieser Integrationsstil verletzt mehrere Prinzipien: Weil Service A alle Interessenten der fachlichen Änderung kennen muss, entsteht fachliche Kopplung. Außerdem müssen auch in diesem Szenario alle Services gleichzeitig verfügbar sein – sie sind somit zeitlich gekoppelt. Beides verletzt das Prinzip der *losen Kopplung* und *Unabhängigkeit*.

Die Änderung einer Kundenadresse setzt damit die Verfügbarkeit aller Interessenten voraus, verletzt damit die *Fehlertoleranz* und ähnlich wie das Problem der kaskadierenden Aufrufe die *Hochverfügbarkeit* des einzelnen Services.

Behandlung von Wiederholungsversuchen

Schlägt der Aufruf von Service A an einen anderen Service B fehl, weil dieser bspw. nicht verfügbar ist oder Netzwerkprobleme aufgetreten sind, liegt es in der Verantwortung von Service A, diesen Aufruf zu gegebener Zeit zu wiederholen, bis dieser erfolgreich war.

Wann ein geeigneter Zeitpunkt für einen erneuten Versuch ist, hängt vom Fehlerfall ab und sollte nicht Teil der Anwendungslogik eines Services sein – insbesondere dann nicht, wenn Service A dem anderen Service B wie im vorigen Absatz beschrieben nur Änderungen von Entitäten mitteilen möchte [Bel20]. Im engeren Sinne verletzt diese zusätzliche Logik also die *Domänenorientierung* des Services.

Punkt-zu-Punkt-Kopplung

Wenn Services ausschließlich über ihre REST-APIs integriert werden, bedingt die Skalierung eines Services A auch die Skalierung aller von ihm angesprochenen Services B-D, weil eine hohe Anzahl von Aufrufen an Service A auch direkt die Anzahl der Aufrufe in Service B-D beeinflusst.

Diese Art der Integration verletzt die *lose Kopplung* und kann zu einem verteilten Monolithen führen. Als verteilter Monolith werden Service-Konstrukte bezeichnet, die zwar als verteiltes System betrieben werden, aufgrund ihrer Architektur aber die Probleme monolithischer Software (siehe Kapitel 2.1.1) haben und nicht von den Eigenschaften von Microservices profitieren können [Bel20].

Überlastung eines Services

Wenn zu einem unerwarteten Zeitpunkt oder in kürzester Zeit sehr viele Anfragen auf einen Service abgesetzt werden, etwa weil dieser wie im vorigen Absatz beschrieben direkt mit einem anderen Service gekoppelt ist, kann es zu einer Überlastung des Services kommen. Eine solche Überlastung tritt auf, wenn die Last schneller steigt, als weitere Instanzen des Services bereitgestellt werden können, um der Last gerecht zu werden. Dann ist das Prinzip der *Hochverfügbarkeit* verletzt und die direkten Anfragen können nicht beantwortet werden [JHB16].

3.1.2 Modellierungsmöglichkeiten von Ereignissen

Nachdem die Probleme betrachtet wurden, die das synchrone Request-Response-Pattern von REST-APIs mit sich bringt, soll nun ein Blick auf Probleme geworfen werden, die direkt aus den Paradigmen, nach denen REST-Schnittstellen entworfen werden, resultieren. Auch in diesem Kapitel wird sich ausschließlich auf Probleme konzentriert, die Einfluss auf die Integration von Services haben und die Umsetzung der vorgestellten Eigenschaften aus Kapitel 2.1.2 erschweren bzw. verhindern.

Verlust von Informationen

REST folgt, wie in Kapitel 2.3.1 beschrieben, dem CRUD-Pattern. Dieses Pattern scheint aus technischer Perspektive sehr intuitiv, kann aus Sicht der fachlichen Domäne aber zu Problemen führen, wenn Daten aktualisiert und damit überschrieben werden. Sind die überschriebenen Daten auch nach der Aktualisierung relevant, muss die Information in der Modellierung besonders behandelt werden und kann bspw. als eigene Entität mit zeitlicher Abgrenzung modelliert werden. Das CRUD-Pattern von REST steigert somit die Komplexität der Modellierung von Domänen und verletzt die *Domänenorientierung* von Microservices [NMAM16].

Vereinheitlichung von Änderungen

REST verfolgt bei der Modellierung der verantworteten Entitäten einen ressourcenorientierten Ansatz und sieht bspw. keine Verben in URLs vor, sondern arbeitet ausschließlich mit HTTP-Verben. Wie bereits in Kapitel 2.3.1 als Herausforderung erwähnt, werden Informationen und Ereignisse aus der Domäne dementsprechend in CRUD-Ressourcen transformiert, um dem REST-Paradigma zu entsprechen. Bei dieser Transformation gehen Informationen verloren, denn die Geschäftsprozesse, die mit den Services abgebildet werden sollen, sind ereignisgetrieben und gehen in ihren Statusänderungen über `CREATE`, `UPDATE` und `DELETE` hinaus. Damit wird das Prinzip der *Domänenorientierung* verletzt, weil die Softwarearchitektur durch technische Paradigma statt Fachlichkeit in der Domäne getrieben wird [Ric19]. Wenn Geschäftsprozesse über die Kombination mehrerer Services abgebildet werden, den Schnittstellen der Services aber jegliche Ereignisorientierung fehlt, entsteht ein Problem bei der Integration genau dieser Services und schlussendlich bei der Abbildung von Prozessen.

3.2 Analyse der Auswirkungen vorgestellter Probleme

In diesem Kapitel soll analysiert werden, welche Eigenschaften von Microservices besonders bedroht sind und welche der zuvor vorgestellten Integrationsprobleme den größten Anteil daran haben. Dafür wird erneut ein Blick auf Abbildung 3.1 geworfen: Dort sind die Eigenschaften und Probleme, die in diesem Kapitel betrachtet werden, mit einem roten Ausrufezeichen markiert. Markiert werden alle Eigenschaften, die durch mehr als ein Problem bedroht sind und alle Probleme, die mehr als eine Eigenschaft bedrohen.

Zu den Eigenschaften, die von drei Problemen bedroht werden, gehört die *lose Kopplung*, die *Domänenorientierung* und die *Hochverfügbarkeit*. Die *Unabhängigkeit* und *Fehlertoleranz* wird durch zwei der vorgestellten Integrationsprobleme bedroht und die *dezentrale*

Datenhaltung durch eines der Probleme. Nur die Eigenschaften *standardisierte Schnittstellen* und *Granularität* sind von keinem der Probleme bedroht. Somit sind sechs der insgesamt acht Eigenschaften durch eines der Integrationsprobleme bedroht, die entstehen, wenn Services hauptsächlich über REST-APIs integriert werden. Innerhalb der Integrationsprobleme lässt sich hingegen eine ganz klare Rangliste angeben, die von folgenden drei Problemen angeführt wird:

1. Abhängigkeit von Konsumenten (4 verletzte Eigenschaften)
2. externe Relationen (3 verletzte Eigenschaften)
3. kaskadierende Aufrufe (2 verletzte Eigenschaften)

Ihr folgen die restlichen sechs Probleme, welche jeweils eine Eigenschaft von Microservices bedrohen. Die schwerwiegendsten drei Probleme verletzen somit insgesamt neun Eigenschaften und sind damit für 60 Prozent der Verletzungen verantwortlich. Lösungsansätze, die zum Ziel haben, die Eigenschaften von Microservices wieder zum Tragen zu bringen, können dies also besonders effektiv tun, indem sie diese schwerwiegenden Integrationsprobleme lösen. Sie werden in dieser Arbeit deshalb besonders gewichtet. Um jedoch alle Eigenschaften wiederhergestellt sind und die Stärken von der Transformation zu Microservices ausgespielt werden können, müssen auch die übrigen Integrationsprobleme gelöst werden. Im folgenden Kapitel wird deshalb der Stand der Technik der Lösungsansätze für diese Probleme vorgestellt.

3.3 Stand der Technik

In diesem Kapitel werden Lösungsansätze des Stands der Technik vorgestellt – die dritte und tiefgelbe Ebene aus dem Überblick in Kapitel 1.3. Dafür wird der Überblick in Abbildung 3.2 erneut erweitert und konzentriert sich nun darauf, welche Lösungsmöglichkeit in der Lage ist, welches Problem zu lösen. Dabei sind die schwerwiegendsten Probleme mit einem roten Ausrufezeichen markiert, während die Lösung, die die meisten Probleme löst, mit einem grünen Ausrufezeichen versehen ist.

Auf den folgenden Seiten werden deshalb fünf Lösungsansätze vorgestellt, die in der Literatur zur Bekämpfung dieser Probleme genannt werden. Ein Lösungsansatz, der im Kontext von Microservices und der Integration dieser verteilten Systeme oft genannt wird, sind sogenannte „Workflow Engines“ wie Camunda[16]. Da der Fokus dieser Workflow Engines jedoch hauptsächlich auf der Orchestrierung bzw. Choreographie von Services zu gesamten Geschäftsprozessen liegt, wird diese Technologie in dieser Arbeit nicht weiter betrachtet.

[16] `https://camunda.com/de/`

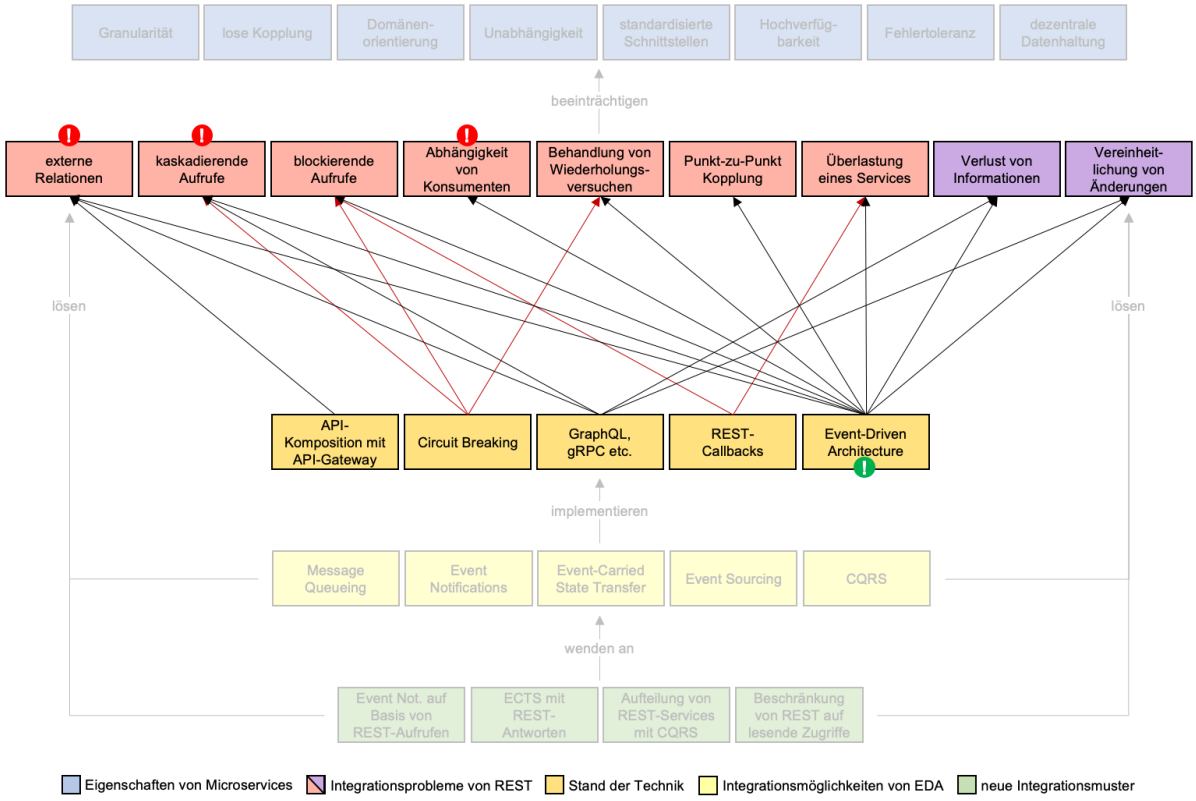

Abbildung 3.2: Auswirkungen des Stands der Technik auf die Integrationsprobleme.

3.3.1 API-Komposition via API-Gateways

Ein API-Gateway ist eine Komponente, die in der Infrastruktur eines API-Ökosystems etabliert werden kann, um zwischen die Aufrufe der Schnittstellen geschaltet zu werden. Statt die API eines Services direkt aufzurufen, wird die API auf dem Gateway registriert und unter einer dem Gateway zugehörigen Zieladresse veröffentlicht. Clients stellen ihre Aufrufe dann gegen das API-Gateway, das die Aufrufe an die registrierte API weiterleitet. Durch diese zwischengeschaltete Komponente können ein besseres Logging und Monitoring des Ökosystems umgesetzt werden, Aufrufe verändert oder angereichert werden und Konsumenten autorisiert werden. Im Kontext dieser Arbeit ist jedoch besonders eine Funktion interessant: Weil alle Schnittstellen des Ökosystems über das Gateway angeboten werden, kann auf dieser Ebene eine Komposition mehrerer Entitäten aus unterschiedlichen APIs erfolgen. Das Gateway stellt dann eine übergeordnete Kompositionsschicht dar [De17].

Diese Komposition löst das Problem der *externen Relationen*, ohne dass eine Anpassung der Services notwendig ist. Sie erfüllt jedoch nicht die dahinterliegende Eigenschaft der *dezentralen Datenhaltung* für die einzelnen Services. Außerdem kann die Komposition innerhalb des API-Gateways je nach Struktur der zusammenzuführenden Daten inperformant sein, darüber hinaus müssen die Services nach wie vor gleichzeitig verfügbar sein, damit eine Komposition zur Laufzeit des Aufrufs erfolgen kann. Unabhängig von der Komposition mehrerer Schnittstellen kann ein API-Gateway einen Flaschenhals darstellen, weil alle Kommunikation über die Komponente erfolgt – sie muss dementsprechend hochverfügbar bereitgestellt werden [NMAM16].

Ein API-Gateway ist eine sinnvolle Komponente, um (REST-)Schnittstellen im Unternehmenskontext zu verwalten. Es kann zu besserer Übersichtlichkeit des Ökosystems beitragen und darüber hinaus einzelne der angesprochenen Probleme lösen bzw. abschwächen. Die Komponente ändert jedoch nicht den Interaktionsstil, mit dem über REST-APIs Daten ausgetauscht werden und stellt deshalb keine geeignete Lösung für die in Kapitel 3.1 angesprochenen Probleme dar.

3.3.2 Circuit Breaking

Eine weitere Lösungsmöglichkeit stellt das sogenannte „Circuit Breaking“ dar. Ähnlich wie eine Sicherung im Stromkasten löst ein Circuit Breaker nach einer bestimmten Anzahl fehlgeschlagener Aufrufe auf eine API aus und schützt sie vor weiteren Zugriffen. Dieser Schwellwert kann je nach Einsatzszenario unterschiedlich definiert werden – denkbar ist bspw. eine Quote von Aufrufen, die von der API mit einem Statuscode `5xx`, also einem Serverfehler, beantwortet werden. Dann werden für eine bestimmte Zeit Aufrufe nicht zum Service der Schnittstelle geleitet, sondern direkt bearbeitet. Während dieser Auszeit können entweder eine Antwort aus dem Cache, eine Standardantwort oder auch eine Fehlermeldung zurückgegeben werden. Die Antwort des Circuit Breakers enthält in der Regel einen HTTP-Header, der Auskunft darüber gibt, wann der Service vorraussichtlich wieder erreichbar ist und erleichtert aufrufenden Services die *Behandlung von Wiederholungsversuchen*. Außerdem werden regelmäßig Health Checks durchgeführt, um den Zustand des fehlerhaften Services zu überprüfen. Dafür können auch eine geringe Menge von Aufrufen genutzt werden, die als Health Checks dienen. Sobald diese Aufrufe von der Schnittstelle erfolgreich verarbeitet werden, wird aufkommender Datenverkehr wieder an den Service geleitet.

Statt eines Timeouts vom anfragenden Service werden Aufrufe nach dem „Fail Fast“-Prinzip Aufrufe direkt beantwortet. Dadurch wird das Problem der *blockierenden Aufrufe* abgeschwächt, was indirekt das Problem der *kaskadierenden Aufrufe* abmildert. Darüber hinaus wird der defekte Service vor weiteren Aufrufen geschützt und hat die Möglichkeit, sich wiederherzustellen. Darüber hinaus kann ein Circuit Breaker auch im Falle einer Wartung für einzelne Services manuell aktiviert werden [New15].

Circuit Breaking stellt ein sinnvolles Werkzeug dar, um die Schwächen synchroner REST-APIs auszugleichen und ist in der Lage, einige der Integrationsprobleme zu lösen. Weil es, ähnlich wie das API-Gateway, jedoch nicht den Interaktionsstil der Schnittstellen ändert, der Grundlage für die Integrationsprobleme ist, eignet es sich nicht als Lösung für den Großteil der angesprochenen Probleme.

3.3.3 Alternative synchrone Schnittstellen

Eine naheliegende Lösung, um die Probleme von REST-APIs zu lösen, ist auf sie zu verzichten. Stattdessen können alternative, synchrone Punkt-zu-Punkt-Schnittstellen verwendet werden, um die Nachteile des REST-Paradigmas aus Kapitel 3.1.2 zu kompensieren. Grundsätzlich kann dies auch über HTTP-APIs gelöst werden, die REST-Konventionen bewusst ignorieren. Weil dies jedoch zu falschen Erwartungen von Konsumenten an die Schnittstelle führen kann, sollen im Folgenden zwei verbreitete Alternativen vorgestellt werden.

Besonders verbreitet und als Alternative zu REST bekannt, ist die von Facebook entwickelte Abfragesprache GraphQL[17], die ihren Ursprung in der Anbindung der ersten Smartphone-Apps hat. Das Projekt wird mittlerweile Open-Source verwaltet und spezialisiert sich auf individuelle Abfragemöglichkeiten durch die API-Konsumenten, um auch bei schlechter Netzwerkqualität mit möglichst wenig Aufrufen und geringer Nachrichtengröße einen hohen Informationsgehalt zu bieten. Darüber hinaus existieren GraphQL-Gateways, die ähnlich wie ein API-Gateway die Schnittstellen mehrerer Services aggregieren und über die Abfragesprache zur Verfügung stellen. Während diese Funktionsweise große Vorteile für externe API-Konsumenten bietet, ist GraphQL weniger auf Interprozesskommunikation ausgelegt und kann deshalb nur wenige der angesprochenen Integrationsprobleme lösen [Kre21]. Ähnlich verbreitet ist gRPC[18], das unter dem Namen „Stubby“ von Google entwickelt wurde und inzwischen ebenfalls Open-Source verfügbar ist. Es nutzt die bekannten Möglichkeiten der Remote Procedure Calls und ruft Methoden eines Services auf, der diese mittels einer IDL beschrieben und verfügbar gemacht hat. gRPC wird als „Incubating“-Projekt, der zweiten von drei Reifheitsgraden der CNCF geführt [Lin21].

Sowohl GraphQL als auch gRPC stellen geeignete Alternativen dar, um die *Vereinheitlichung von Veränderungen* oder den *Verlust von Informationen* durch von REST abweichende Paradigmen zu lösen. Analog zu den vorherigen Werkzeugen lösen Schnittstellen-Technologien jedoch einige der besonders schwerwiegenden Integrationsprobleme nicht.

[17] `https://graphql.org/`
[18] `https://grpc.io/`

3.3.4 Web-Hooks und Callbacks

Ein wiederkehrender Kritikpunkt vorangehender Lösungsmöglichkeiten ist, dass sie die Grundlage der Probleme, den synchronen Interaktionsstil von REST-APIs, nicht ändern. Callbacks bzw. Web-Hooks setzen genau an diesem Punkt an und ermöglichen, dass Aufrufe asynchron verarbeitet werden. Wenn die Antwort auf einen API-Aufruf zeitintensiv ist, standen für den aufgerufenen Service bisher zwei unterschiedliche Optionen zur Verfügung: Der Service kann so lange blockieren, bis die Antwort berechnet wurde oder dem Konsumenten zurückmelden, dass die Anfrage entgegengenommen wurde und einen Endpunkt nennen, unter dem die Antwort nach abgeschlossener Berechnung zur Verfügung steht. Der Konsument der API musste dann, wie in Abbildung 3.3 auf der linken Seite dargestellt, in regelmäßigen Abständen prüfen, ob die Berechnung bereits vollzogen wurde [De17].

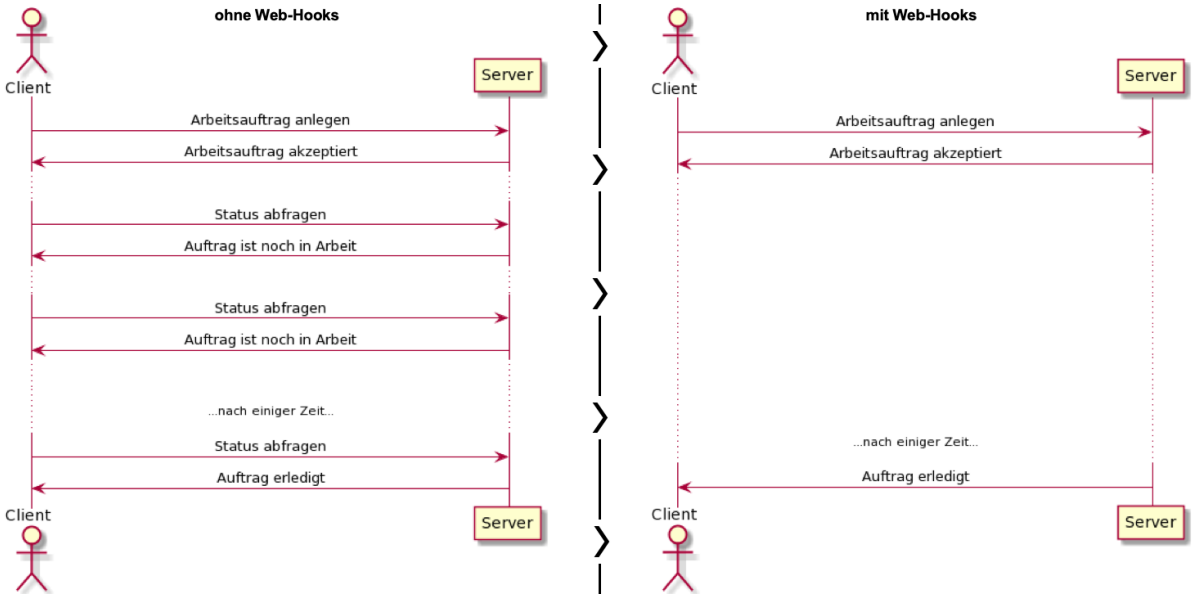

Abbildung 3.3: Benachrichtigung von Clients bei Erledigung zeitintensiver Arbeitsaufträge mittels Web-Hooks bzw. Callbacks.

Mit Web-Hooks besteht eine weitere Möglichkeit, den Konsumenten mit dem Ergebnis des Aufrufs zu versorgen: Statt durch regelmäßige Aufrufe unnötige Last zu erzeugen, hinterlegt der Konsument der API beim Aufruf einer zeitintensiven Operation eine Zieladresse, die als Callback für den aufgerufenen Service dient. Sobald die Berechnung abgeschlossen ist, wird wie auf der rechten Seite von Abbildung 3.3 die Callback-Adresse aufgerufen und das Ergebnis übermittelt. Eine ähnliche Architektur kann auch genutzt werden, um Services von Ereignissen in Kenntnis zu setzen [SI18].

Diese Lösungsmöglichkeit wird direkt von OpenAPI unterstützt: Dort kann der Aufbau der Aufrufe beschrieben werden, die von der beschriebenen Schnittstelle an die vom Konsumenten hinterlegte Callback-Adresse gestellt werden [Sma21]. Mithilfe dieses Werkzeugs kann im Gegensatz zu vorigen Möglichkeiten eine Veränderung der Interaktion von Services herbeigeführt werden, weil die regelmäßigen Aufrufe des Konsumenten überflüssig werden. Die asynchrone Verarbeitung verhindert außerdem *blockierende Aufrufe* und schützt vor einer *Überlastung des Services*. Allerdings besteht nach wie vor ein Problem, wenn der Service, der den Callback angefordert hat, nicht verfügbar ist. Das Problem der *Abhängigkeit von Konsumenten* und die *Behandlung von Wiederholungsversuchen* besteht also weiterhin. Darüber hinaus muss der Service, der die Callbacks ausführt, Geschäftslogik aufbauen, um ausstehende Callbacks zu verwalten – das widerspricht der *Domänenorientierung*. Web-Hooks bzw. Callbacks können damit zwar eine anfängliche Verbesserung der Interaktion herbeiführen, allerdings ist diese nicht tiefgreifend genug, um wesentliche Integrationsprobleme zu lösen.

3.3.5 Event-Driven Architecture

Die fünfte Lösung ist kein direkter Ersatz für synchrone Schnittstellen, die nach dem Request-Response-Pattern Daten austauschen und auch kein Werkzeug, um ihre Schwächen auszugleichen. Stattdessen setzt Event-Driven Architecture auf einen nachrichten- bzw. ereignisbasierten Austausch von Daten. Der Produzent einer Nachricht schickt diese jedoch nicht an den Konsumenten, sondern einen Mediator, der zwischen Sender und Empfänger liegt. Dieser Mediator wird in der Regel als „Message-Broker“ oder „Event-Broker“ bezeichnet. Die Erstellung einer Nachricht wird dadurch von der Verarbeitung durch einen Konsumenten entkoppelt. Der Produzent ist somit nur dafür verantwortlich, dass eine Nachricht erstellt und dem Mediator zugestellt wird, nicht für die Verarbeitung oder Zustellung an Konsumenten. Konsumenten hingegen tragen die Verantwortung dafür, für sie relevante Informationen zu filtern und dann zu verarbeiten. Durch dieses Konzept wird die Abhängigkeit zwischen Produzent und Konsument umgekehrt: Statt dass der Produzent dafür verantwortlich ist, die Informationen an den Konsumenten zuzustellen liegt es in der Verantwortung des Konsumenten, die Informationen vom Mediator zu beziehen. Da die Erstellung einer Nachricht außerdem nicht direkt an die Verarbeitung gekoppelt ist, kann der Datenaustausch über einen Message-Broker asynchron stattfinden und stellt eine Alternative zur asynchronen Verarbeitung via Web-Hooks aus dem vorigen Kapitel dar. Dieser Datenaustausch ermöglicht die Umsetzung alternativer Interaktionsstile, wie sie in Tabelle 3.1 erwähnt wurden. So kann nach wie vor eine Punkt-zu-Punkt-Verbindung zwischen zwei Services aufgebaut werden. Allerdings kann durch den Mediator als zentrale Einheit auch das Publish-Subscribe-Pattern umgesetzt werden, bei dem mehrere konsumierende Services die Nachricht eines Produzenten verarbeiten [BHW11].

Der Einsatz eines Message-Brokers zur Umsetzung der bisher beschriebenen Konzepte wird oft als Message-oriented Middleware beschrieben. Um eine Architektur als ereignisgetrieben zu bezeichnen, muss über den Interaktionsstil der Services hinaus auch die Modellierung einzelner Services und ihrer repräsentierten Domänen ereignisorientiert erfolgen. Nur wenn mit dem Design der Fokus auf die Ereignisse gelegt wird, können diese als Nachrichten über einen Message-Broker verteilt und Grundlage zur Integration einer Systemlandschaft werden [DB10]. In diesem Kontext wird dann von Events gesprochen, die über einen Event-Broker verteilt werden.

Durch diesen alternativen Interaktionsstil können alle der vorgestellten Integrationsprobleme gelöst werden. Durch die Umkehrung der Abhängigkeit zwischen Konsument und Produzent wird die *Abhängigkeit von Konsumenten* mitsamt der *Behandlung von Wiederholungsversuchen* sowie die *Punkt-zu-Punkt-Kopplung* aufgehoben und das Problem der *externen Relationen* gelöst, weil Services Veränderungen an relevanten Entitäten zur Laufzeit aus dem Mediator beziehen können. Weil Daten asynchron ausgetauscht werden können, werden der *Überlastung eines Services* vorgebeugt und *blockierende Aufrufe* sowie *kaskadierende Aufrufe* hinfällig. Darüber hinaus kann durch die ereignisorientierte Modellierung als Alternative zur Ressourcenorientierung und CRUD der *Verlust von Informationen* sowie die *Vereinheitlichung von Veränderungen* verhindert werden. Mit den Vorteilen gehen jedoch neue Herausforderungen einher: Durch die Asynchronität erhöht sich die Komplexität des gesamten Systems, welches durch die lose Kopplung der Services außerdem unübersichtlicher wird [Bel20].

Die bisher vorgestellten Lösungen, insbesondere API-Gateways und Circuit Breaker, stellen eine sinnvolle Ergänzung zu synchronen REST-Schnittstellen dar und können einige ihrer Schwächen ausgleichen – sie ändern jedoch nichts an dem grundlegenden Entwurfsmuster und Interaktionsstil. Wie in Abbildung 3.2 deutlich wird, stellt EDA somit die vielversprechendste der bisher vorgestellten Lösungen dar. Als Alternative zu synchronen Schnittstellen ist sie nicht darauf ausgelegt, diese komplett zu ersetzen, sondern soll stattdessen dort als Erweiterung eingesetzt werden, wo REST-APIs an ihre konzeptionellen Grenzen stoßen. Aus diesem Grund sollen im folgenden Kapitel die Grundlagen von EDAs dargelegt werden.

4 Event-Driven Architecture als Architekturstil

Um EDA als aussichtsreichste Lösung der Integrationsprobleme und alternativen Interaktionsstil zwischen Services näher zu betrachten, sollen in diesem Kapitel die theoretischen Grundlagen für den Architekturstil gelegt werden. Dafür wird zu Beginn eine Einführung in die Ereignisorientierung als Konzept gegeben, um anschließend die Broker-Lösungen am Markt zu vergleichen und abschließend die hellgelbe Ebene aus Abbildung 1.1, die Integrationsmöglichkeiten von EDA, vorzustellen. Diese theoretische Grundlage wird abschließend einer kritischen Bewertung unterzogen, um in den nachfolgenden Kapiteln eigene Ansätze zu erarbeiten.

