Masterarbeit

Vergleich von Architekturqualitätsmerkmalen basierend auf Ereignissen, Process Engines und dem Prozessgesteuerten Ansatz am Beispiel eines Reisebuchungsprozesses

zur Erlangung des akademischen Grades

Master of Science

vorgelegt von

Marc Leon Stiglmayr
Matrikel-Nr.: 00080847

Studiengang:	Informatik
Fakultät:	Informatik
Ausgegeben am:	18.06.2021
Abgegeben am:	17.12.2021
Erstprüfer:	Prof. Dr. Volker Stiehl
Zweitprüfer:	Prof. Dr. Bernd Hafenrichter
Betreuer:	Markus Herhoffer, Dipl.-Inform.

Kurzfassung

Das Ziel dieser Arbeit ist ein Vergleich mehrerer Softwarearchitekturen zur optimalen Implementierung von Geschäftsprozessen. Betrachtungsgegenstand sind dabei ereignisgesteuerte Architekturen, die Process Engine basierte Architektur und der Prozessgesteuerte Ansatz. Zum Zwecke einer aussagekräftigen Analyse werden alle Architekturen anhand eines beispielhaften Reisebuchungsprozesses implementiert und miteinander verglichen. Die unterschiedlichen Implementierungen folgen dabei strikt den vorgegebenen Prozess- und Systemanforderungen. Dadurch wird bereits am Beispiel der Implementierungen deutlich, in welchen Aspekten sich die Architekturen voneinander unterscheiden und mit welchen Vor- und Nachteilen diese einhergehen. Der eigentliche Architekturvergleich findet auf Basis ausgewählter Qualitätsmerkmale gemäß DIN/25010 statt. Anhand der Evaluation ist zu erkennen, dass jede der betrachteten Architekturen einzelne Qualitätsmerkmale besser unterstützt als andere. Es ist dabei deutlich geworden, dass der Einsatz von Process Engines zur Implementierung von Geschäftsprozessen Vorteile im Hinblick auf die Wartbarkeit und Zuverlässigkeit des Systems bieten. Die Implementierung nach dem Prozessgesteuerten Ansatz hat vor allem im Hinblick auf die Langlebigkeit und Nachhaltigkeit des Systems gegenüber den restlichen Architekturen am meisten überzeugt.

Schlagwörter: Prozessautomatisierung, Softwarearchitekturen, Qualitätsmerkmale, Process Engine, Ereignisgesteuerte Architektur, Process Engine basierte Architektur, Prozessgesteuerter Ansatz, Java, Reisebuchungsprozess

Abstract

The goal of this work is a comparison of several software architectures for the optimal implementation of business processes. The objects of consideration are event driven architectures, the process engine based architecture and the process driven approach. For the purpose of a meaningful analysis, all architectures are implemented and compared with each other using an exemplary travel booking process. The different implementations strictly follow the given process and system requirements. This makes it clear from the example of the implementations in which aspects the architectures differ from one another and what advantages and disadvantages they entail. The actual architecture comparison takes place on the basis of selected quality characteristics in accordance with DIN/25010. The evaluation shows that each of the architectures considered supports individual quality features better than others. It has become clear that the use of process engines for implementing business processes offers advantages in terms of maintainability and reliability of the system. Implementation using the process-driven approach was most convincing in terms of system durability and sustainability compared to the remaining architectures.

Keywords: Process automation, software architectures, quality features, process engine, event driven architecture, process engine based architecture, process driven approach, java, travel booking process

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
ARIS	Architektur integrierter Informationssysteme
ATAM	Architecture Tradeoff Analysis Methode
BPM	Business Process Management
BPMN	Business Process Model and Notation
CEP	Complex Event Processing
CRM	Customer Relationship Management
CSS	Cascading Style Sheets
CSV	Comma-separated values
DDD	Domain-Driven Design
DIN	Deutsches Institut für Normung
DLQ	Dead Letter Queue
DMN	Decision Model and Notation
DV	Datenverarbeitung
EDA	Event-Driven Architecture
EDAC	Ereignisgesteuerte Architektur nach dem Choreography Pattern
EDAO	Ereignisgesteuerte Architektur nach dem Orchestration Pattern
HATEOS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
ISO	Internationale Organisation für Normung
JMS	Java Message Service
JPA	Jakarta Persistence API
JSON	Javascript Object Notation
MOM	Message-oriented Middleware
MVC	Model View Controller
OMG	Object Management Group

PDA	Process Driven Application; Process Driven Architecture; Process Driven Approach; Architektur nach dem Prozessgesteuerten Ansatz
PE	Process Engine basierte Architektur
REST	Representational State Transfer
RPC	Remote Procedure Call
SCIL	Service Contract Implementation Layer
SLA	Service Level Agreement
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WSDL	Web Service Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition
YAML	YAML Ain't Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1. Einleitung zum Architekturvergleich

1.1. Motivation eines Architekturvergleichs basierend auf Ereignissen, Process Engines und dem Prozessgesteuerten Ansatz

Geschäftsprozesse sind ein essenzieller Teil eines jeden Unternehmens. Sie sind die Grundlage zur Leistungserbringung und Erfüllung der Kundenbedürfnisse und unterliegen dementsprechend ständigen Veränderungen. In Zeiten der digitalen Transformation stehen Unternehmen vor großen Herausforderungen. Der Wandel hin zu einer agilen Organisation und digitalen Geschäftsmodellen ist unabdingbar. Neue Technologien und sich ändernde Kundenanforderungen erfordern schnelle Anpassungen und eine hohe Flexibilität der Geschäftsprozesse, um die Wettbewerbsfähigkeit als Unternehmen zu wahren. Eine reine Dokumentation der fachlichen Prozesse ist alleine längst nicht mehr ausreichend: „*Im Zentrum des digitalen Unternehmens stehen digitalisierte Prozesse*“ [Appelfeller und Feldmann, 2018, S. 5]. Diese werden durch IT-Systeme gesteuert und einzelne Schritte wenn möglich automatisiert, mit dem Ziel einer Effizienzsteigerung [vgl. Appelfeller und Feldmann, 2018, S. 5]. Doch eine reine „Digitalisierung“ der bestehenden Geschäftsprozesse ist oft nicht zielführend. Zur Lösung von Kundenproblem ist Flexibilität bei der Prozessgestaltung geboten, damit neue digitale Geschäftsmodelle entwickelt werden können. Die optimale Implementierung und Automatisierung der Geschäftsprozesse stellt dabei immer wieder neue Herausforderungen an die IT. Die Softwarearchitektur sollte unterstützend im Hinblick auf die Herausforderungen der digitalen Transformation sein und dazu beitragen Flexibilität und Transparenz in komplexen Prozesslandschaften zu gewinnen.

Große digitale Unternehmen wie Amazon oder Netflix setzten bereits seit Jahren auf Microservice Architekturen in Kombination mit Cloud-Infrastrukturen, um Agilität und Skalierbarkeit zu erzielen [vgl. Wolff, 2018, S. 396]. Dabei existieren unterschiedliche Ansätze, Geschäftsprozesse in Kombination mit Microservices umzusetzen. Fokus liegt hierbei überwiegend auf der Art der Koordination der einzelnen Services. Dies kann sowohl dezentral über den Austausch von Geschäftsereignissen als auch zentral durch den Einsatz eines Orchestrators oder einer Workflow Engine geschehen. Auch ist eine klassische monolithische Anwendung denkbar, worin die Geschäftsprozesse zentral durch Verwendung einer Process Engine implementiert sind.

Zwischen all diesen Möglichkeiten zur Implementierung von Geschäftsprozessen ist abzuwägen, welches Architekturmuster im Hinblick auf dessen Qualitätsmerkmale die geringsten technischen Schulden mit sich bringt und dabei zu einer bestmöglichen technischen Abbildung der fachlichen Ende-zu-Ende-Prozesse beitragen kann. Die Architektur sollte dabei im Hinblick auf die digitale Transformation die Langlebigkeit der Informationssysteme unterstützen und auf Flexibilität und Agilität ausgelegt sein.

1.2. Zielsetzung des Architekturvergleichs

Die Zielsetzung dieser Arbeit ist es, aktuelle Softwarearchitekturen zur Implementierung von Geschäftsprozessen miteinander zu vergleichen. Der Vergleich basiert anhand ausgewählter Architekturqualitätsmerkmale auf Basis eines Beispielprozesses. Im Fokus der Arbeit stehen ereignisgesteuerte Architekturen, Process Engine basierte Architekturen und die Umsetzung nach dem Prozessgesteuerten Ansatz. Auf Grundlage der Erkenntnisse soll es ermöglicht werden, eine Leitlinie zur Implementierung von Geschäftsprozessen im Unternehmen geben zu können.

1.3. Vorgehen zum Architekturvergleich

Im Rahmen dieser Arbeit werden unterschiedliche Softwarearchitekturen zur Implementierung von Geschäftsprozessen miteinander verglichen. Als Grundlage des Vergleichs wird dafür ein fiktiver Ende-zu-Ende Reisebuchungsprozess definiert, der zum Zwecke der realitätsnahen Abbildung die Integration mit einer bestehenden Anwendungslandschaft und externen Systemen voraussetzt. Basierend auf den Anforderungen des Reisebuchungsprozesses wird das Reisebuchungssystem je Architekturvariante implementiert, sodass anhand der Implementierungen ein Vergleich der ausgewählten Architekturqualitätsmerkmale a posteriori stattfinden kann.

Kapitel 2 stellt dafür zunächst die für diese Arbeit relevanten Grundlagen vor. Dazu zählen sowohl grundlegende Konzepte und Prinzipien von Softwarearchitekturen als auch entsprechende Entwurfsmethoden — wie dem „Domain-Driven Design“ (DDD) und der „Architektur integrierter Informationssysteme“ (ARIS). Kommunikationsmechanismen verteilter Systeme werden erläutert, sodass daraufhin die für die späteren Implementierungen relevanten Softwarearchitekturstile definiert und beschrieben werden können. Zusätzlich wird ein kleiner Einblick in das API-Management („Application Programming Interface Management“) und bewährte Schnittstellenprotokolle wie das „Simple Object Access Protocol“ (SOAP) und „Representational State Transfer“ (REST) gegeben. Kapitel 3 untersucht die beiden Entwurfsmethoden Domain-Driven Design und ARIS und stellt deren Gemeinsamkeiten in Bezug auf die Geschäftsprozessmodellierung heraus. Zusätzlich wird die modellgetriebene Entwicklung mit BPMN 2.0 („Business Process Model and Notation“) und dem Einsatz einer Process Engine vorgestellt. Kapitel 4 beschäftigt sich mit dem praktischen Anteil der Arbeit: Der Implementierung des Reisebuchungsprozesses nach der ereignisgesteuerten Architektur, Process Engine basierten Architektur und dem prozessgesteuerten Ansatz. Hierfür wird zunächst die vorgegebene Systemlandschaft, bestehend aus internen, externen, und Infrastruktursystemen definiert. Im Anschluss folgt eine prosaische Beschreibung der Anforderungen an den Reisebuchungsprozess und der technischen Implementierungen. Für ein besserer Verständnis werden Benutzeroberflächen und das Datenmodell vorgestellt, sodass danach die einzelnen Implementierungen behandelt werden können. Kapitel 5 thematisiert den Architekturvergleich. Dort werden zunächst die zu untersuchenden Qualitätsmerkmale ausgewählt und konkretisiert, sodass anschließend basierend auf den Merkmalen der Architekturvergleich stattfinden kann. Zum Abschluss der Arbeit wird in Kapitel 6 ein Fazit über den Vergleich gezogen und zusätzlich ein Ausblick über mögliche, weiterführende Arbeiten gegeben.

2. Theoretische Grundlagen zum Vergleich von Architekturen zur Implementierung von Informationssystemen

Dieses Kapitel stellt die für diese Arbeit relevanten theoretischen Grundlagen vor. Zunächst werden fundamentale Begriffe und Prinzipien der Softwarearchitektur behandelt. Unter anderem wird erklärt was technische Schulden sind (vgl. Kapitel 2.1.3) und nach welchen Kriterien Softwarearchitekturen bewerten werden können (vgl. Kapitel 2.1.4). Daraufhin werden in Kapitel 2.2 Domain-Driven Design und ARIS als Methoden zur Modellierung von Softwarearchitekturen vorgestellt. Nachdem ein kurzer Exkurs hin zu synchroner und asynchroner Kommunikation stattfindet, widmet sich Kapitel 2.4 den verschiedenen Softwarearchitekturstilen. Fokus wird dabei auf Ereignisgesteuerte, Process-Engine basierte und Prozessgesteuerte Architekturen gelegt. Abschließend wird ein Einblick in das API-Management gegeben (vgl. Kapitel 2.5.4). Dort werden Grundlagen von APIs und API-Management-Plattformen thematisiert sowie SOAP und REST als zwei Arten von APIs aufgezeigt.

2.1. Grundlegende Konzepte und Prinzipien von Softwarearchitekturen

2.1.1. Definition: Softwarearchitektur

Eine einheitliche Begriffsdefinition der Softwarearchitektur hat sich bis heute nicht etabliert [vgl. Lilienthal, 2020, S. 1]. Aus diesem Grund werden nachfolgend einige ausgewählte Definition betrachtet und deren Gemeinsamkeiten herausgestellt.

Gernot Starke orientiert sich bei der Definition an dem IEEE-Standard 1471, der Softwarearchitektur wie folgt definiert [Starke, 2020, S. 16]:

> *„Softwarearchitektur: Die grundsätzliche Organisation eines Systems, verkörpert durch dessen Komponenten, deren Beziehung zueinander und zur Umgebung sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.“*

Eine Softwarearchitektur beschreibt demnach die grundlegenden Strukturen und Komponenten eines Systems. Die Komponenten — oder auch Bausteine genannt — können dabei sowohl untereinander, als auch mit ihrer Umgebung kommunizieren. Diese Aspekte greift auch Balzert in seiner Definition zu Softwarearchitektur auf [vgl. Balzert, 2011, S. 23]. Viel mehr beschreibt er, dass die extern sichtbaren Komponenten — also die Komponenten die in einer Beziehung zu ihrer Umgebung stehen — durch Schnittstellen spezifiziert werden. Schnittstellen bilden das Fundament für modulare Systeme, und ermöglichen es ein System in mehrere Bausteine zu untergliedern, die miteinander kommunizieren können [vgl. Starke, 2020, S. 98]. Die einzelnen Bausteine der Software können dabei sowohl physisch innerhalb eines Betriebssystemprozesses ablaufen, als auch verteilt auf mehreren Rechnern und Prozessen. Bei letzterem wird auch von

einem verteilten System gesprochen [vgl. Dunkel u. a., 2008, S. 12].

Robert C. Martin verwendet in seinem Buch „Clean Architecture“ bei der Definition beispielsweise die Analogie einer Form, die dem System von den Entwicklern gegeben wird: „*The architecture of a software system is the shape given to that system by those who build it.*“ [Martin, 2018, S. 136]. Die Ausprägung der Form ist dadurch geprägt, in welcher Anordnung die einzelnen Bausteine bzw. Komponenten zusammengesetzt sind und in welcher Art und Weise diese miteinander kommunizieren.

Alle genannten Definitionen haben gemeinsam, dass sich Softwarearchitektur mit Komponenten und Verbindungen zwischen diesen beschäftigt. Generell wird dort jedoch recht abstrakt über Komponenten eines Systems gesprochen. Die konkrete Ausprägung dieser hängt von der jeweiligen Sichtweise auf das System ab. Orientiert an Philippe Kruchtens „4+1 Sichtenmodell“ [vgl. Kruchten, 1995], unterscheidet Gernot Starke beispielsweise zwischen der Baustein-, Laufzeit-, Verteilungs- und Kontextsicht [vgl. Starke, 2020, S. 155 ff.]. Besonders interessant ist in diesem Kontext die Bausteinsicht. Diese beschreibt die vom System bereitzustellende Funktionalität und dessen Services — also wie das System im Inneren aufgebaut ist. Die Architektur kann dabei sowohl auf grober Ebene, durch Komponenten und Subsysteme, als auch auf feiner Ebene durch Module, Pakete und Klassen modelliert sein [vgl. Starke, 2020, S. 161 ff.].

Innerhalb einer Microservice-basierten Architektur (vgl. Kapitel 2.4.2) kann ein einzelner Service, der beispielsweise über REST mit anderen Services kommuniziert (vgl. Kapitel 2.5.3), als Komponententyp ansehen werden [vgl. Ford und Richards, 2020, S. 103]. Die Komponenten eines Systems werden meist nach domänenspezifischen oder technischen Aspekten partitioniert [vgl. Ford und Richards, 2020, S. 105 ff.]. Kapitel 2.2 stellt mit Domain-Driven Design und ARIS Methodiken vor, die bei der Aufteilung eines Systems in Komponenten unterstützen.

Ein weiterer Punkt der durch die obige Definition berücksichtigt wird, ist dass Systeme üblicherweise Prinzipien unterliegen, die für dessen Entwurf und Weiterentwicklung gelten. Diese Prinzipien werden durch einzelne Entscheidungen, wie beispielsweise die Wahl eines Architekturstils — auch Architekturmuster genannt — oder die Vorgabe von Entwicklungsrichtlinien beeinflusst [vgl. Starke, 2020, S. 16]. Im nächsten Kapitel werden einige relevante Architekturprinzipien vorgestellt.

2.1.2. Vorstellung ausgewählter Architekturprinzipien

Balzert definiert Prinzipien als [Balzert, 2011, S. 29]:

> *„[...] Grundsätze, die man seinem Handeln zugrunde legt. Prinzipien sind allgemeingültig, abstrakt, allgemeinster Art. [...] Prinzipien werden aus der Erfahrung und Erkenntnis hergeleitet und durch sie bestätigt*“.

Prinzipien sind demnach abstrakt formuliert und können deshalb in unterschiedlichen Kontexten Anwendung finden. Das bedeutet, dass einige Leitsätze nicht ausschließlich auf den Softwareentwurf bzw. die Softwarearchitektur eingeschränkt sind, sondern als allgemeine Grundsätze der Softwaretechnik gelten. Geeignete Architektur- und Entwurfsmuster unterstützen typischerweise bei der Umsetzung der Architekturprinzipien [vgl. Balzert, 2011, S. 29].

Balzert definiert eine Reihe verschiedener, grundlegender Prinzipien, die für die Architektur und Entwicklung einer Software gelten sollten. Im Folgenden werden einige dieser Leitsätze

vorgestellt, die für den späteren Verlauf dieser Arbeit von Relevanz sind [vgl. Balzert, 2011, S. 29 ff.; vgl. Balzert, 2009, S. 25 ff.].

Prinzip der Abstraktion

Abstrahieren bedeutet sich vom Konkreten zu lösen. Abstraktion zielt darauf ab, gemeinsame Merkmale zu identifizieren und durch Verallgemeinerung essenzielle Kernpunkte herauszuheben. Dabei wir ein Modell erstellt, welches die reale Welt und deren charakteristische Eigenschaften abbildet. Eine Abstraktion kann mehrere Ebenen umfassen, um Abstufungen bis hin zum Konkreten darzustellen [vgl. Balzert, 2009, S. 26 ff.].

Prinzip der Bindung und Kopplung

Die Softwarearchitekturdefinition aus Kapitel 2.1.1 hat verdeutlicht, dass eine Softwarearchitektur die grundlegenden Strukturen eines Systems beschreibt. Die Bindung der Systemkomponenten und die Kopplung zwischen den Komponenten sind zwei Eigenschaften, die im Wesentlichen zur Strukturbestimmung beitragen [vgl. Balzert, 2009, S. 37].

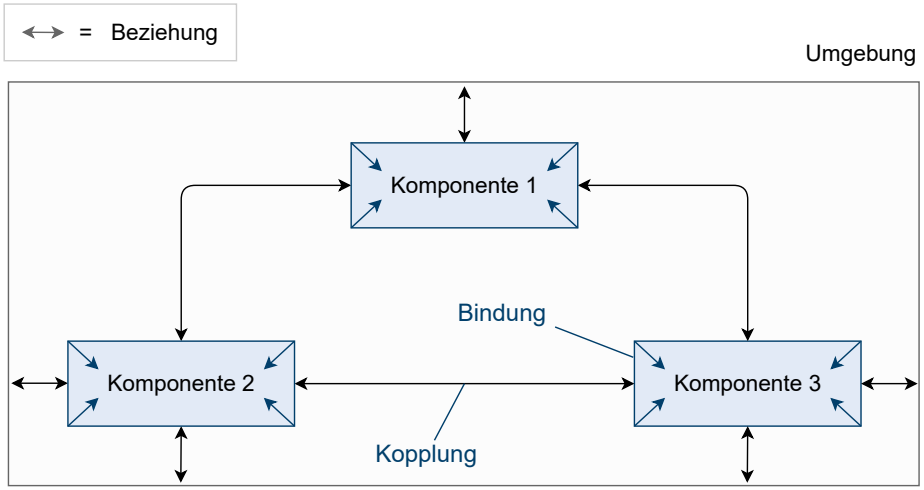

Abbildung 2.1.: Bindung und Kopplung zwischen Komponenten eines Systems [In Anlehnung an Balzert 2009, S. 37]

Die **Bindung** — oder auch Kohäsion genannt — betrachtet die Beziehungen und den Zusammenhalt zwischen den einzelnen Bestandteilen **innerhalb einer Komponente**. Ziel ist es, eine möglichst hohe Bindung zu erreichen, worin alle Elemente einer Komponente inhaltlich eng zusammengehören und in Beziehung zueinander stehen, um Aufgaben zu erledigen (vgl. Abbildung 2.1) [vgl. Balzert, 2009, S. 37 f.].

Im Gegensatz zur Bindung untersucht die **Kopplung** die Abhängigkeiten **zwischen den Systemkomponenten** und der Umgebung (vgl. Abbildung 2.1). Während die Bindung sich mit den Abhängigkeiten innerhalb einer logischen Einheit beschäftigt, geht es bei der Kopplung um übergreifende Beziehungen. Dabei kann es sich sowohl um Schnittstellenaufrufe oder direkte Klassenaufrufe zwischen den einzelnen Services oder Modulen handeln. Bei einem hohen Kopplungsgrad haben Änderungen einer Komponente Auswirkungen auf andere Komponenten. Da es häufig nicht offensichtlich ist, welche Effekte eine Änderung nach sich ziehen kann, versucht man eine lose Kopplung im System zu erreichen. Sind die Komponenten lose gekoppelt, beeinflussen sich diese im Optimalfall nur bei Schnittstellenänderungen. Neben dem

Schnittstellenumfang hat auch die Kommunikationsart eine Auswirkung auf die Kopplung. Eine synchrone Kommunikation (vgl. 2.3) führt beispielsweise zu einer höheren Kopplung, da die aufrufende Komponente Wissen über die Schnittstelle der Zielkomponente besitzen muss und im Kontext eines verteilten Systems auf dessen Verfügbarkeit angewiesen ist [vgl. Balzert, 2009, S. 37 f.].

Um die Komplexität eines Systems gering zu halten, ist es das Ziel eine hohe Bindung und lose Kopplung der Komponenten aufrechtzuerhalten. Das Prinzip der Bindung und Kopplung hat einen direkten Einfluss auf die Modularität eines Systems, da versucht wird zusammenhängende und weitestgehend unabhängige Module zu bilden [vgl. Balzert, 2009, S. 38 f.].

Prinzip der Modularisierung

Das Prinzip der Modularisierung wurde bereits im Jahre 1972 diskutiert, als David Parnas ein Paper veröffentlicht, in dem er die Kriterien und Vorteile des Aufteilens eines Systems in Module bzw. Komponenten beschreibt [vgl. Parnas, 1972].

Ein Modul repräsentiert eine in sich abgeschlossene, funktionale Einheit, die weitestgehend unabhängig von anderen Modulen ist [vgl. Balzert, 2009, S. 40 f.]. Parnas spricht in diesem Kontext auch von dem sogenannten „Information Hiding“: Details der eigentlichen Implementierung und dessen Designentscheidungen bleiben vor der Außenwelt verborgen. Nur einzelne Funktionalitäten werden über wohldefinierte Schnittstellen nach außen bereitstellt [vgl. Parnas, 1972]. Bereits in der Definition zu Softwarearchitektur (vgl. Kapitel 2.1.1) ist die Rede von modularen Komponenten, die über Schnittstellen kommunizieren. Das Aufteilen eines Systems in mehrere Komponenten bringt einige Vorteile in Bezug auf die Wartbarkeit und Verständlichkeit einer Software mit sich [vgl. Balzert, 2009, S. 42]. Modularität steht dabei in enger Relation zu weiterführenden Konzepten wie die Bindung und Kopplung von Komponenten, sowie die Trennung von Zuständigkeiten.

Prinzip der Trennung von Zuständigkeiten

Die Grundidee des Prinzips der Trennung von Zuständigkeiten (engl.: „Separation of Concerns“) basiert auf einer Veröffentlichung aus dem Jahre 1974 von Edsger W. Dijkstra [Dijkstra, 1982b, S. 61], worin er beschreibt, dass das isolierte Betrachten einzelner Aspekte einen effizienteren Denkprozess ermöglicht. In Bezug auf die Softwarearchitektur ist damit gemeint, dass jede Komponente im besten Fall nur für einen Aufgabenbereich zuständig ist. Die Verantwortlichkeit für eine Aufgabe liegt also immer bei einer Komponente und wird nicht über mehrere Komponenten hinweg verteilt.

