HAW HAMBURG

BACHELORTHESIS
Thomas Knaus

Entwicklung eines prototypischen Services zur Realisierung des Event Sourcing Entwurfmusters in einer serverlosen Softwarearchitektur

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Thomas Knaus

Entwicklung eines prototypischen Services zur Realisierung des Event Sourcing Entwurfmusters in einer serverlosen Softwarearchitektur

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr.-Ing. Olaf Zukunft

Eingereicht am: 17. April 2020

Thomas Knaus

Thema der Arbeit

Entwicklung eines prototypischen Services zur Realisierung des Event Sourcing Entwurfmusters in einer serverlosen Softwarearchitektur

Stichworte

Event Sourcing, serverlose Softwarearchitektur, CQRS

Kurzzusammenfassung

Durch Service Anbieter wie Amazon Web Services oder Google Cloud Platform wird die Entwicklung von Microservices immer weiter voran getrieben. Eine Architektur, die durch die Nutzung dieser Service Anbieter ermöglicht wird, ist die serverlose Softwarearchitektur. Diese Bachelorthesis befasst sich mit dem Event Sourcing Entwurfsmuster und wie dieses auf Basis einer solchen serverlosen Architektur umgesetzt werden kann. Hierfür wird ein prototypischer Service entwickelt, der die Funktionalität des Event Sourcings andere Microservices ermöglichen soll.

Thomas Knaus

Title of Thesis

Development of a prototype service to realize the event sourcing design pattern in a serverless software architecture

Keywords

Event Sourcing, serverless software architecture, CQRS

Abstract

Through service providers such as Amazon Web Services or Google Cloud Platform, the development of microservices is being driven forward. One architecture made possible by the use of these service providers is the serverless software architecture. This bachelorthesis deals with the event sourcing design pattern and how it can be implemented

on the basis of such a serverless architecture. For this purpose, a prototypical service will be developed, which will take over the functionalities of event sourcing for other microservices.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit der Thematik des Event Sourcing Entwurfmusters und wie dies in einer serverlosen Softwarearchitektur, als eigenständiger Service, realisiert werden kann. In der Einleitung wird ein erster Einblick in das Thema gegeben und welche Motivation hinter dieser Arbeit steckt. Des Weiteren wird der Aufbau der Arbeit und der einzelnen Kapitel vorgestellt.

1.1 Motivation

Die Entwicklung von Microservices und serverlosen Softwarearchitekturen wird durch Service Anbieter wie Amazon (Amazon Web Services - AWS), Google (Google Cloud Platform) oder Microsoft (Microsoft Azure) immer weiter vorangetrieben. Die auf solchen Plattformen betriebenen Anwendungen bestehen oftmals aus vielen Mircroservices, die gemeinsam ein Gesamtsystem darstellen. So werden beispielsweise mit AWS Lambda Functions Funktionen entwickelt, die nur einen sehr kleinen Kontext besitzen und meistens zur Erfüllung einer einzigen Aufgabe im Gesamtsystem dienen. Ein sehr wichtiger Punkt in der Entwicklung solcher Microservices stellt dabei die Kommunikation zwischen den Diensten dar. Diese findet beispielsweise über eine Message oriented Middlware (MOM) statt. Mit Hilfe der die Dienste per Events kommunizieren. In einer solchen Event getriebenen Softwarearchitektur gewinnt das Entwurfsmuster Event Sourcing zunehmend an Popularität. Event Sourcing dient dabei zum perstitenten Speichern von in einem System auftretenden Events in einem Event Store. Zum Zeitpunkt der Anfertigung dieser Arbeit gibt es noch keine Cloud-basierten Lösungen für einen solchen Dienst.

1.2 Zielsetzung

Ziel dieser Arbeit ist eine prototypische Implementierung des Event Sourcing Entwurfsmusters als eigenständiger Dienst. Dieser Dienst soll dabei auf einer serverlosen Softwarearchitektur basieren. Der entwickelte Dienst wird anschließend zur Evaluation herangezogen, um eine qualifizierte Aussage treffen zu können, ob ein solcher Dienst als Cloud-basierte Lösung umgesetzt werden kann. Welche Herausforderungen dabei bestehen und ob eine serverlose Umsetzung überhaupt sinnvoll ist.

1.3 Aufbau

Der Aufbau der Arbeit gliedert sich in die nachfolgend aufgeführten Kapitel.

Grundlagen

In diesem Kapitel wird das Event Sourcing Entwurfsmuster und die damit in Zusammenhang stehenden Begrifflichkeiten erläutert. Diese sind für das Verständnis der Arbeit relevant.

Anforderungen

In dem Kapitel Anforderungen werden die an den zu implementierenden Dienst gestellten Anforderungen definiert.

Entwurf

Nachdem die Anforderungen an den Dienst beschrieben wurden, werden in dem darauf folgenden Kapitel die Technologieentscheidungen beschrieben, sowie die Architektur des implementierten Prototypen erläutert.

Implementierung

In diesem Kapitel wird die Implementierung des Dienstes erklärt. Dabei wurde sich an die zuvor beschriebenen Technologieentscheidungen und Architektur gehalten.

Qualitätssicherung

Um den ersten Prototypen auf einen akzeptablen Entwicklungsstand zu bringen, wurden die in diesem Kapitel beschriebenen qualitätssichernden Maßnahmen umgesetzt. Zudem

wird auf weitere Maßnahmen eingegangen, die für die Sicherung der Qualität des Dienstes beitragen können.

Evaluation

In diesem Kapitel wird der entwickelte Event Sourcing Service auf die Erfüllung der definierten Anforderungen überprüft. Dabei wird ebenfalls untersucht, ob dieser auch unter hoher Last das gewünschte Verhalten aufweist.

Ausblick und Fazit

Zum Schluss der Arbeit wird auf die weiterführenden Maßnahmen und Möglichkeiten bei der Weiterentwicklung des Prototypen eingegangen. Anschließend wird ein Fazit über den entwickelten Dienst, sowie das Event Sourcing Entwurfsmuster in einer serverlosen Softwarearchitektur getroffen.

2 Grundlagen

2.1 Definitionen

In diesem Kapitel werden wir auf das Event Sourcing Entwurfsmuster eingehen. Davor definieren wir zunächst, die dafür benötigten Begrifflichkeiten. Dabei ist eine Unterscheidung der Kommunikationsart mittels Events und Commands wichtig, worauf im folgenden Kapitel eingegangen wird.

2.1.1 Events

Ein Event ist ein eigenständiges und unveränderliches Objekt. Dieses beinhaltet Informationen über ein Ereignis, welches zu einem bestimmten in der Vergangenheit liegenden Zeitpunkt eintrat.[7]

Events werden für eine unidirektionale Kommunikation verwendet. Das bedeutet, dass durch das Publizieren eines Events, der Sender des Events, keine direkte Antwort des Empfängers erwartet.[34]

Die Erzeugung eines Events kann dabei unterschiedlichste Gründe haben. Wie zum Beispiel, Zustandsänderungen innerhalb einer Applikation, die durch die Interaktion eines Nutzers mit der Applikation entstehen. Ein weiteres Beispiel wäre, ein durch einen Temperatursensor periodisch gemessener Wert, der per Event versendet wird. Das Encoding eines Events kann in verschiedenen Formen auftreten. Als String, JavaScript Object Notation (JSON) oder auch in einem beliebigen Binärformat.[7]

2.1.2 Commands

Bei einem Command handelt es sich, im Vergleich zu einem Event, um eine Aktion, die ausgeführt wird. Unter der Verwendung des Event Sourcing Entwurfmusters ist ein Command als erste Anfrage eines Nutzers/Clients an einen Service zu verstehen, der zunächst von dem Service validiert werden muss. [7] Ist die Validierung erfolgreich, so wird anschließend ein unveränderliches Ereignis - also ein Event, erzeugt und publiziert. Wenn sich beispielsweise ein Nutzer bei einer Webseite registrieren möchte und dieser eine Anfrage mit E-Mail Adresse und Passwort an einen Login Service sendet, so handelt es sich dabei um einen Command. Dieser Command müsste zunächst validiert werden, bevor nach der erfolgreichen Überprüfung der E-Mail Adresse, durch den Login-Service ein entsprechendes Event generiert wird. [7]

2.1.3 Streams

Eine weitere wichtige Definition für die Arbeit mit dem Event Sourcing Entwurfsmuster sind Streams. Dabei handelt es sich um eine Sequenz von aufeinander folgender Events, bei der nicht klar ist, wann diese zu Ende ist. Die Verarbeitung von Streams findet meist mit Streamingplatformen - wie zum Beispiel Apache Kafka oder AWS Kinesis DataStreams - statt und wird als Stream Processing bezeichnet.

2.2 Event Sourcing

Event Sourcing ist ein Entwurfsmuster, dass das persistente Speichern aller in einem System auftretenden Events in einem Event Store ermöglicht. Dabei werden die Events in ihrer chronologischen Reihenfolge unveränderlich gespeichert. [41]

Im Vergleich zu der Speicherung von Aggregaten in einer Art von „current-state-only“ [36] Datenbank, verfolgt Event Sourcing den Ansatz, jedes aufgetretene Event, das ein Aggregat beeinflusst, persistent zu speichern.

Betrachten wir als Beispiel ein Aggregat "Bestellung", dessen Datenbankeintrag in Tabelle 2.1 zu sehen ist. Führen wir nun die folgenden CRUD Operationen auf diese Tabelle aus, so erhalten wir das in Tabelle 2.2 zu sehende Resultat der Operationen.

Durchgeführte CRUD Operationen:

- Erhöhen der Anzahl des Produktes mit der product_id 324 um 1
- Verringern der Anzahl des Produktes mit der product_id 222
- Entfernen des Produktes mit der product_id 531

id	product_id	quantity	timestamp
1	324	1	1565984911
2	222	2	1565984923
3	214	3	1565984957
4	531	1	1565985011

Tabelle 2.1: Ausgangstabelle Bestellung - CRUD

id	product_id	quantity	timestamp
1	324	2	1565986001
2	222	1	1565987321
3	214	3	1565984957

Tabelle 2.2: Ergebnistabelle Bestellung - CRUD

Das selbe Aggregat aus Tabelle 2.1, das in einem Event-Store gespeichert wurde, ist in Abbildung 2.1 zu sehen und hat den selben Ausgangszustand. Wie dabei zu erkennen ist, werden hier alle Zustandsänderungen des Aggregates als einzelne Events abgelegt. Diese Sequenz hält dabei alle Zustandsänderungen in einer unveränderlichen und chronologischen Reihenfolge. Wird nun ein Aggregat, das sich in dem Event-Store befindet, instanziiert, so werden alle Events dieses Aggregates, beginnend vom ältesten Event bis zum neusten Event, verarbeitet. Anschließend hält das Datenobjekt den aktuellen Zustand dieses Aggregates.

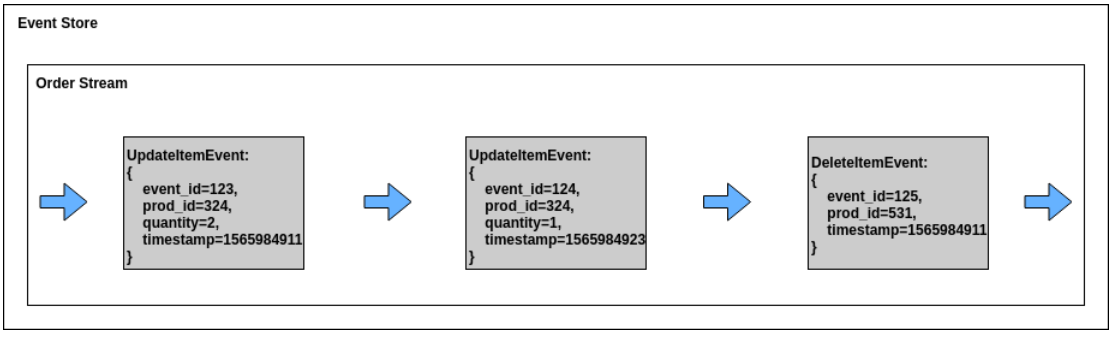

Abbildung 2.1: Event Store mit einem Order Stream

Durch das Speichern aller in einem System aufgetretenen Events, wie es bei dem Event Sourcing Entwurfsmuster betrieben wird, bieten sich einige Vorteile. Es kann dadurch zum Beispiel der Zustand jedes einzelnen Aggregates klar nachvollzogen werden und somit auch der gesamte Systemzustand. Des weiteren ermöglicht es eine einfache Systemanalyse, die für beispielsweise Datenauswertungen relevant sein kann. Durch das wiederholte Einspielen aller Events des Event Stores kann zudem vereinfacht ein parallel laufendes Test-, Staging- oder Development-System aufgebaut werden. Ein weiterer wichtiger Vorteil, der durch Event Sourcing entsteht, ist, dass keinerlei Informationen verloren gehen. Es könnte der Fall eintreten in dem Informationen, die bis zu einem gewissen Zeitpunkt noch keine Rolle für das Unternehmen spielten, plötzlich eine ganz andere Relevanz haben. Im Vergleich zu einem System, das einem klassischen Architekturmuster mit einem relationalen Datenbankmanagementsystem folgt, gehen durch jede Update- und Delete-Operation Informationen verloren. [10]

Durch die Verwendung des Event Sourcing Entwurfmusters kann jedoch das Problem auftreten, dass die Zustände, die aus den Event Streams hervorgehen und der Zustand, den die einzelnen Services eventuell in einer Datenbank halten, differieren. Dieses Problem wird dadurch behoben, dass die Events zur „Source of truth“ [35] gemacht werden und nicht mehr die Datenbankeinträge, die die einzelnen Services führen.

2.2.1 Projections

In SQL-Datenbanksystemen ist eine Abfrage über mehrere Einträge hinweg relativ einfach umzusetzen. Um diese Funktionalität auch in einem Event Store zu verwenden, werden so genannte Projections eingesetzt.

Projections dienen dazu Query Anfragen auf Basis von Event-Streams auszufüren. Dabei wird ein oder mehrere Streams von der Projection-Funktion eingelesen. Die dabei gelesenen Events werden darauf überprüft, ob diese für die Query relevant sind und anschließend auf einen Stream gemapped.[37]

Projections können dabei mit Materialized Views aus dem Bereich der relationalen Datenbanken verglichen werden. Materialized Views stellen das Ergebnis einer Datenbankabfrage in einer eignen Tabelle dar.[42]

2.2.2 Snapshots

Ein weiterer Mechanismus, der für Event Sourcing Relevant ist, ist das Anlegen von Snapshots. In einem System, das Event Sourcing nutzt, können die Event Streams über die Zeit sehr groß werden und dabei sehr viele Events beinhalten. Wodurch sich die Perfomance von Leseoperationen verschlechtert. Dieses Problem kann durch die Verwendung von Snapshots verbessert werden.

Snapshots repräsentieren einen Status des Aggregates zu einem gewissen Zeitpunkt. Wenn eine Anwendung ein Aggregat instanziieren möchte, nutzt diese den letzten Snapshot als initialen Wert und verarbeitet alle darauf folgenden Events des Event Streams. Dadurch muss nicht der gesamte Event Stream durchlaufen werden und die Perfomance verbessert sich. [38]

2.3 CQRS

In Kombination mit dem Event Sourcing Pattern wird sehr häufig das Command Query Responsibility Segregation (CQRS) Pattern verwendet. Grund dafür sind zum einen, die zuvor erwähnten Projections.

In seinen Grundzügen bedeutet CQRS die Auftrennung der Domain Modelle in ein Lese- und in ein Schreib-Modell. Commands, die ein Client an das System sendet, werden somit mittels dem Schreib-Modell verarbeitet. Bei einer Anfrage an den Service erhält der Client dabei das Lese-Modell als Antwort zurück. [5]

Durch diese Aufteilung ist es möglich, die dadurch voneinander unabhängigen Seiten - das Lesen und das Schreiben - hinsichtlich der Performance zu verbessern. Ein Nachteil durch diese Aufteilung ist, dass das Lese-Modell dem Schreib-Modell hinterher hängt. Dies führt zu Eventual Consistancy und somit dazu, dass es passieren kann, dass Nutzer nicht ihre gerade geschriebenen Änderungen direkt lesen können "read-your own writes".[5]

2.4 Serverlose Softwarearchitektur

Wie bereits in der Einleitung dieser Arbeit beschrieben, soll der zu entwickelnde Prototyp auf einer serverlosen Softwarearchitektur basieren. Die von den großen Cloud-

Anbietern bereitgestellten Dienste reichen dabei von Infrastructur as a Service (IaaS) bis hin zu Function as a Service (FasS). Welche unterschiedlichen Komponenten die jeweiligen Dienste beinhalten, sind in Abbildung 2.2 zu sehen. So wird zum Beispiel, bei einem Dienst der IaaS anbietet, die Hardware und Virtualisierung dieses von dem Cloud-Anbieter verwaltet. Bei einem Dienst, der eine Funktion als Service anbietet, wird bis auf die Anwendung selbst alles durch den Anbieter verwaltet. Eine serverlose Softwarearchitektur basiert dabei auf Diensten, die in dem Bereich der Functions as a Service anzusiedeln sind.