4.1 Einführung in die Ereignisorientierung

Ereignisorientierung als Konzept beschränkt sich nicht auf die Integration von Services. In diesem Kapitel sollen deshalb die Bedeutung von Ereignissen in Unternehmen eingeführt werden, bevor die Bestandteile einer ereignisorientierten Architektur erläutert werden und ein Überblick über die Herausforderungen im Umgang mit asynchronen Events gegeben wird.

4.1.1 Bedeutung von Ereignissen in Unternehmen

Wenn Software entwickelt wird, geschieht das nie aus einem Selbstzweck. Stattdessen werden Lösungen entwickelt, die den Menschen dabei unterstützen, Dinge aus der echten Welt digital abzubilden, komplizierte Berechnungen durchzuführen oder Daten zu speichern. Im Kontext großer Unternehmen bedeutet das oft auch, Geschäftsprozesse wie bspw. die Vergabe eines Hauskredits mithilfe von Software zu durchlaufen. Diese Prozesse sind getrieben von Ereignissen: Eine Interessentin bucht ein Beratungsgespräch, ein Kunde ändert seine Adresse oder der variable Zins eines bestehenden Kredits ändert sich. Solche Vorkommnisse aus der realen Welt müssen ihren Weg als Ereignis in das digitale Ökosystem des Unternehmens finden, damit darauf reagiert werden kann. Dabei sind die Ereignisse als Fakten zu verstehen: Sie beschreiben Vorkommnisse, die geschehen sind

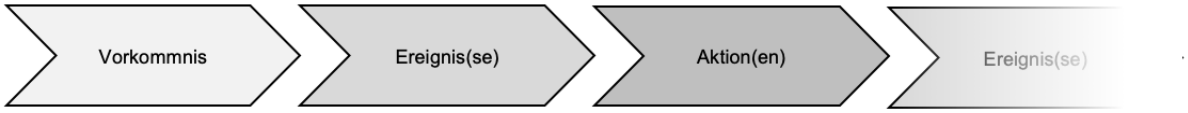

Abbildung 4.1: Veranschaulichung der Reaktion auf Vorkommnisse durch Ereignisse im System.

und sich nicht mehr ändern lassen. So entsteht die in Abbildung 4.1 visualisierte Kette: Aus einem Vorkommnis in der realen Welt wird ein Ereignis im System, auf das eine Aktion folgen kann. Werden diese Ereignisse in einem Event Stream, also einem Strom von Ereignissen, dokumentiert, entsteht eine Single Source of Truth von Ereignissen, die im Kontext des Unternehmens und seiner Geschäftsprozesse geschehen sind [Esp15].

In einem Unternehmen, dessen Prozesse komplett digitalisiert sind, können die reale Welt und ihre Vorkommnisse gut durch die Ereignisse im Event Stream nachgebildet werden und ermöglicht neben der Integration verteilter Systeme diverse Einsatzzwecke: So können auf Basis der Events weitere Geschäftsprozesse angestoßen werden und bspw. nach einer Zinsanpassung Briefe an die Betroffenen versendet werden. Da der Gesamtheit der Ereignisse in einem Event Stream vereint sind, kann dieser auch in Echtzeit analysiert werden, um bspw. verdächtige Kreditkarten-Transaktionen zu erkennen und ein dementsprechendes Ereignis erzeugen. Diese Form der Analyse, die mehrere Events analysiert, um auf Muster reagieren zu können, nennt sich „Event Stream Processing“ oder „Complex Event Processing“. Darüber hinaus ermöglicht die dauerhafte Speicherung des Event Streams auch Analysen vergangener Geschehnisse und eignet sich bspw. mithilfe von Machine Learning zur Entscheidungsunterstützung in Strategiefragen des Unternehmens [DB10].

4.1.2 Bestandteile einer ereignisorientierten Architektur

Um die Vorkommnisse der realen Welt als Ereignis im Ökosystem abzubilden, benötigt es Event-Broker als Infrastruktur, die diese Ereignisse als Mediator verteilen. Abbildung 4.2 zeigt den typischen Aufbau einer solchen Infrastruktur. Im Zentrum steht der Mediator, der Event-Broker. Dieser ist dafür verantwortlich, die Nachrichten von den Produzenten zu den Konsumenten zu transportieren, ohne zu viel Geschäftslogik zu übernehmen – ganz nach dem Prinzip „Smart Endpoints and Dumb Pipes“. Dafür werden innerhalb des Mediators mehrere Kanäle eingesetzt. Dabei ist ein Kanal als abstraktes Konstrukt zu verstehen: Bei Punkt-zu-Punkt-Verbindungen (unterer Kanal in Abbildung 4.2) werden sie in der Regel als „Queue“ bezeichnet, weil sie eine Warteschlange von Nachrichten bilden, die der Konsument abarbeitet.

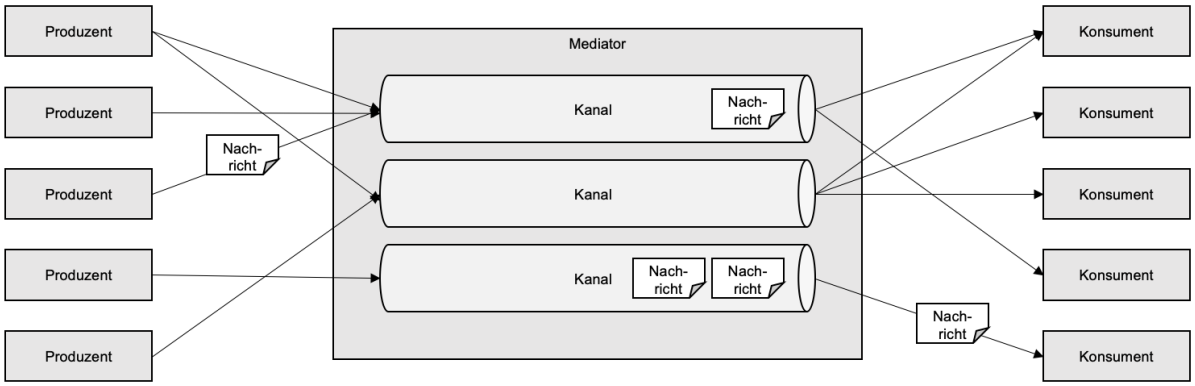

Abbildung 4.2: Abstrakte Komponenten einer ereignisorientierten Architektur [MG20].

In Publish-Subscribe-Szenarien wird in der Regel von „Topics“ gesprochen: Unter einem beschreibenden Namen wie bspw. `CustomerAddressChanged` werden dann Nachrichten bereitgestellt, die ein bestimmtes Event beschreiben und von mehreren Konsumenten verarbeitet werden können [Chr19].

Eine Nachricht als allgemeines, technisches Konzept kann semantisch dementsprechend unterschiedlich interpretiert werden: Wird mit der Nachricht ein geschehenes Ereignis beschrieben, wird von einem Event gesprochen. Das Event wird anderen Services über den Mediator zur Verfügung gestellt, der Produzent hat jedoch gemäß des „Fire-and-Forget“-Patterns keine Erwartungshaltung bezüglich der Verarbeitung des Events. Verfolgt der Produzent mit einer Nachricht jedoch eine klare Absicht und erwartet, dass ein Konsument auf eine bestimmte Weise darauf reagiert, wird von einem Command gesprochen. Events werden dementsprechend typischerweise über Topics verteilt, während Commands über Queues zugestellt werden [MG20].

Als Beteiligte dieser Infrastruktur treten Produzenten und Konsumenten auf. Produzenten, oft auch Ereignisquelle oder Publisher genannt, erstellen Nachrichten und legen diese in einem Kanal innerhalb des Mediators ab. Konsumenten, oft auch Ereignissenke oder Subscriber genannt, stellen eine dauerhafte Verbindung mit dem Mediator her und abonnieren die Kanäle, an deren Inhalt sie interessiert sind. Wenn eine Nachricht in diesem Kanal auftaucht, stellt der Mediator dem Konsumenten diese Nachricht zu. Konsumenten können sich dabei auch im „Competing Consumer Pattern“ zusammenschließen: Dann wird die Nachricht nach einer vordefinierten Logik nur an einen der Konsumenten zugestellt, die eine Subscription für den Kanal beim Mediator hinterlegt haben. Das bietet sich z.B. an, um die Verarbeitung von Commands durch den Einsatz mehrerer Instanzen zu skalieren [BHW11].

Die tatsächliche Implementierung dieses abstrakten Konzepts unterscheidet sich von Broker zu Broker stark und bietet in der Regel auch immer Möglichkeiten über die vorgestellten Grundlagen hinaus. Bevor die konkreten Implementierungen des in Abbildung 4.2 visualisierten Konzepts vorgestellt werden, soll jedoch ein Überblick über Herausforderungen dieser Form des Datenaustauschs gegeben werden.

4.1.3 Herausforderungen im Umgang mit asynchronen Events

Neben den bereits angesprochenen Vorteilen, die in Bezug auf die Integrationsprobleme aus Kapitel 3.1 deutlich werden, erhöht EDA die Komplexität des Ökosystems und bringt Herausforderungen mit sich, die in den folgenden Absätzen angerissen werden sollen, um die erhöhte Komplexität zu verdeutlichen.

Wie bereits in Kapitel 3.3.5 eingeführt, wird durch einen Event-Broker die Erstellung eines Events von seiner Verarbeitung entkoppelt. Im Gegensatz zu synchronen Schnittstellen, bei der auf die erfolgreiche Verarbeitung gewartet wird, bevor eine Aktion als erfolgreich abgeschlossen wird, kann dem Aufrufer eines Services via EDA eine erfolgreiche Rückmeldung gegeben werden, obwohl die Verarbeitung des Aufrufs noch nicht abgeschlossen ist. Dieses Phänomen wird durch den Begriff „Eventual Consistency“ beschrieben und steht im Zusammenhang mit dem CAP-Theorem für verteilte Systeme. Dieses besagt, dass in einem verteilten System die Eigenschaften Konsistenz (**C**onsistency), Verfügbarkeit (**A**vailability) und Partitionstoleranz (**P**artition Tolerance) nicht gleichzeitig voll erreicht werden können – eine der Eigenschaften ist dementsprechend zwangsläufig nicht vollumfänglich erfüllt. Mit dem Einsatz von ereignisgetriebenen Systemen und der entkoppelten Verarbeitung der Events wird die Konsistenz als Eigenschaft nicht vollständig erfüllt – schlussendlich („eventually“) wird die Konsistenz mit Verarbeitung des Events durch alle Konsumenten jedoch hergestellt. In der Zwischenzeit kann es also passieren, dass z.B. eine Veränderung eines Datensatzes noch nicht in allen Datenbeständen des verteilten Systems angekommen ist und dementsprechend ein veralteter Datensatz ausgegeben wird. Ob Eventual Consistency akzeptabel ist, ist also vielmehr eine fachliche als eine technische Entscheidung. Je nach Domäne muss abgewogen werden, ob die Verfügbarkeit des Systems oder die strenge Konsistenz seines verteilten Datenbestands höher zu bewerten ist [Chr19]. Die Kommunikation über Event-Broker ist jedoch auch ohne Eventual Consistency möglich. Dafür würde eine Aktion erst dann erfolgreich abgeschlossen werden, wenn alle Konsumenten die Verarbeitung des Events bestätigen – in diesem Falle gingen jedoch die Asynchronität und Entkopplung verloren.

Eine weitere Herausforderung stellt die Zustellung von Events an Konsumenten in verteilten Systemen dar. Wenn für einen Konsumenten Events in einem von ihm abonnierten Kanal zur Verfügung stehen, bieten Event-Broker unterschiedliche Zustellungsgarantien: *höchstens einmal*, *mindestens einmal*, oder *genau einmal*, wobei letzteres selten unterstützt und in jedem Fall nur mit Mehraufwand und deutlichen Performance-Einbußen

möglich ist. Je nach Zustellungsgarantie muss also damit umgegangen werden können, das Events komplett verloren gehen oder aber Konsumenten darauf eingestellt sein, dass sie identische Events mehrfach zugestellt bekommen. Um diesen Problemen vorzubeugen, ist zusätzliche Logik in den Services nötig, die bspw. die `ID`s bereits verarbeiteter Events speichert [BHW11].

In verteilten Systemen kann außerdem die Verarbeitung von Events in der richtigen Reihenfolge zu einer Herausforderung werden. So muss zum Beispiel bei einem Event Stream für eine Bestellung darauf geachtet werden, dass die Events in der korrekten Reihenfolge abgearbeitet werden – andernfalls könnte das `OrderCancelled`-Event auftreten, bevor die Bestellung überhaupt aufgegeben wurde. Wenn die Reihenfolge der Events Einfluss auf die abgebildete Domäne hat, die Events aber durch mehrere Instanzen parallel verarbeitet werden, wird im Falle einer fehlgeschlagenen Verarbeitung eines einzelnen Events die Einhaltung der Reihenfolge verletzt. Um die Verarbeitung für diese Art von Events trotzdem parallelisieren zu können, werden die Events in partitionierte Kanäle aufgeteilt, deren Untermengen sich nicht beeinflussen. Abbildung 4.3 zeigt, wie die Partitionierung anhand des Beispiels der Bestellungen aussehen könnte: Dort werden Bestellungen mit einer geraden `ID` und die Bestellungen mit einer ungeraden `ID` unterschiedlichen Kanälen zugeordnet und können somit parallel verarbeitet werden. Die Reihenfolge der Events wird damit nach wie vor für eine einzelne `ID` eingehalten, während gleichzeitig die Verarbeitungsgeschwindigkeit verdoppelt werden kann. Spielt die Reihenfolge jedoch keine Rolle, kann die Verarbeitung auch ohne den Einsatz von partitionierten Kanälen durch mehrere Instanzen parallelisiert werden [Ric19].

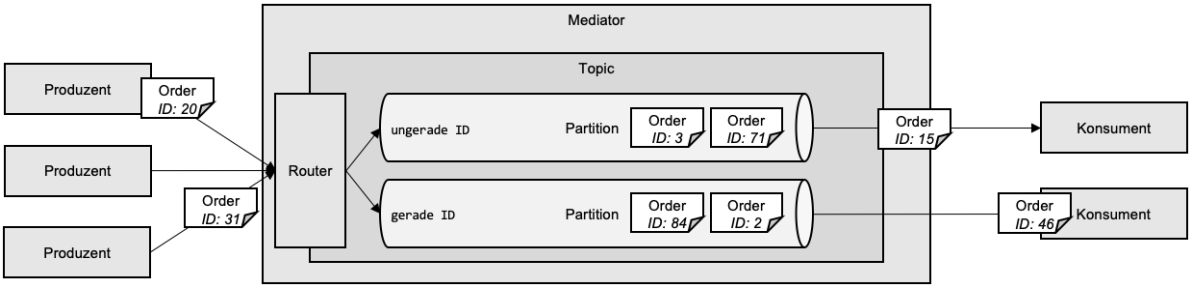

Abbildung 4.3: Parallelisierte Verarbeitung von Events bei Berücksichtigung der Reihenfolge durch Partitionierung.

Wenn die Produzenten von den Konsumenten entkoppelt werden, wird außerdem der Datenfluss innerhalb der Systemlandschaft unübersichtlicher. Ohne zusätzliche Werkzeuge ist nicht ersichtlich, welche Aktionen die Veröffentlichung eines Events auslöst – so können beispielsweise auch Zirkelbeziehungen entstehen, wenn nicht klar ist, welche Konsumenten auf welche Events reagieren. Diese Unübersichtlichkeit sorgt auch für erschwerte Tests einzelner System-Komponenten – die Konsequenzen eines Events sind

nur im Zusammenspiel aller Komponenten ersichtlich. Deswegen bieten Event-Broker oft die Möglichkeit, einen Event Stream, der sich bspw. in einem Produktivsystem ereignet hat, künstlich nachzustellen und bspw. in einem Testsystem erneut abzuspielen, um Fehlerszenarien nachvollziehen zu können [MG20].

Abschließend ist zu beachten, dass mit dem Einsatz eines Event-Brokers als Mediator zwischen Produzent und Konsument ein Single-Point-of-Failure geschaffen wird, der die Kommunikation eines gesamten Ökosystems behindern und bei Lastspitzen zum Flaschenhals des Datenaustauschs werden kann. Weil der Event-Broker immer zwischen Produzent und Konsument steht, ist außerdem eine gewisse zeitliche Verzögerung nicht zu verhindern – synchrone Punkt-zu-Punkt-Schnittstellen sind dementsprechend performanter [Ric19]. Die Verzögerung ist für die meisten Anwendungsszenarien jedoch nicht von Bedeutung: Je nach Event-Broker liegt die Latenz zwischen Produzent und Konsument bei 5 bis 80 Millisekunden [FZY21].

Für all die beschriebenen Herausforderungen müssen Lösungen gefunden werden, die diese Herausforderungen bewältigen, wie die exemplarisch vorgestellte Partitionierung von Kanälen. Diese Lösungen sollen nicht in dieser Arbeit beschrieben werden, verdeutlichen jedoch, dass die Vorteile von ereignisgetriebener Architektur zum Preis erhöhter Komplexität kommt: Durch die asynchrone und entkoppelte Verarbeitung entstehen Szenarien, denen gegenüber synchrone Schnittstellen mit ihrer Einfachheit punkten können. Da außerdem ein Event-Broker benötigt wird, um den Datenaustausch zu realisieren, muss zusätzliche Infrastruktur installiert und gewartet werden. Drei der gängigsten Broker-Lösungen und ihre Alleinstellungsmerkmale sollen deshalb im nächsten Kapitel vorgestellt werden.

4.2 Broker-Lösungen am Markt

Das Herzstück einer ereignisgetriebenen Architektur ist der Mediator, über den Produzenten und Konsumenten Informationen austauschen. Drei Lösungen sollen in diesem Kapitel vorgestellt werden, um Gemeinsamkeiten, Unterschiede und Herausforderungen bei der Verwendung der Infrastruktur zu verdeutlichen. Dabei wird sich in der Gegenüberstellung auf die Kriterien für die Auswahl konzentriert, die von Chris Richardson in „Microservice Patterns“ [Ric19] genannt werden:

1. unterstützte Programmiersprachen
2. unterstützte Protokolle (wie z.B. AMQP, MQTT etc.)
3. Skalierbarkeit
4. Persistierung von Events

5. Latenz
6. Zustellungsgarantie (*mindestens einmal, höchstens einmal, genau einmal*)
7. Erhalt der Reihenfolge von Nachrichten

Als erste Broker-Lösung soll RabbitMQ[19] vorgestellt werden. RabbitMQ bezeichnet sich selbst als „Message-Broker“ und soll in dieser Arbeit näher betrachtet werden, weil es der Broker ist, der am weitesten verbreitet ist. Als zweite Lösung wird Apache Kafka[20] vorgestellt, die ihr Produkt selbst als „Distributed Streaming Platform“ bezeichnen. Kafka ist hat sich insbesondere in den letzten Jahren zu einer sehr verbreiteten Lösung unter großen Unternehmen etabliert und wird deshalb in dieser Arbeit vorgestellt. Abschließend wird NATS JetStream[21] vorgestellt. Diese Lösung, die vage als „connective technology for distributed applications“ bezeichnet wird, ist vor allen Dingen im Kontext der CNCF weit verbreitet und wird aufgrund seiner Leichtgewichtigkeit in dynamisch skalierbaren Szenarien verwendet. Damit sind für eine diverse Landschaft von Broker-Lösungen die am weitesten verbreiteten Stellvertreter der jeweiligen Ausprägungen von Lösungen eingeschlossen. Lösungen wie ZeroMQ[22], die ohne einen Broker als Mediator arbeiten, sollen in dieser Arbeit hingegen nicht genauer betrachtet werden.

Die Unterschiede der drei Broker bei der Bezeichnung der eigenen Lösung stehen repräsentativ für die Differenzen im Protokoll, den angebotenen Funktionen und dem Betrieb der Lösungen. Es existieren jedoch auch einige Gemeinsamkeiten: Alle drei Lösungen werden Open-Source entwickelt und können kostenlos verwendet werden, außerdem unterstützten alle Lösungen die grundlegenden Interaktionsstile Remote Procedure Calls, Work Queues und das Publish-Subscribe-Pattern. Auch die Latenz der Lösungen ist ähnlich und soll deshalb keine weitere Beachtung finden [FZY21].

4.2.1 RabbitMQ

Die verbreitetste und wohl auch bekannteste Broker-Lösung, RabbitMQ, wurde 2007 veröffentlicht und seitdem von vielen Firmen als klassischer Message-Broker und als Message-oriented Middleware eingesetzt. Dabei sind die Vielzahl der unterstützten Programmiersprachen und Messaging-Protokollen der Beliebtheit sicherlich zuträglich gewesen: RabbitMQ implementiert das Advanced Messaging Queueing Protocol (AMQP), unterstützt aber auch Message Queueing Telemetry Transport (MQTT) und WebSockets. Es existieren Bibliotheken für Java, .NET, Python und viele mehr, die zwingend

[19] `https://www.rabbitmq.com/`
[20] `https://kafka.apache.org/`
[21] `https://nats.io/`
[22] `https://zeromq.org/`

notwendig sind, um über den Broker kommunizieren zu können. Als klassisch implementierter Broker ist RabbitMQ zwar für einen hochverfügbaren Betrieb ausgelegt und bietet inzwischen auch einen Kubernetes Operator[23], allerdings wird in dieser Hinsicht deutlich, dass RabbitMQ nicht nativ für elastisch skalierbaren Betrieb konzipiert wurde: Eine einzelne Instanz ist bereits sehr ressourcenintensiv, die Skalierung auf weitere Instanzen braucht einige Minuten, bis sie verfügbar sind. Ebenso bietet RabbitMQ keine Persistierung von Events an und eignet sich somit nicht für Anwendungsfälle, die auf einer langen Historie von Ereignissen aus einem Event Store aufbauen möchten. Aufgrund von AMQP unterstützt RabbitMQ sowohl die Zustellungsgarantien *höchstens einmal* und *mindestens einmal*, ebenso wird der Erhalt der Reihenfolge von Nachrichten unterstützt [VMV21].

Diese weitreichende Unterstützung von Protokollen und Programmiersprachen ist ein deutlicher Vorteil von RabbitMQ. Die Implementierung von AMQP macht den Broker zu einer puristischen Lösung, die sehr nah an der konzeptionellen Idee des Mediators aus Kapitel 4.1.2 ist. Gleichwohl ist die Installation unkompliziert und ein leichtgewichtiger Betrieb möglich – RabbitMQ ist somit ein geeigneter Kandidat, um die Infrastruktur für den neuen Interaktionsstil zwischen Services bereitzustellen. Die mangelnde Skalierbarkeit der Instanzen und die fehlende Persistierung von Events stellt jedoch eine Einschränkung bei der Verwendung von RabbitMQ dar.

4.2.2 Apache Kafka

Apache Kafka weicht mit seiner Implementierung von der konzeptionellen Idee des Mediators ab. Das Produkt wurde ursprünglich von LinkedIn entwickelt, 2012 an die Apache Software Foundation übergeben und seitdem von Confluent als Open-Source-Projekt betreut. Es wird von allen Dingen von mittleren bis großen Unternehmen eingesetzt, die einen sehr hohen Durchsatz von Events bewältigen müssen, die zum Teil jedoch live analysiert werden soll. Als prominentes Beispiel ist Uber bekannt: Der Personenbeförderungsdienst kalkuliert mithilfe von Kafka die Preise für Fahrten in Echtzeit [Apa21a].

Kafka unterstützt zwar viele Programmiersprachen (z.B. Python, C, Go, Node.js, Java) implementiert jedoch kein standardisiertes Messaging-Protokoll, sondern arbeitet mit einem proprietären Protokoll. Es unterscheidet sich deshalb deutlich von Standards wie AMQP und unterscheidet bei Kanälen bspw. nicht, ob sie als Work Queue genutzt werden oder mit dem Publish-Subscribe-Pattern. Auch das Routing der Events unterscheidet sich grundlegend. So unterstützt der Broker bspw. die Partitionierung von Kanälen, wie sie in Abbildung 4.3 dargestellt wird. Dahingegen ist Kafka besser auf Skalierbarkeit ausgelegt: Das Produkt funktioniert selbst als verteiltes System und

[23] `https://www.rabbitmq.com/kubernetes/operator/operator-overview.html`

ist auf viele separate Services aufgeteilt, die zusammen die Broker-Lösung darstellen. Zusätzlich bietet Kafka eine Persistierung von Events, auch unter Angabe einer Aufbewahrungszeit je nach Event-Typ. Sowohl alle Zustellungsgarantien als auch der Erhalt der Reihenfolge von Events werden unterstützt [Apa21b].

Kafka zeichnet sich als Broker-Lösung aus, die sich aufgrund der sehr großen Funktionsvielfalt und hohen Skalierbarkeit insbesondere im Unternehmenskontext zum Industriestandard entwickelt hat. Die Möglichkeiten der Echtzeit-Analyse mittels Kafka Streams[24] sowie die Partitionierung von Kanälen bieten in Kombination mit dem hohen Datendurchsatz und der Möglichkeit, Events zu persistieren, eine Lösung, die für unterschiedlichste Anwendungsszenarien geeignet ist. Als per Design verteiltes System ist es außerdem für die dynamische Skalierung im Cloud-Kontext geeignet. Genau dieser hohe Umfang sowohl in puncto Funktionen als auch im Betrieb der Lösung kann für kleine Unternehmen bzw. einfache Einsatzzwecke jedoch zur Herausforderung werden: Wenn weder Echtzeitanalyse noch ein hoher Datendurchsatz benötigt werden, existieren leichtgewichtigere Alternativen.

4.2.3 NATS JetStream

Eine dieser möglichen Alternativen ist NATS mit seiner Erweiterung NATS JetStream. Als leichtgewichtiger Broker aus dem Cloud-Native-Kontext stellt er den Kompromiss zwischen RabbitMQ in puncto Leichtgewichtigkeit und Kafka in puncto Skalierbarkeit und Persistierung von Events dar. Die erste Version wurde 2011 veröffentlicht, hat einen großen Teil ihrer Entwicklung jedoch ab 2017 erlebt, als sie Teil der CNCF wurde. Dort ist NATS als eines von zwei Projekten im „Incubating"-Status für die Kategorie „Streaming & Messaging". Damit wird es vorwiegend in Unternehmen eingesetzt, deren Infrastruktur mit Kubernetes betrieben wird, unter anderem Mastercard, Tinder, Walmart oder Tesla. Aufgrund der Leichtgewichtigkeit eignet es sich außerdem für Anwendungszenarien aus dem Bereich Internet of Things und Edge Computing sowie für kleine Unternehmen bzw. Projekte [Clo21e].

Die Broker-Lösung unterstützt ebenso eine Vielzahl von Programmiersprachen (z.B. Go, Java, .NET, Node.js), setzt aber auf ein eigenes Protokoll zum Austausch der Nachrichten – lediglich MQTT wird mithilfe einer Erweiterung unterstützt. Damit weichen das Routing und der Aufbau der Event-Kanäle ebenfalls von Standards ab. Im Gegensatz zu AMQP müssen Kanäle bspw. nicht explizit angelegt werden, sondern werden dynamisch erstellt, wenn ein Event mit einem neuen Event-Typ veröffentlicht wird. NATS JetStream ist elastisch skalierbar und kann aufgrund der leichtgewichtigen Instanzen dynamisch auf schwankenden Nachrichtendurchsatz reagieren. NATS Core bietet keine Persistierung von Events, sondern bewegt sich nah an der konzeptionellen Idee

[24] `https://kafka.apache.org/documentation/streams/`

des Mediators aus Kapitel 4.1.2. Durch die Erweiterung mit NATS JetStream wird die Persistierung allerdings ermöglicht. Mithilfe von Streams können Events basierend auf ihrem Typ mit einer Aufbewahrungszeit persistiert werden. Durch diese Persistierung werden neben der Zustellungsgarantie *höchstens einmal* von NATS Core auch die übrigen beiden Zustellungsgarantien sowie der Erhalt der Reihenfolge von Nachrichten ermöglicht [Clo21d]. Die Leichtgewichtigkeit von NATS in Kombination mit der modernen Interpretation eines Event-Brokers positioniert die Broker-Lösung insbesondere für verteilte Systeme, die in Kubernetes-Infrastruktur betrieben werden. Als ressourcensparende Implementierung ist der Broker schnell aufgesetzt und kann durch die Erweiterung NATS JetStream bei Bedarf in seiner Funktionalität erweitert werden. Als Kehrseite des Kompromisses zwischen den zwei vorigen Lösungen bietet NATS weniger Funktionen als Kafka und ist weniger verbreitet als RabbitMQ.

Die Vorstellung dieser drei Broker-Lösungen verdeutlicht, dass sich für das Konzept eines Event-Brokers noch kein einheitlicher Ansatz gefunden hat, der von unterschiedlichen Produkten ähnlich implementiert wird – stattdessen gibt es neben klassischen Brokern wie RabbitMQ, die ursprünglich für den Einsatz als Message-oriented Middleware entwickelt wurden, eine neue Generation an Event-Brokern wie Kafka oder NATS, die auf die Microservice-Bewegung und dem damit einhergehenden Aufschwung von EDAs zurückzuführen ist – das wird vor allen Dingen an Funktionen wie der Persistierung von Events deutlich. Weder die ältere Generation noch die neueren Lösungen sind dabei ähnlich konzeptioniert, geschweige denn standardisiert, sodass die Anbindung für die proprietären Protokolle nur über die angebotenen Bibliotheken der jeweiligen Lösungen funktioniert und von ihnen abhängig ist. Unabhängig dieser Unterschiede implementieren jedoch alle Lösungen mehr oder weniger puristisch die konzeptionelle Idee des Mediators, mithilfe dessen unterschiedliche Integrationsmöglichkeiten umgesetzt werden können. Diese sollen im folgenden Kapitel dargelegt werden.

4.3 Integrationsmöglichkeiten durch Ereignisorientierung

Um einen Überblick über die Möglichkeiten zu schaffen, die EDA zur Integration von Services bietet, werden insgesamt fünf Muster vorgestellt, die im Wesentlichen den Ausführungen von Martin Fowler [Fow17] folgen und durch weitere Literatur ergänzt werden. Wie bereits in Kapitel 3.3 erwähnt, werden die Orchestrierung oder Choreographie von Services zu verteilten Transaktionen oder Prozessen in dieser Arbeit nicht betrachtet. Das „Saga Pattern“, was oft im Zusammenhang mit EDA erwähnt wird, findet in diesem Kapitel deshalb keine Beachtung.