Durch die Trennung von Zuständigkeiten wird das Prinzip der Modularisierung unterstützt. Außerdem trägt es zur Verständlichkeit bei und unterstützt, dass Fachkonzepte aus der Architektur heraus erkennbar werden. Eine zu detaillierte und starke Trennung der Zuständigkeiten kann jedoch negative Effekte haben und zu einer Fragmentierung führen, weshalb eine genaue Analyse bei der Bildung kohärenter Einheiten notwendig ist [vgl. Balzert, 2011, S. 31 f.].

2.1.3. Technische Schulden

Der Begriff „Technische Schulden“ (engl.: „Technical debt“) wurde als Metapher durch Ward Cunningham [vgl. Cunningham, 1993] geprägt. Mit „Technischer Schuld“ ist der zukünftige Mehraufwand gemeint, der mit einer schlechten Umsetzung der Software einhergeht. Die

Metapher orientiert sich an finanziellen Schulden. Wenn in der Finanzwelt ein Darlehen aufgenommen wird, so werden Zinsen für das Darlehen berechnet, bis es abbezahlt ist. Analog hierzu stellen die Zinsen in der Softwareentwicklung den erhöhten Aufwand für Wartung und Erweiterung der Software dar. Der Mehraufwand entsteht durch das Eingehen technischer Schulden, d.h. Treffen suboptimaler technischer Entscheidungen. Die Softwarearchitektur ist ein wesentlicher Bestandteil der dazu beiträgt, technische Schulden im System vorzubeugen und zu vermeiden, sodass Erweiterungen der Software beispielsweise keine neuen Schulden hervorrufen [vgl. Lilienthal, 2020, S. 4 f.; vgl. Fowler, 2019].

Martin Fowler unterscheidet vier Arten von technischen Schulden, die als das „Technische Schulden Quadrant“ bekannt sind (siehe Tabelle 2.1) [vgl. Fowler und Lewis, 2014].

	Rücksichtslos	**Umsichtig**
Wissentlich	Wir haben keine Zeit für Design	Wir müssen jetzt ausliefern und uns später um die Konsequenzen kümmern
Unbeabsichtigt	Was ist mit Schichtenarchitektur gemeint?	Jetzt wissen wir wie wir es hätten implementieren müssen

Tabelle 2.1.: Technische Schulden Quadrant [Fowler und Lewis, 2014]

Technische Schulden werden demnach entweder rücksichtslos oder umsichtig getroffen. Auch kann eine technische Schuld wissentlich oder unbeabsichtigt eingegangen werden. Rückt beispielsweise der Release einer Software näher und es stehen noch einige Features für die Entwicklung an, so können wissentlich technische Schulden aufgenommen werden, um rechtzeitig ausliefern zu können. Eine solche technische Schuld ist umsichtig, da hierbei abgewägt wird, ob der Nutzen für die zusätzlich ausgelieferte Funktionalität größer, als die Kosten für die spätere Behebung der Schulden sind. Ein Beispiel für eine wissentlich, rücksichtslose technische Schuld wäre, wenn sich das Entwicklungsteam trotz Wissen über Architektur- und Entwurfsprinzipien für eine „quick and dirty“ Lösung entscheidet, mit der Begründung, dass keine Zeit für eine saubere Architektur und Implementierung vorhanden ist.

Die mangelnde Zeit wird häufig als Hauptursache für technische Schulden angegeben. Zeitmangel ist jedoch nicht der einzige Grund. Schlechte Prozesse, eine geringe Testabdeckung, nicht genügend qualifizierte Entwickler und eine unzureichende Qualitätssicherung und Dokumentation sich nur einige Aspekte, die technische Schulden verursachen können. Dabei zeigt sich, dass sich technische Schulden nicht nur auf den Code beziehen. Fehlentscheidungen bei der Architektur oder dem Design des Systems spielen eine signifikante Rolle bei der Entstehung und Anhäufung technischer Schulden. Eine gute Architekturwahl kann deshalb das Ausmaß technischer Schulden reduzieren [vgl. Kruchten u. a., 2012, S. 19 f.].

2.1.4. Bewertung von Softwarearchitekturen

Kapitel 2.1.3 hat erklärt was technische Schulden sind und warum diese vermieden werden sollten. Es wurde deutlich, dass technische Schulden mit einem Qualitätsverlust der Software einhergehen. Die Architektur dient dazu sicherzustellen, dass die Anforderungen an die Qualität eines Systems erfüllt werden — auch wenn eine gute Architektur nicht unbedingt eine gute Implementierung impliziert [vgl. Starke, 2020, S. 39 f.]. Dabei stellt sich die Frage, anhand welcher Kriterien die Qualität einer Softwarearchitektur gemessen werden kann. Generell wird

zwischen einer **qualitativen** und **quantitativen** Bewertung unterschieden, dessen wesentliche Konzepte im Folgenden kurz erklärt werden [vgl. Starke, 2020].

Qualitative Bewertung

In der qualitativen Architekturbewertung werden Softwarearchitekturen danach bewertet, inwieweit sie zur Erreichung bestimmter Anforderungen im Hinblick auf spezifische Qualitätsmerkmale oder -eigenschaften, wie die Erweiterbarkeit oder Performance eines Systems, beitragen [vgl. Starke, 2020, S. 310]. Qualitätsmodelle fokussieren sich auf bestimmte Aspekte eines Softwaresystems und bilden die Grundlage für Bewertungen und Analysen [vgl. Gharbi u. a., 2018, S. 147].

Generell gibt es unterschiedliche Modelle, worin Qualitätsmerkmale eines Systems definiert werden. Sie alle haben jedoch gemeinsam, dass sie sich auf zentrale Eigenschaften wie die Wartbarkeit, Zuverlässigkeit oder Effizienz eines Systems beziehen [vgl. Starke, 2020, S. 40 f.]. Im Rahmen dieser Arbeit werden deshalb die Qualitätsmerkmale gemäß DIN/25010 definiert [Starke, 2020, S. 41 ff.]. Dort werden die Merkmale in acht Hauptgruppen unterteilt:

- **Funktionale Eignung**
- **Zuverlässigkeit**
- **Effizienz (engl.: performance efficiency)**
- **Betreibbarkeit**
- **Sicherheit**
- **Kompatibilität**
- **Wartbarkeit**
- **Übertragbarkeit**

Die obigen Hauptgruppen unterscheiden zwischen funktionalen und nicht-funktionalen Anforderungen und lassen sich in weitere Teilmerkmale untergliedern. Dabei können die Eigenschaften in Wechselwirkung zueinander stehen und sich gegenseitig beeinflussen. So kann eine hohe Flexibilität des Systems aufgrund der steigenden Komplexität eventuell zu einer Verringerung der Testbarkeit und Betreibbarkeit führen [vgl. Gharbi u. a., 2018, S. 150]. Eine komplette Auflistung und Beschreibung der Qualitätsmerkmale gemäß DIN/25010 kann dem Anhang A.1 entnommen werden. An dieser Stelle ist zu erwähnen, dass es neben der Auflistung nach DIN/25010 noch weitere Qualitätsmerkmale gibt. So wird die **Skalierbarkeit** beispielsweise nicht explizit als Qualitätsmerkmal nach DIN/25010 genannt, kann aber dennoch bei der Bewertung von Softwarearchitekturen in Betracht gezogen werden [vgl. Gharbi u. a., 2018, S. 149].

In der Praxis werden obige Qualitätsmerkmale und Anforderungen eines Systems über sogenannte Szenarien weiter konkretisiert. Szenarien bilden Qualitätsmerkmale auf konkrete Sachverhalte ab. Angenommen man möchte ein Szenario für die Effizienz eines Systems formulieren, dann könnte dieses folgendermaßen lauten: Das System liefert unabhängig von der Systemlast konstant Antwortzeiten die schneller als eine Sekunde sind. Wie zu erkennen ist, stellt ein Szenario eine konkrete Anforderung an das System, die sich dabei auf ein Qualitätsmerkmal stützt. Das Formulieren von Szenarien ist unter anderem ein Teil der Architekturbewertungsmethodik „ATAM“ („Architecture Tradeoff Analysis Methode“), die jedoch nicht weiter vorgestellt wird, da sie für den Rahmen dieser Arbeit nicht von Relevanz ist [vgl Starke, 2020, S. 314].

Quantitative Bewertung

Neben den bereits vorgestellten qualitativen Merkmalen zur Bewertung einer Softwarearchitektur ist es auch möglich, Artefakte wie den Quellcode oder den Entwicklungsprozess einer Architektur basierend auf quantitativen Metriken zu messen. Die Analyse von Quellcode kann dabei helfen, Architekturabweichungen und Fehler innerhalb eines Systems zu identifizieren.

Klassische Metriken zur Qualitätssicherung von Architekturen betrachten eingehende und ausgehende Abhängigkeiten oder die zyklomatische Komplexität von Komponenten [vgl. Gharbi u. a., 2018, S. 152 f.]. Weitere Maße bezogen auf den Quellcode können auch die Anzahl der Methoden pro Klasse, die Änderungshäufigkeit von Codestellen oder der Grad der Testabdeckung innerhalb eines Pakets sein. Neben dem Quellcode können auch andere Aspekte des Systems nach quantitativen Methoden bewertet werden, wie beispielsweise die Performance des Systems oder die benötigte Zeit zur Umsetzung einer Anforderung [vgl. Starke, 2020, S. 317]. An dieser Stelle ist zu nennen, dass mittlerweile eine Reihe hilfreicher Werkzeuge zur quantitativen Analyse von Softwarearchitekturen existieren [vgl. Gharbi u. a., 2018, S. 153]. Ein Beispiel hierfür ist das von „Hello2Morrow“ entwickelte Tool namens „Sotograph“ [Hello2Morrow, 2021], dass sich gut zum Analysieren von Quellcode eignet.

2.2. Entwurfsmethoden zur Modellierung fachlicher Anforderungen

Wie in Kapitel 2.1 deutlich wurde, spielt im Rahmen von Softwarearchitekturen die Modularisierung, also die Aufteilung eines Systems in Komponenten, eine entscheidende Rolle. Dabei stellt sich die Frage auf welche Art und Weise und nach welchen Kriterien das System in Bausteine unterteilt werden soll. Da sich ein betriebliches System vor allem an den Fachprozessen orientiert, liegt eine fachliche Aufteilung nahe. Dieses Kapitel stellt deshalb nun zwei Entwurfsmethoden vor, die zur Identifizierung von Komponenten auf Basis fachlicher Anforderungen herangezogen werden können.

2.2.1. Domain-Driven Design als fachlich gesteuerte Entwurfsmethodik

Grundkonzepte des Domain-Driven Design

Domain-Driven Design (DDD) beschreibt ein Paradigma für den Entwurf komplexer Softwarearchitekturen. Die Methodik wurde erstmals 2004 von Eric Evans in seinem Buch „Domain-Driven Design: Tackling Complexity in the Heart of Software“ vorgestellt [vgl. Evans, 2004].

Software in Unternehmen dient überwiegend dazu fachliche Prozesse zu unterstützen. Die Fachlichkeit stellt damit einen essenziellen Aspekt der Software dar. Dennoch geht bei der Architektur und Implementierung häufig die Fachlichkeit verloren, da die technischen Modelle keine Rückschlüsse auf die Domäne liefern. Domain-Driven Design versucht dieses Problem zu lösen, indem es Techniken und Entwurfsmuster definiert um das Domänenwissen des Fachbereichs präzise in Software umsetzen zu können. Dabei wird ein einheitliches Modell über die Domäne entwickelt (Domänenmodell), das sowohl als Grundlage für den Fachbereich, als auch für die Architektur und Implementierung der Software dient [vgl. Wolff, 2019].

Die Domäne bezeichnet das Anwendungsgebiet der Software. Einige Domänen können dabei

sowohl Aspekte der realen Welt umfassen, als auch immaterielle Dinge abbilden. Die Domäne eines Reisebuchungssystems umfasst beispielsweise reale Personen, die reale Orte besuchen möchten, während die Domäne eines Buchhaltungssystems Geld und Finanzen beinhaltet [vgl. Evans, 2004, S. 2]. Da die gesamte Domäne eines Unternehmens meist sehr komplex ist, wird diese in der Praxis häufig mit einer Subdomäne gleichgesetzt. Eine Subdomäne stellt einen logischen Teil der Gesamtdomäne dar und dient dazu die Komplexität in überschaubare Teilbereiche zu unterteilen [vgl. Vernon, 2017, S. 44].

In gemeinsamer Zusammenarbeit zwischen den Fachexperten und Entwicklern wird durch einen kontinuierlichen Informationsaustausch das Domänenwissen abstrahiert in das Domänenmodell überführt. Dieser Prozess wird auch als „Knowledge Crunching“ bezeichnet [vgl. Evans, 2004, S. 3]. Eric Evans beschreibt in seinem Buch drei fundmentale Eigenschaften, die ein Domänenmodell hierbei erfüllen soll [Evans, 2004, S. 3 f.]:

- ***„The model and the heart of the design shape each other“***.
  Das Modell bildet die Grundlage für die Implementierung. Die Implementierung kann auf Basis des Modells verstanden werden und spiegelt die fachlichen Anforderungen wieder. Die Wartung und Weiterentwicklung der Software wird dadurch vereinfacht.
- ***„The model is the backbone of a language used by all team members“***.
  Sowohl die Domänenexperten als auch die Entwickler sprechen aufgrund eines einheitlichen Modells über die gleichen Konzepte. Es ist keine Übersetzung zwischen beiden Parteien notwendig, wodurch es seltener zu Missverständnissen kommt. Das Verwenden einer allgemeingültigen, universellen Sprache wird im Bereich des Domain-Driven Design auch als „Ubiquitous Language“ bezeichnet.
- ***„The model is distilled knowledge“***.
  Das Modell repräsentiert die in Übereinstimmung beider Parteien (Entwickler und Fachexperten) erarbeitete Sichtweise der Domäne. Dabei werden die wichtigsten Aspekte der Domäne extrahiert. Durch die enge Bindung des Modells mit der Implementierung können Erfahrungen der Entwickler als Feedback in den Modellierungsprozess einfließen.

Kern des Domain-Driven Designs ist die „Ubiquitous Language“. Dadurch dass auf allen technischen Ebenen — vom Code bis hin zum Datenbankschemata — die Fachterminologie verwendet wird, schafft die Implementierung ein besseres Abbild der Domäne. Dadurch lassen sich Missverständnisse zwischen Entwickler und Domänenexperten reduzieren [vgl. Wolff, 2019, S. 44].

Das Domänenmodell repräsentiert somit eine strukturierte Abstraktion des Domänenwissens. Es kann auf unterschiedliche Art und Weise dargestellt oder kommuniziert werden, wie beispielsweise durch Diagramme oder Dokumentationen. Diagramme unterstützen bei der kontinuierlichen Entwicklung des Modells und helfen bei der Kommunikation im Team. Das eigentliche Modell wird jedoch letztendlich durch den implementierten Code dargestellt [vgl. Vernon, 2017, S. 35; vgl. Evans, 2004, S. 3]. Domain-Driven Design unterteilt sich in zwei Bereiche. Dem Strategischen Design und dem Taktischen Design. Deren wesentlichen Aspekte werden nachfolgend vorgestellt.

Strategisches und Taktisches Design

Das **strategische Design** beschäftigt sich mit dem Architektur- und Modellentwurf, d.h. wie mehrere Domänenmodelle zusammenhängen und das System strukturiert ist. Ziel des Strategischen Designs ist es thematisch unabhängige Fachbereiche — auch „Bounded Context“

genannt — zu identifizieren und in Relation zueinander zu stellen („Context Mapping“). Ein Bounded Context setzt dabei die Grenzen des Gültigkeitsbereichs eines Domänenmodells und ist bestenfalls genau einer Subdomäne zugeordnet. Innerhalb der Kontexte gelten jeweils unterschiedliche Terminologien („Ubiquitous Language“). Eine Bestellung im Kontext der Buchhaltung umfasst beispielsweise Daten zu den Preisen und Steuersätzen, während für den Versand die Größen und Gewichte der Artikel von Relevanz sind [vgl. Starke, 2020, S. 85; vgl. Wolff, 2019, S. 45]

Das **taktische Design** definiert grundlegende Bausteine und Regeln zur Betrachtung innerhalb eines Bounded Context. Mithilfe dieser Konzepte kann eine Domäne auf feingranularer Ebene modelliert und implementiert werden [vgl. Starke, 2020, S. 86]. Neben den von Evans ursprünglichen beschriebenen Bausteinen zur Modellierung der Fachdomäne, wie beispielsweise „Entities“, „Value Objects“ oder „Aggregates“, sind im Laufe der Zeit weitere Konzepte zum Taktischen Design hinzugekommen. Durch „Domain Events“ werden fachliche Ereignisse die innerhalb einer Domäne auftreten modelliert. Die fachlichen Ereignisse implizieren, dass geschäftsrelevante Aktionen in der Domäne passiert sind und sich damit möglicherweise Zustände verändert haben [vgl. Vernon, 2017, S. 97; vgl. Evans, 2015, S. 13]. Domain Events bilden die Grundlage für „Event Sourcing“, worin alle Aktivitäten und Zustände eines Systems durch Domänenereignisse dargestellt werden [vgl. Fowler, 2005]. Einsatz findet dieses Verfahren beispielsweise bei Netflix, um das Herunterladen von Videomaterial zur Offline-Wiedergabe zu ermöglichen [vgl. Casella u. a., 2017].

Workshop-Methoden

Für den „Knowledge-Crunching“-Prozess — also dem Wissensaustausch zwischen Entwickler und Fachexperten zum Erarbeiten eines gemeinsamen Domänenmodells — haben sich in den letzten Jahren zwei beliebte Workshop-Methoden etabliert: Das „Event Storming“ und „Domain Story Telling“.

„**Event Storming**“ ist ein von Alberto Brandolini [Brandolini, 2021] entwickeltes Format, bei dem eine Domäne über das zeitliche Anordnen von Domain Events per Klebezettel modelliert wird. Vorteil diese Methode ist, dass sie schnell von allen Beteiligten verstanden werden kann und nicht das Erlernen einer Modellierungssprache wie der „Unified Modeling Language“ (UML) oder BPMN erfordert [vgl. Vernon, 2017, S. 110 ff.]. Beim „**Domain Story Telling**“ auf der anderen Seite, erzählen — wie der Name schon andeutet — die Domänenexperten ihre Prozessabläufe und Tätigkeiten anhand einer Beispielgeschichte. Die Entwickler zeichnen diese parallel über Piktogramme auf. Schlüsselelemente dieser Piktogramme sind „Actors“ (die beteiligten Personen oder Systeme), „Work objects“ (die relevanten Informationsobjekte oder Daten) und „Activities“ (die Arbeitsschritte bzw. Aktivitäten des Actors). Vor allem aufgrund der konkreten Beispielszenarien ist es einfach möglich das Domänenwissen zu erfassen [vgl. Hofer, 2021]. Für tiefergehende Informationen und den genauen Vorgehensweisen beider Workshop-Formate ist auf [Brandolini, 2021] und [Hofer, 2021] verwiesen.

2.2.2. Das Konzept der Architektur integrierter Informationssysteme (ARIS)

ARIS („Architektur integrierter Informationssysteme“) ist ein von August-Wilhelm Scheer entwickeltes Konzept der Wirtschaftsinformatik zur ganzheitlichen Modellierung von Informationssystemen. Der Architekturansatz beschreibt dabei ein Top-Down Vorgehen, das ausgehend von der Unternehmensstrategie und den betrieblichen Geschäftsprozessen ein Fachkonzept entwickelt, welches detailliert genug ist um als Ausgangsbasis für die Implementierung zu dienen.

Die fachlichen Geschäftsprozesse stehen hierbei im Mittelpunkt, da diese den Grundriss des Informationssystems skizzieren [vgl. Scheer, 2002b, S. 1 f.].

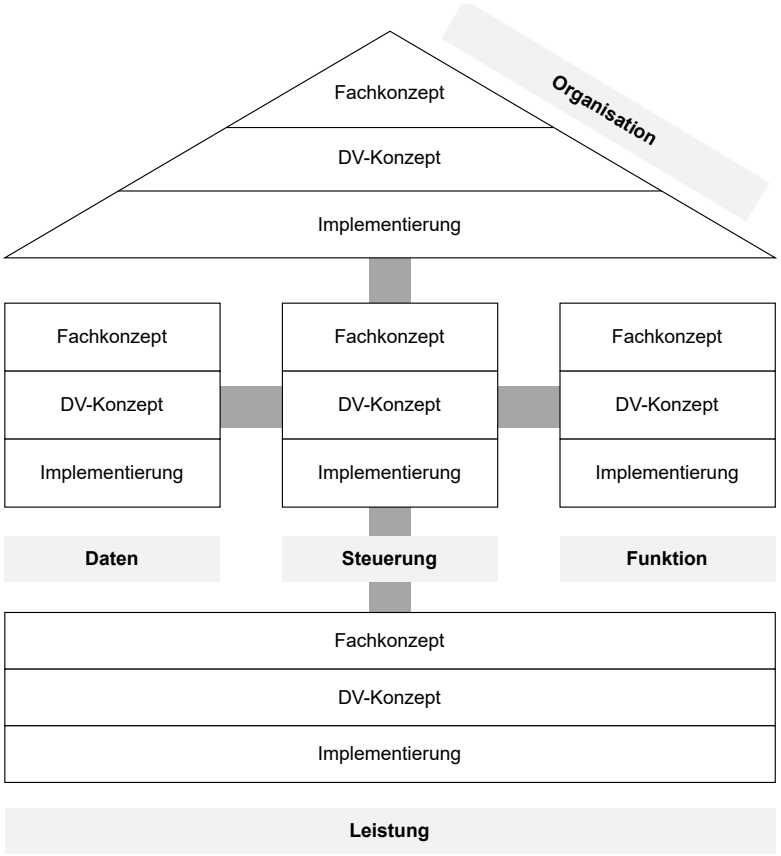

Abbildung 2.2.: Das ARIS-Haus [Scheer, 2002b, S. 41]

Um die Komplexität der Geschäftsprozesse zu erfassen, unterstützt das ARIS-Konzept dabei die unterschiedlichen Beschreibungsaspekte eines Prozesses in mehrere Sichten einzuteilen und ihnen Methoden zuzuordnen [vgl. Scheer, 2002b, S. 2]. Scheer unterscheidet in diesem Kontext zwischen folgenden fünf Sichten, die das sogenannte ARIS-Haus (vgl. Abbildung 2.2) bilden [vgl. Scheer, 2002b, S. 36 f.; vgl. Abts und Mülder, 2017, S. 565 f.]:

- **Organisationssicht**: Definiert die Organisationseinheiten und Verantwortlichkeiten des Unternehmens (menschliche und maschinelle Aufgabenträger) und deren Beziehungen zueinander. Im Rahmen eines Reisebuchungsprozesses liegen die Verantwortlichkeiten beispielsweise bei den Mitarbeitern des Reiseunternehmen und gegebenenfalls zusätzlichen maschinellen Diensten zur Buchungsabwicklung.
- **Funktionssicht**: Beschreibt alle Vorgänge — oder auch Funktionen, Tätigkeiten, Aufgaben genannt — die zur Leistungserbringung dienen. Ein Beispiel für einen Vorgang ist die Reiseprüfung.
- **Datensicht**: Umfasst alle vorgangsbezogenen Daten, sowie alle Ereignisse und Nachrichten die einen Vorgang auslösen. Der Vorgang „Reiseprüfung“ beinhaltet beispielsweise Kunden- und Reisedaten.
- **Leistungssicht**: Alle materiellen und immateriellen Input- und Output-Leistungen. Hierzu zählen auch jegliche Geldflüsse des Unternehmens. Der Kunde gibt beispielsweise sein

Reiseziel an und ist dafür bereit eine bestimmten Betrag zu zahlen. Die Leistungserbringung des Reiseunternehmens besteht dann darin, ein passendes Reiseangebot vorzuschlagen und dessen Buchung abzuwickeln.

- **Steuerungssicht (Prozesssicht)**: Die Steuerungssicht, auch Prozesssicht genannt, steht im Mittelpunkt und stellt alle vorherigen Sichten in Beziehung zueinander, indem es den Geschäftsprozess und dessen zeitlichen Ablaufplan in seiner Gesamtheit abbildet.

Durch entsprechende Diagrammtypen können die einzelnen Sichten modelliert werden. Beispielsweise eignet sich ein Entity-Relationsship-Modell zur Beschreibung der Datenstrukturen, die ereignisgesteuerte Prozesskette oder BPMN (Business Process Model and Notation) von der „Object Management Group“ [The Object Management Group, 2021b] zur Prozessabbildung und Organigramme um die Organisationseinheiten des Unternehmens zu dokumentieren [vgl. Scheer, 2002a, S. 17]. Die ARIS-Sichten betrachten einen Geschäftsprozess aus eher betriebswirtschaftlichen Gesichtspunkten. Um einen engeren Bezug zur IT herzustellen, definiert ARIS zusätzlich ein Phasenmodell. Das Phasenmodell dient dazu, ausgehend vom Fachkonzept, pro ARIS-Sicht die fachlichen Anforderungen schrittweise in informationstechnische Modelle zu übersetzen. So folgt nach dem Fachkonzept die Erstellung des DV-Konzepts (Datenverarbeitungs-Konzept) und im letzten Schritt die technische Implementierung (vgl. Abbildung 2.2) [vgl. Scheer, 2002b, S. 38 ff.]. Detailliertere Informationen bezüglich dem ARIS-Konzept sind [Scheer, 2002b] zu entnehmen.