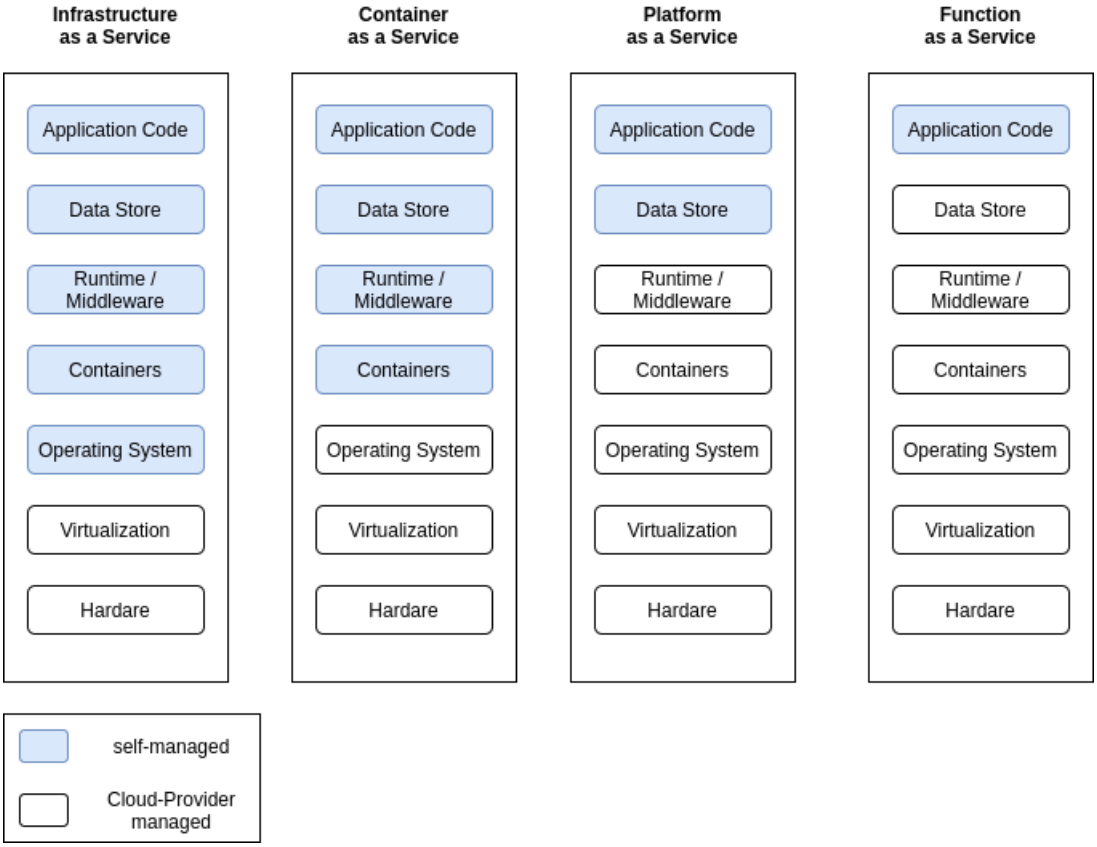

Abbildung 2.2: Abstraktionsebenen von IaaS bis FaaS

3 Anforderungen

Im folgenden Kapitel werden die funktionalen, sowie auch die nicht-funktionalen Anforderungen an den Service definiert. Diese ergeben sich dabei aus Anforderungen, die durch Rücksprache mit dem Unternehmen entstanden sind, sowie aus Anforderungen, die aus der Definition des Event Sourcing Entwurfmusters hervorgehen.

3.1 Funktionale Anforderungen

In diesem Abschnitt wird auf die funktionalen Anforderungen des zu entwickelnden Dienstes eingegangen. Diese Anforderungen beschreiben das gewünschte Verhalten des Dienstes und wie bestehende Dienste diesen verwenden und einbinden können.

3.1.1 Publizieren von Events

Der erste Prototyp des Event Sourcing Services soll eine Eingangsschnittstelle anbieten, über die andere Dienste Events veröffentlichen können. Die über diese Schnittstelle veröffentlichten Events sollen dabei persistiert werden.

3.1.2 Konsumieren von Events

Neben dem veröffentlichen von Events, soll der Event Sourcing Service, eine Möglichkeit zum konsumieren der publizierten Events anbieten.

3.1.3 Replay

Eine weitere Funktionalität, die durch den Service gegeben sein soll, ist das Starten von Replays. Hierfür sind zunächst drei unterschiedliche Verfahren von Replays umzusetzen.

Gesamt-Replay: Spielt alle in dem Event Store hinterlegten Events wieder ein.

Zeitstempel basiertes Replay: Bietet die Möglichkeit, alle Events, die nach einem bestimmten Zeitpunkt eingetreten sind wieder einzuspielen.

Aggregat basiertes Replay: Bietet die Möglichkeit, die Events für ein bestimmtes Aggregat wieder einzuspielen. Wodurch eine einfache Analyse bestimmter Aggregate ermöglicht werden soll, ohne alle Events erneut einspielen zu müssen.

3.2 Nicht-funktionale Anforderungen

Die nachfolgenden Nicht-Funktionalen Anforderungen sollen von dem zu implementierenden Service erfüllt werden.

3.2.1 Serverlose Softwarearchitektur

Dadurch, dass die komplette Softwarearchitektur des bisherigen Gesamt-Systems auf einer serverlosen Softwarearchitektur basiert, ist auch dies für den Event Sourcing Service ein Muss. Das bedeutet weiterführend, dass keine selbst verwaltete Serverinstanz genutzt werden kann, sondern auf die FaaS Angebote des genutzten Cloud Providers zurückgegriffen werden sollte.

3.2.2 Zuverlässigkeit

Die Zuverlässigkeit des Dienstes sollte gegeben sein. Das bedeutet zum einen, dass die über den Service veröffentlichten Events zuverlässig gespeichert werden und das Konsumieren dieser stets möglich ist. Die Verfügbarkeit des Dienstes sollte somit stets gegeben sein.

3.2.3 Erweiterbarkeit

Neben der Zuverlässigkeit des Dienstes, sollte dieser ebenfalls einfach erweitert werden können. Dies bedeutet konkret, dass die zu implementierende REST API einfach um zusätzliche Endpunkte erweitert werden kann. Sowie auch die Komponenten des Dienstes klar definierte Schnittstellen besitzen, um diese gegebenenfalls leicht austauschen zu können.

3.2.4 Benutzbarkeit

Des Weiteren sollen die durch den Dienst angebotenen Endpunkte intuitiv und einfach verwendbar sein. Wodurch gewährleistet werden soll, dass die Nutzung auch für diejenigen möglich ist, die bisher noch keine Kenntnisse über den Dienst haben und wie dieser im Kern aufgebaut ist.

4 Entwurf

In diesem Kapitel wird auf den Entwurf des Event Sourcing Services eingegangen. Dabei werden zunächst die Technologieentscheidungen beschrieben, welche sich zum größten Teil aus den Anforderungen ergeben. Des Weiteren werden die einzelnen Sichten der Softwarearchitektur dargestellt.

4.1 Technologieentscheidungen

Die Technologieentscheidungen, die für die Umsetzung des Event Sourcing Service getroffen wurden, werden nachfolgend beschrieben. Dabei ist die Anforderung diesen Dienst komplett serverlos umzusetzen zu beachten, da nicht jede Technologie dieses Kriterium erfüllt.

4.1.1 Cloud-Anbieter

Die grundlegendste Entscheidung, die getroffen werden muss ist, welcher Cloud-Anbieter für die Umsetzung genutzt werden soll.

Im Fall dieser Arbeit ist diese Entscheidung schnell getroffen. Für die Umsetzung wird Amazon Web Services (AWS) als Anbieter genutzt. Der Grund für diese Entscheidung liegt zum einen darin, dass das Unternehmen in dessen Kooperation diese Arbeit erstellt wird, auf diesen Anbieter setzt. Es ist im Unternehmen somit bereits ein fundiertes Wissen über viele, der von AWS angebotenen Dienste vorhanden, als auch über die Verwendung und Implementierung dieser. Ein weiterer Grund für diese Entscheidung ist, der von AWS angebotene AWS Educate Zugang für Studenten und Professoren. Dieser bietet ein gratis Kontingent von 40$ bis zu 100$ an, wodurch auch Dienste wie beispielsweise AWS Kinesis genutzt werden können [20]. Zusätzlich bietet AWS das Free-Tier Programm an, durch

welches viele von den durch AWS bereitgestellten Dienste, für ein Jahr, gratis genutzt werden können [21].

Einer der bekanntesten Dienste von AWS ist der EC2 Service, welcher das Bereitstellen von Containern ermöglicht und im Bereich Container as a Service anzusiedeln ist. Ein weiterer Service sind die Lambda Functions, welche in den Bereich Functions as a Service gehören. Im weiteren Verlauf dieses Kapitels werden die genutzten Dienste, die zur Realisierung des Event Sourcing Services beitragen vorgestellt.

4.1.2 REST-API

Die Kommunikation mit dem ersten Prototypen des Dienstes soll über eine REST-API stattfinden. Diese API wird über den AWS API Gateway Service in Kombination mit einer AWS Lambda Funktion bereitgestellt.

AWS API Gateway

Der AWS API Gateway Dienst bietet die Möglichkeit des Bereitstellens von REST-APIs und WebSockets. Es können somit API Endpunkte nach dem REST Paradigma definiert werden und mit Backend-Funktionen verbunden werden. Jeder URL Endpunkt wird dabei mittels einer HTTP Anfrage in Kombination mit einem HTTP Verb angesprochen. Diese Endpunkt - HTTP Verb Kombinationen können dabei eigenen Funktionen zugeordnet werden. Die Funktionen, die durch die Endpunkte ausgelöst werden, werden in diesem Fall AWS Lambda Funktionen sein.[11]

4.1.3 Event Verarbeitung

Für die Verarbeitung der gegenüber dem Service veröffentlichten Events wird eine AWS Lambda Funktion verwendet, welche alle publizierten Events abspeichert. Das Weiterleiten der Events hin zu den Projektionen, die diese konsumieren wird durch die Verwendung eines DynamoDB Streams, einer weiteren AWS Lambda Funktion und eines AWS Kinesis Datenstroms umgesetzt.

AWS Lambda Functions

Der AWS Lambda Functions Service bietet die Möglichkeit Funktionen bei bestimmten Ereignissen auszuführen. Dabei werden die Funktionen in einem Container ausgeführt. Ein Container ist dabei eine isolierte Einheit, die sich mit anderen Containern einen Server teilt. Jeder Container erhält dabei eine begrenzte Anzahl an Rechenkapazität, Speicher und Arbeitsspeicher. Bei der Bereitstellung von AWS Lambda Funktionen kann die gewünschte Entwicklungsumgebung ausgewählt werden. Des Weiteren kann die Timeout-Zeit der Funktionen definiert werden, sowie die verfügbare Speichergröße. Die gewählte Speichergröße gibt dabei auch die CPU Rechenkapazität der jeweiligen Funktion vor. [12]

AWS Kinesis

AWS Kinesis Data Streams ist der von Amazon angebotene Datastreaming-Service. Dieser Service bietet die Möglichkeit einer Datenverarbeitung in Echtzeit. Mit Kinesis Data Streams können dabei mehrere Datenströme erstellt werden. Befüllt werden die Datenströme über sogenannte Producer die Events an den Stream Endpunkt von Kinesis senden. Jeder Stream kann dabei von mehreren Konsumenten (Consumern) abonniert werden. Das bedeutet, sobald ein Producer Events über den Stream veröffentlicht, werden diese an die jeweiligen Consumer weitergeleitet.

AWS Kinesis Data Streams bietet dazu die Möglichkeit in einem Stream veröffentlichte Events zu persistieren. Jedoch beträgt die maximale Dauer, in der die Events gespeichert werden 7 Tage.[15] Die maximale Dauer in der die Events gespeichert werden, ist im Fall des Event Sourcing Services jedoch ausreichend, da der AWS Kinesis DataStream nur zur Übermittlung an die Konsumenten dient. Die dauerhafte Persistierung der Events wird mittels einer DynamoDB realisiert.

AWS DynamoDB Streams

Mit Hilfe von DynamoDB Streams können die Aktivitäten einer DynamoDB Tabelle erfasst werden. Werden Operationen wie das Erstellen, Aktualisieren oder Löschen von Tabelleneinträgen durch eine Anwendung ausgeführt, wird somit ein dementsprechendes Event über den DynamoDB Stream gesendet. Diese Informationen werden dabei bis zu 24 Stunden in dem Stream gehalten. Der Stream liefert die Ereignisse in der Reihenfolge

in der diese in der DynamoDB aufgetreten sind.[19] Dabei gibt es jedoch gewisse Einschränkungen, die beachtet werden müssen, auf welche in Kapitel 5 weiter eingegangen wird. Der DynamoDB Streams Dienst ist dabei im Kern sehr ähnlich zu dem des Kinesis DataStreams, jedoch nicht komplett identisch. [29] Beide Dienste nutzen einzelne Shards für die Stream Verarbeitung und bieten eine fast identische API an.

AWS Step Functions

Ein weiterer Dienst, der in dem Event Sourcing Service verwendet werden wird, sind die AWS Step Functions. Durch Step Functions können endliche Automaten definiert werden, welche aus verschiedenen AWS Diensten bestehen können. Wie zum Beispiel AWS Lambda Funktionen.[26] Das Erstellen der Automaten wird dabei mittels einer auf JSON basierten Sprache entwickelt, der Amazon State Language. Dieser Service wird später für die Umsetzung der Replay Verfahren zum Einsatz kommen.[24]

4.1.4 Persistierung

Der Event Sourcing Service benötigt eine Möglichkeit zur Persistierung der Events, da wie bereits erwähnt der AWS Kineis Data Streams Service keine dauerhafte Speicherung der Events ermöglicht. Für das Speichern der Events wird daher, die von AWS angebotene DynamoDB Datenbank verwendet.

AWS DynamoDB

Bei der AWS DynamoDB handelt es sich um eine NoSQL Datenbank. Es gibt verschiedene Typen von NoSQL Datenbanken. Graph-, Dokumenten-, Spalten-basierte und Key-Value Datenbanken. Die AWS DynamoDB ist dabei im Bereich der Key-Value Datenbanken anzusiedeln.[44] Einige Merkmale der DynamoDB sind, dass diese eine schnelle und einfache Skalierung ermöglicht, die nicht selbst verwaltet werden muss. Da die DynamoDB über mehrere Shards verteilt sein kann, spricht man bei der Datenkonsistenz der Datenbank von Eventual Consistency. Dies bedeutet, dass der Fall auftreten kann, dass gerade geschriebene Änderungen bei der nächsten Lese-Operation noch nicht gelesen werden - wenn die Schreib-Operation noch nicht auf allen Shards durchgeführt worden ist. Dennoch bietet DynamoDB die Option von strenger Konsistenz bei Lese-Operationen. Was jedoch mit einigen Nachteilen verbunden ist. Wie zum Beispiel

einer höheren Latenz der Lese-Operationen und eine steigende Anzahl an benötigten Lesezugriffseinheiten.[17]

4.2 Architektursichten

Vor der Implementierung des Event Sourcing Services entwerfen wir zunächst die einzelnen Architektursichten des Dienstes.[32] Starten werden wir dabei mit der Bausteinsicht, die das System zunächst als Blackbox darstellt und auch die Kontextabgrenzung des Dienstes veranschaulicht. Anschließend werden wir die detaillierteren Ansichten der einzelnen Dienste als Whitebox entwerfen. Zum Schluss werden wir noch die Laufzeitsichten der einzelnen Nutzungsfälle des Event Sourcing Services modellieren.

4.2.1 Bausteinsicht

Im folgenden definieren wir die Bausteinsicht des Dienstes. Dabei werden die einzelnen Abstraktionsebenen nach dem Top-down Prinzip dargestellt.[33]

Kontextabgrenzung - Blackbox

In der ersten Abstraktionsebene - Level 0 sehen wir zunächst die Kontextsicht des Event Sourcing Services, hinzu den unterschiedlichen Diensten, die diesen verwenden. Dabei gibt es Dienste, die nur Events konsumieren, konsumieren und publizieren oder Dienste die nur publizieren. Abbildung 4.1 zeigt diese Dienste mit dem Event Sourcing Service als Gesamtsystem. Der Event Sourcing Service und die mit ihm interagierenden Dienste sind dabei als Blackbox dargestellt. Des Weiteren sind die Eingangs- und Ausgangsschnittstellen des Event Sourcing Services zu sehen.

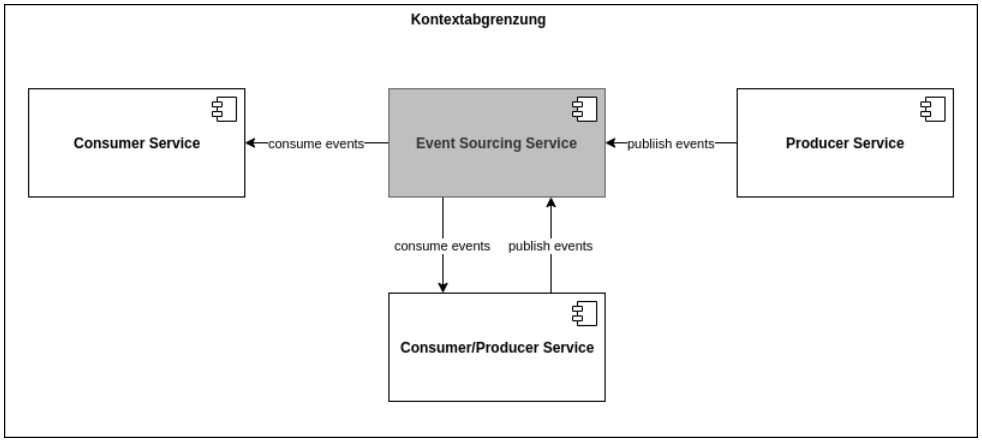

Abbildung 4.1: Event Sourcing Service - Kontextabgrenzung Level 0

Event Sourcing Service - Whitebox

In Abbildung 4.2 ist der Event Sourcing Service nun als Whitebox zu sehen. Dabei ist zu erkennen wie die einzelnen, zuvor erwähnten, Technologien eingesetzt werden. Das API Gateway dient hier als Zugriffsschnittstelle für die Dienste, die den Event Sourcing Service nutzen möchten. Die Endpunkte *publish* und *replay* sind auf die selbe Lambda Funktion *ESService* abgebildet und leiten die entgegengenommenen Anfragen an diese weiter. Wird der Endpunkt *publish* angesprochen, so führt die Funktion eine Schreib-Operation aus und speichert das publizierte Event in der DynamoDB *EventStore*. Nach dem Speichern eines neuen Events in der Datenbank wird automatisch ein Ereignis mit Hilfe des DynamoDB Streams an die Lambda Funktion *EventHandler* veröffentlicht. Die Funktion leitet dieses Event anschließend mittels des Kinesis Datenstroms *ProjectionStream* an alle konsumierenden Dienste weiter.

Bei einer Anfrage für das Starten eines Replay-Verfahrens, wertet die Lambda Funktion *ESService* diese Anfrage aus und startet anschließend den Automaten *Iterator* der mittels AWS Step Functions erstellt wird. Die als Zustand des Automaten definierte Lambda Funktion *Iterator* ließt dabei die dem Replay-Verfahren entsprechenden Events aus der Datenbank und publiziert diese mit Hilfe des Kinesis Datenstroms *ReplayStream*. Dieser stellt ebenfalls eine Ausgangsschnittstelle des Event Sourcing Service dar. Die Auftrennung in zwei Kinesis Datenströme wird dabei bewusst vorgenommen. Diese Entscheidung wird deshalb getroffen, um dem Service, der die Events verarbeitet die Möglichkeit zu bieten zwei parallel betriebene Datenbanken für dessen Projektionen zu führen. Dies ist

eine Lösung um während eines Replay Verfahrens neu aufgetretene Events erkennen zu können. Ein Umschalten der Datenbanken kann somit durchgeführt werden, sobald keine neuen Events mehr während des Replays in den Event Store abgelegt wurden.