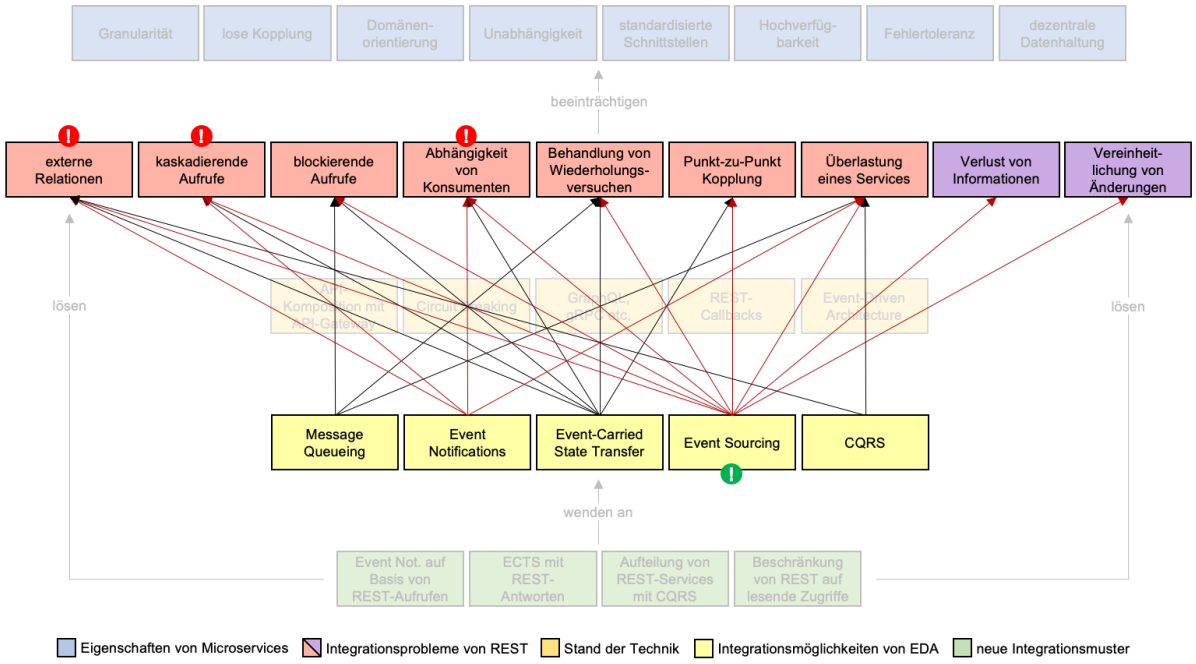

Abbildung 4.4: Auswirkungen der Integrationsmöglichkeiten durch EDA auf Integrationsprobleme.

Die Muster sind in der Reihenfolge ihrer Komplexität und Fokussierung auf Events sortiert. In Abbildung 4.4 wird dafür erneut eine detaillierte Betrachtung auf den Aufbau der Arbeit aus Abbildung 1.1 vorgenommen und verdeutlicht, welche Integrationsprobleme aus Kapitel 3.1 durch die einzelnen Muster gelöst werden. Dabei sind die schwerwiegendsten Probleme mit einem roten Ausrufezeichen markiert, während das Muster, das die meisten Probleme löst, mit einem grünen Ausrufezeichen versehen ist. Nachdem je Muster die Funktionsweise sowie die Vorteile und Herausforderungen dargelegt wurden, werden diese im nachfolgenden Kapitel zusammen mit den in Kapitel 4.2 vorgestellten nötigen Broker-Lösungen einer Bewertung unterzogen.

4.3.1 Message Queueing

Der vermutlich klassischste Ansatz, nachrichten- bzw. ereignisbasierte Kommunikation zwischen Services einzusetzen, ist das sogenannte „Message Queueing“. Statt einen Service wie im Request-Response-Pattern direkt aufzurufen, werden die Aufrufe in eine Queue gelegt, aus der sie asynchron abgearbeitet werden können. In diesem Kontext werden die Events oft als Commands bezeichnet, weil der Produzent der Nachricht mit der Veröffentlichung eine Intention verfolgt und dementsprechend eine bestimmte

Handlung erwartet. Anders als im Publish-Subscribe-Pattern werden die Nachrichten nach dem Competing-Consumer-Pattern in der Regel nur an einen Konsumenten, in dem Kontext oft Worker genannt, zugestellt. Abbildung 4.5 verdeutlicht, wie mehrere Worker die Commands aus einer Queue abarbeiten und sich somit die Arbeitslast teilen. Wenn der Produzent außerdem eine Antwort erwartet, wird vom asynchronen Request-Response-Pattern gesprochen. Der Produzent legt dann eine Queue an, in die Konsumenten die Antworten veröffentlichen. Damit die Commands mit den Antworten verknüpft werden können, wird mit Correlation IDs gearbeitet, die an die Nachrichten gehängt werden. Soll die Nachrichten an mehrere Konsumenten verteilt werden, wird von „Publish-async.-Response-Pattern“ gesprochen. Dann können mehrere Konsumenten ihre Antwort-Nachrichten in die vom Produzenten eingerichtete Queue veröffentlichen [Ric19].

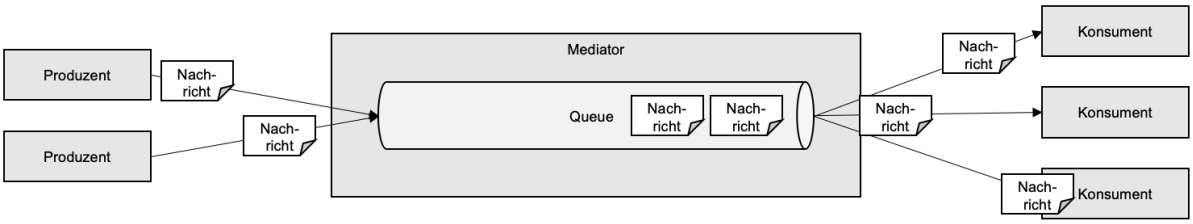

Abbildung 4.5: Abarbeitung einer Message Queue durch mehrere Worker nach dem Competing-Consumer-Pattern.

Die Kommunikation via Message Queue statt synchroner Schnittstelle bietet im Wesentlichen die Vorteile, die schon in Kapitel 3.3.5 angerissen wurden. Da zwischen den kommunizierenden Services ein Event-Broker liegt, übernimmt dieser die Verantwortung für die Zustellung der Nachricht, inklusive eventueller erneuter Versuche. Damit wird das Problem der *Behandlung von Wiederholungsversuchen* gelöst. Darüber hinaus erfolgt kein blockierender Aufruf, weil die Abarbeitung asynchron erfolgt – *blockierende Aufrufe* gehören damit auch zu den gelösten Problemen. Das ist besonders dann praktisch, wenn die Logik, die durch das Command ausgeführt wird, sehr zeit- und rechenintensiv ist. Weil gleichzeitig eintreffende Aufrufe durch die Queue nacheinander abgearbeitet werden, wird der *Überlastung eines Services* vorgebeugt.

Wie in Abbildung 4.4 deutlich wird können über die grundlegenden Vorteile, die durch die asynchrone Verarbeitung und den Event-Broker als Intermediär entstehen, jedoch wenige der in Kapitel 3.1 genannten Integrationsprobleme gelöst werden. Produzent und Konsument sind bspw. nach wie vor eng aneinander gekoppelt, weil sie sich über den Intermediär aufrufen und somit von ihrer Existenz wissen.

4.3.2 Event Notifications

Ein weiteres, eher grundlegendes Muster sind Event Notifications. Im Gegensatz zum Message Queueing werden hier im Sinne der Definition eines Events (siehe Kapitel 4.1.1) keine Antworten erwartet und mit dem Publish-Subscribe-Pattern gearbeitet. Ähnlich wie eine Benachrichtigung auf dem Smartphone enthält das Event aber nicht bereits die komplette Information über das Ereignis, sondern verweist lediglich darauf, dass zum Beispiel eine Bestellung aufgegeben wurde. Services, die sich für dieses Ereignis interessieren, können dann mit einem Aufruf über die etablierten synchronen Schnittstellen des Services den aktuellen Zustand der Entität abrufen. Da das Muster keine Anforderungen an eine zugrundliegende Softwarearchitektur stellt, kann es in einem gewöhnlichen CRUD-Service eingesetzt werden, in dem für bestimmte Ereignisse zusätzlich zu der Datenbank-Operation Events veröffentlicht werden [Fow17].

Event Notifications bringen zwei wesentliche Vorteile mit sich: Wenn alle Services Änderungen an ihren Entitäten per Event Notification veröffentlichen, können andere Services, die diese Entitäten als externe Relationen einbinden, nach jeder Änderung den Zustand abfragen und die Entitäten somit in ihrer eigenen Datenhaltung einbinden. Gleichzeitig wird die *Abhängigkeit von Konsumenten* aufgelöst, weil diese nun eigenverantwortlich den aktuellen Zustand einer Entität abfragen können. Das löst zwei der schwerwiegendsten Probleme: Sowohl *externe Relationen* als auch die damit in Verbindung hängenden *kaskadierenden Aufrufe*, weil zum Zeitpunkt des Aufrufs immer der aktuelle Stand der externen Relationen in den jeweiligen Services existiert. Dadurch kann die Eigenschaft der *dezentralen Datenhaltung* wiederhergestellt werden.

Darüber hinaus können auf Basis der Event Notifications wie bereits beim Message Queueing ressourcenintensive Operationen ausgelöst werden, die aufgrund der Events aus den Services ausgelagert werden können und somit der *Überlastung eines Services* entgegenwirken. Diese implizite „Beauftragung“ kann aber auch die bereits erwähnte Unübersichtlichkeit erhöhen, die EDAs mit sich bringen (siehe Kapitel 4.1.3) – wenn hinter einem Event eine Erwartungshaltung steckt, sollte es als Command modelliert werden. Abgesehen davon sind Event Notifications ein leichtgewichtiges Muster, das bereits wesentliche Integrationsprobleme lösen kann [Fow17].

Die Einfachheit bringt jedoch auch Nachteile mit sich, insbesondere was die Verfügbarkeit und Erreichbarkeit der Services angeht. Weil der Kontext des Events nur beim Aufruf über die bestehenden Schnittstellen ersichtlich ist, müssen Produzent und Konsument gleichzeitig verfügbar sein und die Adresse, unter der die Entität abgerufen werden kann, im Event enthalten und für alle Konsumenten erreichbar sein – eine Anforderung, die in getrennten Netzen zur Herausforderung werden kann.

Darüber hinaus ist nach jeder Veröffentlichung einer Event Notification mit einer Lastspitze im produzierenden Service zu rechnen, der die Verfügbarkeit des Service beinträchtigen kann. Außerdem vergrößert sich durch die notwendige Anfrage das Zeitfenster der Eventual Consistency, was schlussendlich für zusätzliche Komplexität im System sorgen kann.

4.3.3 Event-Carried State Transfer

Die Schwächen, die Event Notifications haben, lassen sich im Wesentlichen mit dem fehlenden Kontext des Events begründen. Event-Carried State Transfer erweitert das vorherige Muster um genau diesen Kontext, den *State*. Statt einer Event Notification, dass eine Bestellung aufgegeben wurde, sind nun auch die Details über die Bestellung in dem Event enthalten – etwa die Zahlungsmethode oder die Lieferadresse. Überträgt man dieses Konzept auf einen üblichen CRUD-Service, würde neben der Information, *dass* eine Entität aktualisiert wurde, auch veröffentlicht werden, *was* genau aktualisiert wurde, also beispielweise der aktualisierte Zustand der betroffenen Entität. Diese Events können entweder direkt im Programmcode des Services erzeugt werden, beispielsweise in einer Controller-Methode oder dem Object-Relational Mapping (ORM) eines CRUD-Services, es gibt aber auch Ansätze, die Änderungen an Datensätzen ohne zusätzlichen Code im Service erkennen. Diese sogenannten Change-Data-Capture-Pattern (CDC-Pattern) beschreiben Möglichkeiten, Zustandsänderungen zu erkennen. Diese Erkennung sowie die Erstellung der jeweiligen Events erfolgt meistens außerhalb der Services und bedarf keiner Veränderung des eigentlichen Services – sie eignen sich daher auch insbesondere für Software von Dritten, deren Programmcode nicht verändert werden kann. Auf diese Pattern soll sich im Folgenden konzentriert werden: Der `query based` Ansatz, der erste von zwei wesentlichen, stellt in regelmäßigen Abständen Datenbankabfragen und erkennt nach unterschiedlichen Kriterien neue oder geänderte Datensätze. Weil diese zusätzlichen Abfragen für Last sorgen, wird oft der alternative `log based` Ansatz gewählt. Dieser basiert auf den AppendOnly-Logs der eingesetzten Datenbanken und extrahiert daraus die Veränderungen an den Datensätzen. Im Gegensatz zum `query based` Ansatz wird die Datenbank keiner zusätzlichen Last ausgesetzt, allerdings hängt die Machbarkeit von der verwendeten Datenbank und deren Log ab [Bel20].

Event-Carried State Transfer löst wie auch Event Notifications das Problem der *externen Relationen*, *Abhängigkeit von Konsumenten* und somit der *kaskadierenden Aufrufe* – die drei Probleme, die die meisten Eigenschaften beeinträchtigen. Da zusätzlich aber auch immer der Kontext des Events mitgesendet wird, müssen sich Services nicht mehr gegenseitig über die synchronen Schnittstellen aufrufen. Das Integrationsproblem der *blockierenden Aufrufe* tritt somit nicht mehr auf, außerdem müssen Services nicht gleichzeitig verfügbar sein und können in getrennten Netzen liegen. Das löst die *Punkt-zu-Punkt-Kopplung* auf. Darüber hinaus treten auch keine Lastspitzen mehr durch konsumierende

Services auf, wie es noch im Muster der Event Notifications der Fall war. Dass der Kontext eines Events immer mitgesendet wird, bedeutet im Zweifelsfall allerdings auch einen unnötigen Verbrauch von Speicherplatz, sollten die Events über längeren Zeitraum gespeichert werden und der Kontext überflüssig sein.

4.3.4 Event Sourcing

Die bisher vorgestellten Muster haben alle eine Gemeinsamkeit: Sie veröffentlichen Events als „Abfallprodukt“, zusätzlich zu der Logik des Services und seinen Datenbanktransaktionen. Die Daten werden in Datenbanktabellen gehalten und mit den typischen CRUD-Operationen verändert. Das bedeutet auch, dass ein Datensatz bei Aktualisierung überschrieben wird. Das Muster „Event Sourcing“ verfolgt hingegen einen komplett anderen Ansatz: Statt den aktuellen Zustand der Daten in Tabellen zu speichern, werden die Änderungen des Zustands in einem Stream von Events gespeichert. Da für jede Änderung ein Event veröffentlicht wird, kann der aktuelle Zustand aus der Summe aller Events gebildet werden. Dabei können Events nicht rückgängig gemacht werden: Weil es nicht gelöscht oder aktualisiert werden kann, existieren in der Regel „Compensating Events“, die die Auswirkungen eines Event rückgängig machen, ähnlich wie eine Korrekturbuchung auf einem Bankkonto. Um den aktuellen Zustand der Daten effizienter bilden zu können und nicht immer ab dem ersten Event seit Bestehen des Systems aufsummieren zu müssen, werden zusätzlich in regelmäßigen Abständen Snapshots wie in Abbildung 4.6 gebildet, die als Startpunkt für neue Berechnungen des aktuellen Zustands dienen [NMAM16].

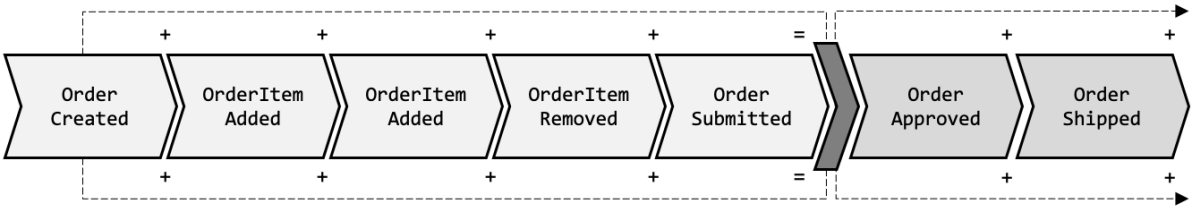

Abbildung 4.6: Verbesserung der Performance der Aufsummierung durch Snapshots.

Event Sourcing hängt als Muster nicht zwangsläufig mit EDA zusammen. Die Events müssen weder in einem Event-Broker veröffentlicht oder gespeichert werden, noch müssen sie asynchron verarbeitet werden. Stattdessen können übliche Datenbanken verwendet werden, mit denen das Paradigma umgesetzt werden kann. Das Paradigma als solches existiert also auch außerhalb von EDA und steht nicht zwangsläufig in Verbindung mit Integrationsmustern. Aufgrund des Fokusses auf Events als Grundlage der Datenhaltung eignet es sich aber besonders, diese Events auch zu veröffentlichen [Bel20].

Als Speicherort eignen sich neben klassischen Datenbanken auch sogenannte „Event Stores“ wie bspw. Event Store[25] oder AxonIQ[26], es können aber auch bereits vorgestellte Event-Broker wie Apache Kafka oder NATS verwendet werden – diese bieten allerdings keine Möglichkeit, Snapshots zu erstellen [Ric19].

Aus der Verwendung von Event Sourcing ergeben sich Vorteile in unterschiedlicher Hinsicht. Da Systeme nicht auf CRUD-Operationen basieren, können wesentlich domänenorientierte Operationen formuliert werden. Event Sourcing wird in der Literatur deshalb oft zusammen mit Domain-Driven Design erwähnt und eignet sich somit insbesondere für komplexe Domänen, bei denen das Problem der *Vereinheitlichung von Änderungen* ins Gewicht fällt. Da Datensätze nie überschrieben werden, tritt auch das Problem der *Verlust von Informationen* nicht auf. Das sorgt für eine wesentlich bessere Nachvollziehbarkeit, weil nicht nur der aktuelle Zustand, sondern auch jeder Schritt dahin festgehalten ist. Das ist insbesondere für Domänen interessant, in denen die Auditierung der Systeme eine Rolle spielt oder in Testinstanzen, in denen reale Event Streams wiederholt werden können, um Fehler nachzuvollziehen. Aber auch aus Integrationsaspekten bietet das Muster Vorteile: Während in den vorherigen Mustern neben einer Datenbanktransaktion zusätzlich ein Event veröffentlicht wurde, werden Events in diesem Muster standardmäßig veröffentlicht. Ein Szenario, in dem eine Datenbanktransaktion erfolgreich war, die Veröffentlichung des Events jedoch fehlgeschlagen ist, kann somit nicht auftreten. Im Gegensatz zu Systemen, die Events zusätzlich veröffentlichen und dies unter Umständen nur für einige, als relevant erachtete Änderungen tun, wird im Event-Sourcing-Ansatz für jede Änderung zwangsläufig ein Event produziert [Fow05]. Werden diese Events in einem Event-Broker veröffentlicht, ist die vom System abgebildete Domäne sehr zugänglich für die umliegenden Services und ermöglicht eine tiefe Integration auf Ebene des Ökosystems. Da das Muster Events zur Grundlage hat, löst es auch alle Integrationsprobleme, die bereits von den vorherigen Mustern gelöst wurden und kann damit, wie in Abbildung 4.4 ersichtlich, die meisten der vorgestellten Integrationsprobleme lösen.

Das Muster bringt neben seinen zahlreichen Vorteilen jedoch auch Nachteile mit sich, die allesamt unter der Überschrift „Gesteigerte Komplexität“ gefasst werden können. Die Herausforderungen sind unterschiedlichster Art und hängen mit dem genauen Einsatz des Musters ab, in diesem Absatz sollen deshalb nur einige mögliche Probleme genannt werden, um die Komplexität zu verdeutlichen. Da es sich bei dem Muster um einen kompletten Paradigmenwechsel handelt, ist die Lernkurve für Entwickelnde steil. Probleme wie gleichzeitige Befehle („Concurrent Commands“), die in herkömmlichen Datenbanken durch Transaktionen und Locking gelöst sind, müssen im Event Sourcing von den Konsumenten der Events gelöst werden. Auch die Prüfung auf die Einzigartigkeit eines bestimmten Attributwerts stellt sich als Herausforderung dar, wenn der aktuelle Zustand nicht in einer Tabelle liegt, sondern aus Events aufsummiert werden muss.

[25] `https://www.eventstore.com/`
[26] `https://axoniq.io/`

Diese Aufsummierung stellt Anwendende vor ein weiteres Problem: Wenn sich in der Lebenszeit der Applikation die Struktur eines Events verändert, muss dieses Event versioniert werden, um die Verarbeitung alter Events nicht zu gefährden. Für jede dieser Versionen muss dann Programmcode vorgehalten werden, der bei der Aufsummierung ausgeführt wird, um alte Versionen interpretieren zu können [Ric19]. Werden diese Herausforderungen adressiert, können mit diesem Muster leistungsstarke und domänenorientierte Anwendungen entwickelt werden.

4.3.5 Command-Query-Responsibility-Segregation

Ein Muster, was in oft in Verbindung mit dem zuvor eingeführten Event Sourcing genannt wird, ist Command-Query-Responsibility-Segregation (CQRS). Kern dieses Musters ist die Aufteilung und Trennung von lesenden und schreibenden Zugriffen auf zwei Services bzw. Schnittstellen. Der für die Lesezugriffe zuständige Query-Service kann dabei auch eine andere Datenbank verwenden, die z.B. für lesende Zugriffe oder Volltextsuche optimiert ist, während der Command-Service ein Datenmodell zugrunde legt, das sich besonders für schreibende Zugriffe und deren Validierung eignet. Ebenso wie Event Sourcing steht CQRS nicht zwangsläufig in Verbindung mit EDA und findet auch aus vielen anderen Gründen und in anderen Abstraktionsebenen Anwendung. Es bietet sich jedoch an, dieses Paradigma mit Event Sourcing oder Event-Carried State Transfer zu kombinieren, weil der Query-Service seine Datengrundlage aus den veröffentlichten Events bilden kann. Insbesondere in Anwendungen, die Event Sourcing verwendet werden, wird ein Service benötigt, der zur Laufzeit den aktuellen Zustand berechnet und anbietet. Event Sourcing und CQRS werden in der Praxis deshalb oft miteinander kombiniert. Der Command-Service produziert und veröffentlicht dann Events, die vom Query-Service konsumiert und zum aktuellen Zustand aufsummiert werden [NMAM16].

Wenn lesende und schreibende Zugriffe von unterschiedlichen Services behandelt werden, können diese auch separat voneinander skaliert werden. Das ist besonders dann praktisch, wenn Lese- oder Schreibzugriffe in stark unterschiedlicher Häufigkeit vorkommen. Außerdem können die Services unterschiedliche Daten halten: Während der Query-Service externe Relationen anbieten möchte und diese deshalb in seine Datenhaltung integriert, können für die Validierung eingehender Mutation im Command-Service andere Informationen relevant sein. Ein Query-Service, der Events aus dem Ökosystem konsumiert, kann Daten dabei effektiver zusammenführen als das in Kapitel 3.3.1 erwähnte API-Gateway. Diese Separierung verbessert außerdem die *hohe Kohäsion* der Services. Im Gegenzug wird die Komplexität jedoch weiter erhöht. Die Aufteilung der Geschäftslogik auf zwei Services bringt wie auch Event Sourcing eine hohe Lernkurve mit.

In diesem Muster wird außerdem die in Kapitel 4.1.3 eingeführte Eventual Consistency zur Herausforderung. Wenn ein Client nach einem schreibenden Zugriff die soeben erstellte Entität abrufen möchte, kann diese aufgrund der Verzögerung noch nicht im Query-Service verfügbar sein [Ric19].

4.4 Kritische Bewertung der zur Verfügung stehenden Integrationsansätze

Die soeben vorgestellten Integrationsmuster sollen gemeinsam mit den nötigen Broker-Lösungen, die bereits in Kapitel 4.2 dargestellt wurden, einer kritischen Bewertung unterzogen werden. Weil die allgemeinen Vorteile von EDAs sowie die Vor- und Nachteile der einzelnen Muster bereits dargelegt wurden, konzentriert sich dieses Kapitel auf die Defizite, die der Stand der Technik bei einer Einführung in bestehende, größtenteils ressourcenorientierte Ökosysteme aufweist. Auf Grundlage dieser kritischen Bewertung werden im nachfolgenden Kapitel dann Kriterien erarbeitet, die an pragmatische und offen zugängliche Integrationsansätze gestellt werden sollen.

4.4.1 Kritik an vorgestellten Integrationsmustern

Der erste Teil der Bewertung beschäftigt sich vor allen Dingen mit den Integrationsmustern des vorigen Kapitels. Diese werfen drei Aspekte auf, die im Folgenden näher erläutert werden:

1. unverhältnismäßige Komplexität für einen Großteil der Applikationen
2. hohe Einführungskosten für ein Ökosystem
3. mangelnde Domänenorientierung bei CDC-Pattern

Die letzten beiden vorgestellten Integrationsmuster Event Sourcing und CQRS können mit Abstand die meisten der beschriebenen Integrationsprobleme (siehe Abbildung 4.4) lösen. Die sich grundlegend von klassischen Anwendungen unterscheidende Architektur bringt jedoch eine Komplexität mit, deren Vorteile nur dann ausgespielt werden können, wenn komplexe Domänen abgebildet werden, die von der Domänenorientierung der Muster stark profitieren. Event Streams statt klassische CRUD-Datenbanken als Speicherort zu betrachten erfordert eine große Umstellung für Systeme und Entwickelnde. Eine Einführung bzw. der Prozess, möglichst viele Services in einem Ökosystem in die Lage zu versetzen, Events zu produzieren und zu konsumieren wird dadurch ebenfalls aufwendig: Wenn diese Muster als Vorbild für eine systemweite Einführung genommen werden, entstehen große Kosten durch nötige Schulungen oder Systemumstellungen.

Dementsprechend eignen sich diese Muster vornehmlich für Neuentwicklungen, denen eine entsprechende Komplexität zugesprochen wird. Das Muster als solches ist also nicht grundsätzlich ungeeignet, aber es eignet sich nicht für eine Einführung in ein gesamtes Ökosystem. Attraktiver scheint hingegen Event-Carried State Transfer unter Anwendung von CDC-Pattern. Die geringe Einstiegshürde kann eine Einführung deutlich vereinfachen, allerdings orientieren sich die vorgestellten Möglichkeiten, Änderungen als Events zu veröffentlichen stark an der Struktur der Daten innerhalb der Datenbank. Wenn diese wie üblich in der dritten Normalform vorliegen, werden die Events in einer Granularität veröffentlicht, wie sie weder ressourcenorientiert noch im Ansatz domänenorientiert geschnitten sind. CDC-Pattern stellen somit zwar eine geeignete Möglichkeit dar, Events mit wenig Aufwand zu erzeugen, müssen aber domänenorientierter arbeiten, um anderen Services einen Mehrwert zu bieten und für die Integration genutzt zu werden.

4.4.2 Kritik am Status quo der Event-Broker-Infrastruktur

Im zweiten Teil der Bewertung wird der Fokus auf die Infrastrukturkomponenten gerichtet, die im Zuge der Einführung von EDA nötig sind. Es wurden vier Aspekte identifiziert, die im Folgenden näher erläutert werden:

1. Bibliothek für Anbindung an Event-Broker erforderlich
2. tiefe Integration des Broker-Protokolls in Service nötig
3. unterschiedliches Nachrichtenformat je Event-Broker
4. mangelnde Übersichtlichkeit durch lose Kopplung

Wenn ein Event-Broker in ein Ökosystem eingeführt wird, um Events als zusätzliche Option zur Integration von Services zu etablieren, müssen diese Services an den Event-Broker angebunden werden. Je nach Broker funktioniert dies über unterschiedliche Protokolle. Broker unterstützen zwar neben ihrem proprietären Protokoll durch Plugins auch standardisierte Protokolle wie AMQP oder MQTT, allerdings werden auch über diese Protokolle proprietäre Attribute gesetzt. Die Anbindung eines Services erfordert dementsprechend eine Bibliothek in der jeweiligen Programmiersprache, um über den Broker kommunizieren zu können. Das ist ein wesentlicher Unterschied zu der Integration via REST, die ausschließlich einen HTTP-Client benötigt. Das kann dazu führen, dass ein bestimmtes Event-Broker-Produkt allein deshalb nicht zur Option steht, weil für eine in der Organisation verbreitete Programmiersprache keine entsprechende Bibliothek existiert. In heterogenen Systemlandschaften ist davon auszugehen, dass immer Systeme existieren, für deren genutzte Programmiersprache keine Bibliothek vorhanden ist und die Systeme somit von der direkten Anbindung an die EDA ausgeschlossen sind. Doch auch wenn eine solche Bibliothek für Services existiert, ist die tiefe Integration in den

Service zu kritisieren: Wenn brokerspezifische Logik direkt in den Services implementiert wird, ist der Austausch eines etablierten Event-Brokers mit erheblichem Aufwand verbunden, weil die Bibliothek und Anbindung im Service mitsamt der proprietären Attribute ausgetauscht werden müssen. Darüber hinaus gestaltet sich die Anbindung eines weiteren Brokers ebenso schwierig. Die beiden ersten Kritikpunkte führen somit zu einer deutlichen Abhängigkeit von einem Event-Broker.

Ferner stellen die unterschiedlichen Nachrichtenformate der Broker eine Herausforderung dar: In verteilten Systemen und heterogener Systemlandschaft kann nicht davon ausgegangen werden, dass Events nur über einen einzelnen Event-Broker verteilt werden. Stattdessen können mehrere, in ihrer Implementierung und ihrem Funktionsumfang unterschiedliche Broker aus diversen Gründen in der Infrastruktur existieren. Unter Umständen müssen Events durch mehrere dieser Broker fließen, um beim Konsumenten anzukommen. Weil die unterschiedlichen Broker eigene Formate für die Events verwenden, müssten Events bei der Einspeisung transformiert werden, um der korrekten Struktur zu entsprechen. Auch dieser Kritikpunkt sorgt somit für eine Abhängigkeit des Ökosystems von einem bzw. mehreren bestimmten Event-Broker und ist mit zusätzlichen Aufwänden bei der Integration verbunden. Der vierte Kritikpunkt orientiert sich im Wesentlichen an der bereits identifizierten Herausforderung in Kapitel 4.1.3: Weil die Kommunikation in EDAs nicht direkt erfolgt und die Produzenten von Events ihre Konsumenten nicht kennen, neigen ereignisorientierte Systeme zu Unübersichtlichkeit. Diese mangelnde Transparenz innerhalb der Systemlandschaft kann zu organisatorischen Problemen bei der Verwaltung der asynchronen Schnittstellen führen.