2.3. Kommunikationsmechanismen verteilter Systeme

Um eine Gesamtfunktionalität zu erbringen, ist üblicherweise die Kommunikation zwischen mehreren Systemkomponenten notwendig. Vor allem innerhalb verteilter Systeme — wie es beispielsweise bei Microservice-basierten Architekturen (vgl. Kapitel 2.4.2) der Fall ist — spielt das Zusammenspiel und die Kooperation der einzelnen Services eine wesentliche Rolle. Dieses Kapitel gibt eine kurze Einführung in synchrone und asynchrone Kommunikationsmechanismen. Fokus wird dabei auf die Nachrichten-basierte Kommunikation gelegt, welche für die später vorgestellten Architekturen von Relevanz ist.

2.3.1. Synchrone Kommunikation

Synchrone Kommunikation zwischen zwei Systemkomponenten ist dadurch gekennzeichnet, dass der sendende Prozess bis zum Empfang des Ergebnisses blockiert ist, d.h. in einem Wartezustand verbleibt. Typischerweise findet eine solche Art der Kommunikation im Rahmen lokaler Funktionsaufrufe statt. Innerhalb verteilter Systeme kann beispielsweise direkt über HTTP-Schnittstellenaufrufe oder Remote Procedure Calls (RPC) synchron miteinander kommuniziert werden. Da der Sender erst nach Rückmeldung des Empfängers weiterarbeiten kann werden bei synchroner Kommunikation die Ressourcen nicht optimal genutzt und es entsteht eine enge Kopplung zwischen beiden Parteien [vgl. Starke, 2020, S. 247 ff.].

2.3.2. Asynchrone Kommunikation

Im Vergleich zum synchronen Austausch wartet bei der asynchronen Kommunikation der Sender nicht auf eine Rückantwort. Die Nachrichtenübertragung passiert hier nach dem Motto „versenden und vergessen“ (engl.: „fire and forget“) [Starke, 2020, S. 247]. Dies ermöglicht es dem Sender direkt nach Versand der Nachricht mit dem Programmablauf fortzufahren, wo-

durch keine Ressourcen unnötig blockiert werden.

Eine solche Nachrichten-basierte Kommunikation wird auch als „Messaging“ bezeichnet [vgl. Hohpe und Woolf, 2015, S. xxx f.]. Die Sender und Empfänger kommunizieren dabei über sogenannte „Message Channels“ miteinander, die durch ein „Messaging-System“ — auch „Message-oriented Middleware“ (MOM) genannt — bereitgestellt und verwaltet werden [vgl. Hohpe und Woolf, 2015, S. xxxi; vgl. Hohpe und Woolf, 2015, S. 60 f.]. Das Messaging-System sorgt dabei für eine lose Kopplung zwischen Sender und Empfänger, indem es den Nachrichtenaustausch koordiniert und durch Zwischenspeicherung der Nachrichten eine sichere Zustellung garantiert. Das ist besonders bei verteilten Anwendungen wichtig, wo es oft zu Systemausfällen kommen kann [vgl. Hohpe und Woolf, 2015, S. 57].

Ein Message Channel kann nach zwei Prinzipien arbeiten: „Point-to-Point“ und „Publish-Subscribe“ [vgl. Hohpe und Woolf, 2015, S. 63]. Bei einem Point-to-Point-Channel — auch Queue genannt — kann die Nachricht eines Senders immer nur von einem Empfänger konsumiert werden. Es findet also eine 1:1-Kommunikation statt [vgl. Hohpe und Woolf, 2015, S. 103 f.]. Anders sieht es bei Publish-Subscribe aus. Dort können sich beliebig viele Empfänger auf eine Nachricht subskribieren und eine Kopie dieser erhalten. Es besteht daher eine 1:n-Beziehung zwischen Sender und Empfänger [vgl. Hohpe und Woolf, 2015, S. 106 f.]. Publish-Subscribe Verfahren werden über „Topics“ realisiert. Welche der beiden Arten eines Nachrichtenkanals besser geeignet ist, hängt von dem jeweiligen Anwendungsfall ab. Im Rahmen ereignisgesteuerter Architekturen (vgl. Kapitel 2.4.3) kommt beispielsweise gerne das Publish-Subscribe Verfahren zum Einsatz, um flexibel auf Events reagieren zu können. Bei einem Backend-Aufruf im Kontext prozessgesteuerter Architekturen (vgl. Kapitel 2.4.5) mag ein Point-to-Point-Channel ausreichend sein.

Eine beliebte standardisierte API für das Messaging in der Java-Welt ist der „Java Message Service“ (JMS) [Qusay H., 2004]. Implementiert wird dieser beispielsweise durch den Open-Source Message Broker „ActiveMQ“ von der Apache Software Foundation [Apache Software Foundation, 2021c]. Neben JMS existieren noch weitere Technologien für das Messaging, wie beispielsweise Apache Kafka [Apache Software Foundation, 2021b], eine ursprünglich von LinkedIn entwickelte Plattform zur Verarbeitung von Datenströmen. Apache Kafka wird auch oft als Event Broker bezeichnet, der eine Erweiterung des klassischen Message Brokers darstellt. Kafka verwaltet ein Handbuch in dem alle Einträge, die als Records bezeichnet werden, jederzeit über Indexe erreichbar sind. Event Broker unterscheiden sich dadurch von klassischen Message Broker, da bei solchen die Nachrichten nach Zustellung üblicherweise gelöscht werden. Erst durch Einsatz eines Event Brokers und damit der persistenten Zustandshaltung werden einige moderne Varianten ereignisgesteuerter Architekturen und Patterns möglich [vgl. Bellemare, 2020, S. 31 f.]. Kafka eignet sich außerdem aufgrund seiner Cluster-Technologie gut für große Enterprise-Anwendungen, die mit einem hohen Datendurchsatz einhergehen. Durch die Replikation der Daten innerhalb des Clusters, kann eine sichere Zustellung der Nachrichten auch bei Ausfall eines Knotens garantiert werden [vgl. Narkhede u. a., 2017, S. XI f.].

2.4. Softwarearchitekturstile

Die Softwarearchitektur Definition aus Kapitel 2.1.1 hat untermauert, dass Architekturen bestimmten Prinzipien unterliegen. Einige grundlegende Prinzipien — mit Fokus auf die Modularität — wurden daraufhin in Kapitel 2.1.2 vorgestellt. Über die letzten Jahre haben sich

Softwarearchitekturmuster gebildet, die bei der inhärenten Umsetzung dieser Prinzipien unterstützen. Deshalb werden in diesem Kapitel zunächst grundlegende Architekturen, wie die Aufteilung in Schichten oder nach Microservices behandelt. Daraufhin findet eine Einführung in ereignisgesteuerte, Process-Engine basierte und prozessgesteuerte Architekturen statt. Diese stellen die Grundlage für die späteren Implementierungen des Reisebuchungsprozesses und des Architekturvergleichs dar.

2.4.1. Schichtenarchitekturen

Schichtenarchitekturen untergliedern die Komponenten eines Systems in mehrere, größere Einheiten — auch Schichten genannt [vgl. Bass u. a., 2013, S. 206]. Jede Schicht ist innerhalb des Systems für eine bestimmte Zuständigkeit verantwortlich und kann unabhängig von anderen Schichten gewartet und weiterentwickelt werden. Schichtenarchitekturen verfolgen damit strikt das „Prinzip der Trennung von Zuständigkeiten“ (vgl. Kapitel 2.1.2). Die Beziehungen zwischen den einzelnen Schichten sind durch Regeln beschränkt. Bei einer strengen Schichtung dürfen Schichten nur mit der nächsten untergeordneten Schicht interagieren, während bei einer schwachen Schichtung mit allen untergeordneten Schichten kommuniziert werden kann. Eine Verwendung von übergeordneten Schichten ist innerhalb einer Schicht nicht zulässig, um zyklische Abhängigkeiten zu vermeiden. Damit eine Kommunikation zwischen den einzelnen Schichten möglich ist, stellt jede Schicht eine Schnittstelle bereit. Änderungen innerhalb einer Schicht, die nicht die Schnittstelle betreffen, wirken sich so nicht auf die Außenwelt aus und sorgen für eine lose Kopplung [vgl. Bass u. a., 2013, S. 205 f.].

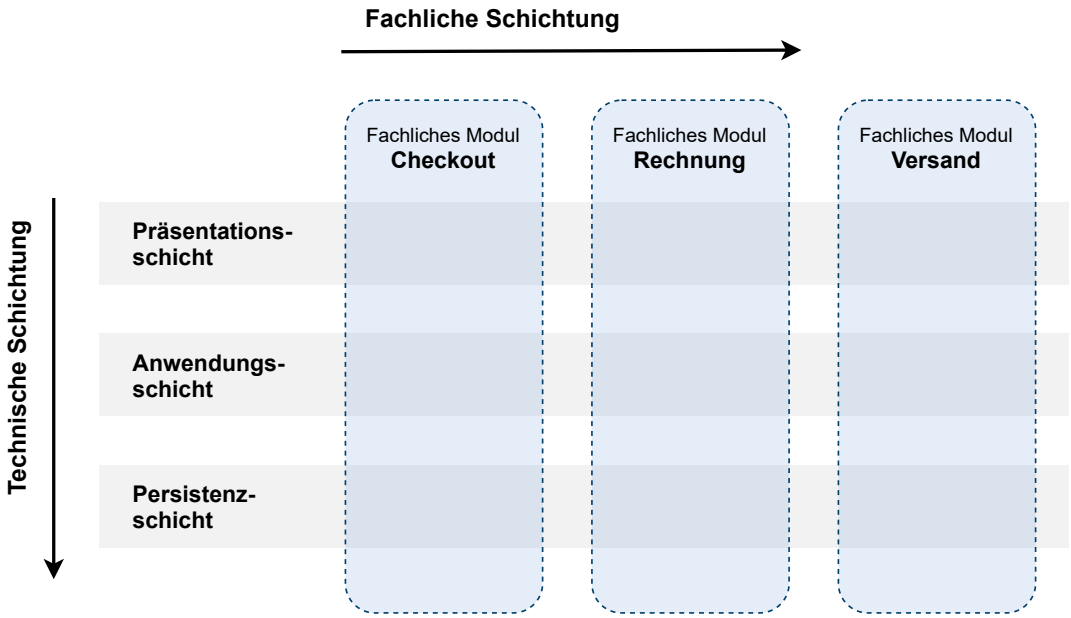

Abbildung 2.3.: Technische und Fachliche Schichtung [In Anlehnung an Lilienthal 2020, S. 99]

Schichtenarchitekturen können nach technischen und fachlichen Aspekten partitioniert sein (vgl. Abbildung 2.3), weshalb im folgenden beide Dimensionen kurz erklärt werden [vgl. Lilienthal, 2020, S. 97].

Technische Schichtung

Eine technische Schichtung teilt die Komponenten eines Systems basierend ihrer technischen Zuständigkeit einer Schicht zu. Dadurch werden die einzelnen technischen Aufgaben eines Systems hierarchisiert und voneinander entkoppelt [vgl. Lilienthal, 2020, S. 97]. Eine typische technische Schichtung für Softwaresysteme ist die sogenannte 3-Schichtenarchitektur die eine Aufteilung nach Präsentations-, Anwendungs- und Persistenzschicht vornimmt (siehe Abbildung 2.3). Die Präsentationsschicht hat als Aufgabe die Benutzeroberfläche bereitzustellen, während die Anwendungsschicht jegliche Geschäftslogik abbildet. Die Persistenzschicht kümmert sich um die Speicherung und Verwaltung der fachlichen Daten in einer Datenbank. Aufgrund der losen Kopplung der einzelnen Schichten, können diese leicht auf verschiedene Rechner verteilt werden, um eine bessere Skalierbarkeit zu erreichen [vgl. Dunkel u. a., 2008, S. 41 f.]. Ein bekanntes Entwurfsmuster das einer technischen Schichtung folgt ist das sogenannte „MVC-Pattern“ („model view controller pattern“), worin eine Anwendung in drei Komponenten unterteilt wird: Der Benutzeroberfläche (view), der Anwendungs- und Geschäftslogik (model) und einer Komponente zur Steuerung der Interaktion zwischen den anderen Komponenten (controller) [vgl. Balzert, 2011, S. 62 f.].

Fachliche Schichtung

Bei großen, komplexen Softwaresystemen reicht eine technische Schichtung alleine oft nicht aus, um ein System modular zu strukturieren. Aus diesem Grund werden Systeme zusätzlich nach fachlichen Modulen bzw. Geschäftsfeldern unterteilt. Ähnlich wie bei der technischen Schichtung kann hier als Regel definiert werden, dass zwischen den fachlichen Modulen eine Beziehung in nur eine Richtung erlaubt ist — um zyklische Abhängigkeiten zu vermeiden [vgl. Lilienthal, 2020, S. 98 ff.]. Abbildung 2.3 zeigt, dass eine fachliche Schichtung das System in vertikale Module unterteilt. Diese stellen die Geschäftsbereiche dar. Jedes fachliche Modul kann dabei intern wiederum eine technische Schichtung aufweisen [vgl. Lilienthal, 2020, S. 99]. Domain-Driven Design (vgl. Kapitel 2.2.1) definiert beispielsweise mit dem strategischen Design eine Methodik um solche fachlichen Module innerhalb einer Domäne zu identifizieren. In der Terminologie von DDD werden diese als „Bounded Context“ bezeichnet. In diesem Zuge spielt die fachliche Schichtung auch bei der Zerlegung eines Systems in Microservices eine essenzielle Rolle, was in Kapitel 2.4.2 genauer thematisiert wird.

2.4.2. Microservices

„Microservices“ bezeichnen einen Architekturstil, bei dem ein Gesamtsystem in mehrere kleine, autonome Services unterteilt ist, die miteinander über leichtgewichtige Schnittstellen verteilt kommunizieren. Die Autonomie der einzelnen Services ist dadurch gekennzeichnet, dass diese in eigenständigen Betriebssystemprozessen laufen und unabhängig voneinander entwickelt, deployed und skaliert werden können. Ein Service wird in diesem Kontext aufgrund seiner kompakten Größe auch als „Microservice“ bezeichnet. Die Kommunikation zwischen den Microservices geschieht häufig über REST-Schnittstellen (vgl. Kapitel 2.5.3) die per HTTP-Protokoll aufgerufen werden. Die Kommunikation kann dabei synchron oder asynchron erfolgen (vgl. Kapitel 2.3). Da die Services getrennt voneinander ablaufen — d.h. jeder in seinem eigenen Betriebssystemprozess und gegebenenfalls auf unterschiedlichen physischen Rechnern — handelt es sich bei Microservice-basierten Architekturen um verteilte Architekturen die mit den Fallstricken eines verteilten Systems einhergehen [vgl. Fowler und Lewis, 2014; vgl. Newman, 2015, S. 22 ff.].

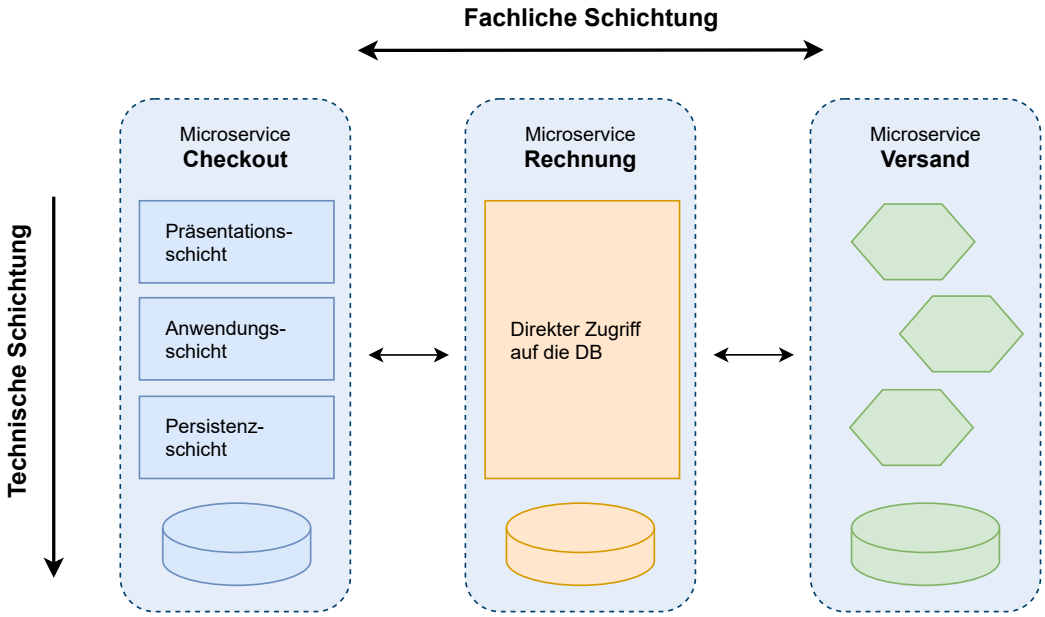

Abbildung 2.4.: Unterschiedliche Microservice Implementierungen [In Anlehnung an Lilienthal 2020, S. 106]

Die Zerlegung eines Systems in Microservices folgt meist einer fachlichen Schichtung, wie sie bereits in Kapitel 2.4.1 über Schichtenarchitekturen behandelt wurde und in Abbildung 2.4 zu sehen ist. Vor allem die Kombination aus Microservices und dem in Kapitel 2.2.1 behandelten Domain-Driven Design erscheint sinnvoll. So kann die fachliche Struktur einer Domäne auf das Softwaresystem übertragen werden, indem ein Microservice jeweils einen Bounded Context abbildet. Dadurch entwickelt sich eine Architektur, dessen Modularität rein aus der Fachlichkeit heraus entsteht [vgl. Lilienthal, 2020, S. 105 f.]. Das Erbringen eines fachlichen Prozesses erfordert oft das Zusammenspiel mehrerer Microservices aus unterschiedlichen Bounded Contexts. Die Pfeile zwischen den Microserivces in Abbildung 2.4 signalisiert die Kollaboration zwischen den Services. Möchte man beispielsweise einen Bestellprozess mit den Microservices aus Abbildung 2.4 implementieren, ist eine sinnvolle Koordination der einzelnen Dienste notwendig. Dies kann über **Orchestrierung** oder **Choreografie** der Services geschehen — auch als „Orchestration Pattern“ und „Choreography Pattern“ bezeichnet [vgl. Bellemare, 2020, S. 135 ff.].

Bei der **Orchestrierung** existiert eine zentrale Stelle, die den Prozessablauf steuert. Die Steuerung kann dabei gegebenenfalls von einer Process Engine (vgl. Kapitel 2.4.4 und 2.4.5) erbracht werden. In dem Beispiel aus Abbildung 2.4 könnte die Steuerungslogik in den Checkout verlagert werden. Dieser wäre dann für die Abwicklung des Bestellprozesses zuständig und würde beispielsweise über synchrone Anfrage/Antwort-Aufrufe mit dem Rechnungs- und Versand-Service kommunizieren. Nachteil eines solchen Stiles ist, dass der Checkout-Service aufgrund seiner zentralen Rolle zum Kernstück des Systems wird. Es entsteht eine hohe Abhängigkeit die dazu führt, dass die anderen Services in ihrer Entscheidungsfreiheit eingeschränkt werden. Anders geregelt ist das bei einem **Choreografie**-basierten Ansatz. Hier gibt es keinen übergeordneten Koordinator, der den Prozessfluss steuert. Stattdessen agiert jeder Service unabhängig voneinander. Typisch ist diese Art der Koordination für klassische ereignisgesteuerte Architekturen. Dort kommunizieren die Services asynchron über Events miteinander, um einen

übergreifenden Prozess zu erbringen. Der Checkout-Service würde in diesem Fall ein Event „BestellungEingegangen“ senden, auf das beliebig viele weitere Services reagieren und Folgeaktivitäten anstoßen. Bekannte Best Practices, die zur Entwicklung von Microservices herangezogen werden können, sind in der sogenannten „Zwölf-Faktoren-App“ beschrieben [vgl. Wiggins, 2017]. Die Konzepte dieser Methode unterstützen dabei, Microservices flexibel und skalierbar zu gestalten.

2.4.3. Ereignisgesteuerte Architekturen

Motiviert sind ereignisgesteuerte Architekturen durch das „Reaktive Manifest“ [Bonér u. a., 2014], worin die Kernqualitäten reaktiver Systeme beschrieben sind. Die ereignisgesteuerte Architektur — zu englisch „Event-Driven Architecture“ (EDA) — ist dadurch gekennzeichnet, dass die Interaktion der Systemkomponenten durch den asynchronen Austausch von Ereignissen stattfindet [vgl. Bruns und Dunkel, 2010, S. 50]. Ein Ereignis (Event) stellt häufig eine Zustandsveränderung eines Objektes dar und kann unterschiedliche Abstraktionsebenen repräsentieren. Technische Ereignisse, wie die Temperaturveränderung einer Maschine, haben einen niedrigen Abstraktionsgrad, während Geschäftsereignisse wie die Stornierung einer Bestellung durch das widerspiegeln fachlicher Geschehnisse eine höhere Abstraktion aufweisen [vgl. Bruns und Dunkel, 2010, S. 48].

Für diese Arbeit sind vor allem Geschäftsereignisse von Relevanz, da sie den fachlichen Prozess beschreiben. In Kapitel 2.2.1 wurde bereits in Zusammenhang mit dem Domain-Driven Design über sogenannte Domain Events gesprochen und wie diese mithilfe des Event Storming innerhalb einer Domäne identifiziert werden können. Domain Events und Geschäftsereignisse werden im weiteren Verlauf der Arbeit als synonyme Begriffe verwendet, da beide letztlich fachliche Sachverhalte widerspiegeln. Außerdem ist zu erwähnen, dass in dieser Arbeit unter EDA das Austauschen einzelner fachlicher Ereignisse zwischen Systemkomponenten gemeint ist, um damit einen Prozessfluss innerhalb eines Systems abzubilden. Dabei ist das „Complex Event Processing“ (CEP) nicht mit inbegriffen. Bei CEP steht das Verarbeiten einer Vielzahl von Ereignissen im Mittelpunkt, die meist innerhalb eines kurzen Zeitraums auftreten. Ziel ist es dann Muster und Zusammenhänge aus der Ereignisfolge zu erkennen, die von Relevanz für die eigentliche Geschäftsanwendung sind [vgl. Hedtstück, 2020, S. VII].

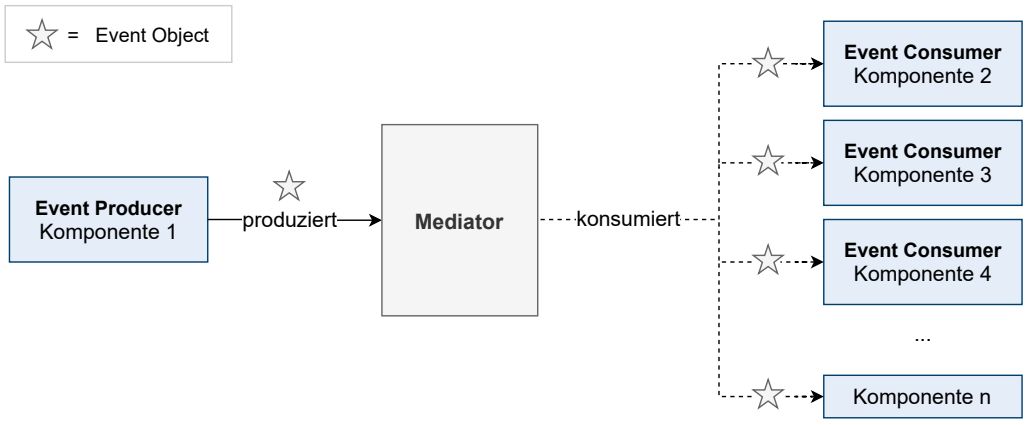

Abbildung 2.5.: Verarbeitungsmodell eines ereignisgesteuerten Systems [In Anlehnung an Bruns und Dunkel 2010, S. 52]

Abbildung 2.5 visualisiert das Verarbeitungsmodell ereignisgesteuerter Systeme. Dieses besteht nach Bruns und Dunkel [vgl. Bruns und Dunkel, 2010, S. 51 f.] aus drei Elementen:

- **Event Producer** (Ereignisquelle): Der Event Producer repräsentiert die Ereignisquelle. Er erstellt das Ereignisobjekt und sendet dieses als Nachricht an einen Mediator weiter, der das Ereignis dann ohne weitere Interaktion mit dem Event Producer weiterleitet.
- **Event Consumer** (Ereignissenke): Event Consumer konsumieren Ereignisse. Trifft ein neues Ereignis bei einem Event Consumer ein, kann es einen Verarbeitungsprozess auslösen.
- **Event Object**: Das Ereignisobjekt enthält keine Informationen über die Verarbeitungsschritte, die bei Eintritt des Ereignisses von den Event Consumern ausgeführt werden. Es dient lediglich als Vermittler von Daten und Objekten zur Darstellung der Ereignisse.

Die Ereignisobjekte werden zwischen Event Producer und Event Consumer über asynchrone Kommunikation ausgetauscht. Der Event Producer weiß also nicht, was bei Eintritt eines Ereignisses im System geschieht. Der Mediator dient als Vermittler zwischen beiden Parteien und sorgt für einen zuverlässigen Transport des Ereignisses. Dieser kann durch ein Messaging-System oder einen Event-Broker repräsentiert werden. Üblicherweise wird dabei das „Publish/Subscribe"-Verfahren verwendet, dass bereits in Kapitel 2.3 behandelt wurde. Das Publish/Subscribe-Verfahren bietet mehr Flexibilität in Hinblick auf die Ereignisverarbeitung. So können beliebig viele Event Consumer ein Ereignis abonnieren und entsprechend darauf reagieren, wie im rechten Teil der Abbildung 2.5 zu erkennen ist [vgl. Bruns und Dunkel, 2010, S. 53 f.].