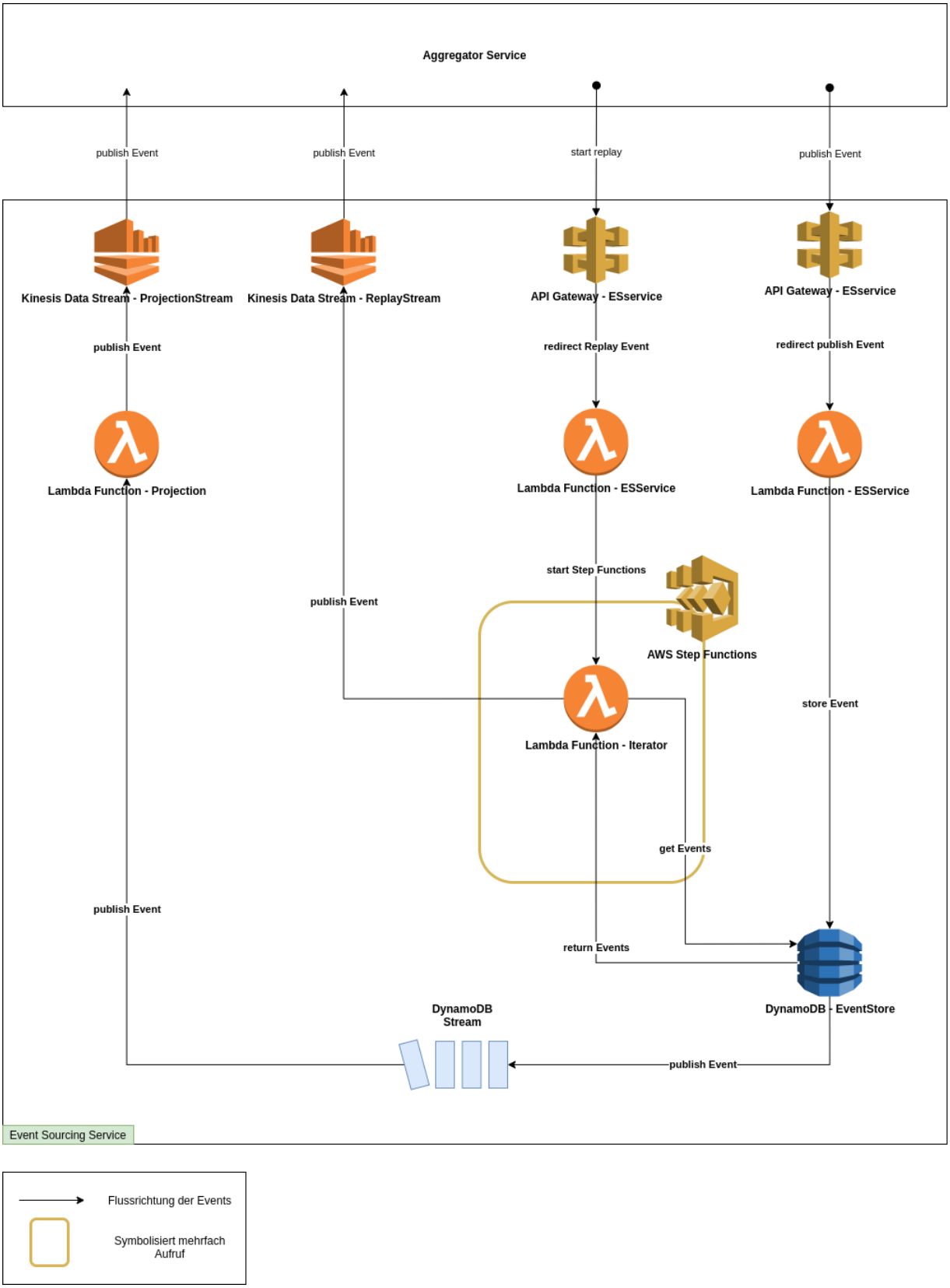

Abbildung 4.2: Event Sourcing Service - Kontextabgrenzung Level 1

Aggregator Service - Whitebox

Nachfolgend in Abbildung 4.3 ist ein prototypischer Service zu sehen, der sowohl von dem Event Souring Service Events konsumiert, als auch Events mittels des Services publiziert. Bei neu erhaltenen Events aus dem *ProjectionStream* des Event Sourcing Services verarbeitet die Lambda Funktion *OrderProjection* diese und speichert ein neues oder das aktualisierte Aggregat in der *Orders* Datenbank ab, oder löscht dieses. Die Lambda Funktion *OrderHandler* enthält in diesem Fall die Geschäftsregeln des Dienstes. Diese erhält Bestellungen, die durch einen externen Dienst per Event gesendet wurden. Auf die übermittelten Bestellungen werden anschließend bestimmte Geschäftsregeln angewendet. Die *OrderHandler* Funktion greift dabei nicht schreibend auf die *Orders* Datenbank zu, sondern liest lediglich die Einträge. Soll eine Bestellung aktualisiert, gelöscht oder neu angelegt werden, so erfolgt dies immer über den Event Sourcing Service per publiziertem Event. Anschließend wird dieses Event durch die *OrderProjection* Funktion verarbeitet und die entsprechende Transaktion mit der Datebank findet statt. Wie dabei zu erkennen ist, erfolgt hier eine Trennung zwischen dem Write- und Read-Model nach dem CQRS Entwurfsmusters. Das Write-Model ist dabei der Event, der gegenüber dem Event Sourcing Service publiziert und anschließend von der *OrderProjection* Funktion verarbeitet wird. Das Read-Model wiederum ist der in der Datenbank abgelegte Eintrag, der von der OrderHandler Funktion gelesen wird. Die Schreib-Modelle werden somit innerhalb des Event Sourcing Services persistiert. Die Lese-Modelle wiederum in dem Konsumierenden Dienst, in diesem Fall dem Aggregator Service. Durch diese Auftrennung in zwei Modelle und die Nutzung des Event Sourcing Services entsteht, die im Grundlagen Kapitel erwähnte Eventual Consistency, was Auswirkungen auf die *OrderHandler* Funktion haben kann. Wird Beispielsweise die selbe Bestellung zweimal hintereinander an die Funktion gesendet, so könnte es passieren, dass der selbe Event zum Erstellen einer Bestellung doppelt an den Event Sourcing Service publiziert wird. Dieser Fall würde dann auftreten, wenn das Read-Model, welches durch den ersten Event angelegt wird, noch nicht erstellt ist und somit bei der Verarbeitung des zweiten Events nicht aus der *Orders* Datenbank gelesen wird. Bei der Implementierung der *OrderProjection* muss somit darauf geachtete werden, dass die Operationen idempotent sind. Was dazu führt, dass das mehrfache Verarbeiten des Events keine Auswirkungen hat.

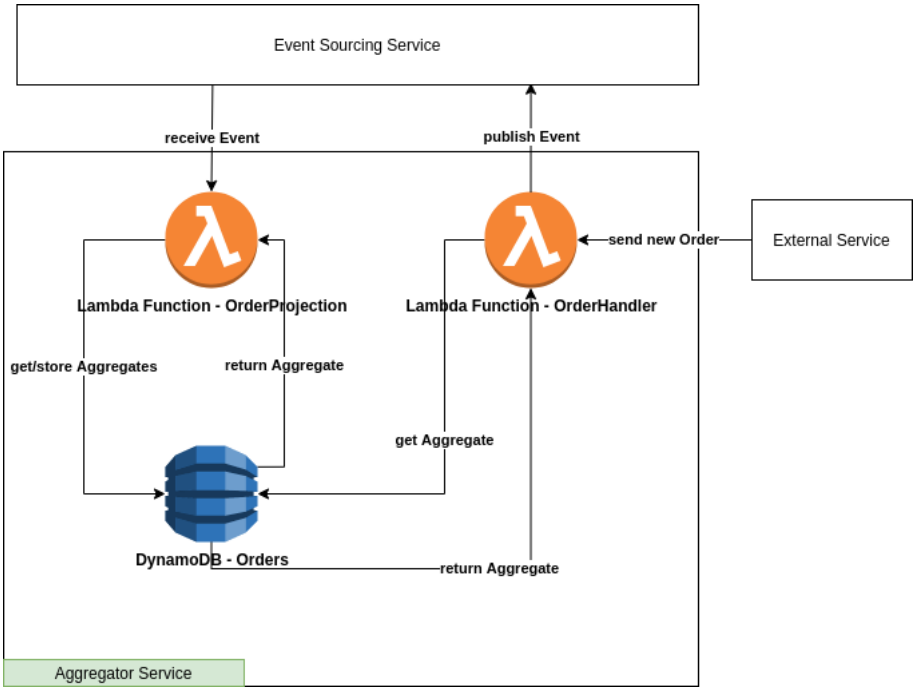

Abbildung 4.3: Aggregator Service - Kontextabgrenzung Level 1

4.2.2 Laufzeitsicht

Aus den einzelnen Interaktionsmöglichkeiten mit dem Event Souring Service ergeben sich die nachfolgenden Laufzeitsichten, die als Sequenzdiagramm dargestellt sind. Diese stellen eine weitere Sicht auf den Service dar und sind wichtig um die Funktionsweise des Dienstes aus der Whitebox Sicht zu verstehen.

Sequenzdiagramm Event Sourcing Service

In Abbildung 4.4 ist das Sequenzdiagramm des Event Sourcing Services zu sehen, im Fall eines neu publizierten Events. Dabei ist der, in Kapitel 4.1 bereits beschriebene Ablauf zu erkennen. Nachdem das Event über das API Gateway empfangen und mittels der *ES-Service* Lambda Funktion in der *EventStore* DynamoDB gespeichert wurde, wird dieses mit Hilfe des DynamoDB Streams an die *EventHandler* Lambda Funktion weitergeleitet. Diese publiziert zum Schluss das Event durch den Kinesis Datenstrom *ProjectionStream* an die Konsumenten.

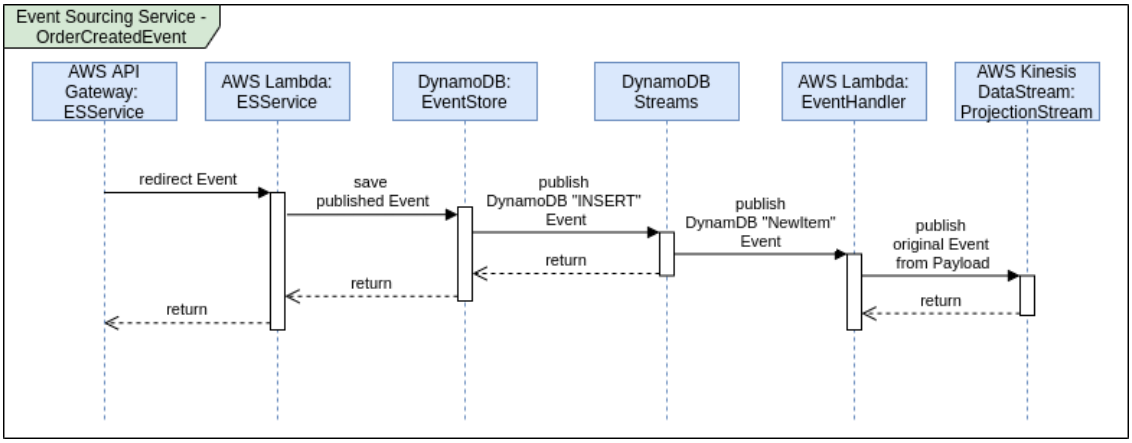

Abbildung 4.4: Sequenzdiagramm Event Sourcing Service: Publish-Endpunkt

Sequenzdiagramm Aggregator Service

In Abbildung 4.5 und 4.6 sind die Sequenzdiagramme für das Erstellen einer neuen Bestellung zu sehen. Der Aggregator Service bekommt durch einen externen Service eine Bestellung per Event geliefert. Da sich aktuell noch keine Bestellung mit der entsprechenden Bestellnummer in der Projektions Datenbank befindet, wird ein OrderCreatedEvent durch die *OrderHandler* Funktion erstellt und mittels des Event Sourcing Services publiziert. Der Aggregator Service erhält dieses Event anschließend über den *ProjectionStream*, der durch die *OrderHandler* Funktion abgearbeitet wird. Diese speichert zum Schluss die neue Bestellung in der *Orders* Datenbank, wie in dem Sequenzdiagramm in Abbildung 4.6 zu sehen ist.

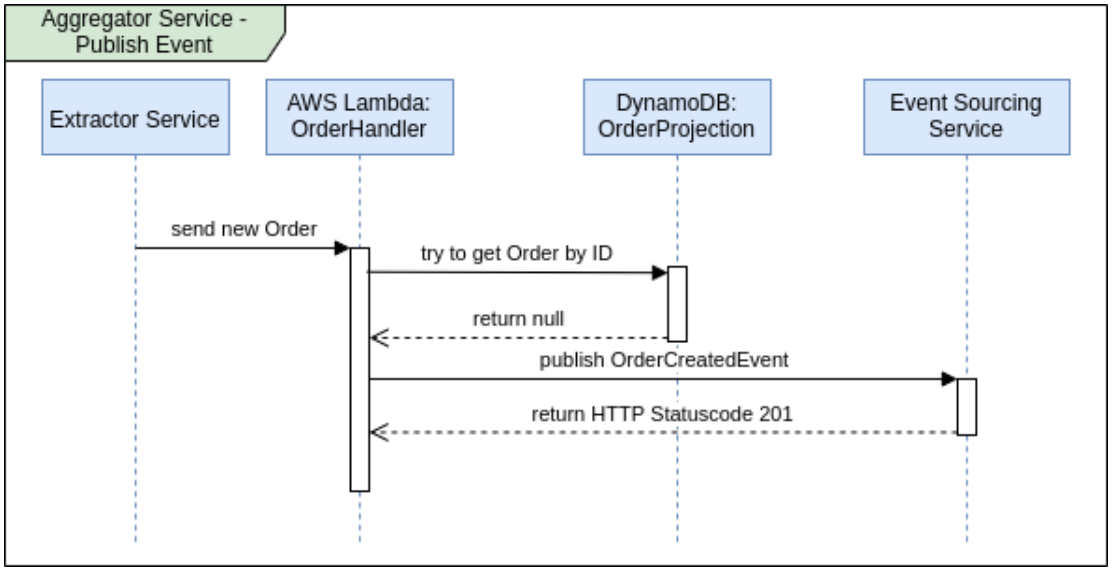

Abbildung 4.5: Sequenzdiagramm Aggregator Service: Publish OrderCreatedEvent

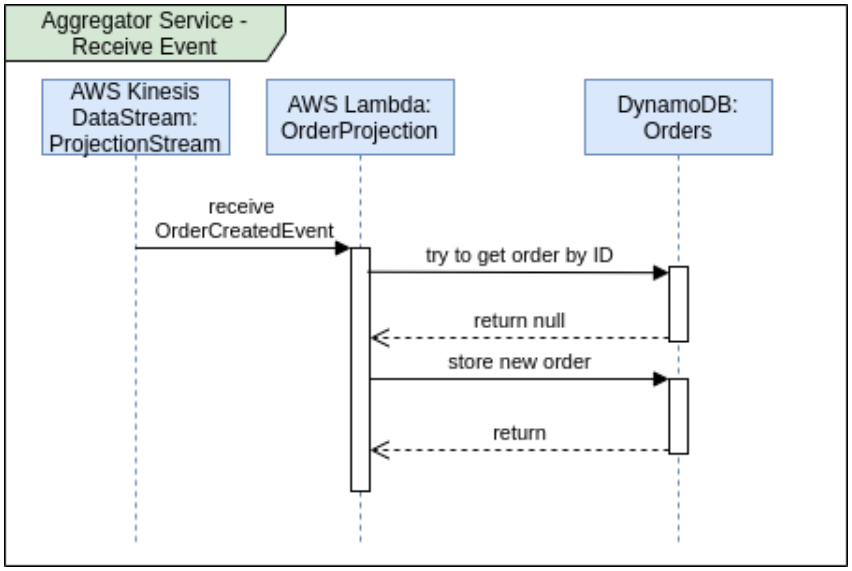

Abbildung 4.6: Sequenzdiagramm Aggregator Service: Receive OrderCreatedEvent

Sollte sich bereits eine Bestellung mit der selben Bestellnummer in der Projektions Datenbank befinden, wird durch die *OrderHandler* Lambda Funktion, auf Basis der definierten Geschäftsregeln, eine Aggregation der Bestellung vorgenommen und ein OrderUpdatedEvent publiziert. Anschließend wendet die *OrderProjection* Funktion, die durch den Event

veröffentlichten Änderungen auf das entsprechende Aggregat in der *Orders* Datenbank an und führt somit die eigentliche Update Operation durch. Die Sequenzdiagramme für diesen Fall sind in Abbildung 4.7 und 4.8 zu sehen.

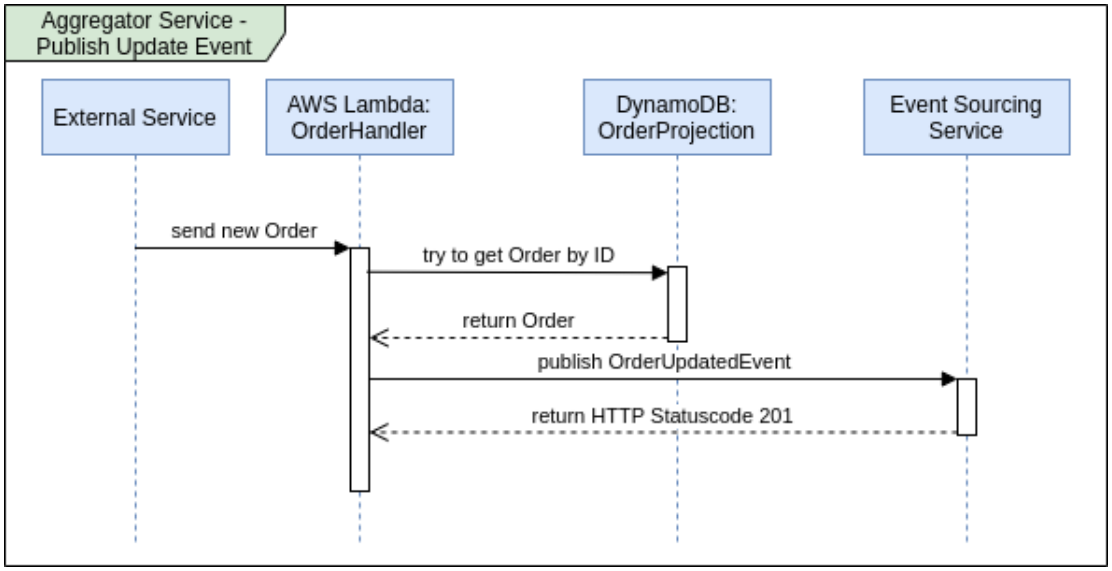

Abbildung 4.7: Sequenzdiagramm Aggregator Service: Publish OrderUpdatedEvent

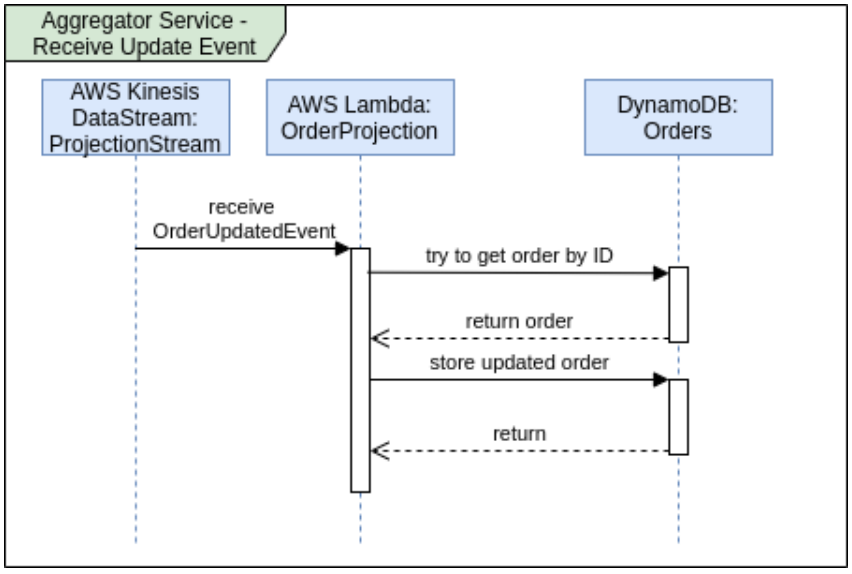

Abbildung 4.8: Sequenzdiagramm Aggregator Service: Receive OrderUpdatedEvent

Sequenzdiagramm Replay

In Abbildung 4.9 ist das Sequenzdiagramm des Replay Mechanismus auf Seiten des Event Sourcing Services zu sehen.