Zusammenfassend sind am Stand der Technik von EDAs zwei übergreifende Aspekte zu kritisieren: Auf Ebene der Integrationsmuster ist **Pragmatismus** nötig, damit eine Einführung von Events in ein Ökosystem mit vertretbarem Aufwand erfolgen kann. In puncto Infrastruktur muss neben der Übersichtlichkeit vor allen Dingen die **Zugänglichkeit** für alle Anwendungen gewährleistet werden, damit Events auch in heterogenen Ökosystemen akzeptiertes Mittel zur Integration werden.

5 Event-Driven Microservices durch lösungsorientierte Integrationsmuster

In den bisherigen Kapiteln wurden die theoretischen Grundlagen für Microservices, ihre Integration über REST sowie die daraus entstehenden Integrationsprobleme vorgestellt. Daraufhin wurden im vorigen Kapitel EDAs eingeführt, die sich als aussichtsreichste Lösung für diese Probleme anbieten und schlussendlich einer kritischen Bewertung unterzogen. Auf Basis all dieser drei Kapitel kann ab diesem Kapitel damit begonnen werden, eigene Ansätze vorzustellen, die Konzepte der vorgestellten Grundlagen anwenden und die Integrationsprobleme aus Kapitel 3.1 lösen.

5.1 Kriterien für pragmatische und zugängliche Integrationsansätze

Aus den im vorigen Kapitel genannten Kritikpunkten am Stand der Technik für EDAs sollen nun Kriterien abgeleitet werden, die den nötigen **Pragmatismus** in Integrationsmustern und die **Zugänglichkeit** der Infrastruktur für Integrationsansätze festlegen.

5.1.1 Anforderungen an Integrationsmuster

Aus den Kritikpunkten, die bzgl. der Integrationsmuster geäußert wurden, entstehen drei konkrete Kriterien, die bei einer Einführung von EDA in heterogene, ressourcenorientierte Ökosysteme relevant sind und nachfolgend näher erläutert werden:

1. geringe Einführungskosten
2. pragmatische, domänenorientierte Lösungen
3. unterschiedliche Ausbaustufen

Um einen Service in die Lage zu versetzen, Events zu produzieren und zu konsumieren, soll möglichst wenig Aufwand nötig sein. Die Integration über REST dient hierbei als gutes Beispiel: Eine REST-API kann zusätzlich zu der bereits bestehenden Geschäftslogik angeboten werden, ohne die Anwendung grundlegend umstellen zu müssen. Je geringer die Einstiegshürde für Services ist, erste Events zu publizieren, desto besser ist die Akzeptanz und Verbreitung von Ereignisorientierung in der gesamten Systemlandschaft. Diese niedrigen Einführungskosten schaffen im Ökosystem eine Grundlage, in der Services bereits rudimentär über Events integriert werden können. Diese geringen Einführungskosten werden durch das zweite Kriterium gestützt: Ökosysteme, die in der Vergangenheit durch REST-APIs ressourcenorientiert geprägt waren, müssen pragmatisch an die Ereignisorientierung herangeführt werden. Das heißt auch, dass Services mit der Einführung von Events nicht direkt ereignisorientiert arbeiten, sondern ihre Geschäftslogik nach wie vor in Ressourcen schneiden und zusätzlich Events veröffentlichen. Dafür eignen sich am ehesten CDC-Pattern – ein Ansatz, der die Ressourcenorientierung von REST-Services berücksichtigt, wird im Kapitel 5.2 eingeführt.

Aufbauend auf diesen ersten Integrationsmustern, die eine kostengünstige Einführung ermöglichen, können sich ressourcenorientierte Services bei Bedarf in Richtung Ereignisorientierung bewegen. Dafür müssen unterschiedliche Ausbaustufen angeboten werden, die von den ersten, niedrigschwelligen Mustern für kleine, einfache Services bis hin zu komplexen Anwendungen reichen, die von Event Sourcing Gebrauch machen möchten. Diese Ausbaustufen ermöglichen neben dem kostengünstigen Einstieg eine bedarfsorientierte Weiterentwicklung im Lebenszyklus eines Services.

5.1.2 Kriterien für offen zugängliche Eventing-Infrastruktur

Unter Berücksichtigung der Kritikpunkte an den Infrastrukturkomponenten ergeben sich drei zentrale Kriterien, die eine Einführung von EDA für das gesamte Ökosystem zugänglich machen:

1. offene Standards für alle Systeme
2. Unabhängigkeit von einem einzelnen Broker
3. Übersicht über alle Produzenten, Konsumenten und ihre Events

Das erste und elementare Kriterium sind standardisierte Protokolle, die allen Services der Systemlandschaft Zugang zur Eventing-Infrastruktur bieten. Das schließt neben Microservices auch bestehende Anwendungen ein, die in einer klassischen dreischichtigen Architektur betrieben werden und im besten Falle sogar Altanwendungen, die bspw. auf Großrechnern betrieben werden. Wenn EDA als Alternative zu REST-APIs in einer historisch gewachsenen, heterogenen Systemlandschaft eingeführt werden soll, um die

Schwächen der synchronen Schnittstellenarchitektur auszugleichen, ist ein ähnlich offener Standard wie REST über HTTP für die Akzeptanz und Verbreitung unverzichtbar.

Das zweite Kriterium adressiert die Kritik an unterschiedlichen Nachrichtenformaten und der tiefen Integration von Event-Brokern in Services: Event-Broker sind eine zentrale Komponente in der Infrastruktur der Systemlandschaft, die über lange Zeit existieren und mit vielen Services kommunizieren wird. Eben weil die Komponente so zentral ist, sollte es zu keinen Altlasten bzw. Einschränkungen kommen, die damit begründet sind, dass sich der Broker schwer austauschen lässt. Services, die Events produzieren und konsumieren, müssen deshalb in der Lage sein, den Broker als Intermediär für Events mit wenig Aufwand wechseln zu können. Ebenso sollte die Struktur eines Events vom Event-Broker entkoppelt werden, um den Aufwand bei einer Umstellung oder dem Transport über mehrere Broker hinweg zu reduzieren. Weil der Funktionsumfang und die Einsatzszenarien der verschiedenen Event-Broker jedoch starke Unterschiede aufweisen, wird eine vollständige Unabhängigkeit nicht möglich sein. Nichtsdestotrotz ermöglichen Maßnahmen, die die Unabhängigkeit fördern, dass die Infrastruktur auch nach mehreren Jahren noch so angepasst werden kann, wie es den Anforderungen der Domäne und der Anwendungen entspricht.

Mit dem dritten Kriterium wird sichergestellt, dass die Einführung von ereignisorientierter, entkoppelter Integration und die damit einhergehende Unübersichtlichkeit nicht zu organisatorischen Problemen bei der Verwaltung von asynchronen Schnittstellen führt. Je nach Produkt können Event-Broker ggf. eine Nutzungsmatrix generieren, die für jedes Event-Topic Produzenten und Konsumenten auflistet und somit einen Teil dieser Anforderung abdeckt. In einer Systemlandschaft, in der unter Umständen aber mehrere Event-Broker existieren, braucht es neben diesen technischen Lösungen aber auch Ansätze, die das gesamte Ökosystem einbeziehen. Auch hier kann nach dem Vorbild von REST-Ökosystemen gearbeitet werden: Unter dem Stichwort „API-Management“ werden diverse Werkzeuge verstanden, die z.B. einen Katalog von allen APIs einer Organisation inklusive Schnittstellenbeschreibung und automatisiertem Monitoring mittels eines API-Gateways umfassen. Ähnliche Ansätze und Werkzeuge können auch für asynchrone Schnittstellen die Transparenz über ausgetauschte Daten erhöhen und damit die Verwaltung der Schnittstellen vereinfachen.

Auf Basis dieser insgesamt sechs Kriterien an Integrationsmuster und Infrastrukturkomponenten soll im Rest des Kapitels 5 und im Kapitel 6 Ansätze erarbeitet werden, die diesen Kriterien genügen. Eine Grundlage für die Anforderungen an Integrationsmuster stellt dabei eine Middleware dar, die im folgenden Kapitel eingeführt wird.

5.2 Entwicklung einer Middleware als Event-Produzent

Bevor die unterschiedlichen Ausbaustufen von Integrationsmustern vorgestellt werden, soll eine Middleware vorgestellt werden, die sich insbesondere dafür eignet, Eventing in Services einzuführen, die eine REST-API anbieten. Dafür wird zwischen den Client und den REST-Service eine Middleware geschaltet, die den ein- und ausgehenden Datenverkehr analysiert und als ein Proxy durchleitet. Aufgrund der Informationen, die in der HTTP-Anfrage und Antwort enthalten sind, können dann Events erzeugt werden, die die Änderungen, die durch den API-Aufruf versucht wurden, enthalten. Die Middleware wird im Folgenden deshalb als „REST-Events-Middleware“ bezeichnet.

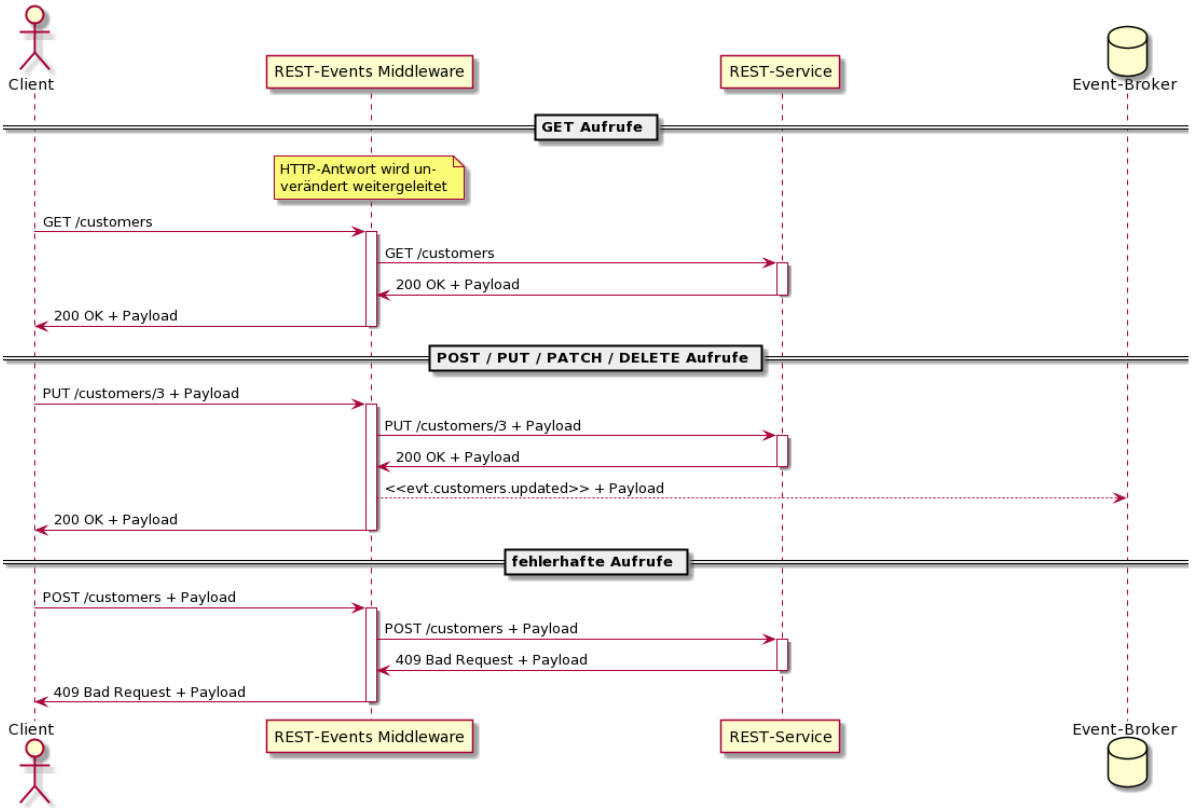

Abbildung 5.1: Einführung einer REST-Events-Middleware als CDCP in REST-Ökosystemen.

Wie in Abbildung 5.1 ersichtlich wird, stellen die HTTP-Verben und Statuscodes, die bei REST-APIs verwendet werden (siehe Kapitel 2.3.1) die Grundlage für die Erkennung von Mutationen dar: Im Falle eines `GET`-Aufrufs werden nur Daten abgefragt, weshalb die Middleware den Aufruf ohne weitere Aktion durchleitet. Auch im Falle einer fehlgeschlagenen Anfrage, die mit einem HTTP-Statuscode größer `299` beantwortet wird,

wird die Middleware nicht tätig. Werden Anfragen jedoch mit den Verben `POST`, `PUT`, `PATCH` oder `DELETE` mit einem HTTP-Statuscode von `200 - 299` beantwortet, wurden im REST-Service Daten verändert. Der neue Zustand des Datensatzes wird nach dem REST-Paradigma im HTTP-Body gesendet und kann somit von der Middleware ausgelesen werden. Auf Basis dieser Informationen kann dann ein Event erzeugt werden: Aus der aufgerufenen Ressource in dem Pfad und dem HTTP-Verb wird ein Topic gebildet. So wird nach dem Beispiel der Abbildung aus der Anfrage `PUT /customers/3` das Topic `evt.customers.updated` erzeugt. Zusätzlich kann aus dem Pfad die ID der betroffenen Entität entnommen werden. Für den Fall, dass die automatische Erzeugung aus den Informationen des HTTP-Aufrufs nicht ausreicht, können zusätzlich Mappings hinterlegt werden, die bspw. für bestimmte Pfade alternative Namen für Ressourcen bestimmen.

Die REST-Events-Middleware kann an unterschiedlichen Stellen implementiert werden. Falls Änderungen am Programmcode des Services möglich sind, kann die Middleware in den HTTP-Controller integriert werden, andernfalls bietet es sich an, die Middleware als eigenen Service innerhalb der Infrastruktur bereitzustellen. Wenn ein API-Gateway eingesetzt wird, besteht je nach Produkt auch die Möglichkeit, an dieser zentralen Stelle eine eigene Middleware zu hinterlegen. Als dezentrale Lösung bietet sich ein Proxy an, der die Aufrufe an Stelle des REST-Services annimmt und dann durchleitet. Die konkrete Implementierung und Bereitstellung in der Infrastruktur wird in Kapitel 7.3.3 erläutert.

Die vorgestellte Middleware folgt dem Vorbild bereits vorsgestellter CDC-Pattern. Im Gegensatz zu bereits vorgestellten Pattern werden die Änderungen jedoch nicht anhand der Datenbank, sondern aufbauend auf Anfragen an die REST-API festgestellt. Sie eignet sich damit insbesondere für Systemlandschaften, deren Schnittstellenlandschaft durch REST dominiert wird und soll im folgenden Kapitel in den dafür entwickelten Integrationsmustern eine zentrale Rolle spielen.

5.3 Konzeptionierung pragmatischer Integrationsmuster

Auf Grundlage der eben vorgestellten Middleware werden in diesem Kapitel vier Ausbaustufen von Integrationsmustern vorgestellt, die Ereignisorientierung in ressourcenorientierten Systemlandschaften einführen. Die Vorschläge sind ihrer Komplexität und Fokussierung auf Events nach sortiert.

Abbildung 5.2 verdeutlicht, welche der in Kapitel 3.1 erwähnten Integrationsprobleme durch die einzelnen Muster gelöst werden können und erweitert damit ein letztes Mal den Aufbau der Arbeit aus Abbildung 1.1. Auch in dieser Abbildung sind die schwerwiegendsten Integrationsprobleme mit einem roten Ausrufezeichen versehen, während das

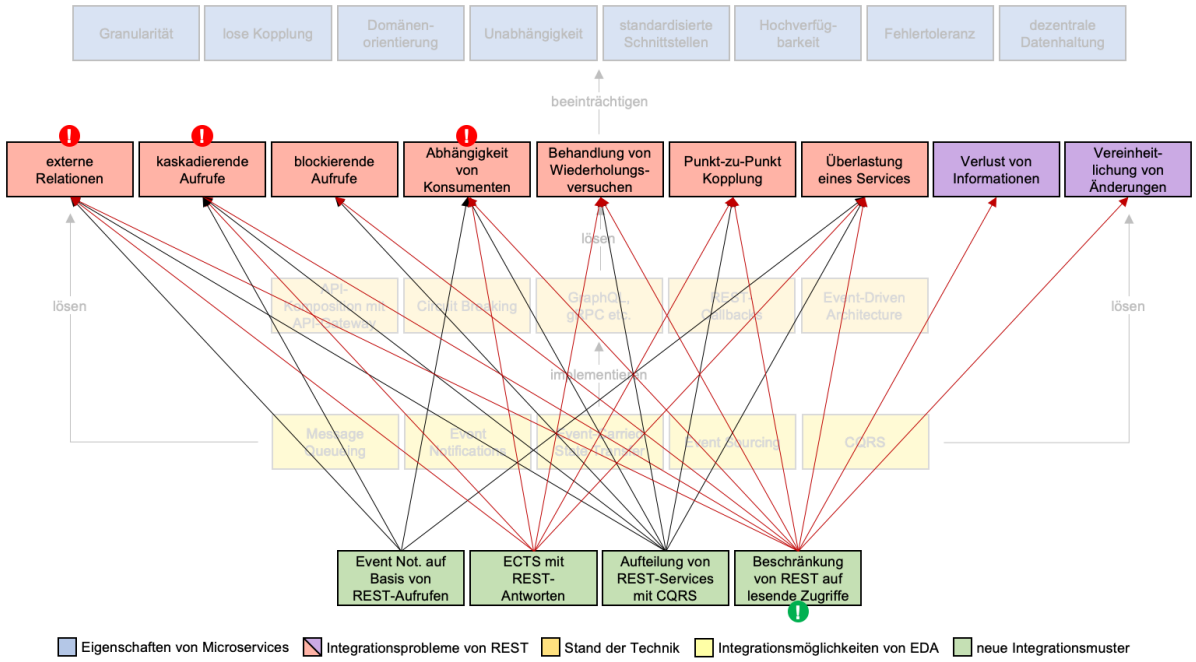

Abbildung 5.2: Auswirkungen der neuen Integrationsmuster auf die Integrationsprobleme.

Integrationsmuster, das die meisten Probleme lösen kann, mit einem grünen Ausrufezeichen versehen ist. Abschließend werden weitere Muster bzw. mögliche Kombinationen bereits vorgestellter Muster dargelegt.

Für jedes der entwickelten Integrationsmuster wird ein Architekturbild vorgestellt, welches einen zentralen CUSTOMERS-Service darstellt, auf den das Integrationsmuster angewandt wird. Darüber werden vereinfachte konsumierende Services dargestellt, die der Veranschaulichung dienen.

5.3.1 Event Notifications auf Basis von REST-Aufrufen

Das erste Muster wendet die in Kapitel 4.3.2 vorgestellten Event Notifications und die REST-Events-Middleware an: Die Middleware erkennt die Mutation von Daten im Service wie beschrieben durch die HTTP-Aufrufe und veröffentlicht Events, die nach dem Vorbild des angewendeten Musters allerdings keine Informationen über den veränderten Zustand enthalten. Die tatsächliche Datenübertragung und Kommunikation der Services erfolgt somit nach wie vor über die REST-Schnittstellen der Services.

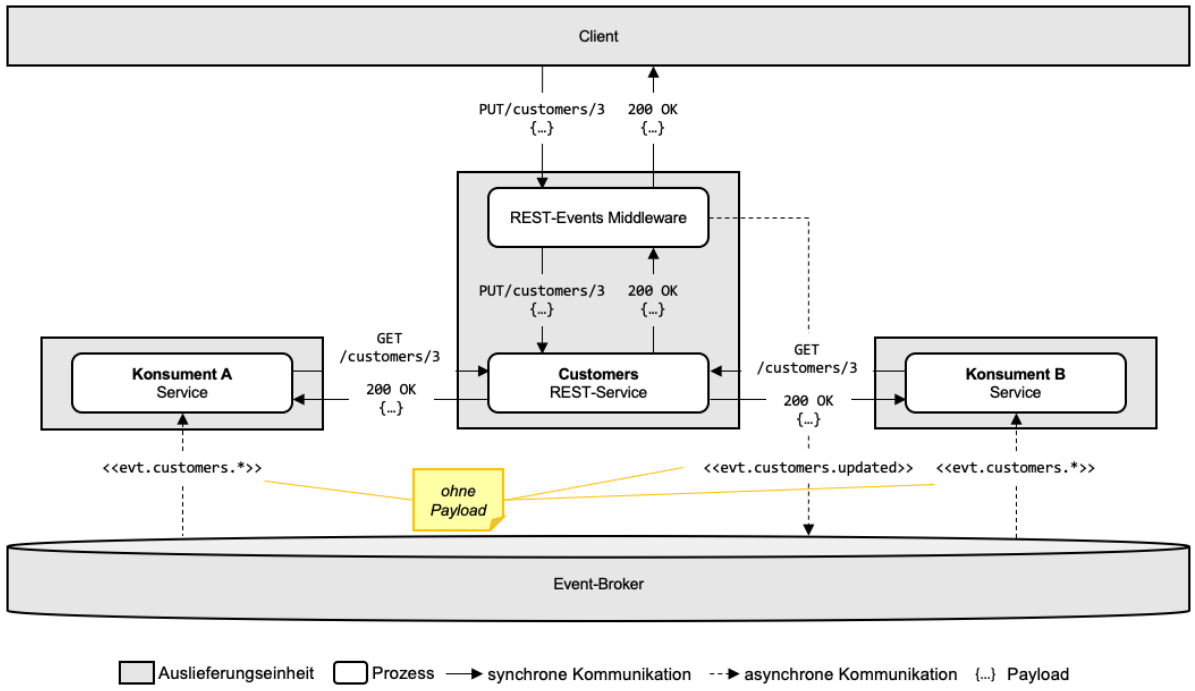

Abbildung 5.3: Integrationsmuster 1: Event Notifications auf Basis der REST-Events-Middleware.

Dieses Muster, das in Abbildung 5.3 visualisiert ist, bringt mehrere Vorteile mit sich. Da der eigentliche Datenaustausch weiterhin über die REST-APIs stattfindet, sind keine Schnittstellenbeschreibungen nötig, die die Struktur von Events dokumentieren. Die Events müssen außerdem weder versioniert werden, noch ist eine Nutzungsmatrix von produzierenden und konsumierenden Services nötig – all diese Anforderungen werden bereits durch das REST-Ökosystem erfüllt. Die Nachteile, die Event Notifications mit sich bringen, bleiben jedoch bestehen: Es muss nach wie vor sichergestellt werden, dass der Produzent und Konsument der Events gleichzeitig verfügbar und erreichbar sind – auch die Lastspitzen nach einem Event stellen weiterhin ein Problem dar. Abbildung 5.2 verdeutlicht, dass dieses Muster trotzdem bereits die drei Integrationsprobleme löst, die in Kapitel 3.2 als besonders relevant identifiziert wurden: *externe Relationen*, *kaskadierende Aufrufe* und *Abhängigkeit von Konsumenten*. Somit wird eine *dezentrale Datenhaltung* der Services ermöglicht. Darüber hinaus können auf Basis der Event Notifications bereits ressourcenintensive Prozesse in eigene Services ausgelagert werden, deren Ausführung durch die Events getriggert wird. Durch diesen Einsatz von Message Queueing wird zusätzlich einer Überlastung des eigentlichen Services bei vielen parallelen Anfragen vorgebeugt.

5.3.2 Event-Carried State Transfer mit REST-Antworten

Die zweite Ausbaustufe ergänzt die Events, wie sie im vorigen Muster veröffentlicht wurden, um den Zustand der veränderten Entität und kombiniert damit den in Kapitel 4.3.3 vorgestellten Event-Carried State Transfer und die REST-Events-Middleware.

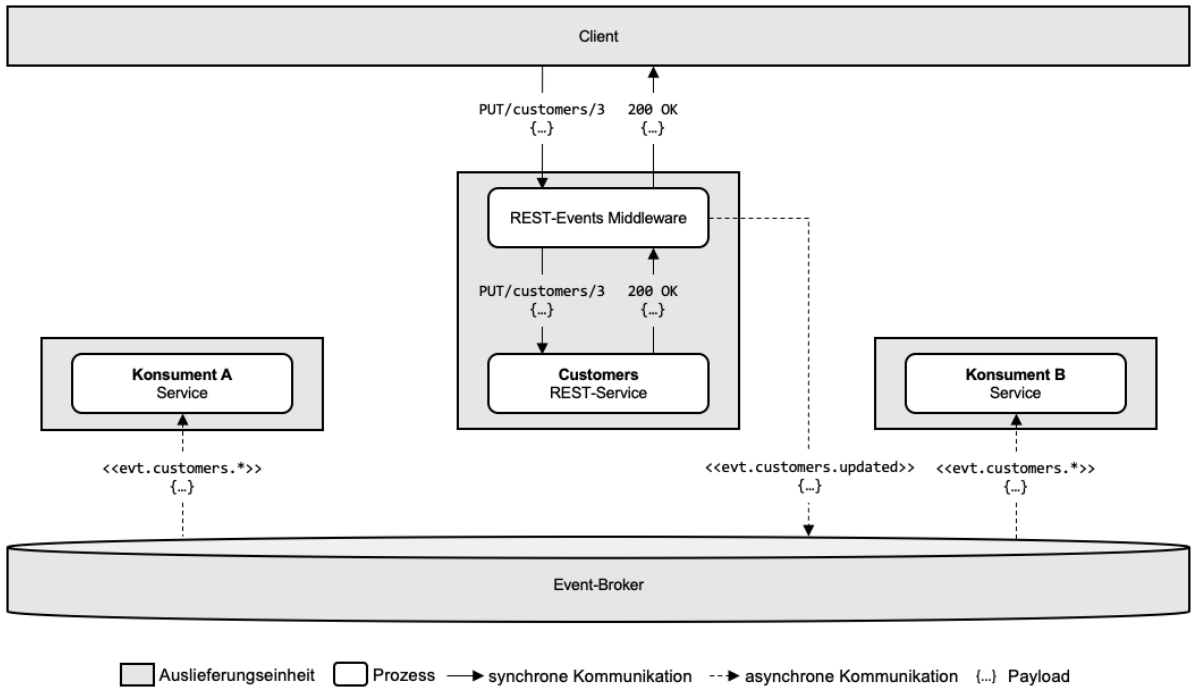

Abbildung 5.4: Integrationsmuster 2: Event-Carried State Transfer auf Basis der REST-Events-Middleware.

Wie in Abbildung 5.4 deutlich wird, wird mit dem veröffentlichten Event auch eine JSON-Struktur mit dem veränderten Zustand als Payload versendet. Dieser kann dem HTTP-Body entnommen werden, den der REST-Service als Antwort auf eine HTTP-Anfrage sendet. Das setzt voraus, dass dieser Body eine vollständige Repräsentation der aktualisierten Entität darstellt. Eignet sich die Repräsentation nicht, weil dem Client bspw. nur eine Untermenge von Attributen gesendet wird, kann die REST-Events-Middleware den Zustand der aktualisierten Entität in einem separaten Aufruf des Services abfragen, um dann das Event zu erstellen. Kommen diese Anpassungen nicht infrage, muss auf das erste Muster zurückgegriffen und auf den Zustand im Event verzichtet werden.

Die Kombination von Event-Carried State Transfer und der REST-Events-Middleware löst neben den Problemen, die bereits durch das vorige Muster gelöst wurden auch die *Behandlung von Wiederholungsversuchen* und *blockierenden Aufrufe*, weil die Kommunikation über die REST-Schnittstellen zum Datenaustausch nicht mehr nötig sind. Damit werden auch die Nachteile des vorigen Musters ausgeglichen. Die Erreichbarkeit der Services sowie die gleichzeitige Verfügbarkeit sind keine Bedingung mehr für einen Austausch der aktualisierten Entitäten. Trotzdem sind wenig Anpassungen nötig, um das Muster in bestehende REST-Ökosysteme zu integrieren. Weil die Struktur der Events der Struktur der REST-Antworten entspricht, ist keine gesonderte Beschreibung und Versionierung der Events notwendig. Eine IDL für die asynchrone Schnittstelle des Services kann nach wie vor Sinn machen, sollte dann aber Verweise auf die OpenAPI-Spezifikation des Services haben.

Durch die REST-Events-Middleware werden ausschließlich erfolgreiche HTTP-Anfragen in Events verwandelt. Eine Stornierungsanfrage, die vom Service abgelehnt wird und deshalb einen HTTP-Statuscode größer `299` hat, kann jedoch ein fachlich relevantes Event sein, an dem andere Services Interesse haben. Um diese fachlich relevanten Ereignisse trotzdem zu identifizieren, kann die REST-Events-Middleware angepasst werden und die relevanten Aufrufe auf Basis der Fehlermeldung erkennen. Darüber hinaus wird auch mit diesem Muster kaum ereignisorientiert gearbeitet, sondern nach wie vor Ressourcenorientierung angewandt, weil die vom Service erzeugten Events auf der REST-Schnittstelle basieren. Somit bleiben die Modellierungsschwächen von REST aus Kapitel 3.1.2 bestehen. Events einer Entität, die über die CRUD-Operationen hinaus gehen, können somit höchstens über die in Kapitel 2.3.1 erwähnten Subressourcen modelliert werden – das Muster ist also weniger geeignet für Anwendungen, die eine stark ereignisgetriebene Domäne abbilden.

5.3.3 Aufteilung von REST-Services mit CQRS

Einen weiteren Schritt in Richtung dieser gewünschten Ereignisorientierung stellt das CQRS-Pattern dar. Um CQRS in einem REST-Ökosystem zu etablieren, soll in diesem Muster eine weitere Middleware eingeführt werden, die es ermöglicht, HTTP-Anfragen mit HTTP-Verb `GET` an den Query-Service und Anfragen mit den Verben `POST`, `PUT`, `PATCH` oder `DELETE` an den Command-Service einer CQRS-Architektur weiterzuleiten. Das Sequenzdiagramm in Abbildung 5.5 zeigt, wie die CQRS-Middleware vor die zwei Services geschaltet werden kann. Dadurch ist es möglich, dem Client eine einheitliche REST-API anzubieten – die Aufteilung in zwei Services bleibt verborgen.