2.4.4. Process Engine basierte Architekturen

Process Engine basierte Architekturen beschreiben einen Architekturstil, bei dem Process Engines als Unterstützung zur Implementierung von fachlichen und technischen Prozessflüssen eingesetzt werden. Die Architektur wird meist in Kombination mit Microservices umgesetzt. Geprägt ist dieser Architekturstil durch Bernd Rücker, einem der Mitbegründer des BPM-Plattform-Herstellers „Camunda".

Eine Process Engine — auch Workflow Engine genannt — wird innerhalb dieser Architektur als zentrale Komponente eingesetzt. Sie steuert den Kontrollfluss und ist auf folgende technische Kernfähigkeiten angewiesen [Ruecker, 2021, S. 23 f.]:

- „**Durable state (persistence)**": Mit Durable state ist gemeint, dass alle Prozessdefinitionen und Prozessinstanzen inklusive deren Zustände historisiert in einer Datenbank abspeichert werden. Diese Fähigkeit ist relevant, damit ein Überblick über alle laufenden und vergangenen Prozessflüsse erhalten werden kann. Zusätzlich ist ein Transaktionsmanagement notwendig, sodass ein gleichzeitiger Zugriff auf einen Prozess möglich ist.
- „**Scheduling**": Scheduling bezeichnet das Steuern zeitlicher Abläufe. Eine Process Engine ist darauf angewiesen, um Wartezustände abzubilden und im Falle einer Tätigkeit aktiv werden zu können.
- „**Versioning**": Bezeichnet, dass Prozesse und dessen Versionen unveränderlich identifizierbar sein müssen. Vor allem bei langlaufenden Prozessen ist es selten, dass ein Zeitpunkt existiert zu dem keine Instanz aktiv ist. Damit bei einer Änderung nicht alle laufenden Prozessinstanzen beeinträchtigt oder unterbrochen werden, sollte eine Process

Engine die Versionierung von Prozessen unterstützten. Im besten Fall werden zusätzlich Möglichkeiten zur Migration bestehender Prozesse auf neuere Versionen angeboten.

Eine Workflow Engine arbeitet auf Grundlage einer Prozessdefinition, worin der Ablauf und die einzelnen Arbeitsschritte (Aktivitäten) des Prozesses definiert sind. Eine konkrete Ausführung einer Prozessdefinition durch die Process Engine wird auch als Prozessinstanz bezeichnet [Ruecker, 2021, S. 7]. Zur Modellierung der Prozesse kann dabei BPMN (Business Process Model and Notation) als Standard eingesetzt werden, der ab Version 2.0 die direkte Ausführbarkeit von BPMN-Modellen auf Process Engines ermöglicht und von der Object Management Group (OMG) verwaltet und weiterentwickelt wird [vgl. The Object Management Group, 2021b; vgl. Freund und Rücker, 2019, S. 7 f.].

Wie oben schon erwähnt steuert eine Process Engine mehrere Aktivitäten, die durch ein Prozessmodell definiert sind. Rücker erweitert dieses Verständnis in seinem Buch über Prozessautomatisierung mit der Aussage: *„[...] workflow engines can orchestrate anything [...]“* [Ruecker, 2021, S. 67]. So kann eine Workflow Engine unter anderem Softwarekomponenten, Entscheidungen, manuelle Tätigkeiten und physikalische Geräte innerhalb eines Prozessmodells orchestrieren [vgl. Ruecker, 2021, S. 92].

Ein klarer Fokus wird dabei auf die in Kapitel 2.4.2 behandelten Microservices in Kombination mit dem Domain-Driven Design (Kapitel 2.2.1) gesetzt. Um einen ganzheitlichen Geschäftsprozess abbilden zu können, müssen typischerweise mehrere Microservices miteinander kommunizieren. Rücker knüpft an diesem Punkt an und verwendet Workflow Engines als Orchestrierungstool, um das Zusammenspiel solcher Services zu koordinieren. Dabei existiert üblicherweise nicht eine zentrale Workflow Engine, die den Ablauf steuert. Microservices können dezentral eine Process Engine eingebettet haben und damit die für dessen Bounded Context gültigen Prozesse automatisieren. Die Workflow Engine ist damit nach außen nicht sichtbar und der Ende-zu-Ende Geschäftsprozess über mehrere Microservices verteilt. Die Autonomie und Isolation eines Microservice bleibt dadurch erhalten [vgl. Ruecker, 2021, S. 70; vgl. Ruecker, 2021, S. 117 f.; vgl. Ruecker, 2021, S. 134].

Abbildung 2.6 visualisiert den Process Engine basierten Architekturstil. Der Customer Onboarding Microservice ist für den Onboarding Prozess verantwortlich und implementiert diesen über eine eingebettete Process Engine. Zur Abwicklung des Prozesses kommuniziert die Workflow Engine des Customer Onboarding Microservice direkt mit den API-Schnittstellen weiterer Systeme oder Services, wie beispielsweise dem CRM-System. Die Kommunikation kann dabei sowohl synchron als auch asynchron stattfinden. Die Prozesse der weiteren Systeme können intern wiederum über eine eingebettete Workflow Engine automatisiert sein.

Neben der Orchestrierung von Softwarekomponenten kann eine Workflow Engine innerhalb einer Process Engine basierten Architektur auch Integrationsaufgaben übernehmen. Rücker unterscheidet beispielsweise nicht zwischen Geschäfts- und Integrationsprozessen, sondern sieht diese eher als Komplement zueinander an [vgl. Ruecker, 2021, S. 10 f.]. Typische Aufgaben, die ein BPMN-Prozess im Rahmen der Integration übernehmen kann, umfassen Resilienz Muster zur Kommunikation mit entfernten Systemen, wie beispielsweise Retry-Verfahren oder Strategien zur Behandlung von Zeitüberschreitungen bei der Anfrage. Auch die Implementierung der „Enterprise Integration Patterns“ [vgl. Hohpe und Woolf, 2015], wie dem „Aggregator Pattern“ zum Sammeln von Nachrichten, sind Anwendungsfälle für Integrationsprozesse [vgl. Ruecker, 2021, S. 173 ff.].

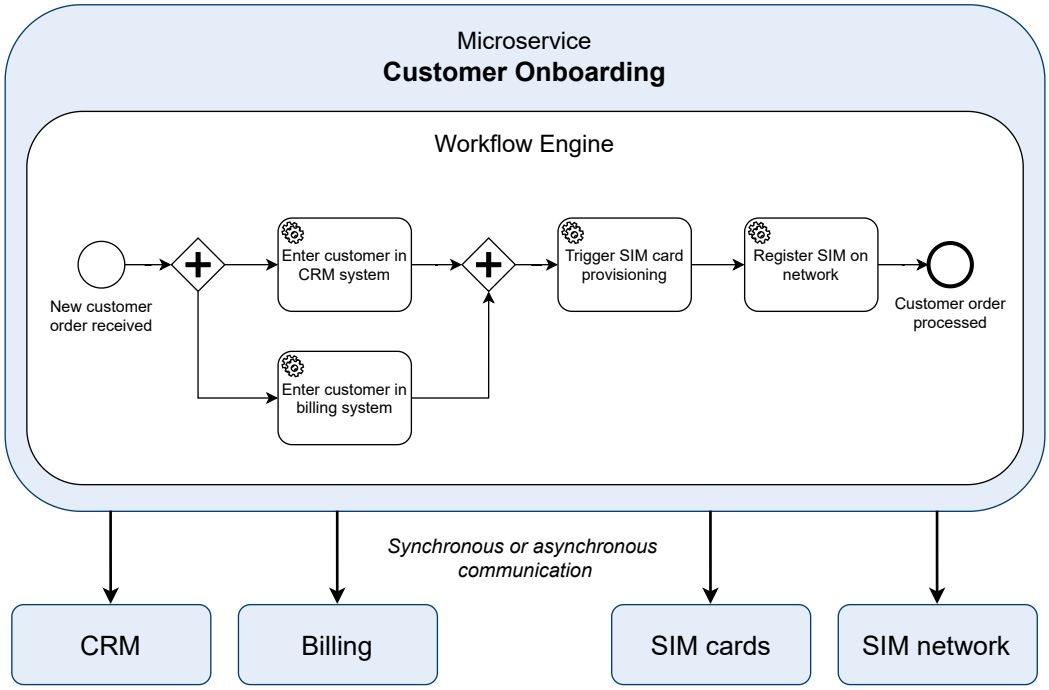

Abbildung 2.6.: Orchestrierung von Microservices durch Einbetten einer Process Engine [In Anlehnung an Ruecker 2021, S. 71]

2.4.5. Prozessgesteuerter Ansatz

Der *Prozessgesteuerte Ansatz* ist eine von Volker Stiehl entwickelte,

> *„[...] ganzheitliche, modellbasierte Projektabwicklungs- und Implementierungsmethodik zur Umsetzung innovativer digitaler Geschäftsmodelle basierend auf Prozessen [Stiehl, 2021].“*

Ziel der Methodik ist die Entwicklung einer sogenannten **prozessgesteuerten Anwendung** (zu englisch „process driven application“, kurz PDA) [vgl. Stiehl, 2021; vgl. Stiehl, 2013, S. 54]. Eine prozessgesteuerte Anwendung ist fachlich orientiert und dadurch gekennzeichnet, dass sie organisationsübergreifende und wettbewerbskritische Ende-zu-Ende-Geschäftsprozesse ganzheitlich implementiert, indem sie Daten und Funktionalitäten bestehender Systeme wiederverwendet [vgl. Stiehl, 2013, S. 24].

Als Vorgehen zur Implementierung einer PDA sieht Stiehl einen Top-Down-Ansatz vor. In enger Zusammenarbeit zwischen Fach- und IT-Abteilung werden zunächst — unabhängig von den existierenden IT-Landschaften — die fachlichen Prozesse modelliert. Bei der Entwicklung steht dabei die direkte Ausführbarkeit der Prozesse innerhalb einer Process Engine im Mittelpunkt [vgl. Stiehl, 2013, S. 54 ff.]. Stiehl spricht hierbei auch von einem prozessgesteuerten Denken, da der Modellierer bereits zu Beginn weiß, dass die fachlichen Prozesse unverändert in einer Workflow Engine ausgeführt werden [vgl. Stiehl, 2021].

Analog zu den in Kapitel 2.4.4 vorgestellten Process Engine basierten Architekturen, bedient sich Stiehl bei der Modellierung und Ausführung der Prozesse auch der BPMN-Spezifikation und einer Workflow Engine. Im Gegensatz zu den Process Engine basierten Architekturen sieht

der Prozessgesteuerte Ansatz jedoch eine strikte Trennung von fachlichen und technischen Prozessen vor [vgl. Stiehl, 2013, S. 92]. Die prozessgesteuerte Anwendung selbst ist ausschließlich für die systemunabhängigen, fachlichen Kernprozesse und dessen zugehörige Funktionalität verantwortlich [vgl. Stiehl, 2021]. Jegliche von außen zu erbringenden Dienste werden ausgehend von den fachlichen Anforderungen der PDA über einen Servicevertrag definiert. Die Umsetzung des Servicevertrags erfolgt dabei in dem sogenannten „Service Contract Implementation Layer“ (SCIL) — zu deutsch „Servicevertrag-Implementierungsschicht“ —, der sich neben der Erfüllung des Servicevertrags auch um sonstige technische Integrationsaufgaben, wie die Unterstützung diverser Kommunikationsprotokolle oder das Transformieren von Daten kümmert [vgl. Stiehl, 2013, S. 29; vgl. Stiehl, 2013, S. 93]. Die Kommunikation zwischen der prozessgesteuerten Anwendung und den Backend-Systemen findet also ausschließlich über den SCIL statt, was zu einer losen Kopplung zwischen PDA und Backend-Systemen führt und die technischen Details von den Fachprozessen über Schnittstellen abstrahiert [vgl. Stiehl, 2013, S. 91]. Die daraus resultierende Schichtenarchitektur wird im Rahmen des prozessgesteuerten Ansatzes auch als **prozessgesteuerte Architektur** bezeichnet und ist in Abbildung 2.7 dargestellt [vgl. Stiehl, 2021].

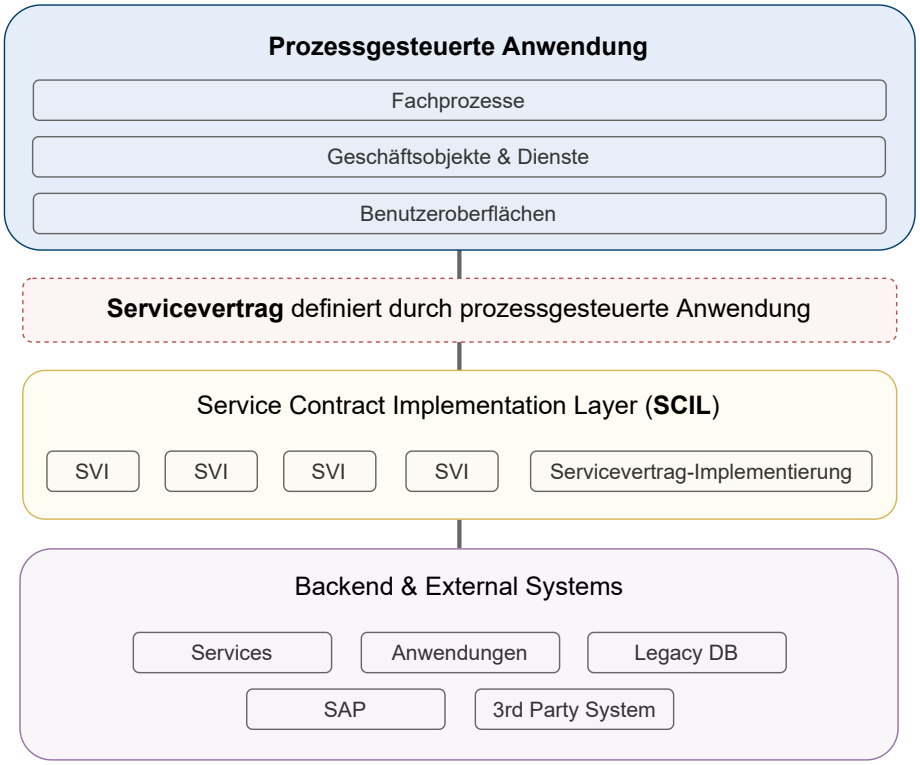

Abbildung 2.7.: Zusammenspiel zwischen prozessgesteuerter Anwendung und Backend-Systemen im Rahmen der prozessgesteuerten Architektur [In Anlehnung an Stiehl 2013, S. 27]

Damit ergibt sich ein weiterer Unterschied zu den im vorherigen Kapitel behandelten Process Engine basierten Architekturen. Dort werden die Backend-Systeme direkt über BPMN-Modelle aus der Process Engine heraus aufgerufen und sind damit nicht wie bei der prozessgesteuerten Architektur über die Servicevertrags-Implementierungsschicht voneinander entkoppelt. Die daraus resultierenden Vor- und Nachteile beider Architekturen werden beim Vergleich in Kapitel 5 genauer untersucht.

2.5. API-Management

2.5.1. Grundlagen von APIs

Ein „Application Programming Interface“, kurz API oder zu deutsch „Anwendungsprogrammierschnittstelle“ genannt, bezeichnet eine Software-zu-Software Schnittstelle, worin ein Programm Funktionalität nach außen bereitstellt, die dann lokal oder über Netzwerkgrenzen hinweg von anderer Software genutzt werden kann [vgl. De, 2017, S. 1]. APIs unterscheiden sich hierbei nach ihrem Kontext. Es gibt Betriebssystem-APIs, die beispielsweise zur Interaktion mit dem Betriebssystem genutzt werden oder aber auch Web-APIs, wo Anwendungen ihre Schnittstelle über das HTTP-Protokoll zur Verfügung stellen [vgl. De, 2017, S. 3].

Eine API definiert üblicherweise einen Vertrag, worin schnittstellenbezogene Informationen festgelegt sind. Dieser kann unter anderem folgende Aspekte einer API beinhalten [vgl. De, 2017, S. 1 f.]:

- Kommunikationsprotokoll
- Informationen zu den bereitgestellten Funktionen, wie beispielsweise Ein- und Ausgabedaten oder Beschreibungen zu den Endpunkten und Datentypen
- Service Level Agreements (SLAs), wie die Verfügbarkeit oder Antwortzeit
- Lizenzbedingungen und Nutzungspreise

Ändert sich der Vertrag einer API beispielsweise durch das Entfernen bestehender Funktionalität, so müssen alle Anwendungen, die die API verwenden, gegebenenfalls angepasst werden. Um diesen Effekt zu vermeiden, stellen APIs deshalb oft mehrere Versionen bereit und gewähren damit eine gewisse Abwärtskompatibilität [vgl. De, 2017, S. 5].

Der Aufbau einer API kann auf Grundlage unterschiedlicher Architekturstile und Kommunikationsprotokolle gestaltet sein. Die wohl bekanntesten Arten, die im Rahmen von Web-APIs eingesetzt werden, sind SOAP und REST, die auch für diese Arbeit von Relevanz sind. An dieser Stelle ist zu erwähnen, dass neben den genannten Arten weitere Möglichkeiten zur Gestaltung von APIs existieren. Ein Beispiel hierfür wäre die von Facebook entwickelte Datenabfragesprache GraphQL [The GraphQL Foundation, 2021], auf dessen Basis relativ einfach komplexe APIs gestaltet werden können oder das von Google entwickelte gRPC Protokoll [gRPC Authors, 2021].

2.5.2. Simple Object Access Protocol (SOAP)

Das „Simple Object Access Protocol“ (kurz: SOAP) ist ein XML-basiertes Kommunikationsprotokoll, das ursprünglich von Microsoft entwickelt wurde. Basierend auf dem Request-Response-Verfahren werden die Daten über SOAP-Nachrichten verschickt. Als Transportprotokoll kommt dabei üblicherweise HTTP zum Einsatz, wobei auch andere Internetprotokolle wie SMTP („Simple Mail Transfer Protocol“) genutzt werden können [vgl. Bengel, 2014, S. 216].

Eine SOAP-Nachricht ist charakterisiert durch drei Bestandteile [vgl. Bengel, 2014, S. 216 ff.]:

- „Envelope“-Element: Ist das Wurzelelement und enthält Namensraumdeklarationen

- „Header“-Element (optional): Zum Angeben zusätzlicher Informationen für die Nachricht, wie beispielsweise Authentifizierungsinformationen
- „Body“-Element: Enthält den eigentlichen Inhalt der Nachricht

Eine beispielhafte SOAP-Nachricht für eine Warenbestellung ist in Listing 2.1 dargestellt.

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Header>
  <t:Transaction xmlns:t="soap-transaction" SOAP-ENV:mustUnderstand="1">
   <transactionID>1234</transactionID>
  </t:Transaction>
 </SOAP-ENV:Header>
 <SOAP-ENV:Body>
  <ns1:purchaseOrder xmlns:n="urn:OrderService">
   <from><person>Guenther Bengel</person></from>
   <article>Java-Magazine</article>
  </ns1:purchaseOrder>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 2.1: SOAP-Nachricht für eine Warenbestellung [Bengel, 2014, S. 218]

Neben einer reinen Übertragung von Daten können über SOAP entfernte Methodenaufrufe realisiert werden [vgl. Bengel, 2014, S. 217]. Im Falle der Warenbestellung aus Listing 2.1 würde dann eine Klasse namens „OrderService“ existieren, die eine Methode `purchaseOrder` bereitstellt. Die „Web Service Description Language“ (WSDL) kann dabei als XML-basierte Sprache zur Beschreibung und Dokumentation der per SOAP bereitgestellten Funktionsaufrufe dienen, sodass ein Nutzer alle zur Verfügung stehenden Dienste eines Webservice zentral einsehen kann [vgl. Bengel, 2014, S. 220 f.].

2.5.3. Representational State Transfer (REST)

Im Gegensatz zu SOAP handelt es sich bei REST („Representational State Transfer“) nicht um ein Protokoll, sondern um einen Architekturstil, der als Abstraktionsstufe auf dem HTTP-Protokoll und dessen Standardmethoden GET, POST, PUT und DELETE aufbaut [vgl. Tilkov u. a., 2015, S. 8 ff.]. Ressourcen einer REST-API werden eindeutig über URIs („Uniform Resource Identifier“) identifiziert. Die Struktur der URI folgt dabei meist einem fest definiertem Schema. Eine beispielhafte URI zur Bereitstellung eines Produktes als Ressource mit der ID 20 kann wie folgt lauten: `https://example.com/products/20` [vgl. Tilkov u. a., 2015, S. 10 f.]. Die Ressourcen einer REST-API können in verschiedenen Repräsentationen vom Server zur Verfügung gestellt werden. Üblicherweise werden die Daten jedoch als JSON (Javascript Object Notation) empfangen. Zur Spezifikation und Dokumentation von REST-Schnittstellen hat sich über die letzten Jahre der OpenAPI Standard [OpenAPI Initiative, 2021] durchgesetzt. Dieser befindet sich zum Stand dieser Arbeit (2021) in Version 3.1.0 und ermöglicht das beschreiben von APIs in JSON oder YAML („YAML Ain't Markup Language“) Format. Vorteil neben der zentralen Dokumentation der Schnittstellen ist, dass aus der Spezifikation auch beispielsweise HTML-Webseiten oder Client-Programmcode zur Interaktion mit der API generiert werden kann. Für weiterführende Informationen zum Thema REST ist auf [Tilkov u. a., 2015] verwiesen.

2.5.4. Einführung in API-Management Plattformen

Innerhalb eines großen Unternehmens existieren üblicherweise viele verschiedene Backend-Systeme, die Schnittstellen zur Interaktion über unterschiedliche Technologien bereitstellen. Auch der derzeitige Trend hin zu Microservice-basierten Architekturen (vgl. 2.4.2) erhöht die Anzahl unabhängiger Services einer Systemlandschaft. Die Verwaltung und Steuerung mehrerer APIs kann dabei zu einer hohen Komplexität und Ineffizienz bei der Entwicklung führen. API-Management-Plattformen haben als Ziel, APIs in ihrem Lebenszyklus zu unterstützen und die damit einhergehende Komplexität zur Bereitstellung und Verwaltung von APIs zu reduzieren. De [De, 2017, S. 16 f.] untergliedert die Aufgaben einer API-Management-Plattform dabei in vier Bereiche:

- **„Developer Enablement for APIs“** umfasst Dienste, die den Entwicklern bzw. Konsumenten den Umgang mit APIs erleichtern sollen. Dazu zählt beispielsweise ein Entwicklerportal, worin alle verfügbaren APIs und deren Spezifikationen mit detaillierten Informationen zu dessen Funktionalität und Nutzung dokumentiert sind. Auch Support oder API-Onboarding Prozesse können innerhalb eines solchen Portals abgewickelt werden, sodass Nutzer die Möglichkeit haben selbstständig Zugriffe auf APIs zu beantragen [De, 2017, S. 25 f.].

- Unter dem Punkt **„Secure, Reliable and Flexible Communications“** bündelt De [De, 2017, S. 17 ff.] alle Aspekte in Bezug auf eine sichere und zuverlässige Kommunikation zwischen den Systemen welche die API implementieren und deren Nutzern. Das sogenannte „API-Gateway“ dient dabei als zentrale Proxy-Komponente zwischen beiden Parteien. Es sichert die dahinter liegenden APIs vor unautorisierten Zugriffen und prüft ob die empfangenen Daten der jeweiligen Spezifikationen entsprechen. Des Weiteren stellt es sicher, dass die Anfragen an die korrekten Systeme weitergeleitet werden (Routing) und schränkt bei Bedarf die Anzahl der erlaubten Anfragen ein. Neben den genannten Aspekten kann ein API-Gateway auch für weitere Integrationsaufgaben, wie die Transformation von Nachrichten, Protokollübersetzungen oder Load-Balancing verantwortlich sein.

- Das **„API Lifecycle Management“** befasst sich mit der Erstellung, Veröffentlichung und Versionierung von APIs. Konsumenten werden benachrichtigt, sobald Änderungen oder neue Versionen einer API erscheinen, um darauf rechtzeitig reagieren zu können. Außerdem können Nutzer neue Anforderungen an eine API stellen, oder technische Fehler und Probleme melden [De, 2017, S. 27 f.].

- **„API Auditing, Logging and Analytics“** spielt vor allem für das Business eine wichtige Rolle, um den geschäftlichen Nutzen und die Qualität von APIs zu messen. Durch das Monitoring der APIs über das API-Gateway können relevante Informationen zu dessen Nutzung und Performance bereitgestellt werden. Anhand dieser Daten kann festgestellt werden, welche APIs stark genutzt werden und damit beispielsweise von höherer Relevanz für das Business sind, oder aufgrund mangelnder Qualität verbessert werden müssen [De, 2017, S. 23 ff.].

Am Markt existieren bereits fertige API-Management-Systeme von großen Herstellern. Darunter Google's Apigee [Google, 2021] oder die von IBM entwickelte API-Connect Plattform [IBM, 2021]. Spotify liefert mit ihrem intern entwickelten Developer Portal namens „Backstage“ [Backstage Project Authors, 2021] beispielsweise eine moderne Plattform zur Verwaltung und Entwicklung von Applikationen und Services sowie deren Schnittstellen. Für weitere Informationen zum Thema API-Management ist auf [De, 2017] verwiesen.