Durch das Senden eines POST Requests gegen den Replay-Endpunkt leitet das API Gateway die entsprechende Anfrage an die *ESService* Lambda Funktion weiter. Wie bereits beschrieben, startet diese den Automaten *Iterator* mit einem entsprechenden Replay Event. Dieser wird von der Lambda Funktion *Iterator* ausgewertet und abgearbeitet. Dabei holt die Funktion, je nach gewünschtem Verfahren, die Events aus der EventStore Datenbank und publiziert die Events schrittweise mit Hilfe des *ReplayStreams*.

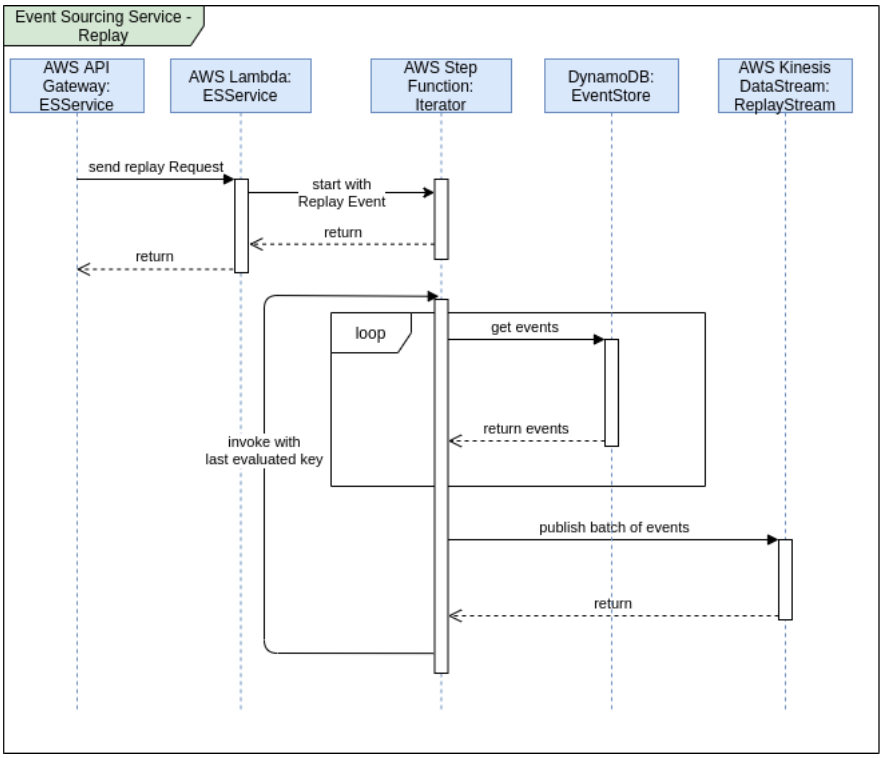

Abbildung 4.9: Sequenzdiagramm Replay-Mechanismus

5 Implementierung

Nach dem Entwurf des Event Sourcing Services wird in diesem Kapitel die Implementierung vorgestellt. Neben der Umsetzung des Event Sourcing Services wird ebenfalls auf die Implementierung des Aggregator Services, als Beispielanwendung eingegangen.

Die Entwicklung des Dienstes wurde dabei in der Programmiersprache Python 3 umgesetzt. Um die Infrastuktur für den Service bereitzustellen, wurde das Softwaretool Terraform verwendet. Terraform ermöglicht es, die Infrastruktur eines Systems per Code zu definieren und anschließend zu provisionieren. Die dabei verwendete Konfigurationssprache ist HCL (Hashicorp Configuration Language).[6] Ein Teil der Provisionierung wird jedoch an das Python Framework AWS Chalice ausgelagert. Bei AWS Chalice handelt es sich um ein Framework, welches zur Entwicklung von serverlosen Applikationen dient.[1]

5.1 Schnittstelle

Für das Bereitstellen der Schnittstelle des Event Sourcing Services wird das oben erwähnte AWS Chalice Framework verwendet. Bei den dadurch bereitgestellten AWS Diensten handelt es sich um das AWS API Gateway und eine AWS Lambda Funktion. Das AWS API Gateway bildet dabei die REST Schnittstelle des Services ab und leitet entgegengenommene Anfragen an die AWS Lambda Funktion weiter, wie bereits in Kapitel 4.1.2 erwähnt.

5.1.1 AWS API Gateway ESService

Die Schnittstellen Endpunkte des Dienstes werden durch das AWS Chalice Framework automatisch in dem AWS API Gateway angelegt. Das Framework erkennt bei dem Deployment der Applikation, die in der Lambda Funktion angegebenen Endpunkte und legt

diese in dem AWS API Gateway Service als Verknüpfung an. In Abbildung 5.1 ist ein Auszug aus dem Webinterface des Dienstes zu sehen. Darin sind in der linken Spalte die beiden Endpunkte des Event Sourcing Services zu erkennen. Der Endpunkt *publish*, der für das entgegennehmen von Events zuständig ist und der Endpunkt *replay*, durch welchen das wieder Einspielen von Events ausgelöst werden kann.

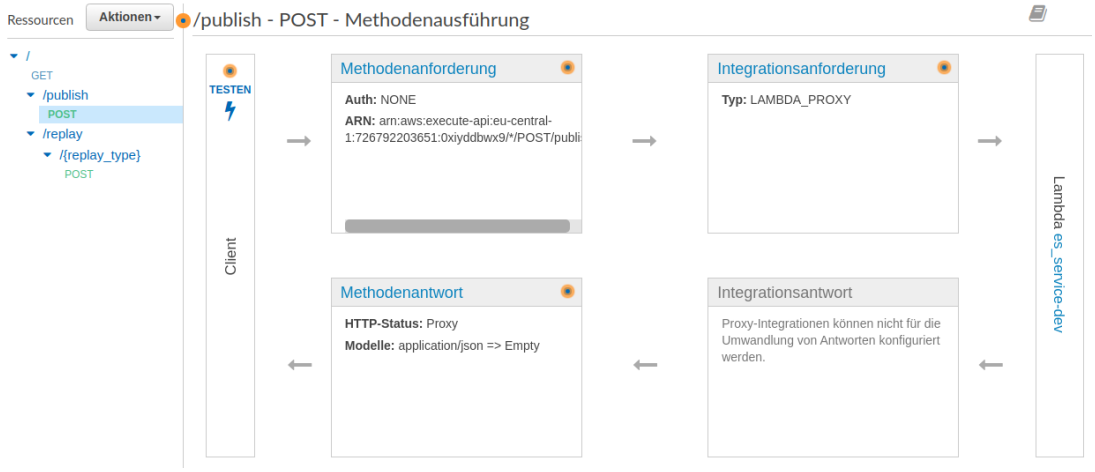

Abbildung 5.1: Event Sourcing Service - REST API Endpunkte in AWS

5.1.2 AWS Lambda - ESService

Wie in Abbildung 5.1 an der rechten Seite zu erkennen ist, wurde durch das Provisionieren mittels AWS Chalice für die Endpunkte *publish* und *replay* eine Lambda Funktion *es_service-dev* erstellt. Bei einer eingehenden HTTP Anfrage wird an diese ein Event gesendet, mit einem in der Anfrage enthaltenem HTTP Body. Damit für die in der AWS Lambda Funktion definierten Endpunkte eine entsprechende Verknüpfung mit dem AWS API Gateway angelegt wird, muss jede Funktion, die einen solchen Endpunkt darstellt den Decorator *app.route* nutzen, wie in Abbildung 5.2 zu sehen ist. Mit dem ersten Übergabeparameter kann dabei der Pfad des Endpunktes angegeben werden, sowie zusätzliche Pfadparameter. Der Übergabeparameter *methods* ermöglicht es, die erlaubten HTTP Methoden des Endpunkts anzugeben.

```
@app.route("/publish", methods=["POST"])
def publish():…


@app.route("/replay/{replay_type}", methods=["POST"])
def replay(replay_type):…
```

Abbildung 5.2: Event Sourcing Service - Endpunkte Lambda Funktion

Publish

Der Endpunkt *publish* verarbeitet an den Service gesendete Events. Mittels einer implementierten *EventStore Client* Klasse, welche die Schnittstelle zu der DynamoDB abbildet, speichert die Funktion die entgegengenommenen Events in der DynamoDB *event_store* ab. Bei einer erfolgreichen Speicherung des Events liefert der Endpunkt zum Schluss eine Antwort mit dem HTTP-Statuscode 201 (Created) an den aufrufenden Service zurück.

Replay

Der Endpunkt *replay* verarbeitet Anfragen zum wieder einspielen von Events. Um eines der implementierten Verfahren zu starten, muss der Typ des Verfahrens als Pfadparameter angegeben werden, wie in Abbildung 5.2 zu sehen ist. Dabei handelt es sich um die in den Anforderungen definierten Verfahren Gesamt-, Zeitstempel-basiert und Aggregat-ID gebunden.

Für das Zeitstempel-basierte und das Aggregat-ID basierte Verfahren muss zusätzlich noch das entsprechende Attribut im Request Body mitgegeben werden. Exemplarische Aufrufe mit Hilfe des Kommandozeilen Tools curl sind in Abbildung 5.3 zu sehen.

```
> curl -X PUT --header "Content-Type: application/json"
  https://rmws36te3f.execute-api.eu-central-1.amazonaws.com/api/replay/timestamp
  -d "{\"timestamp\":1576249816.5055032}"
Replay started successfull!

> curl -X PUT --header "Content-Type: application/json"
  https://rmws36te3f.execute-api.eu-central-1.amazonaws.com/api/replay/id
  -d "{\"aggregate_id\":\"e9d8f483-3631-4100-afe6-d3849c29a8d0\"}"
Replay started successfull!
```

Abbildung 5.3: Event Sourcing Service - Request Replay Verfahren

Die *Replay* Endpunkt-Funktion validiert dabei lediglich den übergebenen Pfad-Parameter und HTTP Body. Anschließend führt diese einen in AWS Step Functions definierten endlichen Automaten aus.

Das eigentliche wieder Einspielen der Events wird dabei durch die, in dem Automat definierte Lambda Funktion durchgeführt. Dies hat den Hintergrund, dass je nach Verfahren eine sehr lange Laufzeit benötigt wird, um alle Events aus dem EventStore zu lesen und anschließend zu publizieren. Dazu mehr in Kapitel 5.4.

Nach dem erfolgreichen Starten des Automaten gibt die Replay Funktion einen HTTP Status Code 200 (OK) zurück und vermittelt dem aufrufenden Dienst damit, dass das Replay Verfahren erfolgreich gestartet wurde.

5.2 EventStore Client

Bei dem *EventStore Client* handelt es sich um eine Hilfsklasse, die Schnittstellen anbietet um mit der *EventStore* Datenbank, sowie den Kinesis Datenströmen interagieren zu können. Sie wird dabei von den AWS Lambda Funktionen *ESService*, *EventHandler* und *Iterator* verwendet. Die Klasse selbst nutzt für die Interaktion mit DynamoDB das Python Framework *pynamodb*. Dabei handelt es sich um ein Framework, das eine ORM (Object-Relational Mapping) artige Schnittelle zu der AWS DynamoDB für Python anbietet [13]. Die angebotenen Funktionen der *Event Store Client* Klasse sind dabei die nachfolgenden:

store_event(event)

Funktion zum Speichern eines Events in dem *EventStore*. Dabei wird ein Objekt vom Typ *EventModel* erstellt und anschließend gespeichert. Das Datenobjekt *EventModel* und damit verbundene Datenschema wird unter Kapitel 5.3.2 vorgestellt.

publish_event(event, stream, sequence_number)

Funktion zum publizieren einzelner Events über den angegebenen Kinesis Datenstrom.

replay_events(events, stream):

Publiziert die übergebenen Events mit Hilfe des Datenstroms *ReplayStream*. Dabei wird ein AWS Kinesis Client verwendet, der mit der Initialisierung des Event Store Clients ebenfalls instanziiert wurde.

get_event(aggregate_id, timestamp):

Liefert das Event mit der angegebenen Aggregat-ID und dem Zeitstempel zurück.

get_all_events(last_evaluated_key, new_start_date=None):

Liefert alle Events aus dem EventStore zurück, sowie den Schlüssel des zuletzt gelesenen Events.

get_events_by_aggregate_id(aggregate_id, last_evaluated_key):

Dient zum auslesen aller Events aus der Datenbank für eine bestimmte Aggregat-ID. Diese muss beim Aufruf der Funktion angegeben werden.

get_events_by_timestamp(timestamp, last_evaluated_key):

Liefert alle Events aus der *EventStore* Datenbank zurück, die nach einem bestimmten Zeitstempel gespeichert wurden.

5.3 Event Verarbeitung

Nachfolgend wird die Implementierung der Event Verarbeitung innerhalb des Event Sourcing Services beschrieben. Durch die Veröffentlichung von Events, über den Endpunkt *publish* werden die nachfolgend beschriebenen Dienste ausgeführt.

5.3.1 AWS Lambda - ESService

Wie bereits in Kapitel 5.1 beschrieben, nimmt die AWS Lambda Funktion *ESService* die Anfragen für das Publizieren von Events und das Starten eines Replay-Verfahrens entgegen. Bei der Implementierung des *publish* Endpunkts wird dabei das empfangene Event sofort mittels der Funktion *store_event* des *EventStore Clients* gespeichert. Da die Events den Zustand des Systems abbilden, ist es besonders wichtig diese sofort zu speichern. Das Publizieren mit Hilfe des AWS Kinesis Datenstroms, direkt nach der Entgegennahme der Events wurde deshalb bewusst nicht innerhalb der *ESService* Funktion implementiert, um die Anzahl der potenzielle Fehlerquellen gering zu halten. Fehler die beispielsweise während der Datenübertragung zwischen den Diensten auftreten könnten, werden somit vermieden. Ein weiterer Punkt der gegen das Veröffentlichen zum selben Zeitpunkt wie das Speichern des Events spricht, ist, dass es zu Inkonsistenzen zwischen den Diensten führen kann. Angenommen ein Event kann nicht in die *EventStore* Datenbank gespeichert werden, wird aber erfolgreich an die konsumierenden Dienste übermittelt, so würde sich der Zustand zwischen dem *EventStore* und den Projektionen unterscheiden. Der *EventStore* wäre dadurch nicht mehr als *Single Source of Truth* zu gebrauchen.

5.3.2 AWS DynamoDB - EventStore

Wie bereits erwähnt, wird als Datenbank für die zu speichernden Events eine AWS DynamoDB verwendet. In dem Quellcodeauszug 5.1 ist die in HCL definierte Datenbank zu sehen.

```
######## Databases ########
resource "aws_dynamodb_table" "event_store" {
  name         = "event_store"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "date"
```

```
  range_key      = "timestamp"

  stream_enabled   = true
  stream_view_type = "NEW_IMAGE"

  attribute {
    name = "date"
    type = "S"
  }

  attribute {
    name = "timestamp"
    type = "N"
  }

  attribute {
    name = "aggregate_id"
    type = "S"
  }

  global_secondary_index {
    name            = "aggregate_idx"
    hash_key        = "aggregate_id"
    range_key       = "timestamp"
    projection_type = "ALL"
  }
}
```

Quellcode 5.1: Terraform Konfiguration - DynamoDB EventStore

DynamoDB bietet zwei verschiedene Betriebsarten für die Kapazität der Lese- und Schreibzugriffe an, On-Demand und Provisioned.

Bei dem On-Demand Modus fallen Kosten für jeden Request an. Die Skalierung findet dabei automatisch durch AWS verwaltet statt. Erhöht sich die Anzahl der Anfragen, so werden zusätzliche Instanzen von AWS gestartet, um die erhöhte Last verarbeiten

zu können.[17] Es müssen somit auch keine Lese- und Schreibkapazitäten angegeben werden.

Der von Terraform verwendete Standard ist jedoch der Provisioned Modus. Durch die Verwendung des Provisioned Modus der DynamoDB muss die Anzahl der benötigten Lese- und Schreibkapazität Einheiten definiert werden.[31] Diese werden in der Konfigurationsdatei mit *read_ capacity* und *write_ capacity* angegeben. Die Einheit Read Capacity Units (RCU) und Write Capacity Units (WCU) geben dabei Auskunft über den benötigten Lese- und Schreibzugriff Durchsatz. Je nach Anfrage steht eine RCU Einheit für einen streng konsistenten Lesezugriff oder zwei schwach konsistente (eventual consistent) Lesezugriffe pro Sekunde, für ein Objekt von bis zu 4 KB.[31] Die Art des Lesezugriffs kann dabei bei der Anfrage an die DynamoDB angegeben werden. Eine WCU Einheit steht für einen Schreibzugriff pro Sekunde für ein Objekt von einer Größe von bis zu 1 KB. Für das transaktionale Schreiben von Objekten in die Datenbank werden 2 WCU Einheiten für ein Objekt mit einer Größe von bis zu maximal 1 KB benötigt.

Für eine Implementierung der Datenbank lässt sich somit die nachfolgende Rechnung aufstellen, um die Anzahl der benötigten RCU und WCU Einheiten zu errechnen. Die in Tabelle 5.1 zu sehenden Werte lassen sich dabei wie folgt berechnen.