Der Command-Service veröffentlicht bei schreibenden Zugriffen für jede geänderte Entität ein Event, bspw. über die REST-Events-Middleware, wie in Abbildung 5.6 auf der rechten Seite visualisiert. Diese Events werden dann vom Query-Service konsumiert, der

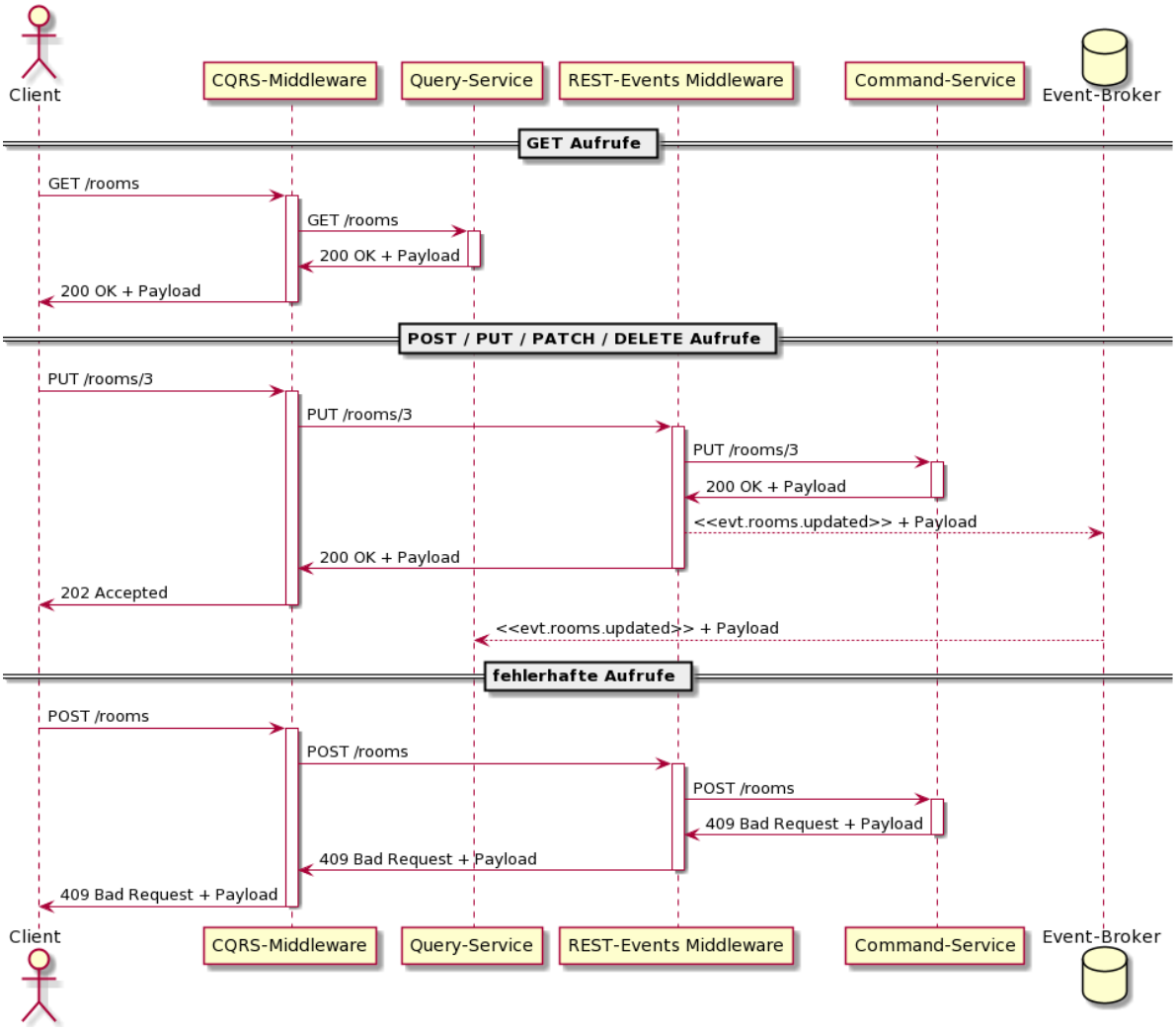

Abbildung 5.5: Sequenzdiagramm zur Kommunikation zwischen CQRS- und REST-Events-Middleware mit dem aufgeteilten REST-Service.

darüber hinaus auch noch Events anderer Services beziehen kann, um seine Datengrundlage zu bilden. Diese Datengrundlage wird dann bei lesenden Zugriffen abgefragt (siehe Abbildung 5.6 links).

Die Vorteile der Aufteilung von REST-Services in zwei einzelne Services ähneln den Vorteilen des CQRS-Pattern. Durch die Separierung kann der Query-Service flexibel auf die Anforderungen der Clients angepasst werden und zusätzliche externe Relationen aus anderen Services anbieten, während der Command-Service nur die Entitäten vorhält, die für die Geschäftslogik relevant sind. Das ermöglicht im Falle von Kaufsoftware, deren Programmcode nicht angepasst werden kann, die Möglichkeit, einen zusätzlichen, eigens

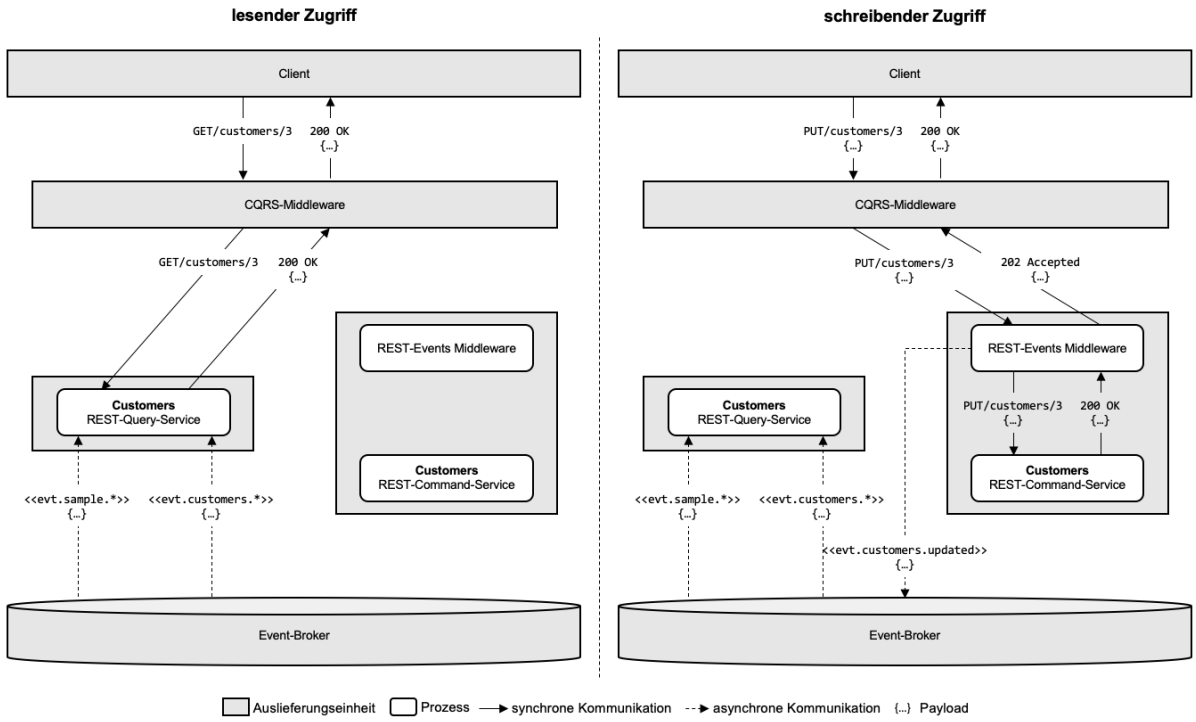

Abbildung 5.6: Integrationsmuster 3: Aufteilung von REST-Services mittels einer CQRS-Middleware.

implementierten Query-Service neben den bestehenden Service zu stellen, um die Daten flexibler abfragen zu können. Darüber hinaus können die Services getrennt voneinander skaliert werden. Diese getrennte Skalierung kann der *Überlastung eines Services* vorbeugen und somit die *Hochverfügbarkeit* des Services sicherstellen. Darüber hinaus löst die Kombination von Event-Carried State Transfer, der REST-Events-Middleware und der CQRS-Middleware alle Problemen, die bereits durch das vorige Muster gelöst wurden – muss aber nach wie vor mit den Modellierungsschwächen von REST auskommen. Es stellt jedoch den idealen Zwischenschritt dar, um sich für schreibende Zugriffe vom REST-Paradigma zu trennen – und damit auch schrittweise von der Ressourcenorientierung.

5.3.4 Beschränkung von REST auf lesende Zugriffe durch CQRS

Mit den bisher vorgestellten Mustern konnten alle Integrationsprobleme gelöst werden, die in Kapitel 3.1 vorgestellt wurden – mit Ausnahmen der Probleme, die auf das REST-Paradigma und dessen Ressourcenorientierung zurückzuführen sind. Um auch diese letzten beiden Probleme lösen zu können, soll in dieser vierten Ausbaustufe ein Integrati-

onsmuster vorgeschlagen werden, das am CQRS-Ansatz festhält, allerdings nur noch den Query-Service nach dem REST-Paradigma designt. Wie in Abbildung 5.7 ersichtlich, ist dementsprechend keine CQRS-Middleware mehr nötig: Die Services sind komplett unabhängig voneinander und werden dem Client auch als unterschiedliche Services verfügbar gemacht.

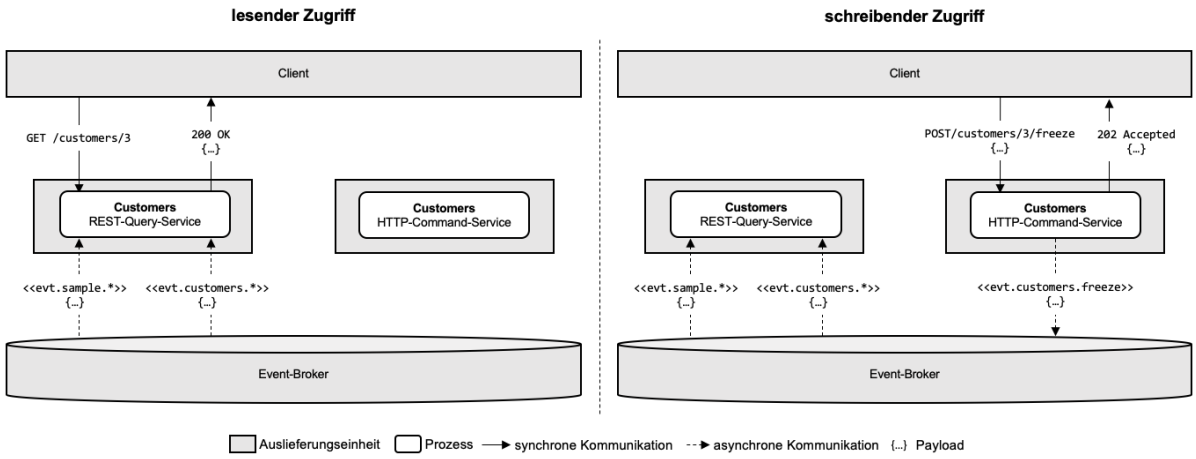

Abbildung 5.7: Integrationsmuster 4: Beschränkung des REST-Paradigmas auf den Query-Service durch CQRS.

Für den Command-Service soll in dieser vierten Ausbaustufe kein konkretes Paradigma vorgeschlagen werden, allerdings müssen gewisse Eigenschaften erfüllt sein: Um die *Vereinheitlichung von Veränderungen* zu verhindern, arbeitet der Command-Service ereignisorientiert und bietet domänenorientierte, über CRUD hinausgehende Operationen an. Außerdem werden die Daten in einer Art und Weise gespeichert, dass die Zustandsänderungen von Entitäten auch nachträglich nachvollziehbar sind, um dem *Verlust von Informationen* vorzubeugen. Hier bieten sich Event Streams an, wie sie im Event-Sourcing-Ansatz verwendet werden. Die Schnittstelle für den Command-Service kann dabei nach wie vor über HTTP angeboten werden, oder aber auf andere Technologien gRPC oder bereits bestehende Schnittstellen anderer Technologien zurückgreifen.

Kapitel 2.3.1 hat gezeigt, dass REST unterschiedliche Schwächen aufweist, die insbesondere bei Schreibzugriffen und der partiellen Aktualisierung von Entitäten zu Herausforderungen im Design führen können. Wenn das Paradigma nun nicht mehr für den gesamten Service, sondern ausschließlich für lesende Zugriffe verwendet wird, können die Schwächen ressourcenorientierter Modellierung ausgeglichen werden, ohne auf die Einfachheit und Akzeptanz des Paradigmas für standardisierte Schnittstellen verzichten zu müssen. Dieses Integrationsmuster löst somit als vierte und komplexeste Ausbaustufe alle Integrationsprobleme, die in Kapitel 3.1 vorgestellt und in Abbildung 5.2 dargestellt sind. Je nach Anwendungsfall kann die Aufteilung eines Services mitsamt verschiedener

Schnittstellen-Paradigmen jedoch auch zu Verwirrung führen, wenn viele Clients nicht nur Daten lesen, sondern auch regelmäßig schreibend auf den Service zugreifen und dabei mit völlig unterschiedlich modellierten APIs interagieren müssen.

Davon abgesehen kann die Abkehr von REST für schreibende Zugriffe mit der vierten und letzten Ausbaustufe ein weiterer Schritt hin zu Services sein, die komplett auf die durch REST geförderte Ressourcenorientierung verzichten und stattdessen nach alternativen Mustern arbeiten. Einige dieser Muster finden neben Kombinationen bereits vorgestellter Muster im folgenden Kapitel Erwähnung.

5.3.5 Weitere Kombinationen der vorgestellten Muster

Mit den zuvor beschriebenen vier Ausbaustufen wird eine Einführung und Transformation zu ereignisorientierter Service-Modellierung ermöglicht. Wenn auch nach Anwendung des Integrationsmusters der letzten Ausbaustufe Bedarf besteht, den Service mittels Ereignisorientierung domänenorientierter zu gestalten oder die vorgeschlagenen Muster aus anderen Gründen nicht infrage kommen, kann auf bereits vorgestellte Alternativen aus der Literatur zurückgegriffen werden. Allen voran ist hier der in Kapitel 4.4.1 kritisierte Event-Sourcing-Ansatz zu nennen. Wurde ein Service über die vorangehenden Ausbaustufen transformiert und würde von zusätzlicher Ereignisorientierung profitieren, ist davon auszugehen, dass die Komplexität des Musters gerechtfertigt ist und seine Vorteile zum Tragen kommen. Aber auch entlang der vorgestellten Ausbaustufen sind Kombinationen mit den in der Literatur vorgestellten Mustern möglich. Da sowohl CQRS als auch Event Sourcing als alleinstehende Konzepte außerhalb des Kontexts von EDA funktionieren, können sie bspw. mit Event Notifications aus der ersten Ausbaustufe kombiniert werden.

Besteht die Möglichkeit, Änderungen am Programmcode des Services vorzunehmen, kann die zweite Ausbaustufe, der Event-Carried State Transfer auch unabhängig von der REST-Events-Middleware umgesetzt werden. Werden Events bspw. im ORM des Services erzeugt, kann die Struktur eines Events unabhängig von der Antwort-Struktur der REST-API gestaltet werden und somit auch granulare Aktualisierungen einer Entität als Event veröffentlicht werden. Ebenso können fehlgeschlagene Anfragen eines Clients, die für die Domäne relevante Ereignisse darstellen, leichter identifiziert und als Event anderen Services zur Verfügung gestellt werden. Dabei kann der Service die Erstellung von Events komplett übernehmen und auf die REST-Events-Middleware verzichten oder nur zusätzliche, über die CRUD-Operationen hinausgehende Events veröffentlichen.

In diesem Kapitel wurden mit der Aufstellung von Kriterien für pragmatische und zugängliche Integrationsansätze Anforderungen gestellt, die von den soeben vorgestellten Ausbaustufen zu erfüllen sind. Mit der REST-Events-Middleware wird dabei eine pragmatische Lösung geschaffen, um bestehende REST-Schnittstellen in die Lage zu versetzen, Events zu publizieren. In Kombination mit der ersten Ausbaustufe kann diese Lösung mit sehr wenig Zeit- und Kostenaufwand eingeführt werden. Inwiefern diese konzeptionellen Vorschläge sich in der Praxis bewähren, soll deshalb mit einer Implementierung eines Prototyps in Kapitel 7 untersucht werden, vorerst sollen jedoch Vorschläge für die Einführung einer ereignisgetriebenen Architektur gemacht werden.

6 Vorschläge zur Einführung einer ereignisgetriebenen Architektur

Nach der Erarbeitung lösungsorientierter Integrationsmuster sollen Ansätze entwickelt werden, um den Anforderungen an die Eventing-Infrastruktur gerecht zu werden, wie sie in Kapitel 5.1.2 gestellt wurden. Dafür werden in diesem Kapitel drei Konzepte vorgestellt, die eine Standardisierung der Event-Struktur und des Transports sowie einer Beschreibung der gesamten Infrastruktur vorsehen.

6.1 Standardisierung der Event-Struktur

Wenn Ökosysteme für die Verteilung von Events auf mehrere Event-Broker setzen, müssen die Events bei der Einspeisung in die jeweiligen Broker transformiert werden, um der Event-Struktur des jeweiligen Brokers zu entsprechen. Dabei sind die Attribute, die Broker in ihrer Event-Struktur definieren, oft sehr ähnlich: Was bei NATS eine `MsgId` ist, wird in AMQP mit dem Header `message-id` beschrieben. Um diesen Transformationsaufwand zu verhindern, sollte die Struktur eines Events von dem Broker entkoppelt werden, damit Events über beliebige Broker verteilt werden können.

Durch diese Unabhängigkeit entstehen mehrere Vorteile: Weil der Standard im gesamten System gilt, verarbeiten Publisher und Subscriber jedes Event auf die gleiche Art und Weise. Weil Fallunterscheidungen je nach Broker oder Event-Typ nicht mehr nötig sind, wird die lose Kopplung von Services und Infrastruktur verstärkt. Auch Events von externen Event-Publishern, wie beispielsweise von einem Cloud-Anbieter, können beim Eingang in das System in ihrer Struktur angepasst werden und dann wie jedes andere Event verarbeitet werden. Unterschiedliche Broker bringen mit ihrem bereits erwähnten unterschiedlichen Funktionsumfang jedoch auch unterschiedliche Vorteile mit sich. Neben den üblichen Attributen, wie sie im vorigen Absatz erwähnt wurden, existieren auch sehr spezifische Attribute an einem Event, die bestimmte Funktionen ermöglichen. Kafka zum Beispiel bietet wie in Kapitel 4.2.2 beschrieben die Möglichkeit der Partitionierung von Events an – solche Funktionen und ihre notwendigen Attribute an einem Event dürfen durch die Standardisierung nicht zunichte gemacht werden.

Eine solche Standardisierungs-Initiative wird aktuell innerhalb der CNCF unter dem Namen „CloudEvents“ entwickelt. Sie ist neben einigen proprietären Event-Formaten von großen Firmen oder Event-Brokern der einzig verbreitete Standard, der unabhängig von Hersteller und Broker-Lösung entwickelt wurde.

6.1.1 Einführung in den CloudEvents-Standard

CloudEvents hat zum Ziel, eine Struktur für Events zu entwickeln, die unabhängig von einem einzelnen Broker ist und gleichzeitig so erweitert werden kann, dass besondere Funktionen einzelner Broker nach wie vor durch Attribute repräsentiert werden können. Das Projekt befindet sich im „Incubating“-Status der CNCF und definiert im Wesentlichen eine Spezifikation, welche die Struktur eines Events beschreibt [Clo21c]. Die Spezifikation ist mit Version 1.0 stabil und wird bereits von einigen großen Firmen verwendet: Sowohl Microsoft Azure als auch GitHub bieten alle Events, auf die Kunden eine Subscription anlegen können, im CloudEvents-Format an. Darüber hinaus arbeitet auch Google in bestimmten Produkten bereits mit CloudEvents und auch Knative[27], ein Erweiterungs-Framework für Kubernetes, das sich unter anderem auf Functions-as-a-Service spezialisiert, nutzt CloudEvents als Standard für Events [Clo21a].

Unabhängig von der Nutzung von namenhaften Akteuren in der Softwarebranche erfüllt der CloudEvents-Standard die nötigen Eigenschaften, die zu Beginn des Kapitels genannt wurden: Es vereinheitlicht die üblichen Attribute innerhalb der Event-Struktur und bietet Möglichkeiten, Events um besondere Attribute zu erweitern. So wurden zum Beispiel mit der Version 1.0 eine Erweiterung für Sequenznummern, Distributed Tracing und die bereits genannte Partitionierung veröffentlicht, welche die grundlegende Struktur um jeweilige Attribute ergänzen können. Darüber hinaus können auch eigene Erweiterungen definiert werden, die beispielweise proprietäre Header für die interne Verwendung innerhalb einer Organisation ergänzen können.

6.1.2 Funktionsumfang von CloudEvents

Ein CloudEvent ist wie gewöhnlich in Header und Payload aufgeteilt. Die Header werden für das Routing verwendet, der Payload aus Gründen der Performance idealerweise nur von Subscribern ausgelesen. Der CloudEvents-Standard soll laut Spezifikation möglichst leichtgewichtig sein und nur wenige Attribute verpflichtend voraussetzen. Aus der Event-Struktur, die in Listing 6.1 dargestellt ist, sind deswegen nur die Felder `specversion`, `id`, `source` und `type` verpflichtend. Die übrigen Header müssen nicht zwangsläufig gesetzt

[27] `https://knative.dev/docs/eventing/`

sein, ebenso muss das `data` Attribut, der Payload des Events, nicht zwangsläufig vorhanden sein. Darüber hinaus kann ein Event neben der dargestellten JSON-Repräsentation auch in den Formaten Protocol Buffers[28] und Avro[29] dargestellt werden.

```
{
    "specversion" : "1.0",
    "type" : "com.github.pull_request.opened",
    "source" : "https://github.com/cloudevents/spec/pull",
    "subject" : "123",
    "id" : "A234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "text/xml",
    "data" : "<much wow=\"xml\"/>"
}
```

Listing 6.1: Aufbau der Event-Struktur eines CloudEvents [Clo21b].

Über die Standardisierung der Event-Struktur hinaus bietet das CloudEvents-Projekt allerdings auch Bibliotheken für diverse Sprachen an, die CloudEvents in einen Broker einspeisen oder aus einem Broker konsumieren. Dabei findet immer auch eine Transformation der Attribute in die jeweiligen proprietären Werte des Brokers statt, damit die Infrastruktur auf Basis der Attribute arbeiten kann. Beim Auslesen werden diese Attribute wieder in die CloudEvents-Struktur gebracht. Unterstützt werden aktuell die Bindings für Kafka, MQTT, AMQP in der Version 1.0, HTTP, WebSockets und NATS. Die Struktur des Events ersetzt also nicht die bereits etablierten Formate der einzelnen Broker und versucht, sich als Alternative zu etablieren, sondern integriert die unterschiedlichen Formate in eine abstrahierte Struktur, die von Publisher und Subscriber unabhängig vom Transportweg verwendet werden kann. So wird das Attribut `id` des CloudEvents in NATS zu einer `MsgId` und in AMQP zu einem `message-id` Header transformiert. Das CloudEvent wird trotzdem in seiner kompletten Ausprägung, also mit Header und Payload serialisiert und als Payload des Broker-Events verschickt.

[28] `https://developers.google.com/protocol-buffers/`
[29] `https://avro.apache.org/`

6.1.3 Herausforderungen im Umgang mit CloudEvents

Kern der CloudEvents-Spezifikation ist die Definition einer standardisierten Event-Struktur. Während diese bereits in einem stabilen Zustand ist, sind die Bibliotheken, die die Transformation in Broker-Events vornehmen, je nach Sprache unvollständig und bieten nur einen Bruchteil der eingangs erwähnten unterstützten Event-Broker an. Darüber hinaus werden nur bestimmte Versionen von verbreiteten Protokollen angeboten: AMQP wird beispielsweise nur in der Version 1.0 unterstützt, RabbitMQ nutzt als die am meisten verbreitete Implementierung jedoch die Version 0-9-1. Im Zweifel müssen die Konnektoren zu diversen Event-Brokern also selbst geschrieben bzw. den Open-Source-Projekten beigesteuert werden.

Während die Verarbeitung von CloudEvents, die über den Event-Broker verteilt werden, mithilfe der mitgelieferten Bibliotheken ohne weitere Konfiguration ausgelesen werden können, werden Broker-Events nicht erkannt bzw. transformiert, wenn ihr Payload nicht den Mindestanforderungen der CloudEvents-Spezifikation genügt. Hier kann entweder mit einer eigenen Bibliothek gearbeitet werden, die das CloudEvent aus den Metainformationen des Broker-Events erstellt oder mit einer strikten Trennung von CloudEvents in eigene Queues. Broker-Events, die nicht als CloudEvent vorliegen, könnten dann von einem Event-Subscriber konsumiert und transformiert werden, um in eine Queue eingespeist zu werden, aus der ausschließlich CloudEvents konsumiert werden. Events von externen Publishern, die ihre Events nicht als CloudEvents publizieren, müssten eine solche Transformation durchlaufen, bevor sie in ein verteiltes System eingespeist werden, das mit dem CloudEvents-Standard arbeitet.

Ein weiterer Aspekt, der nicht Bestandteil des Projekts ist, ist das Routing von Events. Das Attribut `type` bietet zwar die Grundlage dafür, ein Event in eine jeweilige Queue des Event-Brokers einzuspeisen, allerdings wird diese Logik nicht von den Bibliotheken bereitgestellt. Es erfolgt also kein Mapping des `type` eines CloudEvents zu einem `Subject` in NATS oder `Topic` in RabbitMQ. Daraus folgt auch, dass in einer Queue eines Event-Brokers CloudEvents mit unterschiedlichem `type` verteilt werden können [Clo21b].

Die aufgezeigten Defizite sind teilweise sicherlich dem noch recht jungen Zustand des Projekts zuzuschreiben, hängen zum großen Teil aber mit dem Rahmen zusammen, den das Projekt steckt und beispielsweise das Routing von Events bewusst nicht zum Bestandteil der CloudEvents-Spezifikation erklärt. Die einheitliche Event-Struktur bietet nichtsdestotrotz einen großen Vorteil, weil sie entgegen einer Konvention, die nur für das eigene System entworfen würde, innerhalb der CNCF-Landschaft als Standard akzeptiert und verwendet wird.

6.2 Anbindung aller Services mittels HTTP-Proxies

Die Aspekte, die von der CloudEvents-Spezifikation unberührt bleiben, stellen bei der Einführung Eventing-Infrastruktur zwei zentrale Herausforderungen dar: Weil CloudEvents weder die Logik der Subscriptions noch die Zuordnung eines CloudEvent `type`s auf jeweilige Queues eines Event-Brokers löst, müssen diese in den Services implementiert werden, die Events produzieren oder konsumieren möchten. Für den Fall, dass Events in einem verteilten System über mehrere Event-Broker verteilt werden, müssen sie außerdem wissen, aus welchem Broker bestimmte Events zu beziehen sind. Das bedeutet, dass jeder Service für die jeweilige Programmiersprache über eine Bibliothek verfügen muss, die die Verbindung zum Event-Broker herstellt und eben erwähnte Logik abdeckt. In einer heterogenen Systemlandschaft ist allerdings nicht davon auszugehen, dass dies für alle Services zutrifft – Events wären somit moderneren Implementierungen vorbehalten. Zwar gibt es Ansätze wie Spring Cloud Stream[30], die als Abstraktion etablierter Broker eine einheitliche Bibliothek anbieten, allerdings beschränkt sich deren Einsatz erneut auf Umgebungen, in denen Java eingesetzt wird.

Um Events allen Services zugänglich zu machen und eine hohe Akzeptanz im Integrations-Ökosystem zu erreichen, braucht es zusätzlich zur Standardisierung der Event-Struktur also weitere Komponenten, die eine Standardisierung über CloudEvents hinaus etablieren.

6.2.1 Auslagerung der brokerspezifischen Logik in eigene Services

Damit die Event-Produzenten und -Konsumenten von dem spezifischen Event-Broker entkoppelt sind, werden zwei zusätzliche, vorgelagerte Infrastruktur-Komponenten eingeführt (siehe Abbildung 6.1), welche die brokerspezifische Logik kapseln: Die erste Komponente ist für die Einspeisung von Events durch Services in die jeweiligen Broker verantwortlich und trägt dementsprechend den Namen CloudEvents-Ingestor, kurz CE-Ingestor. Die zweite Komponente ist für die Versorgung der Services mit Events aus den jeweiligen Event-Brokern verantwortlich und trägt dementsprechend den Namen CloudEvents-Dispatcher, kurz CE-Dispatcher. Beide Komponenten kommunizieren per HTTP mit Produzenten bzw. Konsumenten – die einzelnen Aufrufe können dem UML-Diagramm in Abbildung 6.2 entnommen werden.