3. Praktische Ansätze zur Implementierung von Geschäftsprozessen

Geschäftsprozesse sind der Kern eines Informationssystems. Um einen hohen Nutzen aus der Implementierung und Automatisierung von Prozessen zu erhalten, ist es essenziell diese realitätsgetreu in Software abzubilden. In Kapitel 2.2 wurden bereits zwei Entwurfsmethoden zur Modellierung von Softwarearchitekturen vorgestellt: Das Domain-Driven Design und ARIS. Dieses Kapitel arbeitet nun Gemeinsamkeiten und Unterschiede beider Methoden heraus. Dabei wird erläutert, inwiefern DDD und ARIS die Implementierung von Geschäftsprozessen unterstützen und wie eine modellgetriebene Entwicklung mit dem bereits in Kapitel 2.4.4 und 2.4.5 behandelten BPMN-Standard bei der Umsetzung herangezogen werden kann.

3.1. Domain-Driven Design und ARIS

Domain-Driven Design und ARIS sind beides Konzepte für den Entwurf komplexer IT-Systeme. ARIS als Architekturmethodik wurde erstmals 1991 vorgestellt und existiert damit bereits einige Jahre länger als DDD. Seitdem wurde ARIS in einer Vielzahl unterschiedlicher Projekte zur Entwicklung komplexer Informationssysteme eingesetzt [vgl. Scheer, 2002a]. Domain-Driven Design auf der anderen Seite repräsentiert einen moderneren Ansatz, der vor allem im Bereich der Softwareentwicklung über die letzten Jahre an Beliebtheit gewonnen hat.

Domain-Driven Design setzt sich als Ziel ein *„[...] Design [zu] schaffen, das effektiv auf die strategischen Bedürfnisse der Organisation eingeht“* [Vernon, 2017, S. 109]. Hierfür arbeiten die Entwickler eng mit dem Fachbereich zusammen, um ein ganzheitliches Domänenmodell zu entwickeln, welches später als Grundlage für die Implementierung der Software dient. Die Fachsprache soll dabei allgegenwärtig in allen Aspekten des Entwicklungsprozesses präsent sein („Ubiquitous Language“). ARIS beschreibt eine ähnliche Vorgehensweise. Ausgehend vom Fachkonzept findet eine schrittweise Übersetzung der fachlichen Anforderungen in technische Konzepte statt (vgl. ARIS-Phasenmodell in Kapitel 2.2.2). Durch das Top-Down Vorgehen wird sichergestellt, dass das System die Geschäftsbedürfnisse erfüllt und einen optimalen Business-Value liefert. Bei DDD und ARIS ist die Gestaltung und Implementierung des Systems also von fachlicher Seite aus getrieben: Eine enge Zusammenarbeit zwischen Fachbereich und IT ist damit für beide Methoden die Grundlage eines guten Architekturentwurfs.

Informationssysteme implementieren Geschäftsprozesse, die oft nicht trivial sind sondern mit einer gewissen Komplexität einhergehen. Um mit dieser Komplexität umzugehen, betrachtet ARIS einen Prozess deshalb in mehreren Sichten (vgl. ARIS-Sichten in Kapitel 2.2.2). Dabei wird eine zentrale Fragestellung des Geschäftsprozessmanagements beantwortet: *„[...] wer macht was wann und womit [...]“* [Becker u. a., 2012, S. 6]. Die ARIS-Sichten liefern auf alle Aspekte dieser Frage eine Antwort. Die Organisationssicht legt beispielsweise Fokus auf die Verantwortlichkeiten und Rollen (WER). Die Funktionssicht beschreibt die einzelnen Aufgaben und Tätigkeiten innerhalb des Prozesses (WAS). Die Datensicht umfasst alle relevanten Date-

nobjekte (WOMIT) und die Steuerungssicht integriert alle vorherigen Sichten, indem sie den Geschäftsprozess ganzheitlich über beispielsweise BPMN modelliert und damit die Aktivitäten in eine zeitliche Abfolge bringt (WANN).

Domain-Driven Design nutzt zur Identifizierung und Modellierung einer Domäne leichtgewichtige Workshop-Methoden wie beispielsweise das in Kapitel 2.2.1 vorgestellte Event Storming oder Domain Story Telling. Diese Ansätze beziehen bei der Gestaltung Softwareentwickler und Fachexperten mit ein und setzten dabei nicht das Wissen spezifischer Modellierungssprachen voraus. So kann schnell eine gemeinsame Sicht auf die Domäne erarbeitet und diese später besser in der Implementierung umgesetzt werden. Bei genauerer Betrachtung der Methoden zeigt sich, dass die im Rahmen von DDD eingesetzten Ansätze zum Erfassen einer Domäne, analog zu ARIS, indirekt dieselbe zentrale Fragestellung des Prozessmanagements beantworten möchten und damit einen Geschäftsprozess beschreiben. Beim Domain Story Telling wird der Geschäftsprozess durch Piktogramme verdeutlicht. Die beteiligten Personen oder Systeme innerhalb des Prozesses (WER) werden darin über „Actors“ dargestellt. „Activities“ beschreiben die einzelnen Arbeitsschritte der Akteure (WAS) und „Work Objects“ die dafür relevanten Informationsobjekte (WOMIT). Domain Story Telling betrachtet damit die Organisations-, Funktions- und Datensicht des ARIS-Hauses (vgl. Abbildung 2.2 in Grundlagenkapitel 2.2.2). Die Steuerungssicht (WANN) kann durch textuelle Annotationen mit Nummerierung der einzelnen Aktivitäten im Piktogramm ergänzt werden. Event Storming als zweite Methodik unterstützt vor allem bei der Funktions- und Steuerungssicht (WAS, WANN). Dort wird die Domäne durch das zeitliche Anordnen von Domain-Events — bzw. fachlicher Ereignisse, die im Kontext der Domäne auftreten — durchdrungen. Hier kann das Modell jedoch auch in nachgelagerten Schritten durch die WER und WOMIT Aspekte ergänzt werden. Im Rahmen der ereignisgesteuerten Implementierungen dieser Arbeit wird der Geschäftsprozess durch Domänen Events abgebildet.

Die im Kontext von DDD erwähnten Techniken, wie Domain Story Telling und Event Storming eignen sich aufgrund ihrer Leichtgewichtigkeit und einfachen Verständlichkeit gut, wenn es darum geht schnell einen Überblick über fachliche Zusammenhänge zu erhalten. Von dieser Agilität profitieren Projekte vor allem zu Beginn der Entwicklung. Mit der Zeit wächst jedoch die Fachlichkeit und damit auch die Komplexität eines Systems. Um Transparenz über die wachsenden Geschäftsprozesse, die in dem jeweiligen System implementiert werden, sicherzustellen, ist eine stetige Anpassung der Prozessdokumentationen und Modelle notwendig. Ein solches Vorgehen ist dabei zeitaufwändig und ineffizient. Zusätzlich kann auf Dauer keine exakte Abbildung der erarbeiteten Modelle in Bezug auf das Softwaresystem garantiert werden. Mit zunehmender Zeit und vermehrten Prozessänderungen können die Software und das eigentliche Domänenmodell voneinander divergieren und Missverständnisse hervorrufen.

3.2. Model-driven Development mit BPMN 2.0

Workflow Engines (vgl. Definition in Kapitel 2.4.4) ermöglichen es Geschäftsprozesse direkt zu implementieren und damit zu automatisieren. Eine Workflow Engine ist eine Art Compiler für Prozessmodelle, wobei die Modelle den Programmcode repräsentieren [vgl. Freund und Rücker, 2019, S. 6]. Dies ermöglicht eine modellgetriebene Softwareentwicklung auf Basis des Prozessmodells. Änderungen des Modells wirken sich im Vergleich zu den vorher behandelten Methoden des DDD direkt auf die Implementierung und Ausführung innerhalb der Process Engine bzw. des Systems aus und dienen nicht nur der reinen Dokumentation.

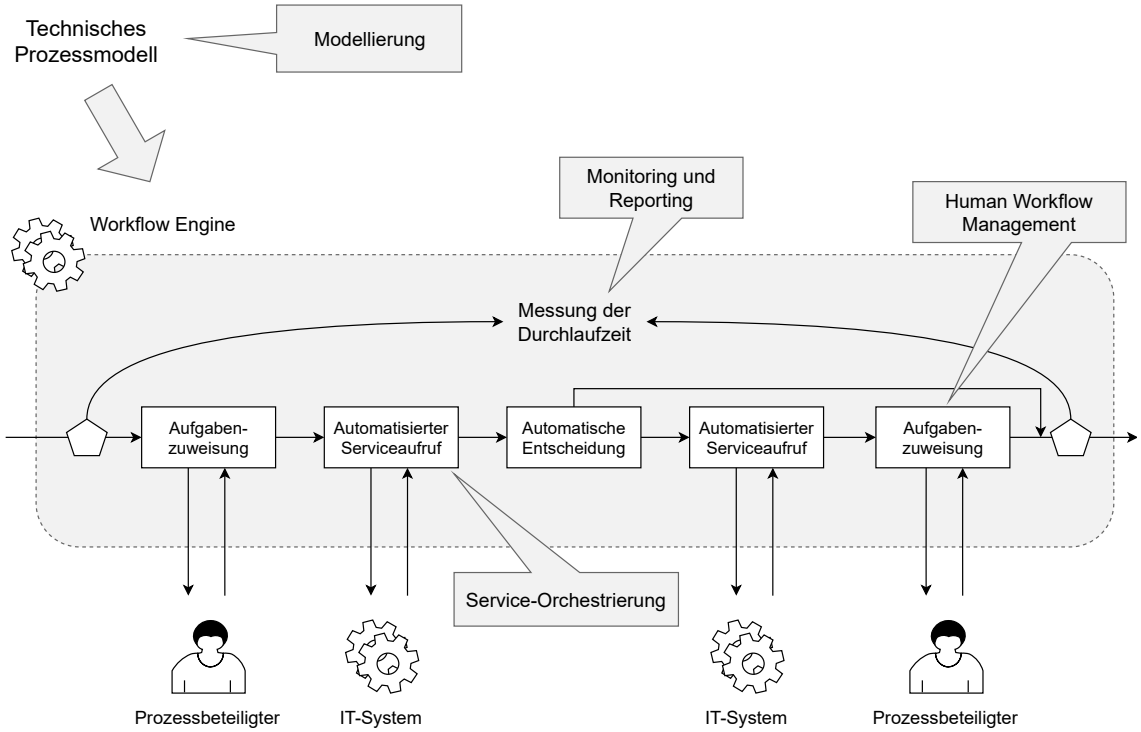

Abbildung 3.1.: Die Workflow Engine als zentrale Komponente zur Implementierung von Geschäftsprozessen [In Anlehnung an Freund und Rücker 2019, S. 6]

Abbildung 3.1 zeigt nach Freund und Rücker [vgl. Freund und Rücker, 2019, S. 6] die grundlegenden Fähigkeiten einer Workflow Engine. Diese steuert anhand eines technischen Prozessmodells sowohl die Aufgabenzuweisung an Prozessbeteiligte als auch die Orchestrierung von Anwendungen oder die Automatisierung von Entscheidungen. Außerdem überwacht eine Process Engine die Prozessinstanzen und ermöglicht es dadurch Durchlaufzeiten zu messen und auf potenzielle Fehler innerhalb einer Prozessinstanz reagieren zu können. Eine Workflow Engine arbeitet auf Basis eines technischen Prozessmodells. Vorteil eines solchen Modells ist es, dass es universell eingesetzt werden kann und sowohl vom Fachbereich als auch den Softwareentwicklern verstanden werden kann. Es trägt damit zu einer besseren Kommunikation zwischen Business und IT bei [vgl. Freund und Rücker, 2019, S. 7].

Als Beschreibungssprache der Prozessmodelle im Rahmen der Implementierung der Process Engine basierten und der prozessgesteuerten Architektur wird der bereits genannte BPMN („Business Process Model and Notation“) Standard verwendet. BPMN eignet sich deshalb gut, weil aufgrund der umfassenden Modellierungselemente Prozesse ganzheitlich und detailliert abgebildet werden können. Mit Version 2.0 bietet der Standard außerdem ein XML-basiertes Format zur Beschreibung der Prozessmodelle an. Die XML-Dateien können so von Process Engines gelesen und ausgeführt werden [vgl. The Object Management Group, 2021b]. Abbildung 3.2 zeigt die visuelle Darstellung ein exemplarischen BPMN-Prozessmodells. Die tatsächliche XML-Repräsentation des Modells ist in Listing 3.1 gezeigt.

Neben BPMN gibt es mittlerweile auch Prozessbeschreibungssprachen von großen Herstellern wie Amazon oder Netflix, die ihre eigenen Workflow Engines anbieten. Dazu zählt beispielsweise AWS Step Functions [Amazon Web Services, 2021] oder Netflix Conductor [Netflix Technology

Blog, 2016]. Hierbei werden die Prozesse textuell über JSON oder YAML Dateien beschrieben. Nachteil dieser Ansätze ist jedoch, dass aufgrund der textuellen Beschreibung, der Prozessfluss nicht auf Anhieb sichtbar ist. BPMN liefert im Vergleich hierzu mit einer grafischen Prozessmodellierung einen großen Mehrwert, um auch komplexe Prozessdiagramme einfach verstehen zu können [vgl. Ruecker, 2021, S. 108].

Abbildung 3.2.: Visuelle Darstellung eines BPMN-Modells [Eigene Darstellung]

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
    xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
    xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
    xmlns:di="http://www.omg.org/spec/DD/20100524/DI" id="Definitions_0lddr47"
    targetNamespace="http://bpmn.io/schema/bpmn" exporter="Camunda Modeler"
    exporterVersion="4.2.0">
 <bpmn:process id="Process_0zzb5yc" isExecutable="true">
  <bpmn:startEvent id="StartEvent_1">
   <bpmn:outgoing>Flow_1tz881t</bpmn:outgoing>
  </bpmn:startEvent>
  <bpmn:sequenceFlow id="Flow_1tz881t" sourceRef="StartEvent_1"
   targetRef="Activity_0iyb7i9" />
  <bpmn:userTask id="Activity_0iyb7i9" name="Book Vacation">
   <bpmn:incoming>Flow_1tz881t</bpmn:incoming>
   <bpmn:outgoing>Flow_0s4a8ac</bpmn:outgoing>
  </bpmn:userTask>
  <bpmn:endEvent id="Event_0a3lpjl">
   <bpmn:incoming>Flow_0s4a8ac</bpmn:incoming>
  </bpmn:endEvent>
  <bpmn:sequenceFlow id="Flow_0s4a8ac" sourceRef="Activity_0iyb7i9"
   targetRef="Event_0a3lpjl" />
 </bpmn:process>
 <bpmndi:BPMNDiagram id="BPMNDiagram_1">
  <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="Process_0zzb5yc">
   <bpmndi:BPMNEdge id="Flow_1tz881t_di" bpmnElement="Flow_1tz881t">
    <di:waypoint x="188" y="120" />
    <di:waypoint x="270" y="120" />
   </bpmndi:BPMNEdge>
   <bpmndi:BPMNEdge id="Flow_0s4a8ac_di" bpmnElement="Flow_0s4a8ac">
    <di:waypoint x="370" y="120" />
    <di:waypoint x="442" y="120" />
   </bpmndi:BPMNEdge>
   <bpmndi:BPMNShape id="_BPMNShape_StartEvent_2" bpmnElement="StartEvent_1">
    <dc:Bounds x="152" y="102" width="36" height="36" />
   </bpmndi:BPMNShape>
   <bpmndi:BPMNShape id="Activity_10m2adb_di" bpmnElement="Activity_0iyb7i9">
    <dc:Bounds x="270" y="80" width="100" height="80" />
   </bpmndi:BPMNShape>
   <bpmndi:BPMNShape id="Event_0a3lpjl_di" bpmnElement="Event_0a3lpjl">
    <dc:Bounds x="442" y="102" width="36" height="36" />
```

```
35      </bpmndi:BPMNShape>
36    </bpmndi:BPMNPlane>
37  </bpmndi:BPMNDiagram>
38 </bpmn:definitions>
```

Listing 3.1: XML-Struktur eines beispielhaften BPMN-Prozesses

4. Beispielhafte Architektur und Implementierung anhand eines Reisebuchungsprozesses

Kapitel 2 hat nun die wesentlichen theoretischen Grundlagen, die für diese Arbeit von Relevanz sind, erläutert. Dabei wurden drei unterschiedliche Softwarearchitekturstile vorgestellt: Ereignisgesteuerte, Process Engine basierte und prozessgesteuerte Architekturen. Ziel dieser Arbeit soll ein Vergleich dieser Stile anhand ausgewählter Qualitätsmerkmale aus Kapitel 2.1.4 sein. Um einen repräsentativen und praktischen Architekturvergleich erzielen zu können, bietet es sich an die unterschiedlichen Architekturmuster anhand eines Beispielszenarios innerhalb einer hypothetischen Anwendungslandschaft umzusetzen. Auf Basis der Implementierung können so Rückschlüsse auf die Qualitätseigenschaften der Architekturstile gezogen werden. Die Betrachtung erfolgt anhand eines Reisebuchungsprozesses, wobei jede Architekturumsetzung sich stets an den geltenden Best Practices zur Implementierung ausrichten soll. Dieses Kapitel definiert hierfür zunächst die hypothetische bestehende Systemlandschaft, in die das Reisebuchungssystem integriert werden muss und die wesentlichen Anforderungen und Rahmenbedingungen an das Reisebuchungssystem, sodass nachfolgend eine Referenzimplementierung durch Einsatz der einzelnen Architekturstile erfolgen kann.

4.1. Technologiewahl

Die Technologiewahl basiert vor allem auf eigenen Präferenzen und bereits gesammelten Erfahrungen. Die Entscheidung zur verwendeten Technologie für die Implementierung aller später genannten Systemkomponenten fällt hierbei auf **Java** in Kombination mit dem **Spring Framework** [VMware, 2021a]. Diese Entscheidung rührt aus dem Fakt, dass der Autor bereits Kenntnisse in dieser Programmiersprache aufweisen kann und damit eine geringere Einarbeitungszeit benötigt. Generell können die jeweiligen Architekturen auch unabhängig von der gewählten Technologie implementiert werden, sodass diese Wahl keine Einschränkung darstellt. Zur Abwicklung asynchroner Kommunikation und dem Austausch von Ereignissen zwischen den einzelnen Services innerhalb der jeweiligen Architekturen wird als Messaging-System die in Kapitel 2.3.2 genannte Event-Streaming-Plattform **Apache Kafka** herangezogen. Die Entscheidung ist vor allem deshalb auf Kafka gefallen, da die Plattform als Event Broker klassische Message Broker um zusätzliche Funktionalitäten erweitert. Im Rahmen der einzelnen Implementierungen könnte Kafka jedoch auch gegen andere Message Broker wie ActiveMQ ausgetauscht werden. Zur Implementierung der Process Engine basierten Architektur (vgl. Kapitel 4.7) und prozessgesteuerten Architektur (vgl. Kapitel 4.8) wird außerdem eine Process Engine benötigt. Aufgrund persönlicher Erfahrung wird hier die in Java entwickelte Open-Source BPMN Process Engine von **Camunda** [Camunda Services GmbH, 2021a] verwendet. Jegliche Benutzeroberflächen (vgl. Kapitel 4.4) werden mit HTML („Hypertext Markup Language“), CSS („Cascading Style Sheets“) und JavaScript entwickelt. Zur Unterstützung wird hierbei **ReactJS** [Facebook Inc., 2021] als Frontend-Framework eingesetzt.

4.2. Vorstellung der hypothetischen Systemlandschaft

In Unternehmen herrscht eine Vielfalt an unterschiedlichen Systemen, wobei jedes System seine eigene Domäne abbildet und dabei eigene Kommunikationsprotokolle und Schnittstellen verwendet. Die Entwicklung eines neuen Informationssystems geht dabei meist mit der Integration in eine bestehende Systemlandschaft einher, da die Systeme zur Erbringung von Funktionalitäten und Geschäftsprozessen miteinander kommunizieren müssen. So kann es beispielsweise notwendig sein, dass Bestelldaten in Rahmen eines Bestellprozesses in einem internen Finanzsystem gespeichert werden müssen, um die Generierung von Umsatzberichten für die Geschäftsführung zu ermöglichen. Auch kann es hilfreich sein einige Funktionalitäten nach extern auszulagern, um sich besser auf das Kerngeschäft konzentrieren zu können. Dort ist eine Integration mit Diensten außerhalb des Unternehmens zu berücksichtigen.

Um eine realitätsnähere Ausgangssituation — wie sie auch in einem echten Unternehmen der Fall wäre — für die Entwicklung des Beispielsystems zu simulieren, erscheint es deshalb sinnvoll, bestehende interne und externe Systeme, die im Rahmen des Reisebuchungsprozesses berücksichtigt werden, zu definieren.

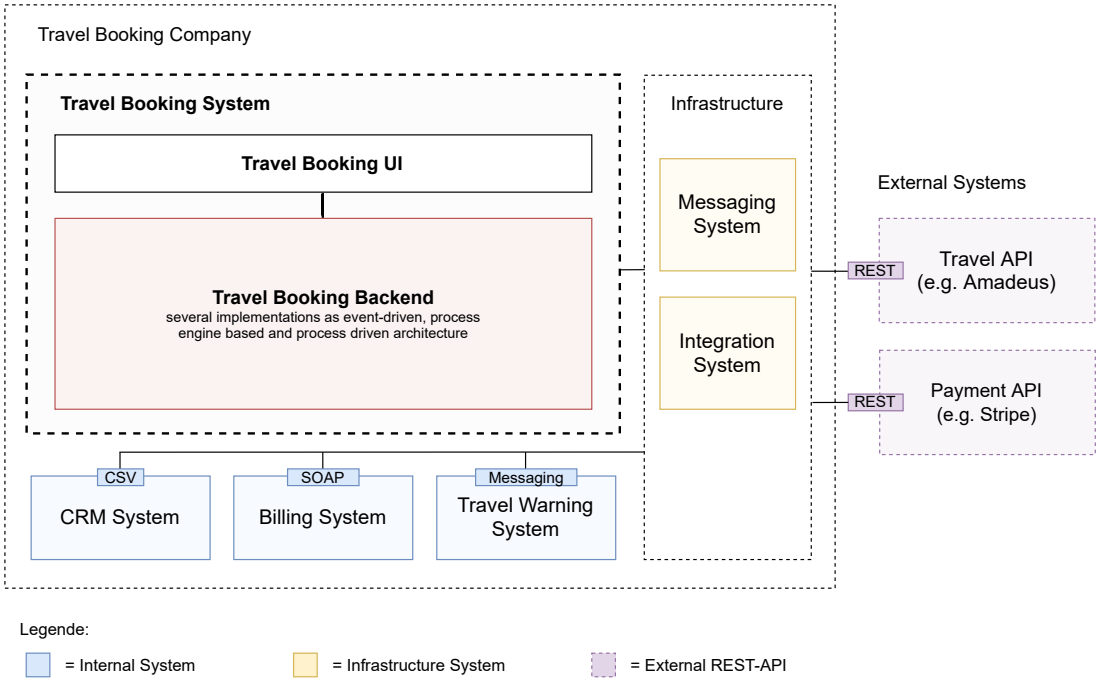

Abbildung 4.1.: Übersicht der hypothetischen bestehenden Systemlandschaft und Einordnung des Reisebuchungssystems im Rahmen der Beispielimplementierung [Eigene Darstellung]

Abbildung 4.1 liefert eine Übersicht aller relevanten Systeme die im Rahmen des Reisebuchungsprozesses zum Einsatz kommen und mit dem Reisebuchungssystem interagieren. Das Reisebuchungssystem („Travel Booking System“) ist verantwortlich für die Implementierung und Abwicklung des Reisebuchungsprozesses und damit der zentrale Betrachtungsgegenstand dieser Arbeit. Das Travel Booking System ist in zwei voneinander getrennte Komponenten

aufgeteilt: Dem „Travel Booking UI“ und dem „Travel Booking Backend“. Das Travel Booking UI ist eine Webapp, die alle Benutzeroberflächen kapselt und mit der der Endanwender über den Browser interagieren kann. Das Travel Booking Backend ist für die technische Umsetzung des Reisebuchungsprozesses verantwortlich. Es ist mehrfach je nach Architekturform (z.B. Ereignisgesteuert, Process-Engine basiert oder entsprechend des Prozessgesteuerten Ansatz) implementiert. Genauere Details zur internen Struktur der Komponenten des Reisebuchungssystems liefern die jeweiligen Implementierungskapitel (vgl. Kapitel 4.4, 4.5, 4.6, 4.7, 4.8).

Neben dem eigentlichen Reisebuchungssystem werden weitere Systeme im Rahmen der bestehenden Systemlandschaft des Reisebuchungsunternehmens definiert, die es im Kontext der Anforderungen an den Reisebuchungsprozess zu integrieren gilt (vgl. Kapitel 4.3.1). Hierzu zählen die in Darstellung 4.1 blau markierten internen Systeme und lila gekennzeichneten externen APIs. Diese Systeme sind je Implementierung in gleicher Art vorgegeben und damit unabhängig von den einzelnen Architekturumsetzungen. Zur Abbildung der Heterogenität einer realen IT-Landschaft in Unternehmen verwenden die Systeme unterschiedliche Protokolle und Formate zur Kommunikation wie z. B. REST, SOAP oder Datei-basiert über CSV („Comma-separated values“). Neben den vorgegebenen internen und externen Systemen der Anwendungslandschaft sind zusätzlich die gelb markierten Infrastruktur-Komponenten zu berücksichtigen, die in allen Implementierungsvarianten zum Einsatz kommen. Im weiteren Verlauf werden nun die einzelnen Systeme der hypothetischen Systemlandschaft und dessen Schnittstellen in Kürze vorgestellt.

4.2.1. Intern: CRM System

Das „CRM System“ soll ein bestehendes Altsystem im Rahmen der Beispielimplementierung simulieren. CRM steht dabei für „Customer Relationship Management“ und dient der Abwicklung von kundenzentrischen Prozessen.

Schnittstellen

Da es sich hierbei um ein Altsystem handeln soll, erscheint es weniger sinnvoll, moderne Web-Schnittstellen anzubieten. Aus diesem Grund ist die Entscheidung auf eine Datei-basierte Schnittstelle, die das Einlesen von CSV-Dateien ermöglicht, gefallen. Integriert werden dabei Kundendaten, die aus den komma-separierten CSV-Dateien extrahiert werden. Die CSV-Datei für den Import der Kundendaten weist pro Zeile folgende Struktur auf:

```
1 <name>,<email>,<age>,<address>,<country>,<creditCardNumber>
```

Ein beispielhafter Import-Datensatz könnte dann wie folgt aussehen:

```
1 John Doe,john@doe.de,35,Beispielstrasse 5,Deutschland,1234567891234567
```

4.2.2. Intern: Billing System

Beim „Billing System“ handelt es sich analog zum CRM System um eine Legacy Anwendung, die eine Schnittstelle zum Speichern von Kundendaten anbietet.

Schnittstellen

Das Billing System stellt zum Anlegen eines neuen Kunden eine SOAP-Schnittstelle (vgl. Kapitel 2.5.2) nach außen bereit. Die Definition der Schnittstelle erfolgt über eine XML Schema Definition (XSD), die in Listing 4.1 dargestellt ist.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://thi.de/soap/travel-booking"
        targetNamespace="http://thi.de/soap/travel-booking">
    <xs:element name="saveCustomerRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="customer" type="tns:customer"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="saveCustomerResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="success" type="xs:boolean"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="customer">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="country" type="tns:country"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="age" type="xs:int"/>
            <xs:element name="creditCardNumber" type="xs:long"/>
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="country">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Deutschland"/>
            <xs:enumeration value="Vereinigtes Koenigreich"/>
            <xs:enumeration value="Frankreich"/>
            <xs:enumeration value="Italien"/>
            <xs:enumeration value="Spanien"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Listing 4.1: XSD-Schema der Kundenschnittstelle des Billing System

4.2.3. Intern: Travel Warning System

Das „Travel Warning System“ ist ein Service der periodisch alle 20 Sekunden eine Reisewarnung in Form eine Nachricht an das Messaging System sendet. Eine Reisewarnung ist dabei immer auf eine Stadt bezogen und kann einen Warnungsgrad von eins bis zehn haben.

Schnittstellen

Die Schnittstelle des Travel Warning System ist durch das asynchrone Senden von Nachrichten auf das Topic `TravelWarningEvent` des Messaging-Systems gegeben. Dieses Topic kann von belieben Services subskribiert werden, um die aktuellen Reisemeldungen zu erhalten.

4.2.4. Infrastruktursysteme

Die Infrastruktur besteht aus dem „Messaging System“ und dem „Integration System“. Das Messaging System wird wie bereits in Kapitel 4.1 erläutert durch Apache Kafka repräsentiert. Das Integration System stellt einen Spring Boot Service dar, der das Apache Camel Framework [Apache Software Foundation, 2021b] zur Implementierung von Integrationslogik verwendet. Das Integration System ist hauptsächlich für die Integration zwischen Travel Booking Backend und der bestehenden Systemlandschaft verantwortlich. Außerdem wird es zur Implementierung der Servicevertrag-Implementierungsschicht im Rahmen der prozessgesteuerten Architektur in Kapitel 4.8 eingesetzt. In Summe umfasst das Integration System 17 Camel Routen zur Umsetzung der Integrationslogik.

4.2.5. Extern: Travel API

Die „Travel API“ stellt ein externes System dar, das es ermöglicht, Hotel- und Flugangebote einzuholen, zu buchen und diese zu stornieren. Das System orientiert sich an den Self-Service APIs der Amadeus IT Group [vgl. Amadeus IT Group, 2021], die im Falle einer realen Implementierung herangezogen werden können.

Schnittstellen

Bei der Travel API handelt es sich um eine REST-API die mehrere Endpunkte zur Verfügung stellt:

- `/offers/flights` : Zum Finden eines Flugangebots
- `/offers/hotels/:city` : Zum Finden einer Liste von Hotelangeboten zu einer bestimmten Stadt
- `/booking/flights` : Zum Buchen eines Flugangebots
- `/booking/hotels` : Zum Buchen eines Hotelangebots
- `/cancellation/flights` : Zum Stornieren eines Flugangebots
- `/cancellation/hotels` : Zum Stornieren eines Hotelangebots

Zur Spezifikation der Schnittstelle bietet sich der in Kapitel 2.5.3 genannte OpenAPI-Standard an. An dieser Stelle ist deshalb auf Anhang A.3.1 zu verweisen, worin eine OpenAPI Beschreibung der Travel API zu finden ist. Die Spezifikation beinhaltet alle notwendigen Informationen zu den REST-Endpunkten, wie beispielsweise die Ein- und Ausgabedatentypen und möglichen HTTP-Antwortcodes.

4.2.6. Extern: Payment API

Die „Payment API“ ist ein externes System, das Dienstleistungen im Bereich der Zahlungsabwicklung anbietet. Verglichen werden kann die beispielhafte API mit dem Online-Bezahldienst „Stripe“ [vgl. Stripe, 2021], der sich darauf spezialisiert hat zahlungsbezogene Geschäftsprozesse digital umzusetzen.

Schnittstellen

Bei der Payment API handelt es sich analog zur Travel API um eine REST-Schnittstelle mit zwei Endpunkten:

- `/payments/charge` : Zum Abbuchen eines Geldbetrags von einer Kreditkarte
- `/payments/refund` : Zum Zurückerstatten von Geldbeträgen auf eine Kreditkarte

Die OpenAPI-Spezifikation der Payment API ist in Anhang A.3.2 einzusehen.

4.3. Anforderungen an das Beispielsystem

4.3.1. Prozessanforderungen

Dieser Abschnitt widmet sich einer prosaischen Beschreibung der Geschäftsprozessanforderungen. Bezogen der in Kapitel 3 behandelten Prozessfragestellungen sollen hierbei WER, WAS und WANN Aspekt des Fachprozesses geklärt werden. Ziel ist es einen leicht zu verständlichen Geschäftsprozess zu beschreiben, der gleichzeitig typische Abläufe widerspiegelt, die auch in realen Fachprozessen auf unterschiedliche Art und Weise zum Einsatz kommen können. Im Kontext dieser Arbeit wird deshalb die Reisebuchung als fachliche Domäne für die Implementierung des Beispielsystems herangezogen, da die Buchung einer Reise einen typischen Prozess darstellt, den schon viele Menschen durchlaufen haben. Damit eignet sich dieser aufgrund seiner Realitätsnähe und einfachen Verständlichkeit gut für vorliegenden Anwendungsfall. Der innovative Aspekt des Fachprozesses liegt in der intelligenten Generierung von Reisepaketen, wobei ein Reisepaket immer eine Kombination aus Hotel und Flug umfasst, die in direkter Korrelation mit den Kundenwünschen steht.

Abbildung 4.2 spiegelt den Standardfall des Geschäftsprozesses — also ohne jegliche Verzweigungen oder Ausnahmebehandlungen — wider, sodass die essenziellen Vorgänge deutlich werden. Die zwei Hauptrollen des Fachprozesses stellen der Kunde und das Reiseunternehmen dar:

- **Kunde**: Der Kunde steht im Mittelpunkt, da er mit dem Bedürfnis, eine Reise unternehmen zu wollen, den Geschäftsprozess initiiert. Das Verhalten des Kunden wird in den Beispielimplementierungen über Aufrufe aus dem webbasierten Travel Booking UI simuliert (vgl. Kapitel 4.4).
- **Reiseunternehmen**: Das Reiseunternehmen erfüllt das Bedürfnis des Kunden und sorgt für die Leistungserbringung. Es ist dafür zuständig dem Kunden geeignete Reisepakete bereitzustellen und die Buchung der Reise abzuwickeln. Die Leistungserbringung erfolgt überwiegend durch Kommunikation mit internen und externen Systemen. Das Reiseunternehmen beinhaltet jedoch auch Servicemitarbeiter, die für alle manuellen Aufgaben im Rahmen des Prozesses verantwortlich sind. Dazu kann sowohl die Genehmigung einer Reisebuchung als auch der Eingriff im Falle eines Systemfehlers zählen. Solche manuellen Aufgaben werden ähnlich wie bei der Kundenrolle durch die Interaktion mit dem Travel Booking UI abgewickelt (vgl. Kapitel 4.4).

Im folgenden findet nun eine Beschreibung des Prozesses anhand der in Abbildung 4.2 dargestellten, übergeordneten Schritte statt, sodass der Ablauf des Geschäftsprozesses gut nachvollzogen werden kann.

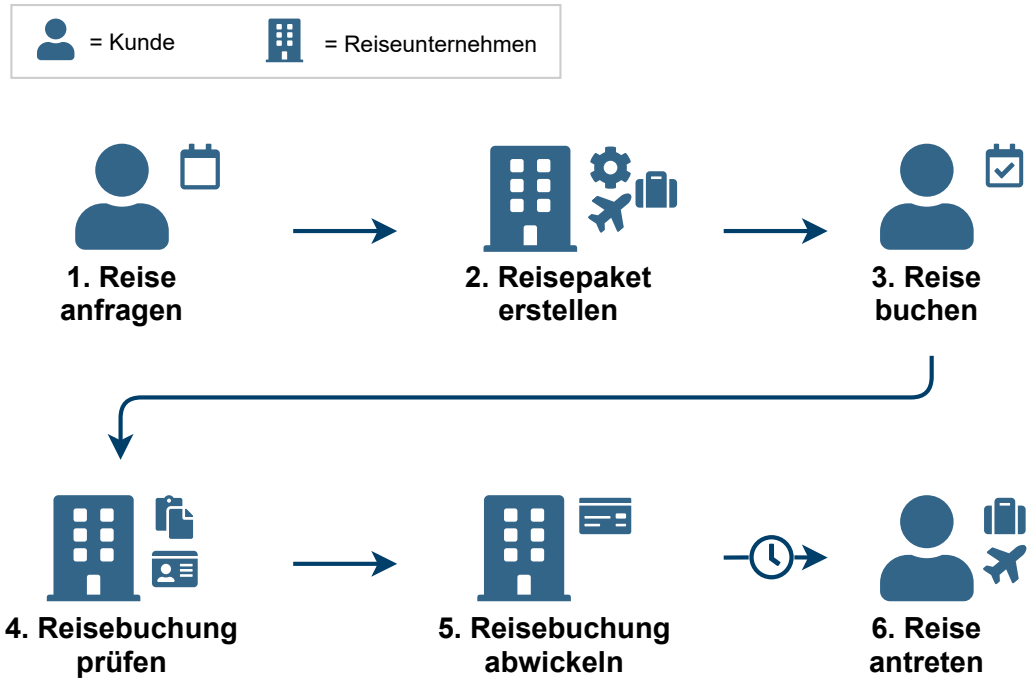

Abbildung 4.2.: Standardfall des Reisebuchungsprozesses [Eigene Darstellung]

1. Reise anfragen

Der Reisebuchungsprozess beginnt damit, dass ein Kunde die Generierung eines neuen Reisepaketes beim Reiseunternehmen anfragt. Dazu gibt der Kunde in einem Webformular seinen Reiseausgangspunkt, sein gewünschtes Reiseziel, den für ihn passenden Reisezeitraum, die Anzahl an Reisenden und ob es eine Premiumreise sein soll an. Die Daten werden nach erfolgreicher Eingabe dann an das Reiseunternehmen gesendet.

2. Reisepaket erstellen

Basierend auf der Kundenanfrage erstellt das Reiseunternehmen ein personalisiertes Reisepaket. Ein Reisepaket besteht dabei immer aus einem Hotelvorschlag, einem Hinflug und einem passendem Rückflug. Die Hotel- und Flugangebote werden von der externen Travel API (vgl. Kapitel 4.2.5) bezogen. Aufgabe ist es nun eine Liste von Hotelangeboten für einen bestimmten Reiseort über die externe API anzufragen und dann ein passendes Hotel für das Reisepaket auszuwählen. Zusätzlich müssen zwei weitere Anfragen an die externe API gestellt werden, um ein Angebot für einen Hinflug als auch einen Rückflug zu erhalten, die Teil des Reisepaketes sein sollen. Aufgrund hoher Systemlasten des externen Dienstes, kann es vorkommen, dass eine API-Anfrage erst recht spät oder sogar fehlerhaft antwortet (vgl. HTTP-Antwortcodes der Travel API in Anhang A.3.1). Das Reisebuchungssystem muss mit solchen Ausfällen umgehen können, indem es ein Retry-Verfahren implementiert und nach maximal zwei fehlerhaften Anfragen eine manuelle Reisepaketerstellung durch einen Mitarbeiter des Reiseunternehmens anstoßt. Der Mitarbeiter hat dann die Möglichkeit über eine Weboberfläche fehlende Reisepaketdaten anzugeben. Im Rahmen des Retry-Verfahrens soll außerdem zwischen den ungültigen Aufrufen mindestens drei Sekunden gewartet werden bevor eine erneute Anfrage versendet wird.

Nachdem die Hotel- und Flugangebote für das Reisepaket eingeholt sind, wird der Gesamtpreis

kalkuliert. Dieser setzt sich aus den Angebotspreisen abzüglich eines dynamischen Rabatts, der von mehreren Faktoren abhängig ist, zusammen. Zu den Faktoren zählt unter anderem die Reisedauer, die Anzahl an Reisenden und ob es sich um eine Premiumreise handelt. Der Rabatt kann maximal 30 % betragen. Je fünf Prozent Rabatt werden gegeben, wenn die Reise entweder für mehr als drei Personen gebucht wurde oder die Reisedauer mehr als fünf Tage beträgt. Weitere fünf Prozent kommen hinzu, wenn es sich bei der Anfrage um eine Premiumreise handelt. Einen zusätzlichen Bonus von 15 % Rabatt erhält ein Kunde außerdem für ein bestimmtes, ausgewähltes Reiseziel, das derzeitig im Angebot ist. Sobald das Reisepaket vollständig erstellt ist und der Preis berechnet wurde, wird dieses dem Kunden zur Verfügung gestellt.

3. Reise buchen

Nachdem ein Angebot für ein Reisepaket erstellt wurde, kann der Kunde dieses innerhalb von drei Tagen über ein Webportal einsehen und buchen. Für die Buchung wird der Kunde zunächst dazu aufgefordert seine Kontakt- und Zahlungsinformationen anzugeben. Darunter fallen Daten wie Name, Email, Alter, Addresse, Land und Kreditkartennummer. Sobald er diese Angaben getätigt hat, kann er die Buchung abschließen, was dazu führt, dass diese vom Reisebuchungssystem weiter bearbeitet wird. Sollte der Kunde ein Reisepaket nicht innerhalb der Zeitspanne von drei Tagen buchen, läuft das Angebot ab und es ist keine Buchung mehr möglich. Ist der Kunde mit dem für ihn erstellten Reisepaket nicht zufrieden, hat er außerdem die Möglichkeit das Angebot abzulehnen.

4. Reisebuchung prüfen

Geht beim Reiseunternehmen eine neue Reisebuchung durch den Kunden ein, werden zunächst die Kundendaten in dem unternehmensweiten CRM-System (vgl. Kapitel 4.2.1) und dem Billing System (vgl. Kapitel 4.2.2) abgespeichert. Zeitgleich wird dem Kunden eine Bestätigungsmail für den Buchungseingang gesendet und eine Kundenprüfung mit Hintergrundcheck durchgeführt. Der Hintergrundcheck basiert auf den Kundendaten und kalkuliert einen Score zwischen 0 und 100, der angibt, wie vertrauenswürdig ein Kunde ist. Erreicht der Score einen Wert, der höher als 50 ist, muss ein Mitarbeiter des Reiseunternehmens die Reisebuchung manuell prüfen und freigeben. Lehnt der Mitarbeiter die Reisebuchung aus bestimmten Gründen ab, wird eine Absagemail an den Kunden versendet und die Reisebuchung abgebrochen.

5. Reisebuchung abwickeln

Falls keine Buchungsprüfung erforderlich ist oder die Reisebuchung akzeptiert wurde, kann im nächsten Schritt die eigentliche Buchung der Reise abgewickelt werden. Die Buchungsabwicklung erfordert dabei, dass mehrere Aktionen sequenziell durchgeführt werden:

1. **Hotelbuchung**: Zunächst erfolgt die Buchung des Hotelangebots über die Travel API. Dabei kann es vorkommen, dass das Hotelangebot nicht mehr gültig ist, was letztendlich zu einem Abbruch der Reisebuchung führt.

2. **Flugbuchung**: Nach erfolgreicher Hotelbuchung werden der Hin- und Rückflug über die Travel API gebucht. Auch hier kann es ähnlich wie bei der Hotelbuchung möglich sein, dass die Flugangebote nicht mehr verfügbar sind. Tritt ein solcher Fehlerfall ein, soll die Hotelbuchung wieder rückgängig gemacht werden.

3. **Zahlungsabwicklung**: Sobald sowohl Hotel als auch die Flüge erfolgreich gebucht sind, soll das Bankkonto des Kunden mit dem Preis des Reisepaketes belastet werden. Die

Abbuchung erfolgt durch die externe Payment API (vgl. Kapitel 4.2.6). Dabei kann es vorkommen, dass die Kreditkarte des Kunden abgelaufen ist, was zu einem Buchungsfehler führt. Ist es aufgrund dessen nicht möglich den Zahlungsbetrag einzuholen, soll das Hotel und die Flüge wieder storniert und die Reisebuchung abgebrochen werden.

Die drei genannten Schritte der Buchungsabwicklung hängen logisch miteinander zusammen. Schlägt ein Schritt fehl, so sollen alle vorherigen Aktionen rückgängig gemacht werden und der Kunde per Mail über den Reisebuchungsfehler informiert werden. Ein klassisches Beispiel das vermieden werden möchte ist, dass der Flug und das Hotel zwar gebucht wurden, aber der Geldbetrag aufgrund eines Verarbeitungsfehlers nicht vom Kundenkonto eingezogen werden konnte. In einem solchem Fall soll deshalb sowohl die Flug- als auch Hotelbuchung wieder storniert werden und ein Abbruch der Reisebuchung erfolgen, da der Kunde nicht gezahlt hat. Ziel ist es dabei Inkonsistenzen innerhalb des Systems zu vermeiden. Aus diesem Grund sollen die Schritte im Rahmen einer verteilten Transaktion abgewickelt werden, in der entweder alle Aufrufe erfolgreich abschließen oder keiner davon. Werden alle Buchungsschritte erfolgreich durchlaufen, soll an den Kunden eine Buchungsbestätigung per Mail versendet werden.

6. Reise antreten

Bevor der Kunde die Reise antritt, hat dieser nach fehlerfreier Buchungsabwicklung die Möglichkeit, bis zu dem Zeitpunkt von sieben Tagen vor Reisebeginn die Reise wieder zu stornieren. Eine Stornierung führt dazu, dass alle in Punkt fünf genannten Buchungen rückgängig gemacht werden müssen. Der Kunde muss das Geld zurückerstattet bekommen und die Flug- und Hotelbuchung muss annulliert werden. Nach Abschluss der Stornierung sollte der Kunde zusätzlich eine Stornierungsbestätigung per E-Mail erhalten.

Ein weiterer Service, den das Reiseunternehmen bis sieben Tage vor Reisebeginn anbietet, ist die Reisewarnmeldung. Reisewarnungen werden durch das Travel Warning System (vgl. Kapitel 4.2.3) bekannt gegeben und beziehen sich immer auf ein bestimmtes Reiseziel. Wenn eine neue Warnmeldung eintrifft, soll geprüft werden ob aktive Buchungen zu diesem Reiseziel bestehen. Wenn dem so ist soll der Kunde per Mail über die Reisewarnung benachrichtigt werden. Unter einer aktiven Reisebuchung versteht sich dabei eine Buchung die bereits erfolgreich abgewickelt wurde und im Zustand verweilt in dem bis zu sieben Tage vor Reisebeginn gewartet wird.

4.3.2. Technische Systemanforderungen

Aus den in Abschnitt 4.3.1 präsentierten Schritten des Reisebuchungsprozesses lassen sich technische, nicht-funktionale Anforderungen an das Reisebuchungssystem ableiten. Die einzelnen Systemanforderungen sind in Tabelle 4.1 aufgelistet und unterteilen sich in sechs Kategorien: **Integration, Resilienz, Auditing, Datenkonsistenz, Performance** und **Flexibilität**. Die adäquate Umsetzung des Reisebuchungsprozesses kann so bei den Implementierungen zusätzlich anhand der Einhaltung der Systemanforderungen gemessen werden. Im folgenden werden die einzelnen technischen Anforderungen und ihre Relevanz in Bezug auf den Reisebuchungsprozess vorgestellt.

Integration in bestehende Systemlandschaft

Das entwickelte System soll mit Systemen einer bestehenden IT-Landschaft kommunizieren. Dabei ist mit einer Heterogenität was die Kommunikationsprotokolle der bestehenden Dienste

Anforderung	Beschreibung	Kategorie
Integration in bestehende Systemlandschaft	Das entwickelte System kommuniziert mit Systemen der bestehenden Anwendungslandschaft	Integration
Integration externer Dienste	Das entwickelte System kommuniziert mit externen Diensten	Integration
Technische Resilienz	Auftretende technische Fehler im Rahmen der Verarbeitung können spezifisch behandelt werden (z.B. Systemausfall, Systemüberlastung)	Resilienz
Fachliche Resilienz	Auftretende fachliche Fehler im Rahmen der Verarbeitung können spezifisch behandelt werden (z.B. ungültige Kreditkarte, inkonsistente Daten)	Resilienz
Fehleridentifikation	Das System sollte Möglichkeiten bieten, um im Fehlerfalle die verursachende Aktion identifizieren zu können, um einen besseren Eingriff zu ermöglichen	Auditing
Nachvollziehbarkeit	Das System sollte Möglichkeiten bieten die ausgeführten Aktionen und Datenänderungen im System nachvollziehen zu können	Auditing