Das Objekt besteht aus vier String Attributen und einem Number Attribut. Laut der Dokumentation des AWS DynamoDB Services gelten für diese Attribute die folgenden Rechenregeln:

String Attribut = Länge des Attribut Namens + die Anzahl der UTF-8 codierten Bytes.

Number Attribute = Länge des Attribut Namens + 1 Byte für alle zwei Zahlen + 1 Byte.[27]

Somit ergeben sich die unten aufgeführten Werte.

Attribut	**Name (Byte)**	**Wert (Byte)**	**Summe**
aggregate_id	12	36	48
date	4	10	14
event_type	10	17	27
payload	7	1403	1410
timestamp	9	10	19
Gesamtgröße:			**1518**

Tabelle 5.1: Attributgrößen in Byte

Für strenge Konsistenz bei einem Lesezugriff, kostet das Lesen eines Objektes 1 RCU Einheit (1.518 Byte / 1.024 = 1,48 Kilobyte). Das Lesen mit schwacher Konsistenz würde somit 0,5 RCU Einheiten kosten. Das Schreiben in die Datenbank würde 2 WCU Einheiten kosten und das transaktionale Schreiben 4 WCU Einheiten.

Da zum aktuellen Zeitpunkt noch nicht bekannt ist, wie viele Events im Schnitt pro Sekunde publiziert werden, wird in der ersten Implementierung jedoch die Option On-Demand gewählt. Dieser kann durch das Schlüsselwort *billing_mode* mit dem Wert *PAY_PER_REQUEST* ausgewählt werden. Ein weiterer Grund für die Entscheidung für den On-Demand Modus ist, dass vermutlich eine höhere Last zu bestimmten Zeitpunkten eintritt und keine kontinuierlich Auslastung. Eine Änderung des Lese- und Schreibkapazitäten Modus ist jedoch einmal nach jeweils 24 Stunden möglich und könnte somit auch nachträglich nochmals konfiguriert werden.[31]

Datenmodell

Bei der DynamoDB handelt es sich wie im Grundlagenkapitel bereits erwähnt, um eine NoSQL Datenbank. Diese ist somit bis auf den Primärschlüssel schemalos. Der Primärschlüssel kann bei DynamoDB aus einem Partitionsschlüssel (hash key) oder aus einer Kombination von Partitionsschlüssel und Sortierschlüssel (sort key) bestehen [28]. Die Attribute, die diesen Schlüssel abbilden, müssen in der Konfiguration der DynamoDB angegeben werden, wie in dem Codeauszug in 5.1 zu sehen ist. Bei der Verwendung des Schemas der Datenbank, mit Hilfe des Frameworks *pynamodb*, müssen diese Attribute ebenfalls angegeben werden, um Objekte des Typs in der DynamoDB speichern zu können. Das durch den Event Sourcing Service verwendete Schema ist in dem Quellcodeauszug 5.2 zu sehen. Dabei sind zum einen die definierten Schlüsselattribute zu sehen, sowie die zusätzlich definierten Attribute. Der Primärschlüssel besteht aus dem Attribut *date* als Partitionsschlüssel (hash key) und dem Attribut *timestamp* als Sortierschlüssel (sort key). Zusätzlich wurde ein weiterer Index (Global Secondary Index) - *aggregate_idx* - angelegt. Dieser muss ebenfalls in der Terraform Konfigurationsdatei definiert werden, sowie die, durch den Index verwendeten Attribute. DynamoDB bietet die Möglichkeit bis zu 20 sekundäre Indizes zu erstellen.[28] Der zweite Index wurde ebenfalls in dem durch *pynamodb* definierten Datenmodell angegeben, siehe Quellcode 5.3. Der Grund für die Definition eines weiteren Indexes hat mit dem Replay Verfahren zu tun und wird in Kapitel 5.4 erläutert.

```
class EventModel(Model):
    """
    DynamoDB event model.
    """

    class Meta:
        table_name = "event_store"
        region = "eu-central-1"

    date = UnicodeAttribute(hash_key=True)
    timestamp = NumberAttribute(range_key=True)
    aggregate_id = UnicodeAttribute(null=False)
    event_type = UnicodeAttribute(null=False)
    payload = JSONAttribute(null=False)

    # Secondary Index
    aggregate_idx = EventByIdIndex()
```

Quellcode 5.2: Event Modell

```
class EventByIdIndex(GlobalSecondaryIndex):
    """Index to query events by timestamp"""

    class Meta:
        read_capacity_units = 2
        write_capacity_units = 1
        index_name = "aggregate_idx"
        projection = AllProjection()

    aggregate_id = UnicodeAttribute(hash_key=True)
    timestamp = NumberAttribute(range_key=True)
```

Quellcode 5.3: Sekundärer Index des Event Modells

5.3.3 AWS DynamoDB Streams

Nach dem erfolgreichen Speichern eines Events in der DynamoDB *EventStore* wird mittels DynamoDB Streams ein *NewItem* Event an den konsumierenden Service gesendet. Um den DynamoDB Stream verwenden zu können, muss dieser mittels des Schlüsselworts *stream_enabled* in der Terraform Konfiguration aktiviert werden. Des Weiteren muss angegeben werden welche Informationen die Events enthalten sollen. Durch das Schlüsselwort *stream_view_type* kann dies bestimmt werden. Im ersten Prototypen des Event Sourcing Services wurde dabei der Typ *"NEW_IMAGE"* gewählt. Dadurch wird das komplette Objekt durch den Stream übermittelt. Eine weitere Option wäre die *"KEYS_ONLY"* Option, wodurch lediglich die ID des neu angelegten Events übermittelt werden würde.[19] Dadurch wäre jedoch eine separate Abfrage des Events aus der Datenbank nötig. Zu einem späteren Zeitpunkt könnte dies eventuell genutzt werden, wenn in der Datenbank abgelegte Events dazu führen würden, dass die maximale Payload Größe von 6 MB überschritten wird. Diese Größe bezieht sich dabei auf die maximale Größe einer Request und Response Anfrage an eine AWS Lambda Funktion. [23]

In den Grundzügen unterscheidet sich der Dynamodb Streams Service kaum von dem des Kinesis DataStreams Dienstes. Eine Problematik, die sich während der Implementierung dessen jedoch auftat, ist die, dass dieser nur sehr wenig Konfigurationsmöglichkeiten bietet. So gibt es zum Beispiel keine Möglichkeit die Anzahl der verwendeten Shards zu konfigurieren. Dies kann dazu führen, dass die Events über mehrere Shards hinweg verarbeitet und weitergeleitet werden, was zur Folge hat, dass die Events in einer unterschiedlichen Reihenfolge bei der konsumierenden Lambda Funktion eintreffen können. Dieses Problem wurde im ersten Protoypen des Dienstes dadurch gelöst, dass das aktuelle Datum als Paritionsschlüssel genutzt wurde. Dadurch werden alle Events mit dem selben Partitionsschlüssel von dem selben Dynamodb Streams Shard verarbeitet und die Ordnung dieser ist garantiert.

5.3.4 AWS Lambda - EventHandler

Die durch den DynamoDB Stream veröffentlichten Events werden von der AWS Lambda Funktion *EventHandler* konsumiert. Damit die Lambda Funktion die Events erhält sobald diese in dem Stream veröffentlicht werden, muss DynamoDB Streams als Event Quelle der Lambda Funktion hinzugefügt werden. In Terraform kann diese Verbindung mittels der Ressource *"lambda_event_source_mapping"* erstellt werden.

```
resource "aws_lambda_event_source_mapping" "dynamodb_stream" {
  event_source_arn  = "${aws_dynamodb_table.event_store.
      stream_arn}"
  function_name     = "${aws_lambda_function.event_handler.arn}"
  starting_position = "LATEST"
}
```

Quellcode 5.4: AWS Lambda und DynamoDB Stream Verbindung

Die Aufgaben der EventHandler Funktion sind dabei die folgenden. Sie konsumiert die Events aus dem DynamoDB Stream, entpack die im Event enthaltenen Objekte, die in die DynamoDB abgelegt wurden und publiziert diese anschließend mittels eines AWS Kinesis Datenstroms.

Für das Publizieren der Events mit Hilfe des Datenstroms *event_stream* wurde das AWS Software Development Kit für Python, *Boto3*, verwendet. Im ersten Prototypen bietet die Schnittstelle des Event Souring Services nur die Möglichkeit des Veröffentlichens einzelner Events. Nachfolgend ist die Funktion des *EventStore Clients* zum veröffentlichen der Events zu sehen.

```
def publish_event(self, event, stream, sequence_number):
    """
    Publish single event to a given kinesis stream.
    """
    try:
        if sequence_number:
            response = kinesis_client.put_record(
                StreamName=stream,
                Data=json.dumps(event),
                PartitionKey="test",
                SequenceNumberForOrdering=sequence_number,
            )
        else:
            response = kinesis_client.put_record(
                StreamName=stream, Data=json.dumps(event),
                    PartitionKey="test"
            )
```

```
    except Exception as e:
        raise e

    if not response["ResponseMetadata"]["HTTPStatusCode"] ==
       200:
        raise PublishingFailedException(f"Error publishing
           events to {stream} stream!!")

    return response
```

Quellcode 5.5: Boto3 Kinesis Client - Publizieren eines Events

Grund für die Implementierung der *EventHandler* Funktion ist der, dass DynamoDB maximal zwei Konsumenten des Streams zulässt.[30] Durch die Verwendung dieser Funktion als Zwischeninstanz und des Kinesis Datenstroms, können somit die Einschränkungen, die durch DynamoDB Streams vorgegeben sind, umgangen werden und es können mehrere Konsumenten des Datenstroms definiert werden. Diese Implementierung entspricht einem von AWS vorgeschlagenen Design Pattern für DynamoDB Streams, dem Lambda fan-out Pattern. [8]

5.3.5 AWS Kinesis - ProjectionStream

Wie bereits erwähnt, werden die Events an die konsumierenden Dienste mittels eines Kinesis Datenstroms übermittelt. Für die Provisionierung des Datenstroms wird dabei nachfolgender HCL Code verwendet.

```
resource "aws_kinesis_stream" "projection_stream" {
  name        = "projection_stream"
  shard_count = 1
}
```

Quellcode 5.6: AWS Kinesis Datenstrom in HCL definiert

Dabei wurde der Datenstrom *ProjectionStream* erstellt, welcher eine Ausgangsschnittstelle für den Event Sourcing Service darstellt. Die Dienste die Events empfangen möchten, wie zum Beispiel der Aggregator Service, können diesen Stream abonnieren, um ihre Projektionsdatenbank zu befüllen.

Enhanced fan-out

Um den konsumierenden Diensten, die Zustellung der Events zu garantieren wird zusätzlich die *enhanced fan-out* Option aktiviert. Dafür muss ein AWS Kinesis Consumer angelegt werden, um die Option nutzen zu können. Diese ermöglicht es jedem Konsumenten, einen Datenkonsum von 2 MB pro Sekunde zu garantieren.[15] Ohne die Verwendung des enhanced fan-outs kann es vorkommen, dass bei der Verarbeitung vieler kurz hintereinander auftretender Events, nicht alle Events bei den konsumierenden Diensten ankommen. Dies hätte zur Folge, dass sich die Systeme in unterschiedlichen Zuständen befinden.

Garantierte Ordnung

Ein weiterer sehr wichtiger Punkt, der bei der Verwendung von AWS Kinesis Data Streams zu beachten ist, ist die Ordnung der Events in der, diese bei den Konsumenten ankommen. Amazon Kinesis Stream garantiert die Ordnung der publizierten Events pro Shard. In diesem Prototypen wird nur ein Shard für den *ProjectionStream* verwendet. Die Ordnung der beim Konsumenten eintreffenden Events ist somit garantiert und entspricht der Reihenfolge in der die Events durch die Lambda Funktion in den Datenstrom abgelegt wurden. Jeder Shard eines Kinesis Datenstroms hat dabei die folgenden Limitierungen:

- Maximal 1.000 Einträge pro Sekunde oder
- Maximalgröße von 1 MiB pro Sekunde. (1 MiB = 1,048576 MB)

Sollte sich die Anzahl der Shards per Stream erhöhen, so muss eine Lösung gefunden werden, um die Ordnung der Events weiterhin zu garantieren. Dies könnte beispielsweise dadurch realisiert werden, dass die Events vor dem Veröffentlichen mit einer Sequenznummer angereichert werden. Der Konsument müsste dann die Sequenznummern der Events auswerten und diese anhand dieser in die korrekte Reihenfolge bringen.

5.4 Replay Verfahren

Wie bereits in Kapitel 4 erwähnt, werden die Replay-Verfahren mit Hilfe eines endlichen Automaten umgesetzt, der durch den AWS Step Functions Service erstellt und ausgeführt werden kann.

5.4.1 Iterator Automat

Eine Schwierigkeit bezüglich des Lesens aller Events aus der DynamoDB Datenbank ist, dass die Lambda Funktion, welche dies übernimmt, nur eine maximale Laufzeit von 15 Minuten hat.[23] Befinden sich jedoch Millionen von Events in der Datenbank, so würde das Lesen aller Events bei dem entsprechenden Replay-Verfahren diese Laufzeit überschreiten. Deshalb wurde ein endlicher Automat implementiert, um dieses Problem zu lösen. In Abbildung 5.4 ist ein Graph dieses Automaten zu sehen. Wie der Name des Automaten bereits andeutet, handelt es sich dabei um einen Iterator, welcher über die Events in der DynamoDB iteriert und diese mittels eines Kinesis Datenstroms erneut publiziert. Bei der Umsetzung wurde sich dabei an dem, im AWS Step Functions Developer Guide, aufgeführten Beispiel - Iterating a Loop using Lambda - orientiert.[25]

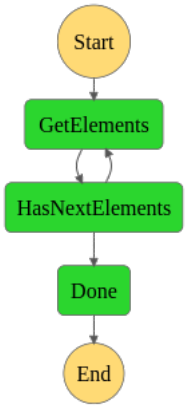

Abbildung 5.4: AWS Step Function - Iterator

Definition Automat

Nachfolgend in dem Quellcodeauszug 5.7 ist der Quellcode des Iterator Automaten aufgeführt. Wie darin zu erkennen ist wird zunächst ein Start Zustand mit Hilfe des Schlüsselworts *StartAt* definiert. Innerhalb des Schlüssels *States* werden anschließend die einzelnen Zustände angegeben. In diesem Fall die Zustände *GetElements*, *HasNextElements* und *Done*.

Bei dem Zustand *GetElements* handelt es sich um den Startzustand, welcher durch die Ausführung des Automaten ausgelöst wird. Der Typ des Zustandes ist dabei *Task*, was

bedeutet, es kann die Logik, die in diesem Zustand ausgeführt wird an eine Ressource ausgelagert werden. In diesem Fall handelt es sich dabei um die Lambda Funktion *IteratorFunction*. Die an die Funktion als Event übergebenen Parameter sind mit dem Schlüssel *Parameters* vorgegeben. Die Funktion, die den Automaten starten möchte, muss somit die entsprechenden Parameter in dem HTTP Body mitliefern. Ansonsten würde der Automat in einen Fehler laufen und somit die Funktion nicht ausgeführt werden. Mit Hilfe des Schlüsselwortes *ResultPath* und dem Wert *$* wird der Rückgabewert der Lambda Funktion an den nächsten Zustand des Automaten weitergeleitet, welcher mit dem Schlüssel *Next* angegeben wird.

Im nächsten Zustand des Automaten *HasNextElements* wird mit Hilfe des Typs *Choice* eine Validierung des Rückgabewertes der Lambda Funktion vorgenommen. Unter *Choices* ist dabei die entsprechende Überprüfung definiert. In diesem Fall wird der Schlüssel *last_evaluated_key* auf Gleichheit mit einer leeren Zeichenfolge geprüft. Trifft dies nicht zu, so wird der Zustand *GetElements* mit den an *HasNextElements* übergebenen Rückgabewerten erneut aufgerufen. Dies bedeutet, solange die Lambda Funktion *IteratorFunction* einen Schlüssel zum Fortsetzen des Replay Verfahrens zurück gibt, so lange wird diese wieder aufgerufen und setzt die Iteration fort. Liefert die Funktion keinen weiteren Schlüssel an den *HasNextElements* Zustand, so wird der Zustand *Done* aufgerufen, welcher den Endzustand des Automaten abbildet.

```
{
  "StartAt": "GetElements",
  "States": {
    "GetElements": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-central-1:726792203651:
          function:iterator_function",
      "Parameters": {
        "ReplayType.$": "$.replay_type",
        "LastEvaluatedKey.$": "$.last_evaluated_key",
        "Timestamp.$": "$.timestamp",
        "ID.$": "$.id"
      },
      "ResultPath": "$",
      "Next": "HasNextElements"
    },
```

```
    "HasNextElements": {
      "Type": "Choice",
      "Choices": [
        {
          "Not": {
            "Variable": "$.last_evaluated_key",
            "StringEquals": ""
          },
          "Next": "GetElements"
        }
      ],
      "Default": "Done"
    },
    "Done": {
      "Type": "Pass",
      "End": true
    }
  }
}
```

Quellcode 5.7: AWS Step Function - Iterator

5.4.2 Implementierung Replay Verfahren

Die Funktion *IteratorFunction* wird von der Lambda Funktion *ESService* aufgerufen, wenn eine entsprechende Anfrage für das Starten eines Replay Verfahrens erfolgreich validiert wurde. Die Funktion prüft dabei die übergebenen Parameter. Anschließend liest diese die Events aus der DynamoDB Datenbank mit Hilfe der *EventStore Client* Klasse aus. Nachfolgend publiziert diese, die durch den *EventStore Client* zurückgegebenen Events über den Kinesis Datenstrom *ReplayStream*. Dieser wurde wie der *ProjectionStream* konfiguriert um, die zuvor benannten Probleme (Garantierte Ordnung, Enhanced fan-out) zu verhindern. Zum Schluss erstellt die Funktion noch eine, auf den Rückgabewerten basierende Antwort. Dadurch wird ein weiterer Aufruf der Funktion eingeleitet oder das Replay-Verfahren abgeschlossen.