Der CE-Ingestor bietet eine HTTP-Schnittstelle an, an die beliebige Services in ihrer Rolle als Event-Produzenten HTTP-CloudEvents senden können, um sie in einen bestimmten Event-Broker zu verteilen. Weil der CE-Ingestor von den Services unabhängig ist, wird er zentral im System verfügbar gemacht und kann bei Bedarf elastisch skaliert werden. In dieser Komponente wird die Verbindung zum Broker hergestellt und

[30] `https://spring.io/projects/spring-cloud-stream/`

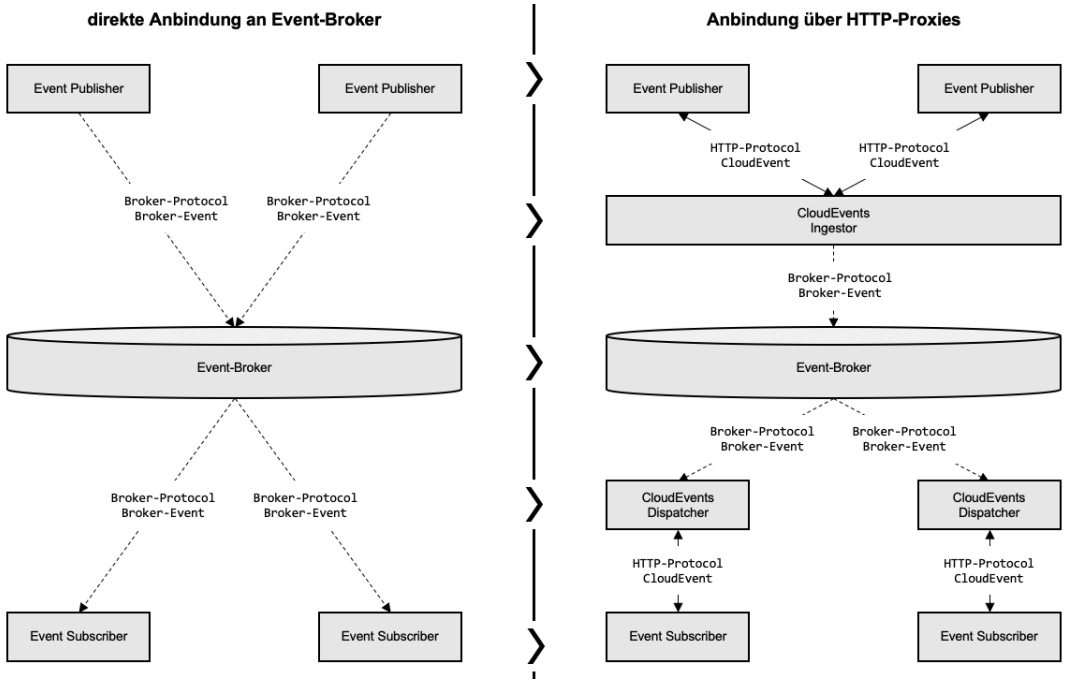

Abbildung 6.1: Kapselung der brokerspezifischen Logik durch den Einsatz von CE-Ingestor und CE-Dispatcher.

es werden eventuelle Ausfälle des Brokers abgefangen. Außerdem erfolgt eine automatische Zuweisung von CloudEvents basierend auf ihrem `type` in die jeweiligen Queues des Brokers – für jeden CloudEvent `type` existiert also eine eigene Queue, in der ausschließlich Events mit diesem `type` verteilt werden. Damit wird die Funktionalität, die das CloudEvents-Projekt in ihren Bibliotheken anbietet, erweitert und standardisiert.

Der CE-Dispatcher hingegen ist eine dezentrale Komponente, die von Services in ihrer Rolle als Event-Konsumenten betrieben werden kann, um mit Events versorgt zu werden. Die Komponente kapselt damit auch die Subscription-Logik, die je nach Event-Broker unterschiedlich implementiert werden muss. Wenn ein CE-Dispatcher verwendet wird, wird diesem eine Konfigurationsdatei bereitgestellt, in der die unterschiedlichen Event-Subscriptions auf den `type` definiert und der HTTP-Endpunkt deklariert ist, an den die Events gesendet werden sollen. Services müssen dann nur eine HTTP-Schnittstelle zur Verfügung stellen, die diese Events verarbeitet. Der durch den Betrieb der zusätzlichen Komponenten entstandene Aufwand und die belastete Rechenkapazität, werden durch zwei wesentliche Vorteile aufgewogen: Neben der bereits erwähnten Kapselung von brokerspezifischer Logik ist nun auch jeder Service, vom modernsten Microservice bis hin zum COBOL-Programm auf einem Großrechner in der Lage, Events zu produzieren oder konsumieren.

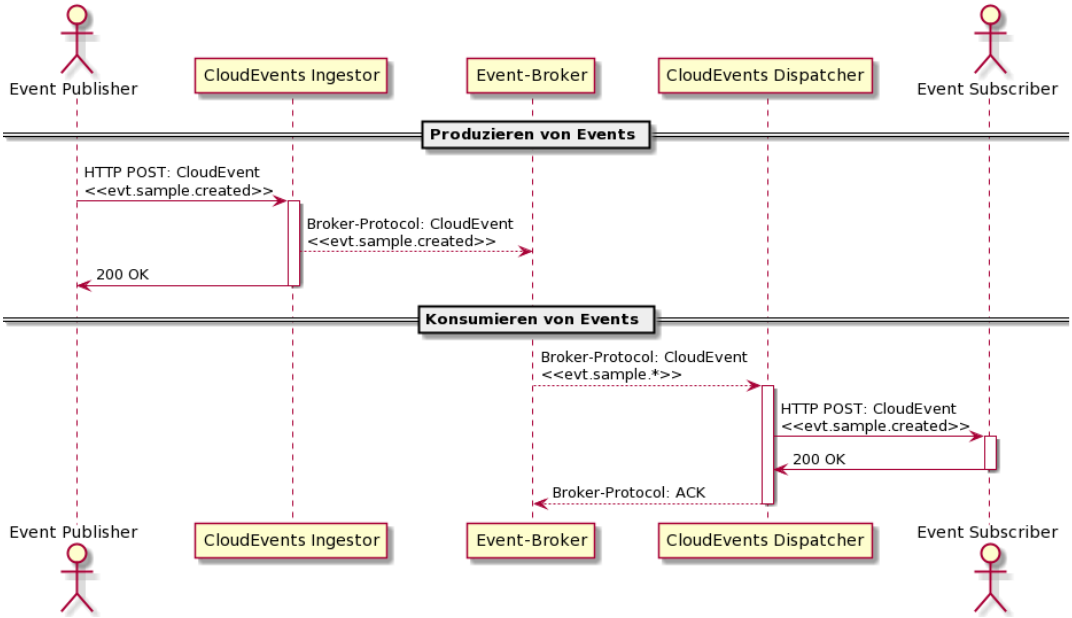

Abbildung 6.2: Kapselung der brokerspezifischen Logik durch den Einsatz von CE-Ingestor und CE-Dispatcher.

Darüber hinaus kann von den zwei eingeführten Komponenten eine Nutzungsmatrix erstellt werden: Der CE-Ingestor hat die Information, welcher Service welchen `type` von Events produziert und der CE-Dispatcher kennt die Services und jeden `type`, den sie konsumieren. Damit kann die mangelnde Übersichtlichkeit, die ereignisgetriebene Architektur mitbringt (siehe Kapitel 4.1.3) reduziert werden. Nichtsdestotrotz entstehen durch die zusätzlichen Komponenten und ihre Kommunikation via HTTP höhere Latenzen, als wenn die Verbindung zum Event-Broker direkt hergestellt würde. Je nach geschäftlicher Anforderung können die Kosten dieser Latenz im Zweifel größer sein als die Ersparnisse, die die Komponenten durch die Standardisierung erreichen.

6.2.2 HTTP-CloudEvents als offener Kommunikationsstandard

Um die Kommunikation der Services und den eingeführten Komponenten und damit auch die Anbindung an die Eventing-Infrastruktur möglichst offen zugänglich zu halten, soll statt über brokerspezifische Protokolle über das durch REST-APIs bereits etablierte HTTP-Protokoll kommuniziert werden. Der CloudEvents-Standard sieht mit dem HTTP-Protocol-Binding[31] einen Transport über HTTP vor, bei dem die Header des Events als HTTP-Header mit einem `ce-` Präfix und der Payload des Events als HTTP-Body versendet werden. Dieses niedrigschwellige Protokoll erlaubt allen Services, die

[31] `https://github.com/cloudevents/spec/blob/v1.0.1/http-protocol-binding.md/`

auch Teil des bisherigen REST-Ökosystems waren, an die Eventing-Infrastruktur angebunden zu werden. Ein ähnlicher Ansatz wird auch von Knative verfolgt, hat seinen Ursprung aber in der Versorgung von Functions-as-a-Service, die üblicherweise über HTTP ausgelöst werden.

Das CloudEvents-Projekt bietet Bibliotheken an, die ein CloudEvents-Objekt für den HTTP-Transport vorbereiten oder ein über HTTP gesendetes Event in ein Objekt der jeweiligen Programmiersprache verwandeln. Im Gegensatz zu Bibliotheken, die eine Verbindung zu einem bestimmten Event-Broker herstellen, sind sie aber aufgrund der bereits dargelegten geringen Komplexität optional – diese Funktionen können stattdessen selbst implementiert werden. Bibliotheken für die Programmiersprachen aller Services, die brokerspezifische Logik enthalten, werden damit überflüssig. Stattdessen können etablierte Protokollelemente von HTTP zum Beispiel genutzt werden, um mittels Statuscodes anzugeben, ob die Verarbeitung eines Events erfolgreich war. Der konkrete Einsatz des Protokolls wird bei der konkreten Umsetzung beider Komponenten im Rahmen des Prototyps in Kapitel 7.3 genauer erklärt.

6.3 Maschinenlesbare Beschreibung eventbasierter Schnittstellen

Nach zwei aufeinander aufbauenden Vorschlägen bezüglich der Anbindung von Services an die Eventing-Infrastruktur folgt in diesem Kapitel der dritte Vorschlag, der sich weniger mit der tatsächlichen Infrastruktur beschäftigt, sondern den Fokus auf die Verwaltung von Event-Driven APIs legt. Diese sollen wie jede andere Schnittstelle im Ökosystem behandelt werden und bedürfen einem gewissen Management, um den in Kapitel 4.1.3 erwähnten Nachteilen entgegenzuwirken. So muss bspw. die Versionierung von Events transparent gelöst werden und eine Übersicht über das gesamte Ökosystem sichergestellt werden, indem Konsumenten und Produzenten von Events mitsamt Event-Struktur bekannt und beschrieben werden. Dementsprechend braucht es API-Management bzw. Event-Management auch für eventbasierte Schnittstellen.

REST zeigt, dass eine maschinenlesbare Schnittstellenbeschreibung wie OpenAPI die Basis für dieses API-Management schafft. Auf den folgenden Seiten wird deshalb „AsyncAPI“[32] und sein Einsatz in ressourcenorientierten Systemlandschaften dargelegt, um der mangelnden Übersicht ereignisorientierter Schnittstellen entgegenzuwirken.

[32] `https://asyncapi.com/`

6.3.1 AsyncAPI als Beschreibungsstandard

Die AsyncAPI-Spezifikation wurde im Jahr 2017 in einer Open-Source-Initiative als Ableger der OpenAPI-Spezifikation gestartet, um eine Interface Description Language für eventbasierte APIs anzubieten. Seit dem Release wurde im Jahr 2019 eine Version 2.0 veröffentlicht und eine AsyncAPI-Initiative gegründet, der große Firmen wie bspw. Adidas, SAP oder Slack angehören. AsyncAPI stellt damit neben wenigen Alternativen die am meisten verbreitete IDL dar, um asynchrone Schnittstellen unabhängig von ihrem Protokoll zu beschreiben. Mit der maschinenlesbaren IDL können Event-Driven APIs mit YAML oder JSON beschrieben werden und verwenden dabei, wie auch OpenAPI JSON Schema, um bspw. auf Strukturen aus anderen Dokumenten referenzieren zu können [Asy21b].

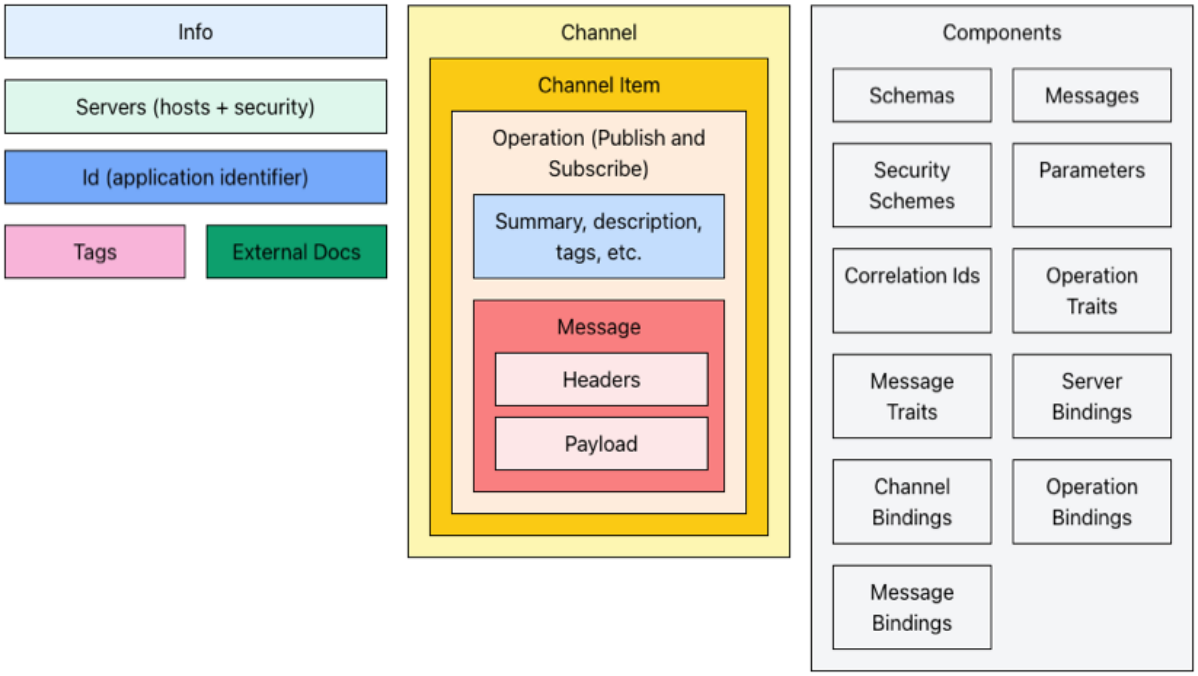

Abbildung 6.3: Aufbau der AsyncAPI-Spezifikation [Asy21b].

Um die Bestandteile einer ereignisgetriebenen Schnittstelle beschreiben zu können, existieren in AsyncAPI unterschiedliche Objekte, die in Abbildung 6.3 dargestellt werden. Neben Meta-Informationen wie Namen, Kontaktinformationen und Lizenzmodell bilden sogenannte „Channels“ als abstraktes Konstrukt ähnlich wie die in Kapitel 4.2 die Kanäle die die oberste Struktur. Innerhalb eines Channels kann definiert werden, ob der beschriebene Services für diesen Channel als Konsument oder Produzent auftritt. Dafür werden jeweils eine `publish` oder `subscribe` Operation definiert. Für jede Operation wird außerdem das versendete Event beschrieben. So können neben den wesentlichen

Informationen wie den Headern und dem Payload unter anderem auch ein Attribut für die Correlation ID und definiert werden. Darüber hinaus besteht auch die Möglichkeit, Anbindungen für Protokolle zu definieren: Server, unter denen die Events produziert und konsumiert werden können sowie protokollspezifische Parameter für Events und Channels. Unterstützt werden die meisten verbreiteten Protokolle, so auch AMQP, NATS und Kafka. Ebenso besteht die Möglichkeit, Angaben zur Authentifizierung, bspw. über OAuth2[33] oder API-Keys zu machen. Es gibt jedoch auch Bestandteile von EDA-Lösungen, die nicht mit AsyncAPI beschrieben werden können. So gibt es keine Möglichkeit, nähere Angaben über den Interaktionsstil von einem Service mit einem Channel zu machen und bspw. anzugeben, ob es sich um eine Queue handelt, aus der mehrere Worker nach dem Competing Consumer Pattern Events abarbeiten. Ebenso können keine Angaben über Event Streams und relevante Informationen wie ihre Retention Policy gemacht werden [Asy21a].

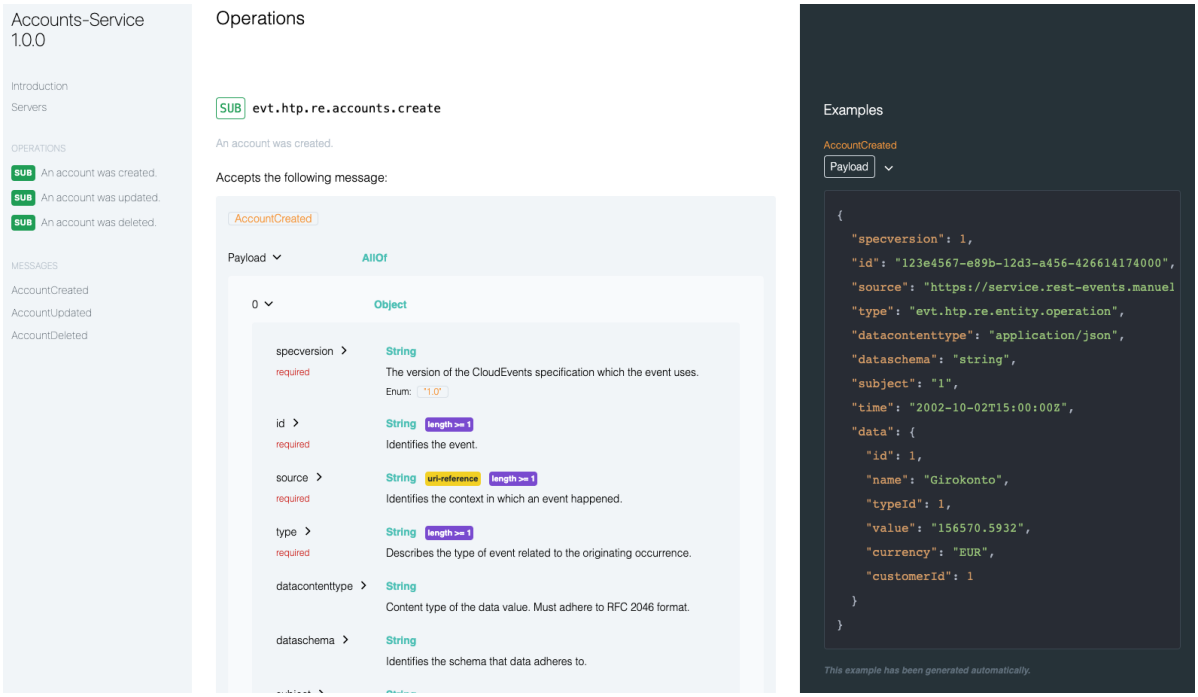

Abbildung 6.4: Generierte Dokumentation auf Basis der AsyncAPI-Spezifikation.

Die Nähe von AsyncAPI zu OpenAPI bringt mehrere Vorteile mit sich: Weil die Spezifikation ebenfalls maschinenlesbar ist, sind ähnliche Tooling-Möglichkeiten wie bei OpenAPI gegeben. So existieren bspw. bereits Tools, um ein AsyncAPI-Dokument zu validieren, menschenlesbare Dokumentation (siehe Abbildung 6.4) oder Test-Suites zu generieren. Aufgrund der Protocol-Bindings kann außerdem Code für konsumierende

[33] `https://oauth.net/2/`

und produzierende Services generiert werden. Darüber hinaus haben einige OpenAPI-Werkzeuge ihre Unterstützung inzwischen auf AsyncAPI ausgeweitet. API-Plattformen wie Gravitee[34], die ursprünglich für REST-APIs entworfen sind, generieren nun ebenfalls HTML-Dokumente auf Basis von AsyncAPI-Spezifikationen. Weil AsyncAPI als Ableger von OpenAPI ebenfalls mit JSON-Schema arbeitet, können des Weiteren Strukturen aus einer OpenAPI-Spezifikation in einer AsyncAPI-Spezifikation referenziert werden. Das ermöglicht eine tiefe Integration in bereits bestehende Schnittstellenbeschreibungen. Die Ähnlichkeit der Spezifikationen bringt außerdem einen Vorteil auf organisatorischer Ebene mit sich: Da sich der Aufbau der jeweiligen Dokumente derart ähnelt, sind Entwickelnde, die bereits mit OpenAPI vertraut sind, mit wenig Einarbeitungszeit in der Lage, Schnittstellenbeschreibungen für asynchrone Schnittstellen zu verfassen. Trotz der Ähnlichkeit gibt es nach wie vor wesentliche Unterschiede zwischen den Spezifikationen. Da AsyncAPI wesentlich jünger und weniger verbreitet ist, existiert dementsprechend weniger Tooling – insbesondere auch aus der Open-Source-Community. Des Weiteren besteht ein Nachteil, der weniger mit AsyncAPI, sondern viel mehr mit den Eigenschaften von EDA zu tun hat. Im Gegensatz zu HTTP-APIs existieren für eventbasierte Schnittstellen zahlreiche Protokolle und Event-Broker, die unterschiedliche Funktionen mitbringen und Standards befolgen. Diese Heterogenität in einer allgemeingültigen IDL zu vereinen, stellt eine große Herausforderung dar: Um kein Protokoll auszuschließen, müssen mehrere Alternativen geboten werden, gleichzeitig aber auch eine Abstraktion der Bestandteile der einzelnen Lösungen stattfinden. Als Dokument, das alle eventbasierten Schnittstellen beschreibt, muss dieser Spagat zwischen Unterstützung und Standardisierung auf Kosten der Verständlichkeit und Einfachheit akzeptiert werden. Mit voranschreitender Standardisierung von EDA ist jedoch davon auszugehen, dass auch die Beschreibung von EDA standardisierter erfolgen kann.

Um die Standardisierung von EDA innerhalb der eigenen Organisation voranzutreiben, eignet sich die AsyncAPI-Spezifikation in Kombination mit den vorangehenden Vorschlägen auch jetzt schon. Im folgenden Kapitel sollen deshalb Vorschläge gemacht werden, wie AsyncAPI in Ökosystemen eingesetzt werden kann, die von REST-APIs geprägt sind.

6.3.2 Einsatz in ressourcenorientierten Ökosystemen

In Systemlandschaften, in denen die Integration von Services vornehmlich über REST-APIs erfolgt, sind Schnittstellenbeschreibungen mit OpenAPI essentiell. In diesem Kapitel wird dementsprechend davon ausgegangen, dass für jeden Service, der eine REST-API anbietet, eine OpenAPI-Spezifikation existiert. Wenn diese Services nach den vorgeschlagenen Ausbaustufen aus Kapitel 5.3 in die Ereignisorientierung eingeführt werden, sollte in jedem Fall ab der zweiten Ausbaustufe (Kapitel 5.4), besser jedoch ab dem ersten

[34] `https://www.gravitee.io/`

veröffentlichten Event eine IDL für die ereignisorientierte Schnittstelle angeboten werden. Darüber hinaus muss jeder Service, der als Event-Konsument auftritt, dies mittels einer IDL beschreiben. Deshalb wird vorgeschlagen, diese IDLs nach der AsyncAPI-Spezifikation zu verfassen. Neben den Vorteilen, die AsyncAPI bereits isoliert beisteuert, entsteht ein weiterer, zentraler Vorteil beim Einsatz in ressourcenorientierten Ökosystemen. Da AsyncAPI mit JSON Schema auf Strukturen aus einem OpenAPI-Dokument referenzieren kann, kann das Schema eines Events, das durch die REST-Events-Middleware erzeugt wurde, direkt auf das REST-Schema aus der OpenAPI-Spezifikation verweisen. Somit muss die Beschreibung der Datenstruktur nicht doppelt erfolgen und ist dementsprechend weniger aufwendig und fehleranfällig. Eine weitere Vereinfachung ist bei den Protocol-Bindings möglich. Da die Integration mittels Events unabhängig von einem bestimmten Broker erfolgen soll und die Einspeisung über den CE-Ingestor erfolgt, werden diese nicht benötigt. Sie tragen außerdem nicht dazu bei, die Übersichtlichkeit von EDA zu verbessern.

Wenn jeder Service, der als Produzent oder Konsument von Event-Driven APIs auftritt, ein AsyncAPI-Dokument bereitstellt, ist analog zu OpenAPI eine Möglichkeit für API-Management von Event-Driven APIs geschaffen. Es herrscht Transparenz über alle existierenden Events mitsamt Produzenten und Konsumenten, sodass diese auf einer API-Plattform dargestellt werden können. So können Verantwortliche anderer Services in einem API-Katalog oder Event-Katalog sehen, welche Events bereits in der Infrastruktur zu Verfügung stehen und in ihre Prozesse integrieren. Auf Basis der IDL kann außerdem eine Nutzungsmatrix erstellt werden. Diese hilft den Anbietern von ereignisorientierten Schnittstellen, ihre Konsumenten zu identifizieren und bspw. bei neuen Versionen oder bevorstehenden Wartungsfenstern zu informieren. Des Weiteren ist durch die Referenzierung auf OpenAPI-Dokumente eine tiefe Integration in REST-Ökosysteme möglich. Wenn Datenstrukturen identisch sind, müssen sie nicht doppelt beschrieben werden, allerdings besteht zu jeder Zeit die Möglichkeit, die Schnittstelle von den Datenstrukturen der REST-API abzukoppeln und die Struktur der Events frei zu bestimmen. Ebenso können jederzeit Events definiert werden, die unabhängig von REST-Operationen erfolgen. Diese Integrationsmöglichkeit der Schnittstellenbeschreibungen ermöglicht analog zu den Ausbaustufen der Integrationsmuster, dass Services schrittweise auf die Ereignisorientierung vorbereitet werden.

Mit diesem Kapitel wurden Vorschläge für die Eventing-Infrastruktur gemacht, die Kriterien aus Kapitel 5.1.2 adressieren. Mit dem Einsatz von CloudEvents und HTTP-Proxies für den Datenaustausch via Events kann dank offener Standards eine hohe Unabhängigkeit vom Event-Broker erreicht werden – dieser Datenaustausch bleibt dank AsyncAPI übersichtlich, wenn Produzenten, Konsumenten und ihre Events dokumentiert werden. Um diese Einschätzung zu validieren, sollen die vier Ausbaustufen von Integrationsmustern aus Kapitel 5.3 mit den drei Ansätzen aus diesem Kapitel gemeinsam anhand eines Prototyps validiert werden.

7 Prototypische Implementierung der vorgestellten Konzepte

Nachdem auf Basis der kritischen Bewertung des Stands der Technik in Kapitel 5 Ausbaustufen zu pragmatischen Integrationsmustern vorgestellt und in Kapitel 6 Vorschläge zu Infrastrukturkomponenten gemacht wurden, sollen diese nun in einem Prototyp kombiniert werden. Anhand dieses Prototyps wird anschließend beurteilt, welche der Kriterien aus Kapitel 5.1.1 durch die Muster und Ansätze erfüllt werden. Außerdem wird der Prototyp zum Anlass genommen, weitere, über den Fokus dieser Arbeit hinausgehende, Anwendungsszenarien für EDA und Erweiterungsmöglichkeiten der vorgestellten Konzepte zu identifizieren.

7.1 Architektur der beispielhaften Services

Für das prototypische System werden dabei drei Services herangezogen, die sich an den zentralen Entitäten aus dem Bankenumfeld orientieren: Der Customers-Service verwaltet die Kunden der Bank, der Accounts-Service verwaltet die Konten und im Assets-Service werden Wertgegenstände verwaltet, die als Sicherheit für Kredite dienen. Dabei beziehen sich Konten und Wertgegenstände immer auf einen Kunden, die Objekte werden also mit der ID des Kunden als Fremdschlüssel gespeichert und für die Kunden-Entität als externe Relationen aufgeführt. Alle drei Services bieten eine REST-API an, über die sie die von ihnen verwalteten Entitäten anbieten. Um eine Umgebung zu simulieren, in der Ereignisorientierung mit möglichst wenig Aufwand eingeführt werden soll, werden in allen drei Services keinerlei Änderungen am Programmcode gemacht – etwa, weil es sich um Kaufsoftware handelt oder keine Entwicklungskapazitäten für Anpassungen verfügbar sind. In der Abbildung 7.1 wird dargestellt, mittels welcher Muster und Infrastrukturkomponenten die Services an die Eventing-Infrastruktur angebunden werden. Der Aufbau und das Zusammenwirken der unterschiedlichen Konzepte wird für die einzelnen Services in den folgenden Absätzen erläutert.

Der Assets-Service, in Abbildung 7.1 blau hinterlegt, produziert in dem fiktiven Szenario REST-Antworten, die nicht geeignet sind, um sie als Events zu veröffentlichen. Er wird deshalb mithilfe von Event Notifications auf Basis von REST-Aufrufen, dem ersten

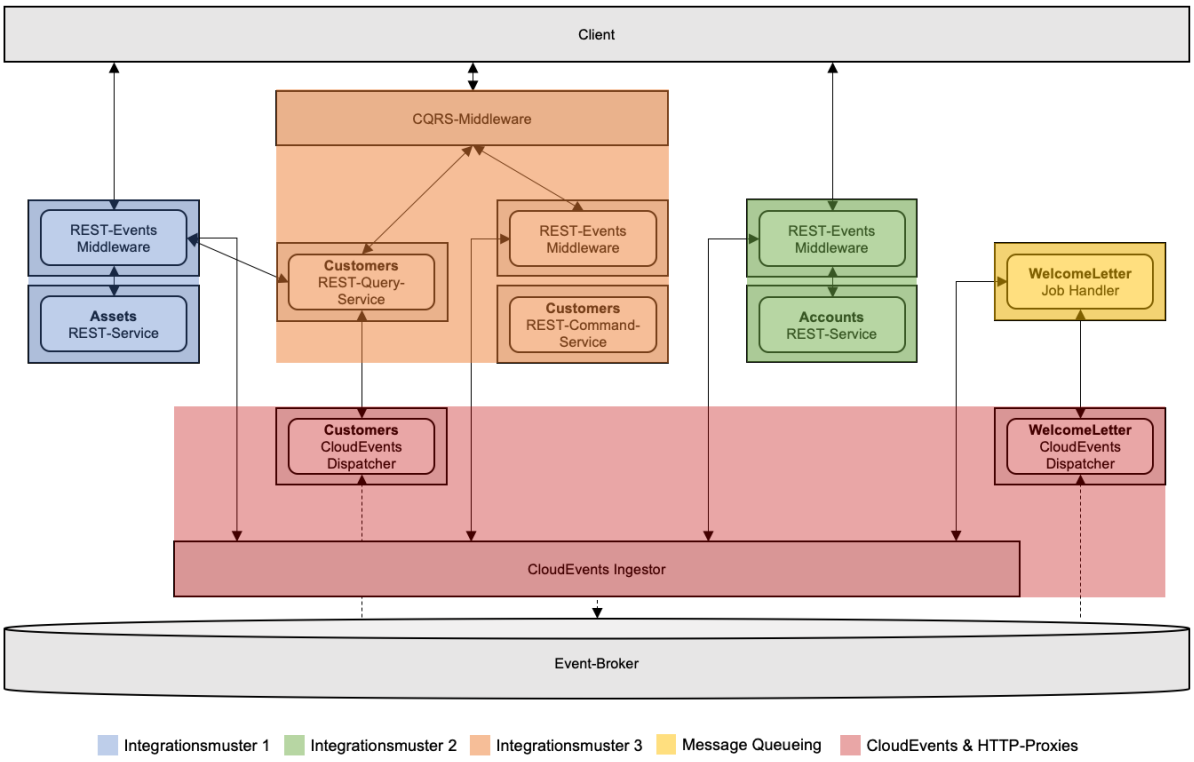

Abbildung 7.1: Beispielhafte Services des Prototyps.

vorgestellten Integrationsmuster aus Kapitel 5.3, angebunden. Dem REST-Service wird dazu die REST-Events-Middleware vorgeschaltet, welche Events per HTTP über den CE-Ingestor publiziert.