Transaktionalität	Verteilte Transaktionen können vom System ausgeführt werden	Datenkonsistenz
Parallelität	Das System kann Aufgabenschritte parallel durchführen	Performance
Reaktivität	Das System kann auf eintretende Ereignisse flexibel regieren und dabei ggfs. die aktuelle Verarbeitung unterbrechen	Flexibilität

Tabelle 4.1.: Technische Systemanforderungen der Beispielimplementierung [Eigene Darstellung]

betrifft zu rechnen. Konkret soll das Reisebuchungssystem mit der in Kapitel 4.2 definierten Anwendungslandschaft integriert werden. Dies erfordert die Unterstützung von SOAP-Schnittstellen und Datei-basierter Kommunikation über CSV-Dateien.

Integration externer Dienste

Neben den internen Systemen soll das Reisebuchungssystem auch mit externen Diensten interagieren. Zur Erbringung des Reisebuchungsprozesses ist eine Kommunikation mit der externen Travel API und Payment API über REST notwendig.

Technische Resilienz

Die technische Resilienz beschreibt, dass das System in der Lage sein soll, technische Fehler, die während der Verarbeitung auftreten, spezifisch behandeln zu können. Anstatt im Fehlerfalle einen undefinierten oder ungültigen Systemzustand einzunehmen, greifen definierte Ausnahmebehandlungen. Bei einer Reisebuchung kann beispielsweise während der Reisepaketgenerierung

ein technischer Fehler in Form eines Systemausfalls, einer Überlastung oder fehlerhaften Antwort der Travel API auftreten. Diese sollen behandelt werden können und nicht dazu führen, dass der Kunde keine Reise buchen kann bzw. seine Anfrage nicht bearbeitet wird. Die technische Resilienz geht damit auch indirekt mit der Anforderung an eine Fehleridentifikation und Reaktivität des Systems einher.

Fachliche Resilienz

Die fachliche Resilienz beschreibt analog zur technischen Resilienz, dass auftretende fachliche Fehler im Rahmen der Verarbeitung spezifisch behandelt werden können. Ein fachlicher Fehler kann beispielsweise durch eine ungültige Kreditkarte oder fehlerhaften Reisepaketgenerierung hervorgerufen werden.

Fehleridentifikation

Das System sollte Möglichkeiten bereitstellen, im Fehlerfall die verursachende Aktion identifizieren zu können. Sollte während der Laufzeit ein unvorhersehbarer Fehler auftreten, kann so schnell darauf reagiert werden.

Nachvollziehbarkeit

Ausgeführte Aktionen und Schritte sollen im System nachvollziehbar sein. Es soll festgestellt werden können welche unterschiedlichen Pfade ein Prozess durchlaufen hat und in welchem Zustand sich dieser befindet. Mit solchen Informationen können wichtige Fragestellungen für das Business beantwortet werden, wie z. B. die Anzahl abgelehnter oder stornierter Reisebuchungen.

Transaktionalität

Eine Transaktion ist eine Menge von Aktionen oder Schritten, in der entweder jede Aktion vollständig abgeschlossen wird oder keine davon [vgl. Starke, 2020, S. 204]. Das entwickelte System soll in der Lage sein verteilte Transaktionen auszuführen. Dies ist wichtig um eine Datenkonsistenz innerhalb des Systems zu wahren. Eine Transaktion im Kontext der Reisebuchung stellt beispielsweise die Buchungsabwicklung dar. Tritt ein Fehler bei der Hotel- oder Flugbuchung auf oder ist die Kreditkarte des Kunden abgelaufen, so sollen alle vorangegangenen, ausgeführten Schritte wieder annulliert werden, sodass das System wieder in einen konsistenten Zustand kommt.

Parallelität

Um möglichst minimale Durchlaufzeiten zu gewährleisten, soll das System Aufgabenstränge parallel durchführen können. Eine schnelle Abarbeitung der Reisebuchung erhöht die Kundenzufriedenheit. Als Beispiel kann hier die Reisepaketgenerierung genannt werden. Um eine möglichst schnelle Generierung zu gewährleisten, können die Hotelangebote gleichzeitig bzw. parallel mit den Flugangeboten angefragt werden.

Reaktivität

Reaktivität bezeichnet die Fähigkeit eines Systems, dynamisch auf unterschiedliche, oft auch unvorhersehbare Ereignisse reagieren zu können [vgl. Escoffier und Finnigan, 2021, S. 4]. Im

Hinblick auf das Reisebuchungssystem bedeutet das, dass eine gewisse Flexibilität in der Verarbeitung gewährleistet werden muss. Dazu zählt die Fähigkeit, auf eingehende Ereignisse wie z. B. eine Reisewarnmeldung reagieren zu können. Auch das Zeitverhalten ist zu berücksichtigen. Der Geschäftsprozess erfordert es an mehreren Stellen in Wartezustände überzugehen und nach Ablauf des Zeitintervalls Funktionen anzustoßen. Das ist beispielsweise nach erfolgreicher Reisebuchung der Fall, wo bis zu sieben Tage vor Reisebeginn auf eintretende Ereignisse wie eine Stornierung oder Reisewarnmeldung gewartet werden muss.

4.4. Benutzeroberflächen

Die Prozessanforderungen aus Kapitel 4.3.1 implizieren eine Möglichkeit zur Interaktion eines Benutzers mit dem Reisebuchungssystem. Als Rollen sind hier der Kunde und der Mitarbeiter des Reiseunternehmens genannt. Die Interaktion mit dem System wird in den Beispielimplementierungen über eine zentrale Webapplikation (Travel Booking UI) gesteuert, die mit dem JavaScript Framework React in Kombination mit der bewährten CSS-Bibliothek „Bootstrap“ [Bootstrap Team, 2021] umgesetzt ist. Die Webapplikation kommuniziert dabei mit HTTP-basierten REST-Schnittstellen einzelner Services der Travel Booking Backend Implementierungen (vgl. Abbildung 4.3).

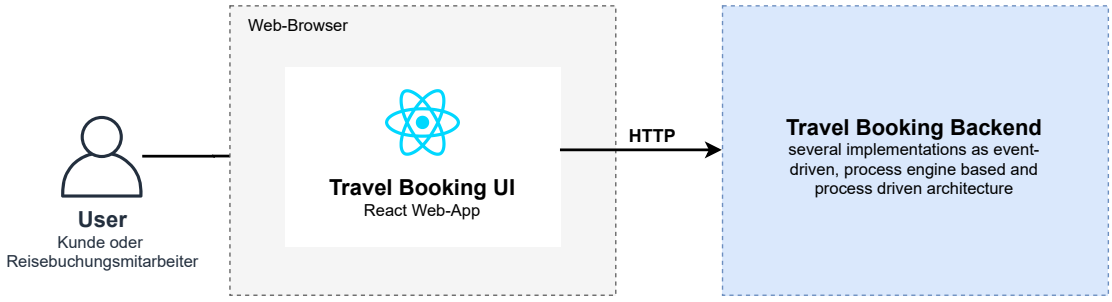

Abbildung 4.3.: Darstellung der Benutzerinteraktion mit dem Travel Booking System [Eigene Darstellung]

Damit ein Benutzer also Aktionen im Rahmen des Reisebuchungsprozesses tätigen kann, fungiert das Travel Booking User Interface als Vermittler zwischen Benutzer und System. Die Webapplikation umfasst insgesamt vier Benutzeroberflächen. Nachfolgend werden die einzelnen Oberflächen vorgestellt.

Benutzeroberfläche zur Anfrage eines neuen Reisepaketes

Abbildung 4.4 zeigt die Benutzeroberfläche zur Anfrage eines neuen Reisepaketes. Hier kann der Kunde seine Wunschdaten für die Reise angeben. Mit Abschicken der Anfrage wird der eigentliche Reisebuchungsprozess initial gestartet.

Benutzeroberfläche zum Einsehen und Buchen eines Reisepaketangebots

Abbildung 4.5 stellt die Benutzeroberfläche zum Einsehen und Buchen eines angefragten Reisepaketes dar. Die angefragten Reisepakete werden auf dieser Seite aufgelistet. Der Kunde hat dabei die Möglichkeit jedes der Reisepaketangebote entweder abzulehnen oder zu buchen. Tätigt der Kunde keine der genannten Aktionen, läuft das Reisepaket ab und beide Buttons

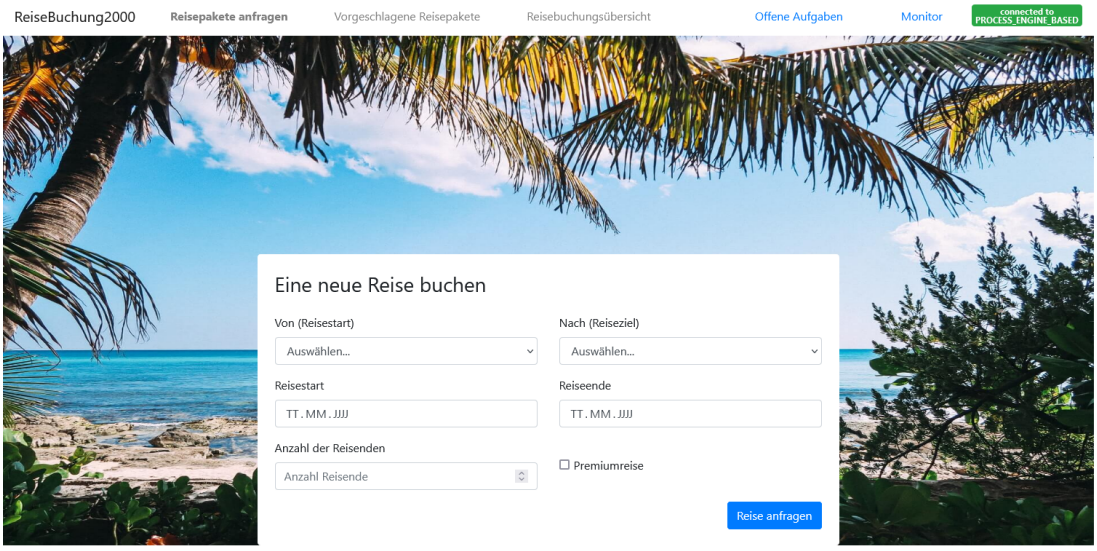

Abbildung 4.4.: Benutzeroberfläche zur Anfrage eines neuen Reisepaketes [Eigene Darstellung]

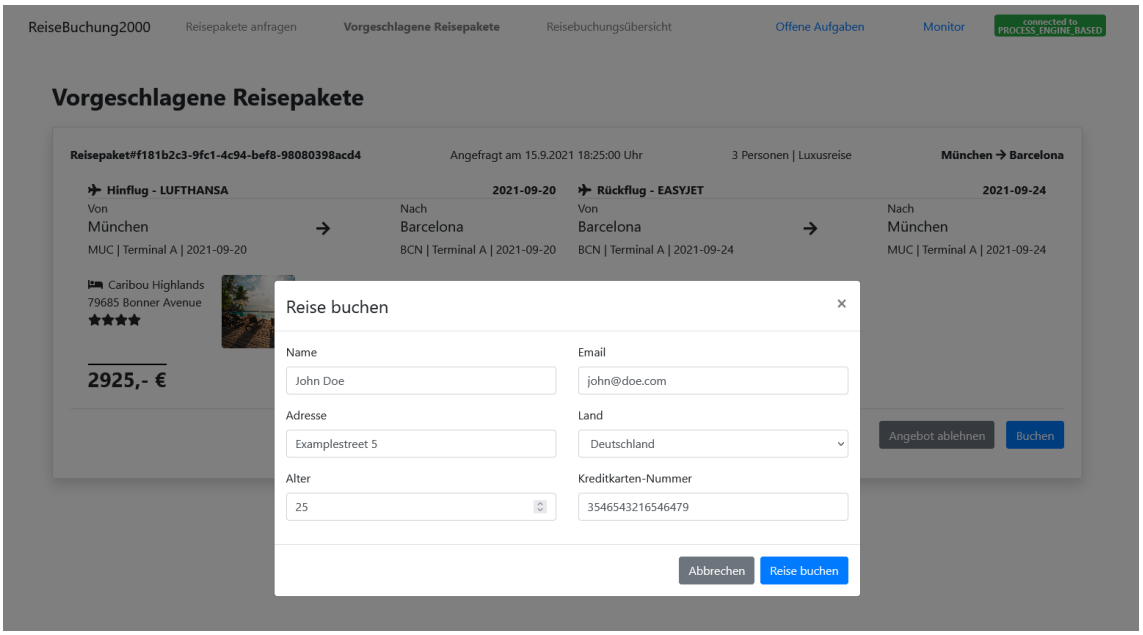

Abbildung 4.5.: Benutzeroberfläche zum Einsehen und Buchen eines Reisepaketangebots [Eigene Darstellung]

zum Buchen und Ablehnen werden durch eine Statusmeldung ersetzt. Entscheidet sich der Kunde für die Buchung eines Pakets erscheint davor ein Modal indem er aufgefordert wird seine Kundendaten anzugeben.

Benutzeroberfläche zum Einsehen und Stornieren von getätigten Reisebuchungen

Gebuchte Reisepakete werden in der Benutzeroberfläche zur Reisebuchungsübersicht angezeigt, die in Abbildung 4.6 zu sehen ist. Dort kann außerdem der aktuelle Status der einzelnen Reisebuchungen nachverfolgt werden und die Buchung gegebenenfalls über einen zusätzlichen Button storniert werden. Die unterschiedlichen Statustypen einer Reisebuchung sind dem Da-

tenmodell aus Kapitel 4.5 zu entnehmen.

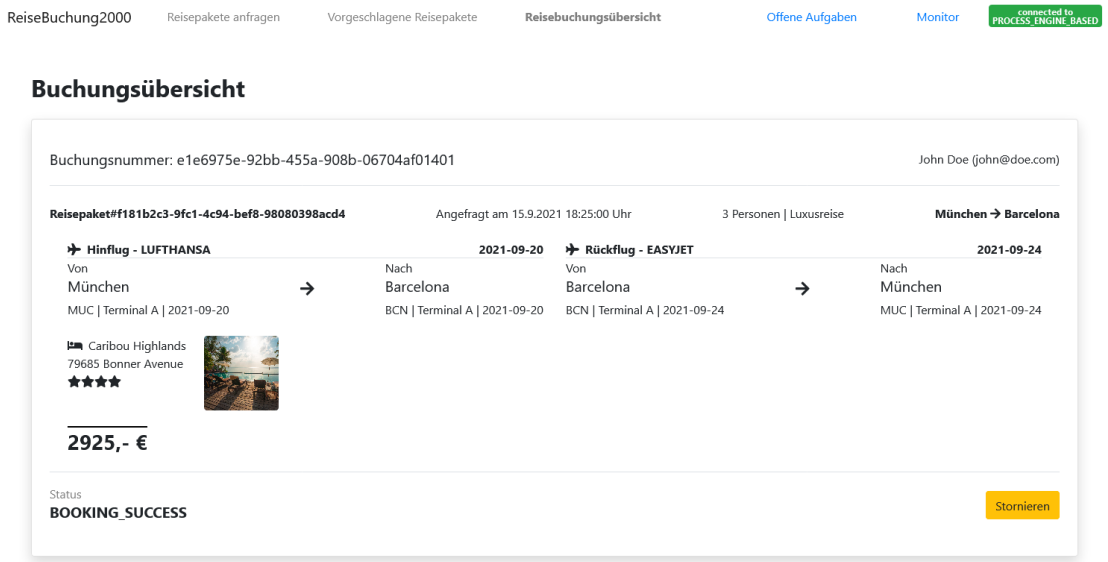

Abbildung 4.6.: Benutzeroberfläche zum Einsehen und Stornieren von getätigten Reisebuchungen [Eigene Darstellung]

Benutzeroberfläche zum Einsehen und Abschließen offener Aufgaben als Mitarbeiter des Reiseunternehmens

Alle offenen Aufgaben die für einen Mitarbeiter des Reisebuchungsunternehmens im Rahmen der Abwicklung des Reisebuchungsprozesses anfallen, werden in der Webapp unter dem Reiter „Offene Aufgaben“ angezeigt. Wie in Abbildung 4.7 zu sehen ist, wird dabei zwischen zwei Aufgabentypen unterschieden: „Reisepaket erstellen“ und „Reisebuchung prüfen“. Steht eine Aufgabe zum Erledigen an, wird diese auf der linken Seite als Box dargestellt und kann dann zur Abarbeitung ausgewählt werden. Das Formular zur Bearbeitung der Aufgabe wird dann auf der rechten Seite dem Benutzer angezeigt.

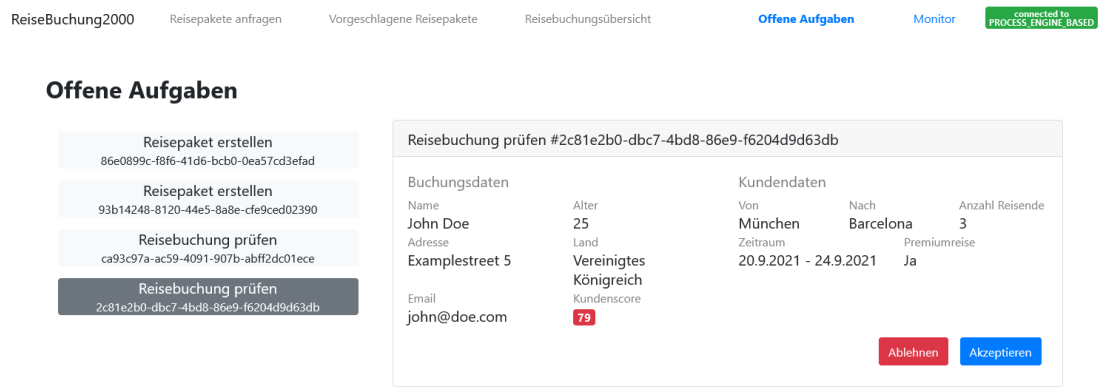

Abbildung 4.7.: Benutzeroberfläche zum Einsehen und Abschließen offener Aufgaben als Mitarbeiter des Reiseunternehmens [Eigene Darstellung]

4.5. Datenmodell

Für die eigentliche Umsetzung des Reisebuchungssystem wird ein einheitliches Datenmodell (siehe Abbildung 4.8) definiert, das den Anforderungen aus Kapitel 4.3.1 gerecht wird. Das Datenmodell hat keinen Anspruch auf Vollständigkeit, sondern dient dazu einen Überblick über die wichtigsten Datenobjekte innerhalb des Systems zu geben. Generell hat ein Kunde

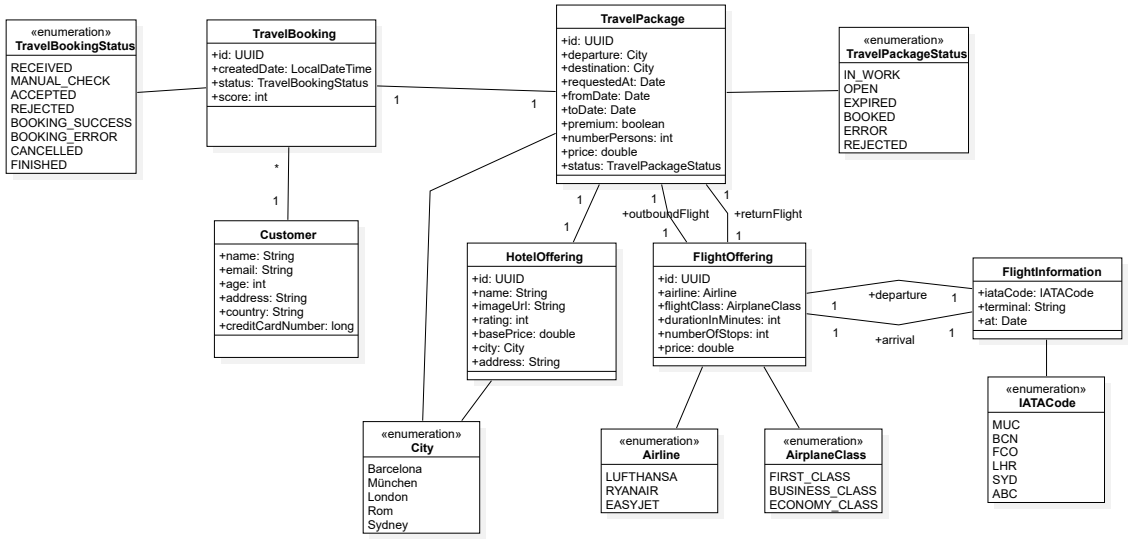

Abbildung 4.8.: Datenmodell des Reisebuchungssystems [Eigene Darstellung]

die Möglichkeit mehrere Reisebuchungen zu tätigen. Eine Reisebuchung ist durch die Klasse `TravelBooking` modelliert und signalisiert durch den `TravelBookingStatus` den eigentlichen Zustand in dem sich eine Buchung befindet. Eine Reisebuchung ist immer auf genau ein Reisepaket (`TravelPackage`) bezogen. Ein Reisepaket setzt sich dabei aus genau einem Hotelangebot (`HotelOffering`) und einem Hinflug (`outboundFlight`) und Rückflug (`returnFlight`), die über die Klasse `FlightOffering` repräsentiert sind, zusammen. Ein `FlightOffering` enthält Daten zum Abflug und zur Ankunft eines Flugangebots. Diese sind dargestellt durch die Assoziation zwischen den Klassen `FlightOffering` und `FlightInformation` . Des Weiteren sind einige Enumerationen definiert, wie z. B. `City` , `Airline` oder `AirplaneClass` . Diese beschränken sich bewusst auf nur einige wenige selektierte Einträge, um den Umfang der Auswahlmöglichkeiten einzuschränken.

Das Datenmodell beschreibt die Objekte des Reisebuchungssystems aus einer gesamtheitlichen Sicht, um eine bessere Verständlichkeit zu gewährleisten. Je nach Best Practices der späteren Implementierungen kann die Verantwortlichkeit für die Implementierung bestimmter Domänenobjekte jedoch auch auf unterschiedliche Services verteilt sein. Zur Einfachheit ist ein Java Modul namens `de.thi.travel.common` implementiert, das einige Datenobjekte zentral kapselt und den einzelnen Services des Reisebuchungssystems zur Verfügung stellt. Die Verwendung eines solchen zentralen Moduls ist innerhalb realer Microservice-Implementierungen nicht zu empfehlen, da es zu einer höheren Kopplung und damit erhöhtem Abstimmungsaufwand zwischen den Teams führen kann. Jegliche Datenbankoperationen zum Abspeichern der Datenobjekte werden innerhalb des Spring Frameworks über die Jakarta Persistence API (JPA) durchgeführt. Als Datenbank wird dabei entweder die „H2“ In-Memory-Datenbank [Müller, 2021] oder „PostgreSQL“ [The PostgreSQL Global Development Group, 2021] verwendet.

4.6. Ereinigsgesteuerte Implementierung

4.6.1. Zielsetzung und Vorgehensweise der Implementierung

Zielsetzung der Implementierung

Zielsetzung der Implementierung ist es, die in Kapitel 4.3 gestellten Prozess- und Systemanforderungen unter Anwendung einer ereignisgesteuerten Architektur (vgl. Kapitel 2.4.3), die später auch abgekürzt als EDA bezeichnet wird, umzusetzen. Aktuelle und typische Best Practices ereignigsgesteuerter Architekturen sind dabei zu berücksichtigen, sodass ein repräsentativer Vergleich mit anderen Architekturen möglich wird.

Vorgehensweise zur Implementierung

Kapitel 2.4.3 hat bereits das Grundkonzept ereignisgesteuerter Architekturen vorgestellt: Services kommunizieren durch das asynchrone Senden und Empfangen von Ereignissen miteinander. Bellemare stellt klar, dass ein Event dabei nicht nur als einfaches Signal zur Kommunikation zwischen Parteien dient, sondern als Übermittler jeglicher Art von Daten. Events bilden damit also das Fundament zur Kommunikation und den Austausch von Daten und damit auch die Grundlage zur Implementierung dieses Architekturmusters [vgl. Bellemare, 2020, S. 14]. Domänen Events werden üblicherweise über Workshop-Methoden des Domain-Driven Designs identifiziert, wie beispielsweise über die in Kapitel 2.2.1 vorstellte Methodik des Event Storming. Da eine detaillierte Beschreibung aller aus den Prozessanforderungen abgeleiteten Domain Events den Rahmen dieser Arbeit jedoch überschreiten würde, werden diese als gegeben angesehen.

Häufig wird die Architektur unter Verwendung von Microservices implementiert. Microservices eignen sich deshalb besonders gut, weil sie inhärent auf die Unabhängigkeit und lose Kopplung untereinander abzielen: Eine Eigenschaft die durch asynchrone, ereignisgesteuerte Kommunikation bekräftigt wird. Eine direkt Kopplung, wie sie durch synchrone Request-Response APIs oder durch das Zugreifen auf eine gemeinsame Datenbank entsteht, kann durch den Austausch von Events umgangen werden [vgl. Bellemare, 2020, S. 14]. Bellemare führt hier den Begriff eines „Event-Driven Microservice“ [Bellemare, 2020, S. 21] ein. Damit ist ein Microservice gemeint, der innerhalb eines Bounded Context operiert und basierend auf ein oder mehreren Input-Events Geschäftslogik ausführt. Als Ergebnis publiziert dieser dann gegebenenfalls wieder ein oder mehrere Output-Events nach außen, die von anderen Microservices zur Weiterverarbeitung konsumiert werden können. Die Kommunikation zwischen den Event-Driven Microservices erfolgt dabei ausschließlich asynchron über Events, während innerhalb eines solchen Microservice auch synchrone Request-Response Aufrufe erfolgen können [vgl. Bellemare, 2020, S. 21]. Im Kontext der Abbildung des Reisebuchungsprozesses bedeutet das also, dass es nicht den einen Service gibt, der den gesamten Geschäftsprozess erbringt, vielmehr existieren mehrere Services, von denen jeder einen Teil oder Ausschnitt des Gesamtprozesses verantwortet und implementiert. Erst die Zusammenarbeit zwischen den Services, die aus dem Senden fachlicher Ereignisketten besteht, ermöglicht die Abbildung des Gesamtprozesses.

Stopford spricht bei dieser Art der Kommunikation auch von dem sogenannten „Event Collaboration Pattern“ [Stopford, 2018, S. 39]. Überwiegend findet die Kollaboration dabei ohne eine zentrale Instanz statt, die den Prozess verwaltet und steuert. Solche Systeme werden auch als Choreografie-basierte Systeme bezeichnet. Im Gegensatz dazu kann der Geschäftsprozess innerhalb eines Systems jedoch auch durch eine zentrale Instanz kontrolliert werden, wie

bereits in dem Grundlagenkapitel zu Microservices genannt wurde (vgl. Kapitel 2.4.2). Dann findet die Kollaboration zwischen den Services durch eine Orchestrierung statt. Bellemare bezeichnet die beiden Varianten auch als „Choreography Pattern“ [Bellemare, 2020, S. 136] und „Orchestration Pattern“ [Bellemare, 2020, S. 139]. Bei der Choreografie bestimmt also die Beziehung zwischen den Services den Geschäftsprozess, während bei der Orchestrierung einzelne Services den Prozessablauf und das Zusammenspiel steuern [vgl. Bellemare, 2020, S. 136 f.]. Da beide Patterns unterschiedliche Ansätze haben und damit unterschiedliche Auswirkungen auf die Gesamtarchitektur ausüben können, ist es notwendig beide Ansätze im Rahmen der ereignisgesteuerten Umsetzung zu implementieren. Im späteren Vergleich können so die Vor- und Nachteile beider Arten untersucht und gegenübergestellt werden.

Zusammengefasst kann das Vorgehen zur Implementierung der ereignisgesteuerten Architektur folgendermaßen abgeleitet werden: Das Reisebuchungssystem wird auf Basis mehrerer Event-Driven Microservices implementiert, die nach dem Event Collaboration Pattern miteinander interagieren. Jeder Service ist dabei für einen bestimmten Teil des Gesamtprozesses innerhalb eines Bounded Context verantwortlich. Um den Geschäftsprozess zu erbringen, werden die zwei Varianten zur Kollaboration zwischen den Services berücksichtigt, weshalb zwei separate Implementierungen — eine basierend auf dem Choreography Pattern und die zweite unter Einsatz des Orchestration Pattern — vorgenommen wird.

Nachrichtenversand an Apache Kafka mit Spring Cloud Stream

In Abschnitt 2.3.2 wurde bereits Apache Kafka und dessen Rolle als Message bzw. Event Broker für ereignisgesteuerte Architekturen vorgestellt. Da der Austausch von Ereignissen die Basis der Implementierung einer EDA darstellt, wird im Folgenden beispielhaft gezeigt, wie im Rahmen des Reisebuchungssystems der Nachrichtenaustausch mit Apache Kafka stattfindet.

In Kapitel 4.1 wurde die Entscheidung getroffen, dass alle Systemkomponenten mit Java und dem Spring Framework umgesetzt werden. Aus diesem Grund werden alle Event-Driven Microservices im Rahmen der EDA Implementierungen als Spring-basierte Microservices implementiert. Spring bietet mit Spring Cloud Stream ein Framework an, das den Umgang mit Messaging Systemen wie Apache Kafka oder RabbitMQ enorm erleichtert. Seit Version 3.x verwendet Spring Cloud Stream standardmäßig das funktionale Programmiermodell von Spring Cloud Function zur Definition von sogenannten „Message Handlern“. Um Nachrichten konsumieren und senden zu können, ist es dabei notwendig, eine Java Funktion zu schreiben und diese mit der Annotation `@Bean` als Spring Bean bereitzustellen. Listing 4.2 zeigt die Definition eines beispielhaften Message Handlers, der eine Eingangsnachricht vom Typ String erhält, diese transformiert und das Ergebnis wieder als Output versendet.