In dem Quellcodeauszug 5.8 ist der Event zu sehen, welcher beim Start des Replay Verfahrens an die Iterator Funktion übergeben wird. Der Rückgabewert der Funktion ist in der Antwort der Iterator Funktion in 5.9 zu sehen. Wie dabei zu erkennen ist, hat der Rückgabewert die selben Schlüsselwörter wie der Übergabewert. Der Wert *LastEvaluatedKey* ist dabei jedoch gesetzt, was zu einem erneuten Aufruf der Iterator Funktion führt. Anhand des Schlüsselwortes *ReplayType* kann die Funktion unterscheiden, welche Art von Replay-Verfahren gestartet werden soll. Nachfolgend ist die Implementierung der einzelnen Replay Verfahren beschrieben.

```
{
    "ReplayType": "all",
    "ID": "",
    "LastEvaluatedKey": "",
    "Timestamp": "",
    "Date": "2020-01-19"
}
```

Quellcode 5.8: Übergabewert zum Starten der Iterator Funktion

```
{
    "ReplayType": "all",
    "ID": "",
    "LastEvaluatedKey": {
        "date": {"S": "2020-01-19"},
        "aggregate_id": {"S": "15"},
        "timetamp": {"N": "1580478288.7424119"}
    },
    "Timestamp": "",
    "Date": "2020-01-19"
}
```

Quellcode 5.9: Rückgabewert der Iterator Funktion

Gesamt-Replay

Für das Replay Verfahren bei dem alle Events aus der Datenbank wieder eingespielt werden sollen, wird die Funktion *get_all_events* des EventStore Clients genutzt. Diese

liest bis zu 400 Events aus der Datenbank und gibt diese an die aufrufende *Iterator-Function* zurück. Der Funktion *get_all_events* kann ein optionaler Übergabeparameter *last_evaluated_key* übergeben werden. Dieser Schlüssel gibt den Primärschlüssel des zuletzt gelesenen Objekts an. Zu Beginn prüft die Funktion ob dieser Parameter übergeben wurde. Ist dies der Fall, so ist klar, dass es sich um eine weitere Iteration innerhalb eines Replay Prozesses handelt. Wird dieser Schlüssel übergeben, so kann das Startdatum für die abzufragenden Events aus diesem extrahiert werden. Wird kein Schlüssel übergeben, so handelt es sich um den ersten Durchlauf der Iteration und es wird ein Standardwert als Startdatum gesetzt. Bei diesem Wert handelt es sich, im ersten Prototypen des Dienstes, um das Datum, an dem die Datenbank erstellt wurde. Der Datumswert muss bei der Abfrage des Indexes mit angegeben werden, da dieser als Hash-Key zählt. Zur Abfrage der Events an den einzelnen Datumswerten werden diese zunächst berechnet. Mit Hilfe des zuvor initialisierten Startdatums und dem aktuellen Datum können die Tage, die es abzufragen gilt, berechnet werden. Diese Aufgabe wird von der Hilfsfunktion *_get_dates_between* übernommen, wobei es sich bei dieser Funktion, um eine Generator Funktion handelt. Diese liefert die Datumswerte in einzelnen Schritten an die aufrufende Funktion zurück, wodurch über diese iteriert werden kann. Dabei fragt die Funktion in den einzelnen Schritten alle Events zu den angegebenen Datumswerten aus der Datenbank ab.

```
def get_all_events(self, last_evaluated_key=None):
    """
    Get all events.
    """
    # Calculate dates between start_date and current_date.
    if last_evaluated_key:
        date_to_query = last_evaluated_key["date"]["S"]
    else:
        date_to_query = START_DATE

    event_list = []

    batch_size = ITEM_BATCH_SIZE

    for date in self._get_dates_between(date_to_query=
        date_to_query):
```

```
        # Query events by calculated date.
        key, events = self._query_events_by_date(
            date=date, last_evaluated_key=last_evaluated_key,
                batch_size=batch_size,
        )
        event_list.extend(events)

        # Return if the maximum amount of records per batch has
            been reached.
        if len(event_list) == ITEM_BATCH_SIZE:
            return key, event_list
        else:
            # Calculate left over batch size
            batch_size = ITEM_BATCH_SIZE - len(event_list)

    return key, event_list
```

Quellcode 5.10: EventStore Client Funktion - get_all_events

Die Abfrage geschieht über die Hilfsfunktion *_query_events_by_date*, welche die eigentliche Abfrage gegenüber der Datenbank ausführt und die Events sowie den Schlüssel des zuletzt abgefragten Events zurückliefert. Das Maximum an abzufragenden Events wird ebenfalls mit angegeben. Dieses wird definiert, da der API Endpunkt des AWS Kinesis Data Stream Dienstes, eine maximale Anzahl von 500 Events pro Stapel (Batch) erlaubt und eine Größe von maximal 5 MB für die gesamte Abfrage.[22] Nach dem Lesen der Events aus der Datenbank wird deshalb anschließend die Anzahl der bereits erhaltenen Events überprüft. Bei erreichtem Limit wird die Liste der Events, mit dem Schlüssel des zuletzt gelesenen Events, an die Iterator Funktion zurück geliefert.

Die eigentliche Abfrage mittels des *pynamodb* Frameworks ist in dem Codeauszug in 5.11 zu sehen. Durch das Setzen des Sortierschlüssels *timestamp*, welcher einen nummerischen Wert enthält, ist eine Sortierung der zurückgegebenen Events nicht mehr nötig. Die Events werden bereits in aufsteigender Reihenfolge, basierend auf dem Wert des Attributes *timestamp*, zurückgegeben.

```
def _query_events_by_date(
    self,
    date,
```

```
    last_evaluated_key=None,
    range_key_condition=None,
    batch_size=ITEM_BATCH_SIZE,
):
    """
    Helper Funktion to query timestamp index by date.
    """
    result = EventModel.query(
        hash_key=date.strftime(DATE_TIME_FORMAT),
        last_evaluated_key=last_evaluated_key,
        range_key_condition=range_key_condition,
        limit=batch_size,
    )

    items = [i for i in result]
    print(f"Retrieved events from event store: {items}")

    # Check if we need to process more items.
    if result.last_evaluated_key:
        return json.dumps(result.last_evaluated_key), items
    else:
        return "", items
```

Quellcode 5.11: Pynamodb Index Abfrage

Nachdem die Iterator Funktion die Liste der wieder einzuspielenden Events erhalten hat, werden diese mittels der *replay_ events* Funktion publiziert. Im Vergleich zu dem publizieren neuer Events über den *publish* Endpunkt wird hier der *PutRecords* Endpunkt, der Kinesis Data Streams API genutzt. Dieser hat die oben benannten Limitierungen, ermöglicht es jedoch im Vergleich zu dem *PutRecord* Endpunkt einen Stapel von Events zu publizieren, anstatt nur einzelne Events. Dadurch kann das Replay Verfahren beschleunigt werden und die Anzahl der benötigten Requests minimiert.

Replay-Verfahren - Zeitstempel basiert

Die Implementierung des Zeitstempel basierten Replay-Verfahrens ist dem des Gesamt Replay-Verfahrens sehr ähnlich. Die Iterator Funktion ruft dabei die Schnittstellen Funktion *get_events_by_timestamp* des *EventStore Clients* auf. Diese ist nachfolgend zu sehen.

```
def get_events_by_timestamp(self, timestamp, last_evaluated_key=
    None):
    """
    Query event_store table for events that are greater than or
       equal to timestamp.
    """
    start_date = None
    if not last_evaluated_key:
        # Extract date from given timestamp
        start_date = str(datetime.fromtimestamp(float(timestamp)
            ).date())
    else:
        start_date = last_evaluated_key["date"]["S"]

    batch_size = ITEM_BATCH_SIZE
    events = []

    for date in self._get_dates_between(date_to_query=start_date
        ):
        # Define range key condition to enable server side
            filtering of events.
        range_key_condition = EventModel.timestamp >= float(
            timestamp)

        key, result = self._query_events_by_date(
            date=date,
            last_evaluated_key=last_evaluated_key,
            range_key_condition=range_key_condition,
            batch_size=batch_size,
        )
```

```
        events.extend(result)

        if len(events) == ITEM_BATCH_SIZE:
            return key, events
        else:
            batch_size = ITEM_BATCH_SIZE - len(events)
    return key, events
```

Quellcode 5.12: EventStore Client - Funktion zur Zeitstempel basierten Abfrage

Im Vergleich zu der Funktion, um alle Events aus der Datenbank abzufragen, verwendet diese, bei der ersten Abfrage das Datum des übergebenen Zeitstempels. Dadurch wird verhindert, dass alle Events die älter als der Zeitstempel sind, gelesen werden. Da durch den Paritionsschlüssel *date* des Indexes jedoch alle Events eines Tages aus der Datenbank gelesen werden, müssen diese noch auf ihren Zeitstempel überprüft werden. Um dies zu ermöglichen wird eine *KeyConditionExpression* Bedingung definiert. Die DynamoDB API bietet diese Option an, um eine server-seitige Filterung der Events zu ermöglichen. Dadurch werden anschließend nur die Events der Partition zurückgegeben, die diese Bedingung erfüllen.[18]

Die Bedingung für die Zeitstempel basierte Abfrage ist dabei folgendermaßen definiert. Das Attribut **timestamp** des Events muss größer oder gleich dem übergebenen Zeitstempel sein. Die Hilfsfunktion **_query_by_date** bietet hierfür den optionalen Übergabeparameter **range_key_condition** an. Somit liefert die DynamoDB API lediglich die Events zurück, die nach einem bestimmten Zeitstempel Wert in der Datenbank gespeichert wurden.

Replay-Verfahren - Aggregat-ID basiert

Zuletzt wurde das Replay Verfahren für das wieder einspielen von Events eines bestimmten Aggregates implementiert. Dabei werden alle Events mit der übergebenen Aggregat ID aus der EventStore Datenbank ausgelesen. Die Abfrage basiert dabei auf dem sekundären Index der Datenbank *aggregate_idx*, der zusätzlich angelegt wurde und ist in dem Quellcodeauszug 5.13 zu sehen. Auch auf diesem Index ist eine aufsteigende Sortierung der Events nicht mehr nötig, da als Sortierschlüssel ebenfalls das Attribut *timestamp* definiert ist.

```
def get_events_by_aggregate_id(self, aggregate_id,
    last_evaluated_key=None):
    """
    Query event_store table for events that have a specifc
        aggregate_id.
    """
    items = []

    result = EventModel.aggregate_idx.query(
        hash_key=aggregate_id,
        last_evaluated_key=last_evaluated_key,
        limit=ITEM_BATCH_SIZE,
    )
    events = [i for i in result]

    if result.last_evaluated_key:
        return json.dumps(result.last_evaluated_key), events
    return "", events
```

Quellcode 5.13: EventStore Client - Funktion zur ID basierten Abfrage

5.5 Aggregator Service

Um einen ersten Dienst zu entwickeln, der es ermöglicht den Event Sourcing Service in einem beispielhaften Szenario zu testen, wurde der Aggregtor Service implementiert. Nachfolgend ist die Implementierung der einzelnen Komponenten dieses Dienstes beschrieben. Die Provisionierung findet dabei, wie auch bei dem Event Sourcing Service, über Terraform statt.

5.5.1 AWS Lambda Funktion - OrderHandler

Bei der *OrderHandler* Funktion handelt es sich um eine Lambda Funktion, die einzelne Bestellungen erhält und diese verarbeitet. Je nach Anwendungsfall werden diese neu in der *Orders* Datenbank erstellt, aktualisiert oder gelöscht.

Zu Beginn findet durch die *OrderHandler* Funktion eine Überprüfung der übergebenen Bestellung statt. Dabei wird mit Hilfe der übermittelten Aggregat ID in der Datenbank nach der entsprechenden Bestellung gesucht. Sollte eine Aktualisierung einer Bestellung stattfinden, so muss sich eine Bestellung mit der entsprechenden ID in der Datenbank befinden. Ebenso wenn eine Bestellung gelöscht werden soll.

Für das Abfragen der Datenbank *Orders* wird dabei wie bereits bei dem Event Sourcing Service das pynamodb Framework verwendet.

Publizieren des Events

Nach der erfolgreichen Validierung der Bestellung und der durchzuführenden Operation wird das Event erstellt und der *publish* Endpunkt des Event Sourcing Dienstes aufgerufen.

Der Aggregator Service enthält für jeden Event Typen eine dem entsprechende Klasse. Die Event Typen umfassen dabei die folgenden:

- **OrderCreatedEvent**
- **OrderUpdatedEvent**
- **OrderDeletedEvent**

Für eine vereinfachte Serialisierung und Deserialisierung der Events wurde zusätzlich ein Event Schema definiert. Dieses wird durch das Python Framework *marshmallow* umgesetzt und dient somit als Datentransferobjekt (DTO) der Events. *Marshmallow* bietet dabei die Möglichkeit komplexe Datentypen und Klassen in die, in Python enthalten Datentypen zu konvertieren.[9] In dem Quellcodeauszug 5.14 ist das Event Schema zu sehen.

```
class EventSchema(Schema):
    date = fields.String(allow_none=False)
    aggregate_id = fields.String(allow_none=False)
    event_type = fields.String(allow_none=False)
    payload = fields.Dict()
    timestamp = fields.Float(allow_none=True)
```

Quellcode 5.14: Event Schema für die Datenübertragung zwischen den Diensten

Nach der Serialisierung des Events mit Hilfe des oben gezeigten Schemas wird der Endpunkt zum publizieren eines Events aufgerufen. Dies geschieht durch die, in Python integrierte *request* Library, welche das Senden von HTTP Anfragen ermöglicht.[14]

5.5.2 AWS Lambda Funktion - OrderProjection

Wurde das Event erfolgreich mittels des EventSourcing Services publiziert, wird dieses von der *OrderProjection* Funktion durch den AWS Kinesis Stream entgegengenommen. Danach findet zunächst eine Deserialisierung statt, um anhand des Typs des übermittelten Events die entsprechende Funktion und die damit verbundene Operation anzuwenden. Für jeden Event Typ wurde dabei eine eigene Hilfsklasse erstellt, um diesen zu verarbeiten.

In dem Quellcodeauszug 5.15 ist die Funktion für das Erstellen einer neuen Bestellung zu sehen. Wie bereits erwähnt, wird dabei wieder auf das ORM Framework *pynamodb* zurück gegriffen. Die weiteren Funktionen für das Aktualisieren und Löschen von Bestellungen sind simultan aufgebaut und verwenden ebenfalls die dafür vorgesehenen API Endpunkte des *pynamodb* Frameworks.

```
def _create_order(event):
    """
    Create new order according to OrderCreatedEvent.
    """
    try:
        model = _get_order_by_id(order_id=event.aggregate_id)
        if not model:
            vehicle_order = Order(**event.payload["
                attribute_values"])
            vehicle_order.id = event.aggregate_id
            vehicle_order.save()
            print(f"Order created: {vehicle_order}")
        else:
            print(f"Order already exists")
    except Exception as e:
        raise CreateNewOrderException(event)
```

Quellcode 5.15: Erstellen einer neuen Bestellung mit Hilfe des pynamodb Frameworks

Die Hilfsfunktion zum Lesen aus der *Orders* Datenbank wird sowohl von der *OrderProjection*, als auch von der *OrderHandler* Funktion verwendet. Diese führt die Anfrage für den Zugriff auf die Dynamodb aus und gibt die Bestellung mit der übergebenen ID zurück. Dabei wird ein streng konsistenter Lesezugriff durchgeführt. Dies ist mit Hilfe des Übergabeparameters *consistent_read* möglich.

```
def _get_order_by_id(order_id):
    """
    Get order by id from order table.
    """
    return Order.get(hash_key=order_id, consistent_read=True)
```

Quellcode 5.16: Streng konsistenter Lesezugriff auf die DynamoDB

Die Verwendung des streng konsistenten Lesezugriffs auf die Datenbank hat dabei jedoch einige Nachteile [17]

- Hohe Latenz im Vergleich zu schwach konsistenten Lesezugriffen
- Nicht auf sekundäre Indizes anwendbar
- Erhöhter Verbrauch von Lesekapazitätseinheiten (RCU)

Diese Nachteile müssen jedoch vorerst in Kauf genommen werden, um zum einen ein mehrfaches Publizieren der Events durch die *OrderHandler* Funktion zu vermeiden. Wie auch ein mehrfaches Abspeichern des selben Aggregates in der *OderProjection* Funktion zu verhindern.

5.5.3 AWS DynamoDB - Orders

Die *Orders* Datenbank, in der die Bestellungen abgespeichert werden, ist in nachfolgendem Terraform Code Ausschnitt zu sehen. Dabei wurde die Aggregat ID als Primärschlüssel gewählt, da lediglich über diese auf die Datenbank zugegriffen wird.

```
### DynamoDB ###
resource "aws_dynamodb_table" "orders" {
  name           = "orders"
  read_capacity  = 5
  write_capacity = 5
```

```
  hash_key       = "id"

  attribute {
    name = "id"
    type = "S"
  }
```

Quellcode 5.17: Terraform Definition der Orders Datenbank

6 Qualitätssicherung

In diesem Kapitel wird auf die Maßnahmen zur Sicherung der Qualität des Event Sourcing Services eingegangen. Dabei handelt es sich im ersten Prototypen hauptsächlich um das Testen mit Hilfe von Unit- und Integrationstests. Des Weiteren wird die kontinuierliche Integration des Dienstes mittels eines CI&CD Tools umgesetzt.

6.1 Tests

Für das Testen des Dienstes wird das Python Framework *pytest* verwendet. Die einzelnen Tests sind dabei in drei Teststufen unterteilt, beginnend mit den Unit Tests. Die nächste Stufe umfasst die Integrationstests zusammengefasster Komponenten und somit bestimmter Funktionen des Systems. Bei der dritten Teststufe handelt es sich um die Systemtests, welche die komplette Funktionsweise des Systems testen und zwar aus der Sicht der Dienste, die den Event Sourcing Service nutzen. Da es aktuell noch keine Dienste gibt, die den Service nutzen, wird dies durch die Evaluation in Kapitel 7 abgedeckt.

6.1.1 Unit Tests

Bei den Unit Tests handelt es sich um die Tests, mit denen einzelne Funktionen und Klassen getestet werden. [39] Die Tests werden dabei meist von dem Entwickler selbst geschrieben. Wann diese geschrieben werden, ist dabei nicht relevant. Ein Ansatz für das Schreiben der Tests ist der des Test Driven Developments (TDD). Bei dieser Vorgehensweise werden die Tests für eine Komponente bereits vor der Entwicklung dieser geschrieben.[39] Dadurch müssen sich bereits vor der Implementierung der eigentlichen Komponente Gedanken darüber gemacht werden, wie zum Beispiel die Schnittstelle, die Rückgabewerte und somit das gewünschte Verhalten der Komponente aussehen soll.

Beispiel Event Sourcing Service

Bei der Entwicklung des Event Sourcing Services wurden für alle Funktionen und Klassen Tests geschrieben. Eine Einheit, die dabei besonders beachtet wurde, ist die Klasse *EventStore Client*. Diese ist ausschlaggebend für die Interaktion des Dienstes mit der Datenbank, wie auch für das Publizieren der Events und stellt somit eine besondere Schwachstelle dar, sollten sich bei der Weiterentwicklung dieser Fehler einschleichen.