Der Accounts-Service, in Abbildung 7.1 grün hinterlegt, produziert REST-Antworten, die als Events geeignet sind. Er kann deshalb durch den Event-Carried State Transfer mit REST-Antworten (Kapitel 5.4) angebunden werden. Im Gegensatz zum Assets-Service wird den publizierten Events also auch die veränderte Entität als Payload angefügt. Dem REST-Service wird ebenfalls die REST-Events-Middleware vorgeschaltet, die die Events ebenfalls per HTTP über den CE-Ingestor publiziert.

Der Customers-Service, in Abbildung 7.1 orange hinterlegt, bietet in seiner bestehenden Form keine Möglichkeit an, die bereits angesprochenen externen Relationen darzustellen. Um in lesenden Zugriffen zu einem Kunden auch die zugehörigen Wertgegenstände und Konten darstellen zu können, implementiert der Service die dritte Ausbaustufe aus Kapitel 5.3 und verfolgt dementsprechend die Aufteilung von REST-Services mit CQRS in einen Customers-Query- und Customers-Command-Service. Den Services wird deshalb die CQRS-Middleware vorgeschaltet, die HTTP-Aufrufe je nach HTTP-Methode zu einem der beiden Services umleitet. Dabei wird der Customers-

Command-Service identisch zum Accounts-Service per REST-Events-Middleware und Event-Carried State Transfer in die Lage versetzt, Events über den CE-Ingestor zu publizieren. Der Customer-Query-Service konsumiert diese Events zusammen mit den Events der anderen beiden Services über den CE-Dispatcher. Dabei können Events, die vom Customers- oder Accounts-Service konsumiert werden, direkt verarbeitet werden, für die Verarbeitung der Events des Assets-Services ist jedoch ein weiterer Aufruf der REST-API nötig, um den aktuellen Zustand der Entität abzufragen.

Der WelcomeLetter-Job ist ein vierter Service, der kein konkretes Integrationsmuster umsetzt, sondern einen Vorteil veranschaulicht, der bereits ab der ersten Ausbaustufe zum Tragen kommt und im zugehörigen Kapitel erwähnt wurde: Durch die Verwendung von Events besteht die Möglichkeit, ressourcenintensive Prozesse in eigene Services auszulagern und somit die Verfügbarkeit des Services zu sichern. Der WelcomeLetter-Job stellt genau so einen Prozess dar. In dem fiktiven Szenario soll für jeden neuen Kunden ein Brief generiert und versandt werden, was im Prototyp durch eine Textdatei dargestellt wird, deren Erstellung zehn Sekunden braucht. Weil der Customers-Service über die REST-Events-Middleware nun Events produziert, kann dieser Prozess in einen eigenen Service ausgelagert werden. Ebenso wie der Customers-Query-Service wird der WelcomeLetter-Job über den CE-Dispatcher an die Eventing-Infrastruktur angebunden. Der Job produziert außerdem ein Event, wenn eine Datei erstellt wurde.

Neben den in den Services angewendeten Integrationsmustern aus Kapitel 5.3 werden in diesem Prototyp auch die Vorschläge aus Kapitel 6 angewendet. Alle Events werden gemäß dem ersten Vorschlag im CloudEvents-Standard strukturiert und über die bereits erwähnten HTTP-Proxies, in Abbildung 7.1 rot hinterlegt, produziert und konsumiert. Außerdem verfügt jeder Service über eine OpenAPI- und AsyncAPI-Spezifikation, die an den geeigneten Stellen über die in Kapitel 6.3.2 vorgeschlagenen Quer-Referenzen verfügt.

Um auf die Realisierung der Services und insbesondere der CQRS- und REST-Events-Middleware sowie dem CE-Ingestor und CE-Dispatcher einzugehen, wird im folgenden Kapitel dargelegt, welche Technologien für den Prototyp zum Einsatz kommen.

7.2 Verwendete Technologien

Der in dieser Arbeit erarbeitete Prototyp hat den Anspruch, eine heterogene Systemlandschaft nachzubilden, wie sie im Unternehmenskontext üblich ist. Diese Heterogenität bedeutet in der Praxis auch, dass Services aus unterschiedlichen Betriebsumgebungen miteinander integriert werden. Weil der Fokus dieser Arbeit auf der Integration von Microservices liegt, die vorrangig über REST integriert werden und der Umfang des Prototyps überschaubar bleiben soll, beschränkt sich der Prototyp auf eine Betriebsumgebung,

in denen Container via Kubernetes provisioniert werden (siehe Kapitel 2.2.2). Nichtsdestotrotz können die vorgestellten Integrationsmuster und Architekturvorschläge auch in anderen Umgebungen eingesetzt werden. An den Stellen, an denen Kubernetes als Betriebsumgebung Auswirkungen auf die Implementierung der Komponenten hat, werden in diesem und dem folgenden Kapitel alternative Möglichkeiten für andere Plattformen erwähnt. In den folgenden Absätzen werden Technologien und Produkte eingeführt, die im Prototyp eingesetzt werden.

7.2.1 Service-Prototyping mit JSON-Server

Die Grundlage des Prototyps stellen die drei vorgestellten Services dar, die an die Eventing-Infrastruktur angebunden werden sollen. Sie werden als Container mithilfe einer Bibliothek namens JSON-Server[35] implementiert, der statt einer Datenbank eine einfache JSON-Datei nutzt, um die Daten zu persistieren. Weil aus bereits dargelegten Gründen keinerlei Änderungen am Programmcode vorgenommen werden sollen, können die Services als Black-Box betrachtet werden. Sowohl für den Prototyp als auch im Unternehmenskontext spielt die konkrete Implementierung bzw. die gewählte Programmiersprache keine Rolle, weil jegliche Kommunikation über die REST-API stattfindet. Alle drei Services exponieren ihre Entitäten über die gängigen HTTP-Endpunkte, wie sie auch in Tabelle 2.1 beschrieben sind.

7.2.2 Betriebs-Infrastruktur auf Basis von Kubernetes

Als Betriebs-Infrastruktur kommt wie eingangs erwähnt Kubernetes zum Einsatz. Dafür wird auf einer virtuellen Maschine die Kubernetes-Distribution K3S[36] installiert. Im Gegensatz zu klassischen Distributionen eignet sich K3S für Testzwecke, Internet of Things oder anderen Szenarien, in denen nur eine Maschine benötigt wird. K3S enthält bereits den Ingress-Controller Traefik[37], der auch in diesem Protoypen zum Einsatz kommt. Beide sind Projekte im „Sandbox“-Status der CNCF. Wie in Listing 7.1 deutlich wird, ist die Installation überaus simpel: Mit einem einzigen ausgeführten Befehl können auf der Maschine Container per Kubernetes betrieben werden.

Um zum Zeitpunkt der Entwicklung, aber auch für die nachträgliche Betrachtung den kompletten Überblick über die im Cluster befindlichen Kubernetes-Objekte zu haben, wird für den Prototyp das GitOps-Pattern verwendet. Dabei werden alle deklarativ beschriebenen Kubernetes-Objekte in einem Git-Repository abgelegt und mithilfe von

[35] https://github.com/typicode/json-server/
[36] https://k3s.io/
[37] https://traefik.io/solutions/kubernetes-ingress/

```
curl -sfL https://get.k3s.io | sh -
```

Listing 7.1: Befehl für die Installation der Kubernetes-Distribution K3S.

ArgoCD[38], ebenfalls „Incubating“-Projekt der CNCF, im Cluster installiert. ArgoCD agiert als Continuous-Delivery-Tool und synchronisiert die im Repository existierenden Objekte mit den Objekten, die im Cluster installiert sind.

Zwar bietet ArgoCD bereits Möglichkeiten, die Logs von Containern über eine Oberfläche einzusehen, allerdings fehlen erweiterte Funktionalitäten wie Filter oder Aggregation der Logs mehrere Container. Deswegen wird eine zusätzliche Logging-Komponente eingesetzt. Während in produktiven Umgebungen bspw. Grafana[39] in Kombination mit Grafana Loki[40] zum Einsatz kommen, soll für diesen Prototyp auf die Software-as-a-Service Lösung LogDNA gesetzt werden, um die Komplexität gering zu halten. LogDNA bietet die erwähnten Funktionalitäten und ermöglicht somit einen Blick auf den Fluss von Events über mehrere Container und Services hinweg.

7.2.3 NATS JetStream als Event-Broker

Als Event-Broker soll in dem Prototyp NATS JetStream eingesetzt werden. Im Gegensatz zu RabbitMQ kann das Produkt mit der Persistierung von Events und ressourceneffizienter Skalierung punkten und ist aufgrund seiner Orientierung besser auf den Einsatz in EDAs ausgelegt. Apache Kafka stellt mit seinen gebotenen Funktionen und dem verteilten Betriebsmodell zwar einen geeigneten Kandidaten für eine tatsächliche Umsetzung für ein Unternehmen dar, NATS ist aufgrund seiner Leichtgewichtigkeit jedoch angemessener für einen Prototyp.

NATS kann mithilfe des Helm-Charts[41] über ArgoCD mit nur einer Konfigurationsdatei installiert werden, in der die Anzahl Instanzen sowie Angaben zur Persistierung von Events gemacht werden müssen. Daraufhin ist NATS im Cluster verfügbar und kann über einen Service angesprochen werden. Darüber hinaus soll der konkrete Broker in diesem Prototyp keine gesonderte Rolle spielen, da die erarbeiteten Konzepte zum Ziel haben, EDA unabhängig von der Implementierung des Event-Brokers umzusetzen. In dem folgenden Kapitel wird sich deshalb auf die Implementierung der Komponenten konzentriert, mithilfe denen dieses Ziel erreicht werden soll.

[38] https://argoproj.github.io/argo-cd/
[39] https://grafana.com/
[40] https://grafana.com/oss/loki/
[41] https://github.com/nats-io/k8s/tree/main/helm/charts/

7.3 Implementierung der notwendigen Infrastrukturkomponenten

Um die in Kapitel 7.1 vorgestellten Services und die jeweiligen Integrationsmuster anwenden zu können, müssen die nötigen Infrastrukturkomponenten implementiert werden. Dafür werden nach der Auslieferung der REST-Services zunächst die HTTP-Proxies für CloudEvents als Grundlage für jegliche Kommunikation über Events betrachtet, bevor die REST-Events-Middleware sowie die CQRS-Middleware implementiert werden. Neben der Implementierung der jeweiligen Komponenten soll vor allen Dingen auch auf die nötige Konfiguration eingegangen werden, um die Komponenten in der Kubernetes-Infrastruktur zu integrieren. Dafür werden fortlaufend Ausschnitte aus Programmcode und Kubernetes-Ressourcen gezeigt – die vollständigen Dateien befinden sich im GitHub Repository[42].

7.3.1 Auslieferung der REST-Services

Weil es sich bei den REST-Services um Services handelt, deren Programmcode nicht verändert werden kann bzw. soll, ist über die grundlegende Erstellung (Kapitel 7.2.1) hinaus nur die Auslieferung in das Cluster interessant. Die Umgebung wird in den folgenden Abschnitten schrittweise durch die Infrastrukturkomponenten ergänzt, um die Services an den Event-Broker anzubinden.

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: accounts-app
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: accounts-app
          image: ghcr.io/manuelottlik/htp-rest-events-accounts
          volumeMounts:
            - name: accounts-pv-storage
              mountPath: /etc/data/db
```

Listing 7.2: Beispielhaftes Deployment für die REST-Services.

[42] https://github.com/manuelottlik/hsh-thesis-prototype/

In Listing 7.2 ist stellvertretend für alle Services eine Kurzform des DEPLOYMENT-Objekts für den ACCOUNTS-Service zu sehen. Alle drei REST-Services exponieren ihre REST-API über den Port `8080`. Dementsprechend wird ein Kubernetes-SERVICE angelegt, der den Port innerhalb des Clusters verfügbar macht und als Load-Balancer für die Instanzen des DEPLOYMENTs dient. Er wird außerdem von der INGRESSROUTE genutzt, um die Services außerhalb des Clusters verfügbar zu machen. Damit die REST-Services ihre Daten über den Neustart des PODs persistieren können, wird außerdem ein PERSISTENTVOLUME angelegt, das über einen PERSISTENTVOLUMECLAIM in die Container gehängt wird.

7.3.2 Implementierung des CloudEvent-Ingestors

Bevor die Services mittels der REST-Events-Middleware an den Event-Broker angebunden werden können, muss der CE-INGESTOR implementiert und ausgeliefert werden. Dieser soll als zentraler Service im Cluster zur Verfügung stehen, um über seine HTTP-Schnittstelle CloudEvents über den in 7.2.3 installierten NATS-Server zu veröffentlichen. Wie bereits in Kapitel 6.1.3 erwähnt, sind die Bibliotheken von CloudEvents zur Anbindung an Event-Broker aktuell unvollständig und bieten darüber hinaus nur eingeschränkte Funktionalitäten in Bezug auf die Übersetzung eines CloudEvents in die Event-Struktur des jeweiligen Brokers. Für den CE-INGESTOR, der in NODE.JS entwickelt werden soll, wird deshalb eine eigene Bibliothek entwickelt, die für die Verbindung mit dem NATS-Server verantwortlich ist und auch die Übersetzung und Veröffentlichung eines CloudEvents in NATS übernimmt. Mithilfe dieser Bibliothek kann dann ein HTTP-Server auf Basis von EXPRESS.JS[43] entwickelt werden, der über einen `POST`-Aufruf CloudEvents nach dem vorgegebenem HTTP-Protocol-Binding entgegennimmt, validiert und je nach Event-Typ in das jeweilige `Subject` von NATS veröffentlicht. Dafür werden die Header des CloudEvents als HTTP-Header mit dem Präfix `ce-` versehen und der Payload des Events als HTTP Body versendet. Wenn das CloudEvent korrekt aufgebaut ist und erfolgreich im Event-Broker veröffentlicht wurde, wird der HTTP-Aufruf mit dem Statuscode `200 OK` beantwortet – ist das Event fehlerhaft aufgebaut oder Event-Broker nicht erreichbar, wird mit einem dementsprechenden Fehlercode (`400 - 599`) geantwortet.

Die Auslieferung des CE-INGESTORs erfolgt mit den gleichen Kubernetes-Objekten wie bei den REST-Services, allerdings kann auf die INGRESSROUTE verzichtet werden, weil der CE-INGESTOR nicht öffentlich verfügbar sein soll. Außerdem wird dem CE-INGESTOR über eine Umgebungsvariable die Zieladresse des NATS-Servers bekannt macht, mit der die Verbindung aufgebaut werden soll.

[43] `https://expressjs.com/`

Die Zieladresse liegt in einer ConfigMap und kann somit von mehreren Kubernetes-Objekten referenziert werden. Eine ähnliche ConfigMap wird auch für die Zieladresse des CE-Ingestors angelegt, um diese Services verfügbar zu machen, die Events produzieren möchten.

7.3.3 Einführung der REST-Events-Middleware

Sobald der CE-Ingestor im Cluster verfügbar ist, können Events in den Event-Broker eingespeist werden. Wie im Aufbau des Prototyps (Kapitel 7.1) erläutert, soll dies über den Einsatz der REST-Events-Middleware geschehen, die als Proxy vor die REST-Services geschaltet wird und die Antworten analysiert. Dafür wird eine weitere Node.js-Applikation entwickelt, die mithilfe der Bibliothek Express.js einen HTTP-Server anbietet und alle eingehenden Anfragen mittels Axios[44] an die Zieladresse des REST-Services weiterleitet.

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: accounts-app
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: accounts-re-middleware
          image: ghcr.io/manuelottlik/htp-re-middleware
          env:
            - name: DESTINATION_URL
              value: http://localhost:8080
            - name: CE_INGESTOR_URL
              valueFrom:
                configMapKeyRef:
                  name: nats-config
                  key: ce-ingestor-url
        - name: accounts-app
          image: ghcr.io/manuelottlik/htp-rest-events-accounts
```

Listing 7.3: Integration der REST-Events-Middleware in das Deployment eines REST-Services.

[44] https://axios-http.com/

Die Antwort wird dann wie in Kapitel 5.2 beschrieben analysiert und im Zweifelsfall ein CloudEvent erzeugt. Dieses CloudEvent wird an den CE-Ingestor gesendet, dessen Zieladresse der REST-Events-Middleware über eine Umgebungsvariable bekannt gemacht wird. Diese Umgebungsvariable wird mithilfe der zuvor beschriebenen ConfigMap befüllt. Der HTTP-Server, den die REST-Events-Middleware bereitstellt, wird dann unter dem Container-Port `8088` exponiert.

Um dafür zu sorgen, dass die REST-Events-Middleware anstelle des REST-Services HTTP-Anfragen entgegennimmt, kann in einer Kubernetes-Infrastruktur auf das in Kapitel 2.2.2 eingeführte Sidecar-Pattern zurückgegriffen werden. Das Deployment der REST-Services wird damit wie in Listing 7.3 gezeigt, um einen weiteren Container im Pod ergänzt. Außerdem verweist das zum Deployment zugehörige Service-Objekt nicht mehr auf den Port `8080` des REST-Services, sondern den Port `8088` der REST-Events-Middleware. Weil der Zugriff auf den REST-Service in Kubernetes in der Regel ausschließlich über einen Service erfolgt, kann die ursprüngliche Schnittstelle „versteckt“ werden. Um die HTTP-Anfragen an den Container des REST-Services weiterleiten zu können, wird der REST-Events-Middleware außerdem der Port des HTTP-Servers vom REST-Service über eine Umgebungsvariable verfügbar gemacht – in diesem Fall `8080`. Die Angabe des Ports reicht in diesem Fall aus, weil die Container eines Pods wie in Kapitel 2.2.2 beschrieben untereinander über den `localhost` erreichbar sind. Weil der Pod in Kubernetes die kleinste Einheit ist, skaliert die REST-Events-Middleware automatisch mit der Anzahl der Instanzen des REST-Services.

Wenn die REST-Events-Middleware außerhalb von Kubernetes eingesetzt werden soll, kann mit einem klassischen Proxy gearbeitet werden, der die HTTP-Anfragen annimmt und an den Service weiterleitet. In Kombination mit Firewall-Regeln, die den ursprünglichen Service „verstecken“, kann eine ähnliche Architektur erreicht werden.

7.3.4 Entwicklung des Customers-Query-Services

Die Events, die durch die REST-Events-Middleware erzeugt und durch den CE-Ingestor in den Event-Broker eingespeist werden, sollen über einen CE-Dispatcher vom Customers-Query-Service konsumiert werden, um für lesende Zugriffe externe Relationen anbieten zu können. Bevor die Implementierung des CE-Dispatchers vorgenommen wird, soll deshalb die HTTP-Schnittstelle des Query-Services entwickelt werden, die die CloudEvents konsumiert. Der Query-Service exponiert somit zwei HTTP-Schnittstellen: Die erste Schnittstelle bearbeitet unter dem Port `8080` die lesenden Zugriffe von Clients, die Informationen über Kunden abrufen möchten, die zweite ist unter Port `8081` nur für die Verarbeitung von CloudEvents verantwortlich, deren Inhalt in die Datenbank geschrieben werden, aus der die lesenden Zugriffe beantwortet werden.

```
ces.post('/relations/assets/created', async (req, res) => {
  const evt = HTTP.toEvent({ headers: req.headers, body: req.body });
  const { data: entity } = await
   ↪  axios.get(`http://${evt.source}/${evt.subject}`);
  await assetService.create(entity);

  res.sendStatus(200);
});
```

Listing 7.4: Verarbeitung eines CloudEvents per HTTP-Endpunkt.

Während die REST-API des Services öffentlich verfügbar sein soll, darf die Schnittstelle für die Verarbeitung von Events nur innerhalb des Clusters für den CE-Dispatcher verfügbar sein. Die REST-API wird ebenfalls mit JSON-Server angeboten und soll deshalb nicht weiter betrachtet werden. Für die CloudEvents-Schnittstelle wird erneut ein HTTP-Server mit Express.js angeboten, der pro Event-Typ einen Endpunkt bereitstellt, unter dem CloudEvents entsprechend des HTTP-Protocol-Bindings via `POST`-Aufruf verarbeitet werden können. Der Customers-Query-Service wird als eigenes Deployment im Cluster installiert und über einen Service verfügbar gemacht, damit er separat vom Command-Service skaliert werden kann – einer der Vorteile von CQRS. Abgesehen von der Skalierbarkeit ist eine ähnliche Auslieferung auch ohne Kubernetes möglich.

Um den Query-Service gemeinsam mit dem Command-Service dem Client gegenüber als eine REST-API anbieten zu können, fehlt nun noch die CQRS-Middleware. Bevor die eingeführt wird, soll jedoch der CE-Dispatcher implementiert werden, um die soeben erläuterte CloudEvents-Schnittstelle mit Events zu versorgen.

7.3.5 Implementierung des CloudEvent-Dispatchers

Um die produzierten Events an den Customers-Query-Service zu senden, soll im nächsten Schritt der CE-Dispatcher entwickelt werden. Im Gegensatz zum CE-Ingestor wird dieser nicht zentral, sondern pro Service ausgerollt, der mit Events versorgt werden soll. Er wird jedoch als separater Service installiert – skaliert also nicht automatisch mit der Anzahl der konsumierenden Service-Instanzen. Da die Verarbeitung der Events jedoch in den Services erfolgt, an die die Events versendet werden, ist eine Skalierung in der Regel nicht notwendig. Der CE-Dispatcher ist ebenfalls als Node.js-Applikation entwickelt, bietet aber keinen HTTP-Server an, weil er ausschließlich Anfragen an die konsumierenden Services mittels Axios sendet. Für die Verbindung mit dem Event-Broker arbeitet er ebenfalls mit der Bibliothek, die auch für den CE-Ingestor ent-

wickelt wurde und bekommt dementsprechend die Zieladresse des Event-Brokers per Umgebungsvariable eingespeist. Mithilfe der Bibliothek werden die Events zurück in CloudEvents übersetzt, mit der Sequenznummer aus dem Broker angereichert und dann an die Zieladresse der Schnittstelle des konsumierenden Services gesendet – zum Beispiel an die des Customers-Query-Service. Diese Zieladresse wird in einer Konfigurationsdatei zusammen mit den zu konsumierenden Event-Topics in den CE-Dispatcher eingespeist. Dabei wird für jedes Topic ein Endpunkt angegeben, unter dem der konsumierende Service das Event verarbeiten möchte. Außerdem wird ein Name vergeben, um die Subscription auch über einen Neustart hinweg aufrecht zu erhalten.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: customers-query-subscriptions-config
data:
  target-url: http://customers-query-service:8081
  subscriptions.yml: |
    - path: /relations/assets/created
      subject: "evt.htp.re.assets.created"
      consumer: customers-query-assets-created
    - path: /relations/assets/updated
      subject: "evt.htp.re.assets.updated"
      consumer: customers-query-assets-updated
    - path: /relations/assets/deleted
      subject: "evt.htp.re.assets.deleted"
      consumer: customers-query-assets-deleted
```

Listing 7.5: ConfigMap für die Bereitstellung eines CE-Dispatchers.

Wenn den CE-Dispatcher ein Event erreicht, das den abonnierten Topics entspricht, wird es nach dem bereits erwähnten HTTP Protocol-Binding für CloudEvents als HTTP-Aufruf an den für das Event konfigurierten Endpunkt der Zieladresse des Services gesendet. Gleichzeitig wird mit diesem Aufruf dem Event-Broker signalisiert, dass sich das Event jetzt in der Verarbeitung befindet. Wird der Aufruf vom konsumierenden Service mit `200 OK` beantwortet, wird dem Event-Broker die erfolgreiche Verarbeitung mitgeteilt – wird mit einem Fehlercode geantwortet, bekommt der Event-Broker dies ebenfalls mitgeteilt, um das Event zu einem späteren Zeitpunkt erneut zuzustellen. Dieser Ablauf wurde bereits in dem UML-Diagramm aus Abbildung 6.2 visualisiert.

Um den CE-Dispatcher in der Infrastruktur bereitzustellen, sind nur ein Deployment und eine ConfigMap (siehe Listing 7.5) nötig, in der die angesprochene Konfigurationsdatei liegt. Weil der CE-Dispatcher weder innerhalb noch außerhalb des

Clusters angesprochen werden soll, sind weder ein Service noch eine IngressRoute nötig. Da dementsprechend kaum besondere Funktionen von Kubernetes genutzt werden, kann ein CE-Dispatcher ebenso unkompliziert in einer Umgebung bereitgestellt werden, die kein Kubernetes nutzt. Die Konfigurationsdatei könnte dann statt als ConfigMap direkt als Datei in der Applikation bereitgestellt werden.

7.3.6 Entwicklung der CQRS-Middleware

Sowohl der Customers-Query-Service als auch der zugehörige Command-Service sind nun einsatzbereit. Um beide Services nach außen hin als eine REST-API darzustellen, soll im nächsten Schritt die CQRS-Middleware entwickelt und in der Infrastruktur installiert werden, um Anfragen je nach HTTP-Verb umzuleiten. Dafür wird eine Node.js-Applikation implementiert, die ähnlich wie die REST-Events-Middleware HTTP-Anfragen mit Express.js annimmt und mit Axios weiterleitet, allerdings je nach HTTP-Verb an die Zieladresse des Query- oder des Command-Services. Beide Zieladressen werden per Umgebungsvariable in die Applikation eingespeist. Herkömmliche REST-APIs senden nach einem manipulierenden HTTP-Aufruf den aktuellen Zustand der veränderten Entität im HTTP-Body. Weil in einer Trennung von lesenden und schreibenden Zugriffen die Hoheit über die Repräsentation der Daten jedoch beim Query-Service liegt, der die vom Command-Service veröffentlichten Events interpretiert, wird der HTTP-Body durch die CQRS-Middleware aus der Antwort des Command-Services entfernt. Außerdem wird der HTTP-Statuscode angepasst: Statt `200 OK` oder `201 Created` wird dem Client der Statuscode `202 Accepted` zurückgegeben. Somit wird signalisiert, dass die Anfrage erfolgreich verarbeitet wurde, die Interpretation aufgrund der Eventual Consistency in CQRS aber erst zu einem späteren Zeitpunkt erfolgt.

Um die CQRS-Middleware im Cluster bereitzustellen, wird wie üblich ein Deployment und davor liegender Service benötigt. Damit eingehende Anfragen verarbeitet werden können, muss im nächsten Schritt die IngressRoute, die auf den Service ursprünglichen Customers-Services so konfiguriert werden, dass sie Anfragen stattdessen auf den Service der CQRS-Middleware leitet. Dieser müssen dann nur noch die Adressen der Services des Query- und Command-Services als Umgebungsvariable verfügbar gemacht werden. Außerhalb einer Kubernetes-Umgebung kann eine ähnliche Bereitstellung erfolgen – auch dann muss die CQRS-Middleware als Service verfügbar sein und allen eingehenden Datenverkehr entsprechend an die Zieladressen des Query- und Command-Services weiterleiten.

7.3.7 Auslagerung des WelcomeLetter-Jobs

Wie bereits in Kapitel 5.3.1 erwähnt, entsteht bereits ab Ausbaustufe 1 der willkommene Nebeneffekt, dass ressourcenintensive Prozesse in separate Services ausgelagert werden können. Ein solcher Prozess ist im Prototyp die Erstellung eines Willkommensbriefs für neue Kunden. Um den Prozess unabhängig vom Customers-Service skalieren zu können, wird eine Node.js-Applikation entwickelt, die einen HTTP-Server mit einem einzelnen Endpunkt anbietet, über den ein CloudEvent verarbeitet werden kann. Der Programmcode dieses Endpunkts ähnelt dabei stark dem Listing 7.4. Dafür wird nach dem Vorbild des Customers-Query-Services ein CE-Dispatcher in der Infrastruktur bereitgestellt, der so konfiguriert wird, dass Events über neue Kunden an den Endpunkt des WelcomeLetter-Jobs gesendet werden. Um die Erstellung des Briefs zu simulieren, wird nach einer Wartezeit von zehn Sekunden eine Textdatei mit dem Namen des Kunden in ein Verzeichnis gelegt. Daraufhin wird ein Event veröffentlicht, das die Infrastruktur über die Erstellung einer Datei informiert. Listing 7.6 zeigt, wie dieses Event mit nur einem Axios-Aufruf veröffentlicht werden kann.

```
await axios({
  method: 'post',
  url: process.env.CE_INGESTOR_URL,
  headers: {
    'ce-source': 'https://welcomeletter.rest-events.manuelottlik.de',
    'ce-type': 'evt.htp.re.file.created',
    'ce-subject': fileId,
  },
  data: {
    id: fileId,
    path: filePath,
    content: fileContent,
  }
});
```

Listing 7.6: Erstellung und Veröffentlichung eines CloudEvents.

Im Gegensatz zum Customers-Query-Service hat dieser Service keine REST-API, die er parallel anbietet. Der Service läuft dementsprechend dauerhaft und wartet darauf, dass vom CE-Dispatcher Events über den Endpunkt zugespielt werden. Inwiefern dieser Umstand noch optimiert werden kann, wird im Ausblick in Kapitel 8.3 betrachtet.

7.3.8 Erstellung der AsyncAPI für alle Services

Abschließend soll für alle Services, die Events konsumieren und / oder produzieren, eine AsyncAPI-Spezifikation angelegt werden, um die Übersicht über den Datenfluss zu behalten. Für folgende vier Services wird dementsprechend eine Schnittstellenbeschreibung angelegt:

1. Assets-Service (als Produzent)
2. Accounts-Service (als Produzent)
3. Customers-Service (als Produzent & Konsument)
4. WelcomeLetter-Job (als Konsument)

Darüber hinaus existiert für die ersten vier Services eine OpenAPI-Spezifikation, die alle Endpunkte der REST-API sowie die zu erwartende Struktur der Antworten enthält. Auf diese Struktur soll in der AsyncAPI-Spezifikation verwiesen werden, weil die Events auf Basis dieser Antworten erzeugt werden. Listing 7.7 zeigt, wie ein Event, das vom Customers-Query-Service konsumiert wird, beschrieben werden kann. So wird zu Beginn eine `publish` Operation für den Channel `evt.htp.re.accounts.created` definiert.