In dem nachfolgenden Codeausschnitt ist einer der Tests für die Funktion *get_all_events* der Klasse des *EventStore Clients* zu sehen.

```
def test_get_all_events_success(self, mocker, event_mock):
    # Mock _get_dates_between function
    dates_to_query_mock = mocker.patch(
        "chalicelib.eventstore.client.EventStore.
            _get_dates_between",
        return_value=["2020-02-19"],
    )

    # Create dummy events and event list.
    event1 = EventModel(**event_mock)
    event1.aggregate_id = "first-event-id"
    event2 = EventModel(**event_mock)
    event2.aggregate_id = "second-event-id"
    event3 = EventModel(**event_mock)
    event3.aggregate_id = "thrid-event-id"

    event_list = [event1, event2, event3]

    query_events_by_date_mock = mocker.patch(
        "chalicelib.eventstore.client.EventStore.
            _query_events_by_date",
        return_value=("third-event-id", event_list),
    )

    es_client = EventStore()
```

```
    result_key, result_list = es_client.get_all_events()

    assert len(result_list) == 3
    assert result_key == "third-event-id"
    assert result_list[0] == event1
    assert result_list[1].aggregate_id == event2.aggregate_id
    assert query_events_by_date_mock.called_once()
    assert dates_to_query_mock.called_once()
```

Quellcode 6.1: Test der get_all_events Funktion des Event Store Clients

Dabei wurde der Fall für das erfolgreiche Erhalten aller Events mit der Funktion *test_get_all_events_success* getestet. Wie in dem Codeausschnitt zu erkennen ist, wurde für die Hilfsfunktion *_get_dates_between* zunächst ein Mock Objekt erstellt. Ein Mock-Objekt ist eine Art Platzhalter, welcher für das isolierte Testen einer bestimmten Einheit eingesetzt werden kann.[43] Dabei wird der Aufruf dieser Einheit - in diesem Fall der Aufruf der Funktion *_get_dates_between* - abgefangen und der zuvor definierte Rückgabewert zurückgeliefert. Eine weitere Funktion für die ein Mock-Objekt erstellt werden musste, ist die *_query_events_by_date* Funktion. Diese führt den eigentlichen Aufruf der DynamoDB aus und wird in einem separaten Test getestet. Um eine valide Rückgabe dieser Funktion zu erhalten, wurde dabei zunächst eine Liste mit drei Events angelegt, die als Dummys dienen. Anschließend findet der eigentliche Aufruf, der zu testenden Funktion statt. Die Rückgabewerte der Funktion werden anschließend auf die erwarteten Werte überprüft. Neben der Überprüfung der Rückgabewerte findet hier auch eine Prüfung der aufgerufenen Hilfsfunktionen statt. In diesem Test dürfen diese nur einmalig aufgerufen werden. Für das Testen der Funktion *get_all_events* sind jedoch noch weitere Tests nötig, um alle Handlungsstränge dieser abzudecken. Diese sind Zusammengefasst in Abbildung 6.1 zu sehen.

```
def test_get_all_events_success(self, mocker, event_mock):⋯

def test_get_all_events_multiple_dates_success(self, mocker, event_mock):⋯

def test_get_all_events_batch_size_exceeded(self, mocker, event_mock):⋯

def test_get_all_events_last_evaluated_key_success(self, mocker, event_mock):⋯

def test_get_all_events_empty_result(self, mocker, event_mock):⋯
```

Abbildung 6.1: Übersicht aller Tests des EventStore Clients

Testabdeckung

Um schnell und einfach erkennen zu können, ob die erstellten Unit Tests den gesamten Programmcode abdecken, wurde in diesem Projekt eine Erweiterung des *pytest* Frameworks *pytest-cov* verwendet. Mit Hilfe dieser kann die prozentuale Anzahl der durch Tests abgedeckten Codezeilen ausgegeben werden. Die Erweiterung bietet zudem eine grafische Übersicht durch die schnell, nicht vollständig getestete Funktionen ersichtlich sind. Die Übersicht der Testabdeckung für die implementierte Bibliothek ist in Abbidlung 6.2 zu erkennen.

Coverage report: 100%

Module ↓	statements	missing	excluded	coverage
chalicelib/eventstore/__init__.py	0	0	0	100%
chalicelib/eventstore/client.py	102	0	0	100%
chalicelib/exceptions/exceptions.py	14	0	0	100%
chalicelib/models/__init__.py	0	0	0	100%
chalicelib/models/event.py	35	0	0	100%
chalicelib/models/order.py	69	0	0	100%
chalicelib/schema/__init__.py	0	0	0	100%
chalicelib/schema/event_schema.py	40	0	0	100%
Total	**260**	**0**	**0**	**100%**

coverage.py v5.0.3, created at 2020-02-15 13:31

Abbildung 6.2: Testabdeckung der Event Sourcing Service Bibliothek

6.1.2 Integrationstests

Nach dem Testen der einzelnen Einheiten werden im nächsten Schritt diese in Zusammenarbeit miteinander getestet. Diese Art von Tests werden Integrationstests genannt. Dabei wird davon ausgegangen, dass die einzelnen Komponenten, die Bestandteil dieser Tests sind, bereits durch Unit Tests isoliert voneinander getestet wurden. [40]

In dem Event Sourcing Service wurden diese Tests auf Ebene der einzelnen Funktionen durchgeführt. So wurden Tests für alle AWS Lambda Funktionen geschrieben und überprüft, ob die Interaktion dieser mit dem *EventStore Client* korrekt sind. Wie bereits bei den Unit Tests müssen dabei die Aufrufe gegenüber der Datenbank simuliert werden.

Für den Integrationstest der *Iterator* Funktion bietet es sich dabei an den Test zu parametrisieren. Das Test-Framework *pytest* ermöglicht dies mit Hilfe des Decorators *@parametrize*. Dabei werden die Übergabe- und Rückgabewerte definiert. Eine Parametrisierung des Tests ermöglicht es bestimmte Tests mehrfach mit verschiedenen Übergabewerten auszuführen. In dem Quellcodeauszug 6.2 ist einer der parametrisierten Werte zu sehen.

```
class TestIteratorFunction:
    @pytest.mark.parametrize(
        "replay_event, expectation, events",
        [
            pytest.param(
                {
                    "ID": "",
                    "ReplayType": "all",
                    "LastEvaluatedKey": "",
                    "Timestamp": "",
                },
                {
                    "response": {
                        "id": "",
                        "replay_type": "all",
                        "last_evaluated_key": "",
                        "timestamp": "",
                    },
                    "kinesis_call_count": 1,
                    "dynamodb_call_count": 1,
                },
                pytest.lazy_fixture("multiple_events_mock"),
                id="replay-all-events-success",
            ),
```

Quellcode 6.2: Parametrisierung des Integrationstests der Iterator Funktion

Für den Test der *Iterator* Funktion wurden dabei die möglichen Replay-Verfahren als Parameter übergeben. Der erste Parameter *replay_event* beschreibt dabei den Event, welcher an die Lambda Funktion übergeben wird. Der zweite Parameter *expectation* ent-

hält die erwarteten Rückgabewerte der Funktion und weitere Kennzahlen die während der Ausführung des Tests erwartet werden. Diese umfassen in diesem Fall die Anzahl der Aufrufe für das Publizieren von Events, mittels des Kinesis Datenstroms und die Anzahl der Aufrufe der DynamoDB Datenbank, mittels des *pynamodb* Frameworks.

6.2 Continuous Integration und Continuous Delivery

Eine weitere Praxis, die zur Sicherung der Qualität des Dienstes beitragen kann, ist die der kontinuierlichen Integration und der kontinuierlichen Auslieferung (CI/CD).

Unter *Continuous Integration* versteht sich die Praxis, dass neu entwickelter oder überarbeiteter Code so schnell wie möglich in den bestehenden Code integriert wird.[4] Vorteile, die sich dadurch ergeben, sind zum einen das automatisierte Testen von Änderungen. Dies umfasst dabei sowohl die Unit Tests, als auch die Integrationstests, was dazu beitragen kann, Fehler in der Integration oder in den einzelnen Komponenten schnell zu entdecken. Für die Umsetzung können dabei Integrationstools, wie beispielsweise *CircleCI*, *GoCD* oder *Jenkins* zur Hilfe genommen werden. Diese ermöglichen es bei jeder Code Änderung, die in ein Versionsverwaltungstool, wie beispielsweise *git* gespeichert werden, automatisiert zu Testen.

Für den Event Sourcing Service wurde dabei im ersten Prototypen das Tool *CircleCI* verwendet. Dabei handelt es sich um eine cloud-basierte Lösung, welche eine einfache Integration in das Projekt ermöglicht, da keine zusätzlichen Serverinstanzen erstellt werden müssen und somit schnell mit der Umsetzung begonnen werden kann.

Neben der kontinuierlichen Integration kann mit Hilfe des Tools *CircleCI* auch das Prinzip von *Continuous Delivery* umgesetzt werden. Bei der Verwendung dieser Praxis wird noch ein Schritt weiter gegangen und der entwickelte Code sollte zu jeder beliebigen Zeit ausgeliefert werden können. Dies setzt Continuous Integration als Basis voraus.[3]

Im ersten Prototypen des Event Sourcing Services wurde dabei zunächst die kontinuierliche Integration angewendet. In dem Nachfolgenden Codeausschnitt ist der in einer .yaml Datei definierte Integrationsfluss zu sehen. Dabei werden zunächst die benötigten Abhängigkeiten für den Dienst installiert, bevor anschließend die Tests ausgeführt werden.

```
version: 2.1

executors:
  runtime:
    docker:
      - image: circleci/python:3.7

workflows:
  version: 1

  master:
    jobs:
      - install
      - test:
          requires:
            - install

jobs:
  install:
    executor: runtime
    steps:
      - checkout
      - run:
          name: Install the python dependencys
          command: |
            python -mvenv venv && source venv/bin/activate
            pip install --progress-bar=off -U pip
            pip install -r service/requirements.txt
            cd service
            pip install .
      - persist_to_workspace:
          root: ~/project
          paths:
            - venv
  test:
```

```
    executor: runtime
    steps:
      - checkout
      - attach_workspace:
          at: ~/project
      - run:
          name: Run tests for Event Sourcing Service
          working_directory: service
          command: |
            source ~/project/venv/bin/activate
            pytest tests/
```

Quellcode 6.3: Konfiguration für kontinuierliche Integration

Für die weitere Entwicklung des Event Sourcing Services wird die Umsetzung des *Continuous Delivery* Prinzips angestrebt.

6.3 Weitere qualitätssichernde Maßnahmen

Neben den bereits beschriebenen Maßnahmen zur Sicherung der Qualität des Dienstes sollte zudem auch über die folgenden Maßnahmen nachgedacht werden. So wäre zum Beispiel, die Implementierung eines Monitoring Systems hilfreich. Dies hätte den Vorteil, Fehler die während des Betriebs des Dienstes auftreten, möglichst schnell zu entdecken. Der von AWS bereitgestellte Dienst *AWS CloudWatch* [16] bietet, die dafür benötigten Funktionalitäten um die, vom Event Sourcing Service genutzten Dienste zu überwachen. Eine weitere Maßnahme, die zur Qualität des entwickelten Programmcodes beitragen würde, wäre die Verwendung eines Code-Analystools, wie beispielsweise *SonarQube*. Durch dieses wird eine Analyse des Codes hinsichtlich verschiedener Merkmale, wie zum Beispiel die Komplexität, die Code-Formatierung oder potentielle Fehler durchgeführt und aufmerksam gemacht.

7 Evaluation

In diesem Kapitel wird der erste Prototyp des Event Sourcing Services evaluiert. Dabei wird überprüft, ob der Dienst die in Kapitel 3 definierten Anforderungen erfüllt. Des Weiteren werden die einzelnen Bestandteile des Dienstes getestet und somit überprüft, ob diese das gewünschte Verhalten abbilden.

7.1 Publizieren von Events

Die erste Anforderung, die der Dienst erfüllen sollte, ist, dass die Events erfolgreich mit Hilfe des Event Sourcing Services publiziert werden können, diese abgespeichert werden und anschließend von einem Konsumenten in der Reihenfolge, in der diese veröffentlicht wurden empfangen werden.

Um dies zu testen wurde ein Python Script entwickelt, welches eine Reihe von Events erstellt und diese anschließend in der erstellten Reihenfolge publiziert. Das publizieren wurde mit Hilfe des Scripts in drei Durchläufen getestet. Wobei im ersten Durchlauf 100, im zweiten 1.000 und im dritten 10.000 Events publiziert wurden. Um eine Validierung der korrekten Reihenfolge zu erleichtern, wurde dabei eine aufsteigende Zahlenfolge als Aggregat-IDs verwendet.

In allen drei Fällen wurde das Ergebnis auf folgende Punkte überprüft:

- Wurden die Events in der korrekten Reihenfolge in der Datenbank abgelegt?
- Konnten alle Events in der korrekten Reihenfolge aus dem Datenstrom gelesen werden?

Um festzustellen ob die Events in der korrekten Reihenfolge in die Datenbank geschrieben wurden, wurden mit Hilfe des Scripts zur Evaluation, alle Events aus der Datenbank gelesen und die daraus resultierende Liste anschließend auf Korrektheit geprüft.

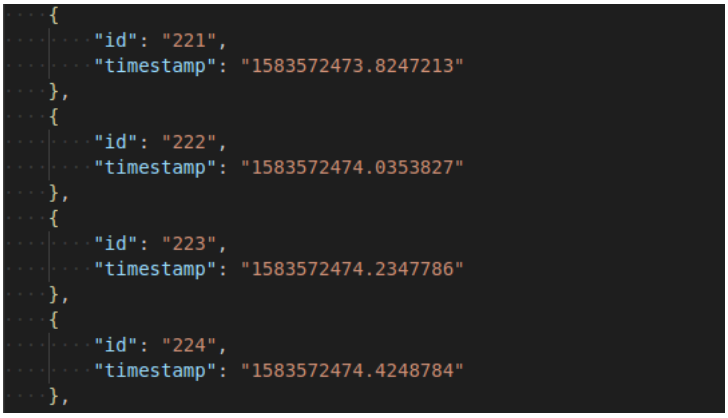

Abbildung 7.1: Ergebnisliste der konsumierten Events reduziert auf die Attribute ID und timestamp.

In nachfolgendem Codeausschnitt ist der Algorithmus zum vergleichen der Events zu sehen. Dabei wird jeweils das aktuelle Event mit dem vorherigen verglichen. Hierfür wird das Attribut Zeitstempel (*timestamp*) des Event Objektes herangezogen. Der Zeitstempel des zu prüfenden Events darf dabei nie kleiner als der des vorherigen Events sein. Durch das Setzen der Aggregat-ID in einer aufsteigenden Zahlen-Reihenfolge beim publizieren der Events, sollte somit diese um jeweils eins erhöht sein.

```
def compare(event_list):
    """
    Helper function to compare a list of items.
    """
    print("Comparing result...")

    # Error list
    error_list = []

    # Set the first value from result list to prev
    prev = event_list[0]

    # Iterate over result list and compare timestamp and
    # aggregate id.
    for i in tqdm(event_list[1:]):
        if int(i["aggregate_id"]) < int(prev["aggregate_id"]):
```

```
            error_list.append(f"ID Error: {i} > {prev}")
        elif i["timestamp"] < prev["timestamp"]:
            error_list.append(f"Timestamp Error: {i} > {prev}")

        prev = i
    print("Done.")
    return error_list
```

Quellcode 7.1: Hilfsfunktion compare: Zum validieren der Ergebnisliste

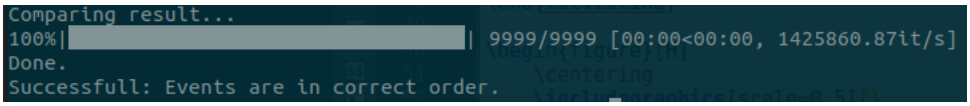

Abbildung 7.2: Ergebnis des Vergleichs der Events in der Datenbank

7.2 Konsumieren von Events

Mit der Funktion *compare*, aus dem Quellcodeausschnitt 7.1, wurden ebenfalls die durch den Projektions Datenstrom erhaltenen Events, auf Korrektheit geprüft. Um die zu vergleichende Liste zu erstellen, wurde dabei ein weiterer Konsument des Datenstroms implementiert. Dieser dient dazu die erhaltenen Events in einer Datei abzuspeichern, um diese anschließend mit der *compare* Funktion überprüfen zu können. Die durch den Datenstrom erhaltenen Events sind in Abbildung 7.3 bis 7.5 als Diagramm zu sehen. Anhand der monotonen Steigung ist zu erkennen, dass alle Events in der korrekten Reihenfolge nacheinander konsumiert wurden. Die X-Werte des Diagramms geben dabei den Zeitstempel und die Y-Werte die Aggregat-ID des Events an.

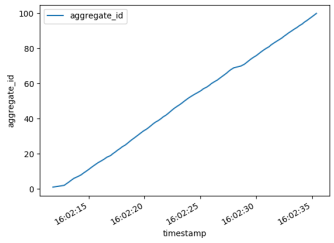

Abbildung 7.3: 100 publizierte Events

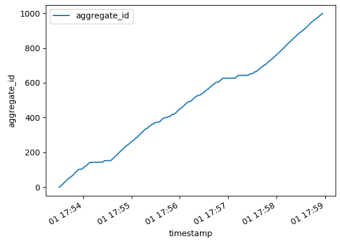

Abbildung 7.4: 1.000 publizierte Events

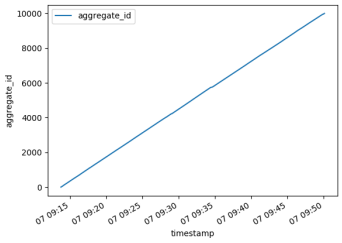

Abbildung 7.5: 10.000 publizierte Events

7.3 Wiedereinspielen von Events

Um die einzelnen Replay-Verfahren zu testen, wurden zunächst 10.000 Events mittels des Dienstes publiziert, um einen befüllten Event Store zu erhalten. Anschließend wurde wie zuvor ein weiterer Konsument für den Datenstrom, in diesem Fall für den Replay Datenstrom, angelegt. Der Konsument schreibt dabei ebenfalls die Events in eine Liste, um anschließend diese der zum vergleichen von Events erstellten Funktion *compare* zu übergeben.

7.3.1 Gesamt Replay-Verfahren

In Abbildung 7.6 sind die einzelnen Stapel von Events zu sehen, welche durch das Ausführen des Gesamt Replay-Verfahrens an den Replay Datenstrom übermittelt wurden. Die Y-Achse gibt dabei die Summe der erhaltenen Events an. Die einzelnen Punkte des Koordinatensystems zeigen die durch den Datenstrom erhaltenen Stapel von Events. Wie bereits in Kapitel 5 der Implementierung des Dienstes, erwähnt, werden die Events in

einer Stapelgröße von 400 Events mittels des Replay Streams übermittelt. In dem Diagramm ist jedoch zu erkennen, dass teilweise weniger Events übermittelt wurden. So kommt es zum Beispiel vor, dass in einem Stapel sich nur 211 Events anstelle von 400 Events befinden. Der Grund hierfür konnte nicht nachvollzogen werden. In Abbildung 7.7 ist das Ergebnis der *compare* Funktion zu sehen. Dieses zeigt, dass alle Events in korrekter Reihenfolge übermittelt wurden.