```yaml
channels:
  evt.htp.re.accounts.created:
    publish:
      summary: "An account was created."
      message:
        $ref: "#/components/messages/AccountCreated"
components:
  messages:
    AccountCreated:
      payload:
        $ref: "#/components/schemas/AccountCloudEvent"
  schemas:
    AccountCloudEvent:
      allOf:
        - $ref: "defaults.yml#/components/schemas/DefaultCloudEvent"
        - type: object
          properties:
            data:
              $ref: "../openapi/accounts.yml#/components/schemas/Account"
```

Listing 7.7: Referenz innerhalb der AsyncAPI-Spezifikation auf ein OpenAPI-Dokument.

Die Bezeichnung der Operationen sind dabei nicht aus Sicht des beschriebenen Services, sondern aus Sicht des Lesenden gewählt. Das heißt, dass Anwendungen mit dem beschriebenen Service interagieren können, indem sie ein Event über diesen Channel publishen. Innerhalb der Operation wird auch auf eine `message` beschrieben, deren `payload` sich aus den Standard-Feldern eines CloudEvents (siehe Listing 6.1) zusammensetzt und darüber hinaus das Attribut `data` spezifiziert. Statt einer Beschreibung des Schemas wird an dieser Stelle jedoch mittels JSON-Schema-Referenz auf die OpenAPI-Spezifikation des Accounts-Services verwiesen. Aktualisiert sich das referenzierte Schema, wird automatisch auch die Beschreibung des Event-Schemas angepasst. Mit dem AsyncAPI-Generator[45] und dem dazugehörigen HTML-Template[46] kann dann eine menschenlesbare Dokumentation erstellt werden, wie sie bereits in Abbildung 6.4 dargestellt wurde.

7.4 Beurteilung der Ansätze anhand der vorgestellten Kriterien

In Kapitel 5.1 wurden jeweils drei Kriterien für die Integrationsansätze und die Eventing-Infrastruktur erarbeitet, die erfüllt sein müssen, um EDA erfolgreich in ressourcenorientierten Ökosystemen einzuführen. Dafür wurden in Kapitel 5 und 6 Ansätze vorgestellt, die in Kapitel 7 mithilfe des Prototyps erprobt wurden. In diesem Kapitel soll nun anhand dieses Prototyps beurteilt werden, inwiefern die Ansätze den Kriterien genügen. Dabei werden die Ansätze nicht einzeln, sondern als zusammenhängende Lösung beurteilt.

7.4.1 Bewertung der Integrationsmuster

Wesentlichen Bestandteil dieser zusammenhängenden Lösung stellen die Integrationsmuster dar. Das erste Kriterium, „geringe Einführungskosten“, wird durch die REST-Events-Middleware in Kombination mit den vier aufeinander aufbauenden Integrationsmustern umgesetzt. Durch den Einsatz der Middleware ist keine Anpassung der Services nötig, die Events produzieren sollen, weil es sich um eine fertige Komponente handelt, die nur einer einmaligen Entwicklung bedarf und anschließend nur noch in das Deployment der Services integriert werden muss. Der Prototyp zeigt mit Listing 7.3, dass diese Integration in Kubernetes-Infrastruktur mit wenig Aufwand erfolgen kann, aber auch außerhalb der Provisionierung von Software mittels Kubernetes stellt die Bereitstellung eines Proxies keinen großen Aufwand dar. Durch die REST-Events-Middleware werden Integrationsmuster ermöglicht, die in der ersten Stufe (Kapitel 5.3.1) mit sehr geringem

[45] `https://github.com/asyncapi/generator/`
[46] `https://github.com/asyncapi/html-template/`

Aufwand und niedrigen Kosten umgesetzt werden können, während der Datenaustausch nach wie vor über die etablierten Schnittstellen erfolgt. Deshalb müssen die Antworten des Services weder geeignet sein, um Event-Payloads daraus zu generieren noch muss eine eigene IDL erstellt werden. Somit werden zwar auch nur wenige der Vorteile von EDA ausgeschöpft, jedoch sind bereits die drei kritischsten Integrationsprobleme gelöst (siehe Abbildung 5.2). Dadurch können Eigenschaften wie die *lose Kopplung*, *Fehlertoleranz* und *Unabhängigkeit* von Microservices voll wiederhergestellt werden. Das Kriterium ist somit voll umgesetzt für Systemlandschaften, in denen die Integration vornehmlich über REST-APIs erfolgt, da sie die Grundlage der REST-Events-Middleware und der darauf aufbauend Muster darstellen.

Mit dem zweiten Kriterium werden aufgrund der mangelnden Domänenorientierung existierender CDC-Pattern „pragmatische, domänenorientierte Lösungen“ gefordert. Dieses Kriterium wird durch die REST-Events-Middleware erfüllt, weil sie als CDC-Pattern an der Schnittstelle eines Services statt an seiner Datenbank angebunden wird und dementsprechend auf einer Repräsentation der Daten aufsetzt, wie sie dem Konsumenten ohnehin bereitgestellt werden. Analog zum ersten Kriterium ist diese Anforderung voll umgesetzt, wenn in der Umgebung hauptsächlich mit REST-Schnittstellen gearbeitet wird. Weil mit der Middleware jedoch zur Laufzeit der Datenverkehr mit den Konsumenten analysiert wird, kostet die dadurch gewonnene Domänenorientierung unter Umständen jedoch Performance. Im Prototyp verzögert sich die Antwort des Services aufgrund der Middleware um 5 - 10 Millisekunden – in den meisten Anwendungsfällen ist das Problem somit vernachlässigbar.

Als drittes und letztes Kriterium werden unterschiedliche Ausbaustufen gefordert, die durch die vier aufeinander aufbauenden Integrationsmuster aus Kapitel 5.3 umgesetzt werden und somit derm Kritikpunkt der unverhältnismäßigen Komplexität gerecht werden. Über die erste Stufe, die maßgeblich für die Ermöglichung geringer Einführungskosten verantwortlich ist, besteht durch die weiteren Stufen die Möglichkeit, Bestandssoftware schrittweise und bedarfsorientiert in die Ereignisorientierung einzuführen. Durch den Prototyp wird deutlich, wie mit der CQRS-Middleware aus Kapitel 7.3.6 im Falle des Customers-Services Bestandssoftware ereignisorientierter ausgerichtet werden kann. Dabei wird die komplette Bandbreite bis hin zu Event Sourcing abgedeckt und somit neben bestehender Software auch für Neuentwicklungen mit dem dritten bzw. vierten Muster Möglichkeiten geboten, Software pragmatisch ereignisorientiert zu entwickeln. Die vorgestellten Muster erfüllen das Kriterium, weil weniger Annahmen über die Vorteile von Mustern gemacht werden müssen, die eine große Komplexität mit sich bringen und große Umstellungsaufwände bedeuten würden. Stattdessen kann sich schrittweise in Richtung Ereignisorientierung bewegt werden und nach jeder Ausbaustufe neu abgewogen werden, ob Probleme bestehen, die durch die nächste Ausbaustufe behoben würden. Somit werden der unverhältnismäßigen Komplexität vorgebeugt und die Einführungskosten weiter gesenkt.

Die pragmatischen Alternativen können jedoch auch zu einer Trägheit bei der Umstellung der Systeme auf ereignisorientiertes Design führen: Weil auch ressourcenorientierte REST-APIs in die Lage versetzt werden, Events zu produzieren, könnte statt ereignisorientierter Ansätze beim Design von Neuentwicklungen weiter auf REST-APIs gesetzt werden, die mit der REST-Events-Middleware kombiniert werden. Zwar wäre dieses Vorgehen nach wie vor kostengünstig, kann jedoch nicht die gleichen Vorteile hervorbringen wie eine Anwendung, die nativ ereignisorientiert entworfen wurde und dementsprechend besser geschnittene Events veröffentlichen kann. Um diesem Verhalten vorzubeugen, braucht es organisatorische Vorgaben, die Ereignisorientierung für Neuentwicklungen verankern.

Abschließend lässt sich sagen, dass mit der REST-Events-Middleware und vorgestellten Integrationsmustern alle Kriterien erfüllt werden, die in Kapitel 5.1.1 gefordert wurden somit dem Anspruch an **Pragmatismus** gerecht werden, ohne dabei weitere Eigenschaften von Microservices zu beeinträchtigen. Inwiefern die Anforderungen an die Eventing-Infrastruktur durch die in Kapitel 6 vorgestellten Komponenten erfüllt werden, wird im folgenden Kapitel beurteilt.

7.4.2 Evaluation der eingeführten Infrastrukturkomponenten

Um die Integrationsmuster möglichst nachhaltig in den Services implementieren zu können, wurden auch für die Eventing-Infrastruktur Kriterien erarbeitet, deren Erfüllung in diesem Kapitel bewertet werden soll. Zentrale Voraussetzung für diese Nachhaltigkeit stellen „offene Standards für alle Systeme“ dar, das erste der drei Kriterien. Diese Anforderung wird durch die Verwendung von CloudEvents zur Standardisierung in Kombination mit HTTP-Proxies umgesetzt. Durch ein Nachrichtenformat, das unabhängig vom Broker definiert ist, bestehen keine Probleme mehr bei dem Transport eines Events über mehrere Broker – außerdem können Events von Services erstellt und interpretiert werden, ohne auf eine Bibliothek zurückgreifen zu müssen. Dies wird im Prototyp durch Listing 7.6 deutlich. In Kombination mit den HTTP-Proxies, die diese CloudEvents einspeisen und zustellen, kann jeder Service, der zuvor bereits mit REST in der Lage war, über HTTP zu kommunizieren, an die Eventing-Infrastruktur angebunden werden. Somit ist keine Bibliothek mehr nötig und es kann eine hohe Akzeptanz erreicht werden, weil kein System ausgeschlossen wird, wenn es aufgrund des Alters oder der verwendeten Programmiersprache nicht unterstützt wird. Um einen Service mit Events zu versorgen muss somit nur noch ein CE-Dispatcher installiert werden, der mittels Kubernetes deklarativ konfiguriert werden kann und nur einem Deployment und einer ConfigMap bedarf. Der Prototyp hat mit Listing 7.5 gezeigt, wie mit gerade einmal drei Angaben pro Subscription ein Service mit Events versorgt werden kann. Dadurch werden die Einführungskosten – eigentlich ein Kriterium der Integrationsansätze – weiter gesenkt. Die

Kombination von CloudEvents und den HTTP-Proxies erfüllt die Anforderung an offene Standards somit vollständig. Während CloudEvents einen Standard für das Event bietet, bilden die HTTP-Proxies einen Standard für den Weg dieser Events. Somit sind alle Services, die bereits mit REST-APIs Teil der Integrationslandschaft waren in der Lage, Teil des Ökosystems zu werden.

Die „Unabhängigkeit von einzelnen Event-Brokern“ ist als zweites Kriterium letztendlich ein Effekt der offenen Standards: Auch hier kann aufgrund von standardisierter Event-Struktur, die über HTTP-Proxies transportiert wird, verhindert werden, dass die tiefe Integration eines Event-Brokers aufgrund seines Nachrichtenformats und der verwendeten Bibliotheken den Austausch eben dieses Brokers erschwert. Durch die standardisierte Event-Struktur ist es egal, in welchen Broker ein Event gesendet oder aus welchem es konsumiert wird. Das zeigt der Prototyp in Listing 7.6 und Listing 7.4. Dadurch wird die Struktur vom Broker entkoppelt und es müssen keine Anpassungen gemacht werden, sollte der Event-Broker ausgetauscht werden. Durch die Abwesenheit einer Bibliothek des Brokers in den Services findet auch keine tiefe Integration des Broker-Protokolls mehr statt, weil die brokerspezifische Logik in den HTTP-Proxies gekapselt wird. Die Weiche, in welchen Broker ein Event gesendet wird oder aus welchem es konsumiert werden soll, kann dann in den Proxies gestellt werden. Somit sind die End-Konsumenten und -Produzenten unabhängig vom Event-Broker und es müssen dementsprechend wenig Anpassungen beim Wechsel des Brokers gemacht werden.

Die Kombination dieser zwei Ansätze ermöglicht insgesamt eine starke Entkopplung der Services vom Broker und stellt somit die Eigenschaft *lose Kopplung* und *Unabhängigkeit* von Microservices wieder her. Weil Broker immer aber auch proprietäre Funktionen mit sich bringen, die in CloudEvents über Erweiterungen modelliert würden, ist ein Austausch eines Brokers, der keinerlei Anpassungen und Aufwände erfordert, mehr Vision und ein Ziel, das angestrebt wird, als realistisch umsetzbar. Mit CloudEvents als offen anerkanntem Standard und der Kommunikation über HTTP, einem der am meisten verbreiteten Protokolle in modernen Ökosystemen ist dieses Kriterium mit dieser Einschränkung voll erfüllt.

Mit dem dritten Kriterium wurden „Übersicht über Publisher, Subscriber und Events“ gefordert, um der mangelnden Übersicht durch die lose Kopplung in EDA entgegenzuwirken. Dafür wurde in Kapitel 6.3 AsyncAPI als IDL vorgestellt. Diese kann für jeden Service erstellt werden, der Events produziert oder konsumiert und stellt durch die Maschinenlesbarkeit die Grundlage für unterschiedlichste Einsatzszenarien dar. So können durch die existierenden Open-Source-Werkzeuge Dokumentation generiert werden, wie im Prototyp und in Abbildung 6.4 dargestellt. Wie bereits in Kapitel 6.3.2 detailliert ausgeführt, setzt AsyncAPI auf JSON-Schema und ist deshalb ideal geeignet für Ökosysteme, die eine Transition von ressourcenorientierten REST-APIs zu ereignisgetriebenen Systemen vornehmen. Die Voraussetzung dafür ist, dass für diese REST-APIs bereits OpenAPI-Spezifikationen in guter Datenqualität vorhanden sind, damit die

Datenstrukturen referenziert werden können. Ist dieser Umstand gegeben, erfüllt AsyncAPI die Anforderung, eine Übersicht über alle Komponenten der EDA zu schaffen, da mit geeigneten Werkzeugen nun der Fluss von Events unabhängig von Event-Brokern durch das System nachvollzogen werden kann. Außerdem kann die Spezifikation in einem Event-Katalog oder auf einer API-Plattform in menschenlesbarem Format bereitgestellt werden, um die Transparenz für alle Nutzenden herzustellen.

Nachdem die Kriterien der Integrationsmuster aus Kapitel 5.1.1 erfüllt wurden, lässt sich nach der Implementierung des Prototyps mit dieser Beurteilung zusammenfassen, dass auch die Vorschläge zur Einführung einer ereignisgetriebenen Architektur für die nötige **Zugänglichkeit** der Eventing-Infrastruktur sorgen. Durch die Einführung weiterer Infrastrukturkompnenten werden die in Kapitel 2.1.2 erläuterten Eigenschaften von Microservices wieder voll erfüllt. Somit sind auch die Kriterien aus Kapitel 5.1.2 für REST-Ökosysteme erfüllt.

8 Fazit und Ausblick

Aufbauend auf der Beurteilung der erarbeiteten Integrationsansätze sollen in diesem abschließenden Kapitel nach einer Zusammenfassung der Arbeit der resultierende Handlungsbedarf ermittelt und ein Ausblick auf mögliche zukünftige Herausforderungen und weitere Anwendungsszenarien der geschaffenen Infrastruktur gegeben werden.

8.1 Zusammenfassung

In dieser Arbeit wurden Lösungsansätze zur pragmatischen Einführung offen zugänglicher ereignisorientierter Architektur für Organisationen erarbeitet, die eine Transformation zu Microservices vollziehen, in der Vergangenheit jedoch hauptsächlich auf REST-APIs für standardisierte Schnittstellen der Microservices gesetzt haben.

Um die Integrationsprobleme zu verdeutlichen, die bei einer solchen Schnittstellen-Architektur auftreten, wurden in Kapitel 2 neben einiger konzeptioneller Grundlagen zu REST und dem Betrieb in Containern die Eigenschaften von Microservices erläutert, von denen einige durch die Integrationsprobleme eingeschränkt werden. Als effektivste Lösungsmöglichkeit stach dabei Event-Driven Architecture aus dem Stand der Technik in Kapitel 3 heraus, weil es einen fundamental unterschiedlichen Integrationsstil zugrundelegt. In Kapitel 4 wurden dieser Integrationsstil, die nötigen Broker-Lösungen und die dadurch entstehenden Integrationsmöglichkeiten vorgestellt und einer Bewertung unterzogen, deren Ergebnis jeweils drei Kriterien für Integrationsmuster und die Eventing-Infrastruktur der Integrationsansätze waren. Aufbauend auf diesen Kriterien wurde in Kapitel 5 die REST-Events-Middleware mitsamt vier aufeinander aufbauenden Integrationsmustern als kostengünstiges Mittel für die Einführung von ereignisgetriebenen APIs in von REST dominierten Systemlandschaften vorgestellt.

Diese Integrationsmuster wurden in Kapitel 6 durch die Anbindung mittels CloudEvents über HTTP-Proxies und einer Beschreibung der ereignisgetriebenen APIs mit dem AsyncAPI-Standard für offen zugängliche Eventing-Infrastruktur ergänzt. Die vorgestellten Konzepte wurden daraufhin in Kapitel 7 als eine zusammenhängende Lösung mithilfe des Prototyps implementiert, um die Erfüllung der Kriterien für die einzelnen Integrationsmuster und Architekturvorschläge für die Infastrukturkomponenten zu prüfen.

Nachdem in der abschließenden Bewertung deutlich wurde, dass die Kriterien an pragmatische und offen zugängliche Integrationsansätze mit den vorgestellten Konzepten erfüllt werden, soll nun Fazit gezogen und der resultierende Handlungsbedarf erörtert werden.

8.2 Fazit und Handlungsbedarf

Das Ziel dieser Arbeit – eine schrittweise Transformation zu ereignisgetriebener Architektur, ohne den Status quo bestehender Schnittstellenlandschaften zu missachten – wurde durch die Lösungsansätze erfüllt. Sie können die erarbeiteten Integrationsprobleme lösen und die beeinträchtigten Eigenschaften von Microservices wiederherstellen, ohne andere Eigenschaften zu beeinträchtigen. Auch die in der Einleitung aufgestellte Frage, wie EDA sinnvoll in durch REST dominierte Ökosysteme eingeführt werden kann, lässt sich klar beantworten: Durch die Verwendung der REST-Events-Middleware, CloudEvents und den übrigens Ansätzen entstehen Event-Driven Microservices, die je nach Anforderung in der Lage sind, Daten als Ressource oder Event zur Verfügung zu stellen. Diese Flexibilität eröffnet Möglichkeiten, die über den Schwerpunkt dieser Arbeit, die Integration von Services, hinausgeht.

Event-Driven Architecture und die dafür existierenden Broker-Lösungen sind zwar schon lange kein neues Konzept mehr, allerdings wurden den Technologien und dem Integrationsstil mit der Microservice-Bewegung der letzten Jahre bis hin zu dem FaaS-Ansatz neuer Rückendwind gegeben. Als Resultat existieren viele neue Lösungen am Markt, die untereinander, aber auch im Kontrast zu länger bestehenden Lösungen wie bspw. RabbitMQ große Unterschiede aufweisen. Es ist davon auszugehen, dass in diesem Bereich in den nächsten Jahren eine Standardisierung stattfinden wird, mit der sich Ereignisstruktur und verwendete Protokolle aneinander annähern und der Markt der Broker-Lösungen konsolidiert. CloudEvents und AsyncAPI stellen den Anfang dieser Standardisierung dar und können Zukunftssicherheit für die eingeführte Eventing-Infrastruktur bedeuten.

Die vorgestellte REST-Events-Middleware und die darauf aufbauend Integrationsmuster eignen sich dabei für alle Organisationen, deren Schnittstellenlandschaft in der Vergangenheit hauptsächlich auf das REST-Paradigma vertraut hat. Die Lösungen können für diese Organisationen ein kostengünstiges und aufwandsarmes Vehikel sein, um die schrittweise Transformation dieser Systemlandschaft zu einer ereignisgetriebenen Architektur zu schaffen. Insbesondere die Verzahnung der Schnittstellenbeschreibung über JSON Schema stellt dabei sicher, dass die Definition der angebotenen Daten über die REST- und ereignisgetriebene API nicht auseinander läuft.

Die Verwendung von CloudEvents als standardisierte Event-Struktur und die Beschreibung dieser Events in AsyncAPI sind hingegen auch unabhängig von der bereits bestehenden Infrastruktur einer Organisation und seines Status quo eine sinnvolle Investition.

Sie können mit der Verwendung von HTTP-Proxies als offen zugänglichen Ein- und Ausgangskanal für Events für eine starke Unabhängigkeit der verwendeten Broker-Lösung sorgen. Mit der Einführung dieser Technologien wird die Eventing-Infrastruktur standardisiert und kann auf weitere zukünftige Standardisierungen sowie entstehende Anwendungsszenarien leichter reagieren. Welche Szenarien sich auf Basis dieser Technologien bereits jetzt anbieten, soll deshalb im Folgenden erläutert werden.

8.3 Ausblick

Ein besonders naheliegendes Szenario stellt der bereits erwähnte FaaS-Ansatz dar. Die Motivation hinter der Idee, Software noch kleiner als Microservices zu schneiden und einzelne Funktionen als Service anzubieten, wird bereits im Prototyp anhand des WelcomeLetter-Jobs (Kapitel 7.3.7) deutlich: Der Service beinhaltet bereits nur eine einzige Funktion, läuft jedoch ununterbrochen. Mithilfe von FaaS-Technologie könnte dieser Service nur dann gestartet werden, wenn ein Event in der Queue liegt, das durch den Service verarbeitet werden muss. Der Programmcode wird also nur ausgeführt, wenn ein Event auftritt, dass die Ausführung auslöst. Dieser Ansatz wird bspw. von dem bereits erwähnten Knative oder OpenFaaS verfolgt, die die dafür nötige Technologie bereitstellen. Diese Technologien gehen sogar teilweise noch einen Schritt weiter: Mit sogenannten „Serverless Functions“ wird der Programmcode der Funktion komplett von seiner Umgebung entkoppelt. Wenn eine Funktion ausgeführt werden soll, übernimmt die Technologie die Aufgabe, eine passende Server-Instanz für die Ausführung zu finden.

Neben den Integrationsmöglichkeiten, die durch EDA geboten werden, wird mit der Veröffentlichung von Events aber auch eine neue Datengrundlage geschaffen, aus der Informationen extrahiert werden können. So werden mithilfe von „Complex Event Processing“ in Echtzeit Event Streams analysiert und bei bestimmten Mustern oder Korrelationen von Events komplexere Events erstellt. So kann bspw. der Missbrauch einer Kreditkarte festgestellt werden, wenn innerhalb eines gewissen Zeitfensters zu viele auffällige Transaktionen getätigt wurden. Die zweite Möglichkeit, Informationen aus den veröffentlichten Events zu extrahieren, wird oft als „Business Intelligence“ beschrieben. Wenn alle Events für viele Jahre in einem sogenannten „Data Lake“ gespeichert werden, können Events über große Zeiträume analysiert werden und so bspw. für Strategie-Entscheidungen des Unternehmens genutzt werden. Auch wenn diese Analyse zur Einführung von EDA für einige Organisationen noch keine Priorität haben mag: Solange die Events gespeichert werden, ist die Einführung einer Langzeitanalyse auch noch Jahre später möglich.

Angesichts der vielen Vorteile von Event-Driven Architecture lohnt es sich für viele Organisationen, mit den in dieser Arbeit erarbeiteten Ansätze den ersten Schritt zu gehen und Ereignisorientierung in ihrem Ökosystem einzuführen. Das zeigen die Ergebnisse dieser Untersuchung und der Ausblick auf die weiteren möglichen Anwendungsszenarien.

Literaturverzeichnis

[Apa21a] Apache Software Foundation. Apache kafka. `https://kafka.apache.org/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Apa21b] Apache Software Foundation. Apache kafka – documentation. `https://kafka.apache.org/documentation/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Asy21a] AsyncAPI Foundation. 2.1.0 asyncapi specification. `https://www.asyncapi.com/docs/specifications/v2.1.0/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Asy21b] AsyncAPI Foundation. Asyncapi initiative for event-driven apis. `https://www.asyncapi.com/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Bel20] Adam Bellemare. *Building Event-Driven Microservices*. O'Reilly, Sebastopol, 2020.

[BHW11] Kyle Brown, Gregor Hohpe, and Bobby Woolf. *Enterprise integration patterns : designing, building, and deploying messaging solutions*. Addison-Wesley, Boston, Mass. u.a., 2011.

[Chr19] Binildas Christudas. *Practical Microservices Architectural Pattern : Event-Based Java Microservices with Spring Boot and Spring Cloud*. Apress, Berkeley, 2019.

[Clo21a] Cloud Native Computing Foundation. Cloudevents | a specification for describing event data in a common way. `https://cloudevents.io/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Clo21b] Cloud Native Computing Foundation. Cloudevents specification. `https://github.com/cloudevents/spec/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Clo21c] Cloud Native Computing Foundation. Cloudevents – cncf cloud native interactive landscape. `https://landscape.cncf.io/card-mode?selected=cloud-events`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Clo21d] Cloud Native Computing Foundation. Nats docs. `https://docs.nats.io/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Clo21e] Cloud Native Computing Foundation. Nats.io – cloud native, open source, high-performance messaging. `https://nats.io/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Clo21f] Cloud Native Foundation. Kubernetes documentation. `https://kubernetes.io/docs/home/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[CS16] Cloves Carneiro and Tim Schmelmer. *Microservices from day one : build robust and scalable software from the start*. Apress, New York, 2016.

[DB10] Jürgen Dunkel and Ralf Bruns. *Event-Driven Architecture : Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Springer, Berlin, 2010.

[De17] Brajesh De. *API Management : An Architect's Guide to Developing and Managing APIs for Your Organization*. Apress, New York, 2017.

[Esp15] Dino Esposito. Cqrs and events: A powerful duo. *MSDN magazine*, 30(8):10–10, 2015.

[Fam15] Bob Familiar. *Microservices, IoT, and Azure : Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*. Apress, Berkeley, 2015.

[FL14] Martin Fowler and James Lewis. Microservices. `https://martinfowler.com/articles/microservices.html`, 2014. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Fow05] Martin Fowler. Event sourcing. `https://martinfowler.com/eaaDev/EventSourcing.html`, 2005. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Fow17] Martin Fowler. What do you mean by event-driven? `https://martinfowler.com/articles/201701-event-driven.html`, 2017. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[FZY21] Guo Fu, Yanfeng Zhang, and Ge Yu. A fair comparison of message queuing systems. *IEEE Access*, 9:421–432, 2021.

[Gar21] Gartner Inc. Microservice definition. `https://www.gartner.com/en/information-technology/glossary/microservice/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[HI17] Thomas Hunter II. *Advanced microservices : A Hands-on Approach to Microservice Infrastructure and Tooling*. Apress, Berkeley, 2017.

[IH19] Bilgin Ibryam and Roland Huß. *Kubernetes patterns : reusable elements for designing cloud-native applications*. O'Reilly, Beijing, 2019.

[JHB16] Pooyan Jamshidi, Abbas Heydarnoori, and Armin Balalaie. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[Kre21] Dominik Kress. *GraphQL : eine Einführung in APIs mit GraphQL*. dpunkt.verlag;, Heidelberg, 2021.

[Lin21] Linux Foundation. grpc documentation. `https://grpc.io/docs/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[MG20] Moisés Macero García. *Learn Microservices with Spring Boot : A Practical Approach to RESTful Services Using an Event-Driven Architecture, Cloud-Native Patterns, and Containerization*. Apress, Berkeley, 2020.

[New15] Sam Newman. *Building Microservices*. O'Reilly, Beijing, 2015.

[New19] Sam Newman. *Monolith to Microservices : Evolutionary Patterns to Transform Your Monolith*. O'Reilly, Sebastopol, 2019.

[NMAM16] Irakli Nadareishvili, Ronnie Mitra, Michael Amundsen, and Matt McLarty. *Microservice architecture : aligning principles, practices, and culture*. O'Reilly, Beijing, 2016.

[Ope21] OpenAPI Foundation. Home - openapi initiative. `https://www.openapis.org/`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Pat17] Sanjay Patni. *Pro RESTful APIs : Design, Build and Integrate with REST, JSON, XML and JAX-RS*. Apress, New York, 2017.

[Ric19] Chris Richardson. *Microservices patterns : with examples in Java*. Manning, Shelter Island, 2019.

[RR07] Sam Ruby and Leonard Richardson. *RESTful Web Services, Safari Books Online*. O'Reilly, Beijing, 2007.

[SI18] Prabath Siriwardena and Kasun Indrasiri. *Microservices for the Enterprise : Designing, Developing, and Deploying*. Apress, New York, 2018.

[Sma21] SmartBear Inc. Callbacks, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[TM21] Martin Thompson and Todd Montgomery. How did we end up here? `https://www.youtube.com/watch?v=oxjT7veKi9c&t=1578s`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Var16] Ervin Varga. *Creating Maintainable APIs : A Practical, Case-Study Approach*. Apress, New York, 2016.

[VMV21] VMVare Inc. Documentation – rabbitmq. `https://www.rabbitmq.com/documentation.html`, 2021. Letzter Zugriff am 25.10.2021, 13:37 Uhr.

[Voh16] Deepak Vohra. *Kubernetes Microservices with Docker, The expert's voice in open source*. Apress, New York, 2016.

[Voh17] Deepak Vohra. *Docker Management Design Patterns : Swarm Mode on Amazon Web Services*. Apress, Berkeley, 2017.

[ZRV+19] Svetoslav Zhelev, Anna Rozeva, George Venkov, Vesela Pasheva, and Nedyu Popivanov. Using microservices and event driven architecture for big data stream processing. *AIP Conference Proceedings*, 2172(1):8, 2019.