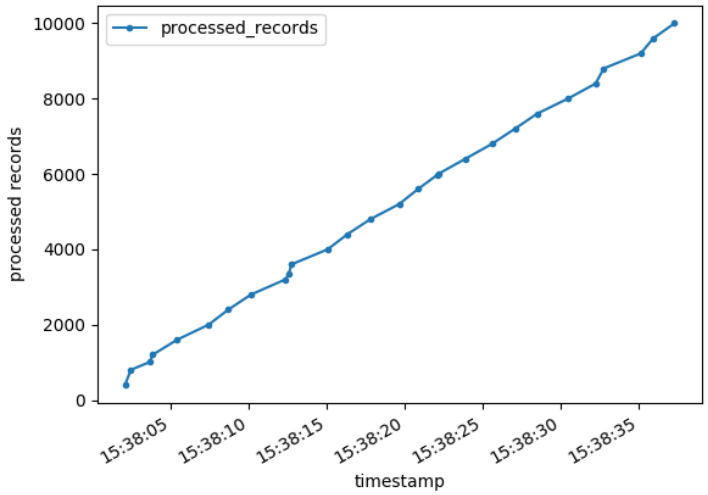

Abbildung 7.6: Verlauf Gesamt-Replay Verfahren

```
Comparing result...
100%|                    | 10000/10000 [00:00<00:00, 1688392.24it/s]
Done.
Successfull: Events are in correct order.
```

Abbildung 7.7: Vergleich Gesamt Replay-Verfahren

7.3.2 Zeitstempel basiertes Replay-Verfahren

Um das Zeitstempel basierte Replay-Verfahren zu testen, wurde der Zeitstempel des Events mit der *Aggregat-ID* 4999 genommen und bei der Anfrage übermittelt. In Abbildung 7.8 sind die einzelnen Stapel, in denen die 5000 Events übermittelt wurden, zu sehen. Dabei gibt es wie bereits bei dem Gesamt Replay-Verfahren Schwankungen, was die Stapelgrößen betrifft. Jedoch sind auch hier alle Events in ihrer korrekten Reihenfolge übermittelt worden, was durch die *compare* Funktion geprüft wurde.

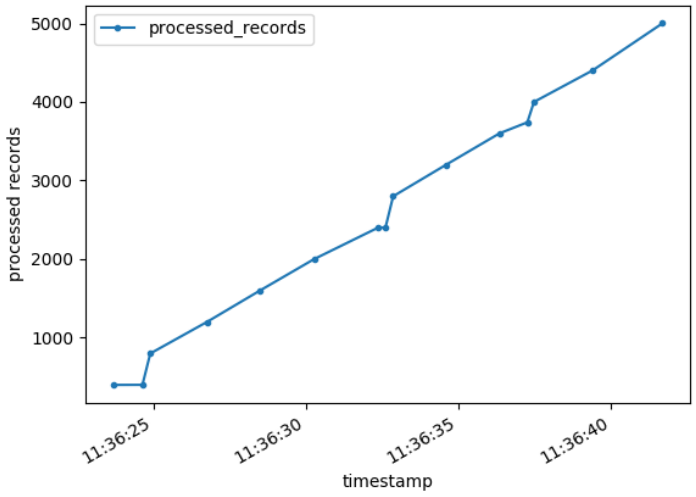

Abbildung 7.8: Verlauf Zeitstempel-Replay Verfahren

```
Comparing result...
100%|                    | 5000/5000 [00:00<00:00, 1610097.50it/s]
Done.
Successfull: Events are in correct order.
```

Abbildung 7.9: Vergleich Zeitstempel basiertes Replay-Verfahren

7.3.3 Aggregat-ID Replay-Verfahren

Für die Evaluation des ID basierten Replay-Verfahrens wurden zunächst 50 Events mit der selben *Aggregat-ID* angelegt. Anschließend wurde das wieder Einspielen dieser Events gestartet. Eine visuelle Darstellung der empfangenen Stapel ist im Fall dieses Replay-Verfahrens nicht interessant, da die 50 Events innerhalb eines einzelnen Stapels übermittelt werden. Der Vergleich zeigt jedoch wieder, dass diese in der korrekten Reihenfolge übermittelt wurden.

```
Comparing result...
100%|                    | 50/50 [00:00<00:00, 277034.61it/s]
Done.
Successfull: Events are in correct order.
```

Abbildung 7.10: Vergleich Aggregat-ID basiertes Replay-Verfahren

7.4 Sonstige Anforderungen

Neben den funktionalen Anforderungen wird der Event Sourcing Service auch auf die nicht-funktionalen Anforderungen überprüft.

7.4.1 Serverlose Softwarearchitektur

Alle Dienste, die der Service nutzt werden durch AWS verwaltet und es sind keine selbst konfigurierten Serverinstanzen wie zum Beispiel, AWS EC2 Container verwendet worden. Somit ist das Kriterium den Dienst, als serverlose Softwarearchitektur umzusetzen erfüllt.

7.4.2 Zuverlässigkeit

Wie durch die Evaluation der funktionalen Anforderungen gezeigt, erfüllt der Dienst auch das Kriterium der Zuverlässigkeit. Es wurden während der Tests immer alle Events zuverlässig und in der erwünschten Reihenfolge übermittelt.

7.4.3 Benutzbarkeit

Durch die Implementierung einer einfachen REST-API, welche aktuell nur zwei Endpunkte besitzt, ist der Dienst einfach zu nutzen. Eine Hürde könnte jedoch das Konsumieren der Events mittels des Kinesis Datenstroms darstellen. Es erfordert eine gewisse Grundkenntnis über die richtige Verwendung von AWS Kinesis Datenströme.

7.4.4 Erweiterbarkeit

Die Erweiterbarkeit des Event Sourcing Services ist ebenfalls gegeben. Durch die Verwendung des *chalice* Framworks können sehr schnell neue Endpunkte für den Dienst hinzugefügt werden. Dies erfordert kein Wissen über das AWS API Gateway, um weitere Endpunkte hinzuzufügen.

8 Ausblick und Fazit

Wie durch die Evaluation gezeigt wurde, ist die Umsetzung eines serverlos betriebenen Event Sourcing Services in der AWS Cloud realisierbar. Die an den Dienst gestellten Anforderungen konnten erfolgreich umgesetzt werden. Dennoch gibt es einige Herausforderungen, die eventuell gegen die in dieser Arbeit vorgestellte Umsetzung sprechen. Gewisse Teile des entwickelten Dienstes sollten bei der Weiterentwicklung nochmals überdacht werden.

In Bezug auf die angebotene REST API des Dienstes könnten nachfolgende Erweiterungen Sinnvoll sein. So würde beispielsweise ein Endpunkt, der die Statusabfrage eines laufenden Replay-Verfahrens ermöglicht, die Implementierung von Replay-Mechanismen auf Seiten der konsumierenden Dienste erleichtern. Ein weiterer Endpunkt für das Publizieren eines Stapels von Events würde sich ebenfalls anbieten. Durch diesen könnte die Anzahl an Requests gegenüber dem Dienst gering gehalten werden. Während der Evaluation sind zwar keine Fehler bei dem Publizieren vieler Events innerhalb weniger Sekunden aufgetreten, dennoch würde dieser zusätzliche Endpunkt die Performanz des Dienstes und die der publizierenden Dienste erhöhen.

Neben der Implementierung weiterer Schnittstellen-Endpunkte würde es sich ebenfalls anbieten, die Umsetzung der REST API und der dazugehörigen Lambda Funktion mittels des *chalice* Frameworks abzulösen. Eine einheitliche Bereitstellung der Infrastruktur mittels Terraform würde den Dienst übersichtlicher und durchschaubarer für Dritte machen. Ein Kritikpunkt, der gegen die Verwendung von Terraform spricht, ist, dass gewisse Funktionen des AWS Kinesis DataStreams nicht über Terraform konfiguriert werden können. So ist es nicht möglich, die benötigte enhanced-fanout Option mittels Terraform zu provisionieren. In Bezug auf die Provisionierung des Dienstes sollte ebenfalls noch die kontinuierliche Auslieferung (Continues delivery) umgesetzt werden bevor der Dienst in einer produktiv Umgebung eingesetzt wird.

Ein weiterer Punkt, der vor der Inbetriebnahme des Dienstes beachtet werden sollte, ist die Verarbeitung von unzustellbaren Events. Im ersten Prototypen gibt es keinen Mechanismus, der mit unzustellbaren Events umgeht. Diese sollten jedoch abgefangen werden und beispielsweise an eine AWS SQS Warteschlange gesendet werden. Welche somit als Dead-Letter-Queue dient. Events, die nicht zugestellt werden können, werden somit schnell ersichtlich und könnten manuell verarbeitet oder mittels eines Replay-Verfahrens erneut eingespielt werden.

Die Umsetzung des Event Sourcing Entwurfsmusters als eigenständiger Dienst, basierend auf einer serverlosen Softwarearchitektur, ist prinzipiell realisierbar. Die Implementierung hat dabei jedoch gezeigt, dass die Wahl der richtigen Technologien ausschlaggebend ist. Bestimmte Dienste, welche in dieser Arbeit verwendet wurden, eignen sich jedoch meines Erachtens nach weniger für die Realisierung des Event Sourcing Entwurfsmusters. Diese Dienste sollten mit Alternativen verglichen und wenn möglich ersetzt werden. So halte ich die Verwendung des Kinesis DataStream Services und somit auch des DynamoDB Streams Dienstes für ungeeignet. Durch die Limitierung auf einen Shard pro Datenstrom ist diese Technologie nicht geeignet, um die veröffentlichten Events an die Konsumenten zu übermitteln. Der Vorteil der horizontalen Skalierbarkeit welcher durch die Verwendung einer serverlosen Softwarearchitektur erreicht werden soll, ist durch diese Limitierung nicht mehr gegeben. Die Übermittlung von Events über mehrere Shards hinweg ist nicht mehr möglich, was den maximalen Datendurchsatz einschränkt. Des Weiteren könnte dies ebenfalls Auswirkungen auf die Eventual Consistency haben, welche durch die Verwendung des CQRS Entwurfmusters eingeführt wird. Bei einer extrem großen Anzahl an publizierten Events könnte die Aktualisierung der Projektionsdatenbanken sehr lange dauern. Dies würde somit zu einer längeren Zeitspanne bis zum Erreichen der Datenkonsistenz zwischen den Datenbanken führen. Um die Kinesis Datenströme ersetzen zu können, gibt es verschiedene Lösungsansätze. So könnten die Dienste welche die Projektionsdatenbanken halten die Events aus dem Event Store periodisch oder bei Bedarf abfragen. Die Projektionsdatenbanken könnten somit als Snapshot der Aggregate dienen. Als Alternative zu Kinesis DataStreams bietet AWS inzwischen einen verwalteten Apache Kafka Service an. Mittels Apache Kafka könnte das Persistieren der Events und publizieren mit Hilfe eines Datenstroms in einem Dienst erfolgen. Apache Kakfa bietet die Möglichkeit, die in den Datenstrom gelegten Events für immer zu speichern.[2]

Ich halte den Ansatz des Event Sourcing Entwurfmusters in einer serverlosen und Event basierten Softwarearchitektur für sehr hilfreich. Der Mehrwert liegt meiner Meinung nach in den folgenden Punkten. Neu entwickelte Dienste können Ereignisse verarbeiten, die

bereits vor der Entstehung dieser eingetreten sind und somit in den gleichen Systemzustand gelangen, wie parallel betriebene Dienste. Durch die Fähigkeit Ereignisse, die bereits in der Vergangenheit liegen erneut zu prozessieren, bietet sich die Möglichkeit einer schnellen und einfachen Datenanalyse. Des Weiteren ist ein einfaches Debuggen von Fehlern möglich. So können Fehler im Systemverhalten schnell lokalisiert werden, indem ein parallel betriebenes System mit den Events aus dem Event Store erneut befüllt wird.

Literaturverzeichnis

[1] AWS Chalice. Aws chalice serverless framework. https://chalice.readthedocs.io/en/latest/index.html. Besucht: 25.10.2019.

[2] Apache Software Foundation. Apache kafka, log compaction basics. http://kafka.apache.org/documentation.html#compaction. Besucht: 05.03.2020.

[3] Martin Fowler. Continues delivery. https://martinfowler.com/bliki/ContinuousDelivery.html. Besucht: 10.2.2020.

[4] Martin Fowler. Continues integration. https://martinfowler.com/articles/continuousIntegration.html. Besucht: 10.2.2020.

[5] Martin Fowler. CQRS. https://www.martinfowler.com/bliki/CQRS.html. Besucht: 22.10.2019.

[6] HashiCorp. Terraform. https://learn.hashicorp.com/terraform/getting-started/intro. Besucht: 01.12.2019.

[7] Martin Kleppmann. *Desinging Data-Intensive Applications*. O'Reilly, 2019. Kapitel: 11 - Stream Processing, Seite: 459.

[8] Aravind Kodandaramaiah. Lambda fan-out pattern. https://aws.amazon.com/de/blogs/database/how-to-perform-ordered-data-replication-between-applications-by-using-amazon-dynamodb-streams/. Besucht: 26.1.2020.

[9] Marshmallow. Marshmallow: simplified object serialization. https://marshmallow.readthedocs.io/en/stable/index.html. Besucht: 26.1.2020.

[10] Microsoft. Event sourcing pattern. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn589792(v=pandp.10). Besucht: 20.10.2019.

[11] Danilo Poccia. *AWS Lambda in Action*. Manning Publications Co., 2017. Kapitel: 3.1 Introducing the Amazon API Gateway, Seite: 39.

[12] Danilo Poccia. *AWS Lambda in Action*. Manning Publications Co., 2017. Kapitel: 1.1 Introducing AWS Lambda, Seite: 7.

[13] PynamoDB. Pynamodb. https://pynamodb.readthedocs.io/en/latest/. Besucht: 12.1.2020.

[14] Requests. Python requests library. https://requests.readthedocs.io/en/master/. Besucht: 30.3.2020.

[15] Amazon Web Services. Amazon kinesis data streams - developer guide. https://docs.aws.amazon.com/streams/latest/dev/kinesis-dg.pdf. Besucht: 26.11.2019.

[16] Amazon Web Services. Aws cloudwatch. https://aws.amazon.com/de/cloudwatch/. Besucht: 22.2.2020.

[17] Amazon Web Services. Aws dynamodb - consistency. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html. Besucht: 12.12.2019.

[18] Amazon Web Services. Aws dynamodb - query. https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html. Besucht: 26.1.2020.

[19] Amazon Web Services. Aws dynamodb streams. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html. Besucht: 12.12.2019.

[20] Amazon Web Services. Aws educate. https://aws.amazon.com/de/education/awseducate. Besucht: 25.10.2019.

[21] Amazon Web Services. Aws free-tier. https://aws.amazon.com/de/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc. Besucht: 01.12.2019.

[22] Amazon Web Services. Aws kinesis date stream - putrecords. https://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecords.html. Besucht: 26.1.2020.

[23] Amazon Web Services. Aws lambda limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html. Besucht: 26.1.2020.

[24] Amazon Web Services. Aws step functions. https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. Besucht: 24.1.2020.

[25] Amazon Web Services. Aws step functions - iterating a loop using lambda. https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-create-iterate-pattern-section.html. Besucht: 24.1.2020.

[26] Amazon Web Services. Aws step functions - service integrations. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-service-integrations.html. Besucht: 24.1.2020.

[27] Amazon Web Services. Dynamodb - item size. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/CapacityUnitCalculations.html. Besucht: 26.1.2020.

[28] Amazon Web Services. Dynamodb - primary key. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html#HowItWorks.CoreComponents.PrimaryKey. Besucht: 26.1.2020.

[29] Amazon Web Services. Dynamodb streams. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.KCLAdapter.html. Besucht: 26.1.2020.

[30] Amazon Web Services. Limits in dynamodb. https://docs.aws.amazon.com/de_de/amazondynamodb/latest/developerguide/Limits.html#limits-dynamodb-streams. Besucht: 26.1.2020.

[31] Amazon Web Services. Read/write capacity mode. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html. Besucht: 26.1.2020.

[32] Gernot Starke. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden, 7. Auflage.* Carl Hanser Verlag, 2015. Kapitel: 5.4 Sichten, Seite:153.

[33] Gernot Starke. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden, 7. Auflage.* Carl Hanser Verlag, 2015. Kapitel: 5.6 Bausteinsicht, Seite: 163.

[34] Ben Stopford. *Desinging Event-Driven Systems.* O'Reilly, 2018. Kapitel: 5 Events: A Basis for Collaboration, Seite: 31.

[35] Ben Stopford. *Desinging Event-Driven Systems.* O'Reilly, 2018. Kapitel: 7 Event Sourcing, CQRS, and Other Stateful Patterns, Seite: 59.

[36] Scott Millet und Nick Tune. *Patterns, Principles, and Practices of domain-driven Design.* John Wiley and Sons, 2015. Kapitel: 22 Event Sourcing - In: The Limitations of Storing State as a Snapshot, Seite: 596.

[37] Scott Millet und Nick Tune. *Patterns, Principles, and Practices of domain-driven Design.* John Wiley and Sons, 2015. Kapitel: 22 Event Sourcing - In: Projections, Seite: 599.

[38] Scott Millet und Nick Tune. *Patterns, Principles, and Practices of domain-driven Design.* John Wiley and Sons, 2015. Kapitel: 22 Event Sourcing, Seite: 599.

[39] Andres Spillner und Tilo Linz. *Basiswissen Softwaretest.* dpunkt Verlag, 2019. Kapitel: 3.4.1 Komponententest, Seite: 63.

[40] Andres Spillner und Tilo Linz. *Basiswissen Softwaretest.* dpunkt Verlag, 2019. Kapitel: 3.4.1 Integrationstest, Seite: 71.

[41] Wikipedia. Event sourcing. https://de.wikipedia.org/wiki/Event_Sourcing. Besucht: 15.06.2019.

[42] Wikipedia. Materialized views. https://en.wikipedia.org/wiki/Materialized_view. Besucht: 15.10.2019.

[43] Wikipedia. Mock-objekt. https://de.wikipedia.org/wiki/Mock-Objekt. Besucht: 10.2.2020.

[44] Andreas Wittig. *Amazon Web Services in Action.* Manning Publications Co., 2016. Kapitel: 10 Programming for the NoSQL database service: DynamoDB, Seite: 253.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name _____

Vorname _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Entwicklung eines prototypischen Services zur Realisierung des Event Sourcing Entwurfmusters in einer serverlosen Softwarearchitektur

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original