```
1 @Bean
2 public Function<String, String> uppercase() {
3   return s -> s.toUpperCase();
4 }
```

Listing 4.2: Message Handler Funktion im Kontext von Spring Cloud Stream [Anandan u. a., 2021]

Zur Definition einer wie in Listing 4.2 gezeigten funktionalen Spring Bean muss diese vom Typ `java.util.function.[Supplier|Function|Consumer]` sein. Jedem Typ ist dabei eine unterschiedliche Semantik zugeordnet:

- **Consumer**: Consumer Funktionen werden aufgerufen, wenn Nachrichten von dem Messaging System empfangen werden.

- **Supplier**: Supplier Funktionen produzieren Nachrichten an das Messaging System. Dies geschieht standardmäßig in einem Intervall von einer Sekunde.
- **Function**: Das Function Interface wird für Methoden verwendet, die sowohl als Konsument und Produzent von Nachrichten agieren.

Die in Listing 4.2 vorgestellte Spring Bean ist vom Typ `java.util.function.Function`, da die Methode sowohl Daten vom Messaging System konsumiert als auch neue Nachrichten produziert. Die eigentliche Integration mit den externen Messaging Systemen geschieht über sogenannte „Destination Binders“, die die Message Handler aufrufen. Diese Binder kümmern sich um das Routing von Eingangs- und Ausgangsnachrichten und um Datentyp-Mappings. Sie fungieren also als Brücke zwischen den externen Messaging Systemen und dem Applikationscode (vgl. Abbildung 4.9).

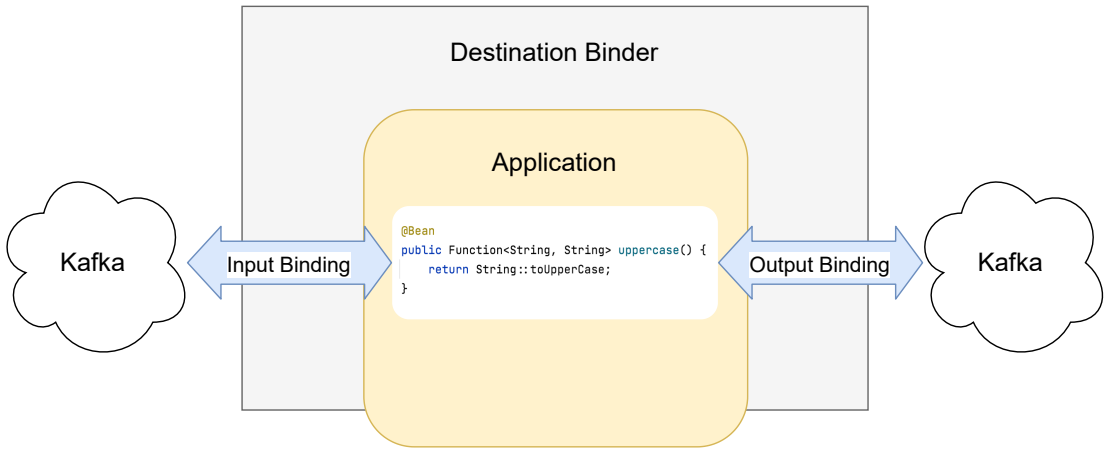

Abbildung 4.9.: Spring Cloud Stream Destination Binder als Brücke zur Anbindung externer Messaging Systeme [In Anlehnung an Anandan u. a. 2021]

Um die Message Handler Funktionen nun einem Topic oder einer Queue zuordnen zu können, muss ein Binding erstellt werden. Dies kann bei Spring Boot beispielsweise in der Applikationskonfiguration angelegt werden (z. B. `application.yml` oder `application.properties`). Dabei wird zwischen Input und Output Bindings unterschieden. Deutlich wird dies anhand eines exemplarischen Bindings für die Funktion aus Listing 4.2, worin `my−input−topic` als Topic für das Input Binding und `my−output−topic` als Topic für das Output Binding konfiguriert wird:

```
1 spring.cloud.stream.bindings.uppercase−in−0.destination=my−input−topic
2 spring.cloud.stream.bindings.uppercase−out−0.destination=my−output−topic
```

Listing 4.3: Konfiguration eines exemplarischen Input und Output Bindings für eine Spring Cloud Stream Message Handler Funktion [In Anlehnung an Anandan u. a. 2021]

Die Konvention für die Namensgebung eines Input und Output Bindings bildet sich nach folgendem Schema [Anandan u. a., 2021]:

- Input Binding: - `<functionName> + −in− + <index>`
- Output Binding: - `<functionName> + −out− + <index>`

`in` und `out` definieren den Binding Typ und `index` gibt den jeweiligen Index der einzelnen Message Handler Funktionen an. Hat ein Binding nur eine definierte Message Handler Funktion, so liegt der Index für den Input und Output bei 0 [vgl. Anandan u. a., 2021].

4.6.2. Architektur nach dem Choreography Pattern

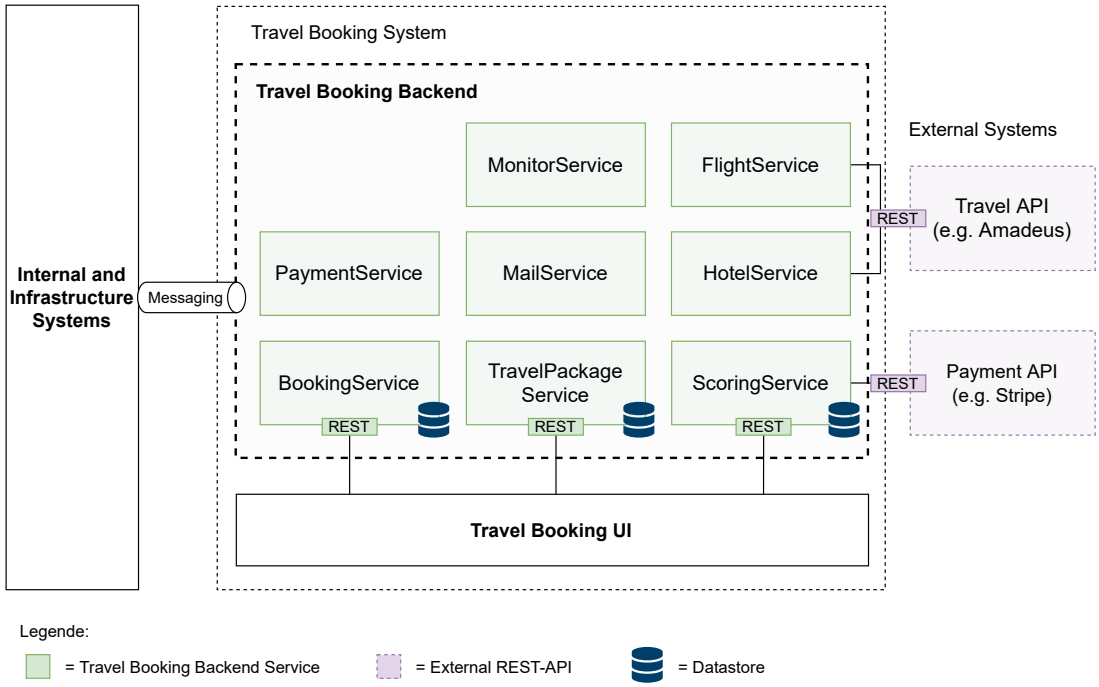

Abbildung 4.10.: Architekturschaubild des Reisebuchungssystems nach einer ereignisgesteuerten Architektur entsprechend dem Choreography Pattern [Eigene Darstellung]

Schaubild 4.10 zeigt die ereignisgesteuerte Architektur des Reisebuchungssystems nach dem Choreography Pattern. Die Architektur umfasst acht Spring Boot Backend Services, die als Event-Driven Microservices implementiert sind. Die Abwicklung des Reisebuchungsprozesses in seiner Gesamtheit geschieht durch die Kooperation und den Austausch von insgesamt 24 Domain Events, wobei jeder einzelne Microservice selbstständig für die Erbringung eines Teilaspekts des Geschäftsprozesses verantwortlich ist. Entsprechend dem von Bellemare und Stopman empfohlenen „Single Writer Principle“ [Bellemare, 2020, S. 28; Stopford, 2018, S. 105] existiert für jedes Event genau ein Microservice, der für das Produzieren des Events verantwortlich ist. Dadurch ist die Hoheit zum Verwalten der Ereignisstruktur innerhalb eines Services angesiedelt und die Quelle des Ereignisses immer eindeutig [vgl. Bellemare, 2020, S. 28; vgl. Stopford, 2018, S. 106]. Auch ist ein Topic bestenfalls immer nur einem Event-Typ bzw. Event-Schema zugeordnet, sodass keine Mischung mehrerer Event-Typen innerhalb eines Event-Streams entsteht. Ein typisches Anti-Pattern in Bezug darauf wäre es, einem Event ein Attribut namens „type“ zuzuweisen, das je nach Wert die Struktur und das Schema des Events ändert [vgl. Bellemare, 2020, S. 46 f.].

Ein weiterer Punkt, der als Best Practice für das Event Handling berücksichtigt wurde, ist, dass die einzelnen Events nicht nur einfache Indikatoren für Geschäftsereignisse sind. Die Ereignisse beinhalten alle relevanten Daten und Ergebnisse von potenziellen Berechnungen direkt als Payload mit dem Event. Das Event ist also die „Single Source of Truth“ für Konsumenten,

sodass diese für ihre weitere Verarbeitung nicht zusätzlich Daten von anderen Services abfragen müssen [vgl. Bellemare, 2020, S. 46; vgl. Bellemare, 2020, S. 51].

In Abbildung 4.10 ist zu erkennen, dass mehrere Services für die Datenhaltung der Geschäftsobjekte verantwortlich sind: Der „TravelPackageService“ speichert alle Informationen in Bezug auf Reisepakete ab. Der „BookingService“ verwaltet die Buchungs- und Kundendaten und der „ScoringService“ hält relevante Daten für die Buchungsprüfung. Essenzielle Geschäftsobjekte, wie beispielsweise die Reisebuchung (TravelBooking) oder das Reisepaket (TravelPackage), die im Kontext des Datenmodells vorgestellt wurden (vgl. Kapitel 4.5), werden über Events zwischen den Services ausgetauscht und gegebenenfalls redundant gespeichert, um die Kopplung innerhalb des Systems zu reduzieren. Die Daten der einzelnen Services und damit verbundene Geschäftsfunktionen werden per REST-API für das Travel Booking UI zur Verfügung gestellt, sodass der Benutzer mit dem System interagieren kann.

4.6.3. Architektur nach dem Orchestration Pattern

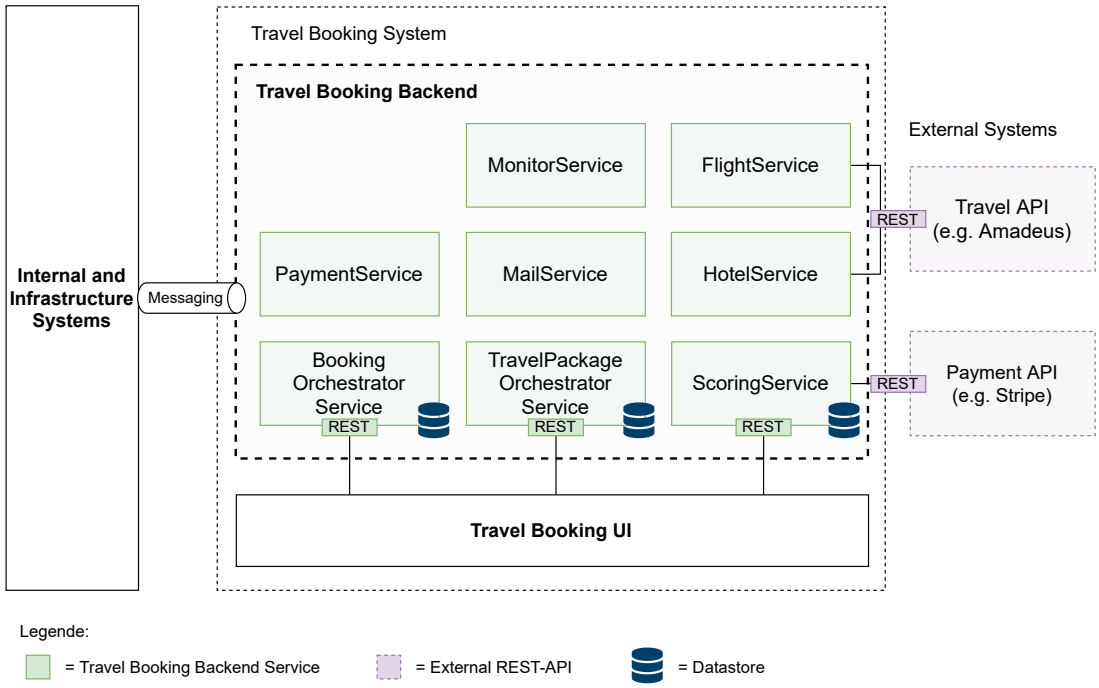

Abbildung 4.11.: Architekturschaubild des Reisebuchungssystems nach einer ereignisgesteuerten Architektur entsprechend dem Orchestrator Pattern [Eigene Darstellung]

Die Architektur nach dem Orchestration Pattern besteht analog zur Choreografie-basierten Implementierung aus insgesamt acht Spring Boot Services, die als Event-Driven Microservices implementiert sind. Im Vergleich zum Choreografie Ansatz existieren hier zwei Orchestrator Services: Der „BookingOrchestratorService“ ist für die Steuerung der Buchungsabwicklung und der „TravelPackageOrchestratorService“ für die Orchestrierung der Reisepaketerstellung verantwortlich. Beide Services verwalten zentral die Prozesslogik ihrer zugehörigen Domäne und steuern basierend darauf andere Event-Driven Microservices. Die eigentliche Businesslogik, die seitens der jeweiligen Event-Driven Microservices ausgeführt wird, wird nicht von den Orchestrator Services übernommen.

Die Orchestrierung erfolgt nach einem asynchronen Request-Response Pattern. Bellemare und Stopford schlagen dazu die Verwendung von sogenannten „Commands“ vor: Ein Command repräsentiert eine Aktion oder eine Anfrage, dass eine Operation von einem anderen Service ausgeführt werden soll [vgl. Bellemare, 2020, S. 139 f.; vgl. Stopford, 2018, S. 30 f.]. Meist wartet der Orchestrator Service nach Versenden eines Commands auf eine Antwort des Services in Form eines Events, sodass basierend auf der Antwort der gegebene Prozessfluss fortgesetzt werden kann. Die Implementierung des Reisebuchungssystems auf Basis der Orchestrierung umfasst insgesamt 11 Commands und 18 Events, die unter den Services ausgetauscht werden. Daran ist auch zu erkennen, dass eine Architektur nach dem Orchestration Pattern meist eine Mischung beider Kommunikationsstile (Orchestrierung über Commands und Choreografie mittels Events) ist.

4.6.4. Validierung der technischen Systemanforderungen

In Kapitel 4.3.2 wurden technische Systemanforderungen definiert, die von allen Implementierungen erfüllt sein müssen. Im Folgenden wird erläutert, inwiefern die einzelnen Anforderungen durch die beiden ereignisgesteuerten Implementierungen umgesetzt sind. Zur Visualisierung des Ereignisflusses werden an bestimmten Stellen Sequenzdiagramme eingesetzt. An dieser Stelle ist zu erwähnen, dass die zeitliche Abfolge der Ereignisse innerhalb der Sequenzdiagramme in der Realität unterschiedlich ausfallen kann, je nachdem wie lange die Verarbeitungszeiten der einzelnen Services sind. Auch ist die Parallelität im Falle wenn mehrere Services ein Ereignis gleichzeitig konsumieren nicht detailgetreu in den Sequenzdiagrammen dargestellt und dient nur einer exemplarischen Demonstration des groben Sequenzflusses.

Integration in bestehende Systemlandschaft

Das Integration System (vgl. Kapitel 4.2.4) dient als Adapter zur bestehenden Systemlandschaft im Rahmen der ereignisgesteuerten Architektur. Das Integration System konsumiert Ereignisse und stößt basierend darauf Integrationsaufgaben — wie beispielsweise den CSV-Export oder SOAP-Schnittstellen-Aufrufe — an. Im Kontext der konkreten Implementierungen ist das Integration System zur Integration mit dem bestehenden CRM und Billing System notwendig, damit Kundendaten abgespeichert werden können. Hierzu sind für beide Implementierungsvarianten Camel Routen definiert worden, die in folgenden Java Klassen implementiert sind:

```
1 de.thi.travel.integrationsystem.eda.SaveCustomerToSystemsRouteEDAChoreography
2 de.thi.travel.integrationsystem.eda.SaveCustomerToSystemsRouteEDAOrchestration
```

Integration externer Dienste

Die für die jeweilige Domäne zuständigen Event-Driven Microservices kommunizieren im Rahmen ihrer Verarbeitung direkt über REST mit den verfügbaren externen APIs (siehe Verbindung zwischen „FlightService“, „HotelService“ und „ScoringService“ mit externen Systemen in Abbildung 4.10 und 4.11). Da die Services mit dem Spring Framework implementiert sind, kann für die HTTP Kommunikation die Klasse `org.springframework.web.client.RestTemplate` verwendet werden. Listing 4.4 zeigt wie der Aufruf der Travel API zur Abfrage aller Hotelangebote für eine bestimmte Stadt unter Verwendung der Klasse `RestTemplate` aussieht.

```
1 HotelOfferingDTO[] resp = restTemplate.getForObject(travelApiUrl + "/offers/hotels/" +
      payload.getDestination(), HotelOfferingDTO[].class);
```

Listing 4.4: HTTP Anfrage zur Ermittlung der Hotelangebote mittels Spring RestTemplate

Technische Resilienz

Technische Resilienz zur Behandlung von Systemausfällen oder technischen Fehlern ist im Rahmen der ereignisgesteuerten Implementierung durch zweierlei Arten gegeben:

- **Retry-Verfahren**: Spring Cloud Stream bietet standardmäßig Funktionalitäten an um Nachrichten im Fehlerfalle mehrmals zuzustellen. Dieser Mechanismus kann über die zentrale Spring Konfigurationsdatei mit dem Parameter `consumer.maxAttempts` pro Message Handler konfiguriert werden [vgl. Anandan u. a., 2021]. Retry-Verfahren werden bei der Anfrage der Hotel- und Flugangebote von dem „HotelService“ und „FlightService“ eingesetzt, da wie in Kapitel 4.2 genannt wurde, mit Ausfällen bei der externen Travel API zur Angebotsgenerierung gerechnet werden kann (vgl. Anforderungskapitel 4.3.1). Über den Parameter `consumer.backOffInitialInterval` kann das Warteintervall zwischen den fehlerhaften Aufrufen konfiguriert werden. Pro Retry wird das Intervall zusätzlich um den in `consumer.backOffMultiplier` konfigurierten Wert multipliziert.

  Im Rahmen der Choreografie-basierten Implementierung sieht die Konfiguration des Retry-Verfahren mit maximal zwei Versuchen und einer initialen Wartezeit von drei Sekunden zwischen den Aufrufen für die Hotelangebotsanfrage folgendermaßen aus (siehe `src/main/resources/application.yml` des Choreografie-basierten „HotelService“):

```yaml
spring:
  cloud:
    stream:
      bindings:
        travelPackageRequestReceivedEvent-in-0:
          consumer:
            # configures the number of attempts when error occurs
            # (after attempts message gets sent to dlq)
            maxAttempts: 2
            backOffInitialInterval: 3000
          destination: TravelPackageRequestReceivedEvent
```

  Kann die Nachricht des Topics `RequestTravelPackageEvent` also beim ersten Mal nicht verarbeitet werden, so wird die Verarbeitung nach drei Sekunden ein weiteres Mal von dem zugehörigen Message Handler (`de.thi.eda.choreography.hotel.messaging.MessageConsumer`) wiederholt.

- **Dead Letter Queue**: „Dead Letter Queues“ (DLQs) beschreiben ein Konzept aus dem Messaging, indem fehlerhafte oder unzustellbare Nachrichten auf einen separaten Message Channel gelegt werden, sodass Verarbeitungsfehler explizit behandelt werden können [vgl. Hohpe und Woolf, 2015, S. 115 ff.]. Im Kontext von Kafka bedeutet das, dass die fehlerhaften Nachrichten auf ein eigenes Topic platziert werden [vgl. Stopford, 2018, S. 127]. DLQs werden hierbei direkt von Spring Cloud Stream unterstützt [vgl. Anandan u. a., 2021]. Um bei dem Beispiel der Hotelangebotsanfrage zu bleiben, schlägt die Anfrage mehrfach fehl, sodass alle Retries aufgebraucht sind, wird die Nachricht auf die konfigurierte Dead Letter Queue umgeleitet. Aktiviert wird die DLQ für einen Message Handler über die Spring Konfigurationsdatei mit den Attributen „enableDlq“ und „dlqName‘ (siehe `src/main/resources/application.yml` des Choreografie-basierten „HotelService“):

```yaml
spring:
  cloud:
```

```
3   stream:
4     kafka:
5       bindings:
6         travelPackageRequestReceivedEvent-in-0:
7           consumer:
8             enableDlq: true
9             dlqName: HotelOfferingDLQ
```

Die Konfiguration führt im Rahmen der Implementierung dazu, dass alle fehlerhaften Nachrichten zur Anfrage von Hotelangeboten in das Kafka Topic `HotelOfferingDLQ` gespeichert werden. Andere Services können diese Nachrichten lesen und basierend darauf Fehlerbehandlungen einleiten.

Fachliche Resilienz

Fachliche Fehler können über Dead Letter Queues, Ereignisse oder einfache If-Anweisungen im System kommuniziert werden. Als Beispiel zeigt Abbildung 4.12 wie im Rahmen der Choreografie-basierten Implementierung ein Fehler bei der Reisepaketgenerierung über eine Dead Letter Queue behandelt wird. Erhält der „FlightService“ eine fehlerhafte Antwort von der Travel API,

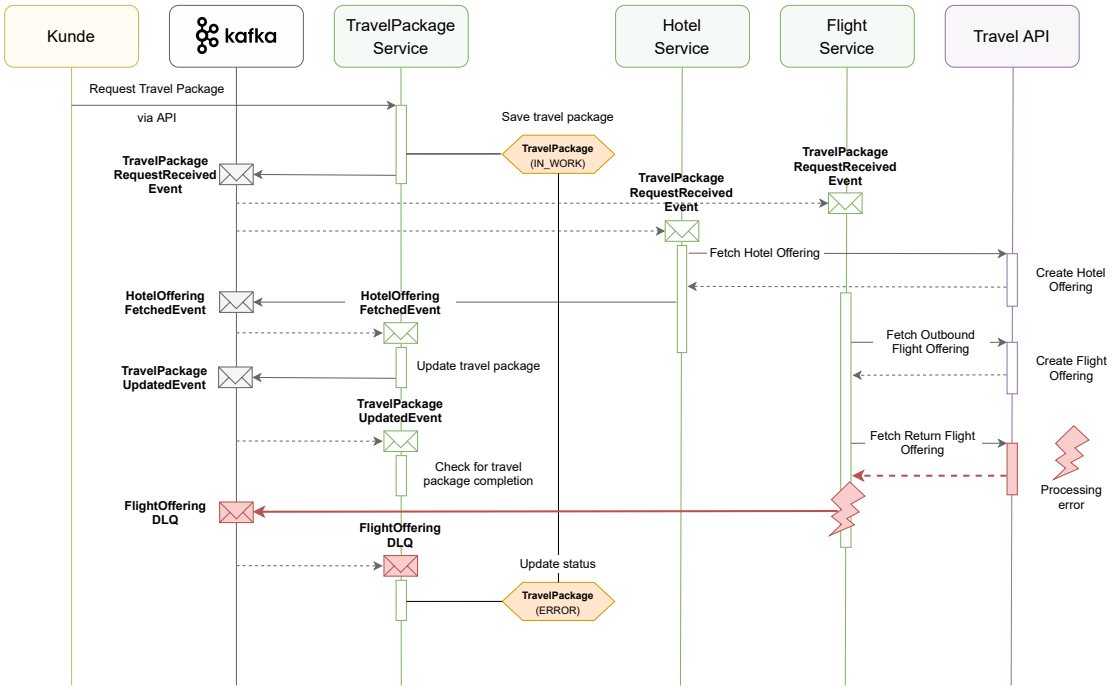

Abbildung 4.12.: Sequenzdiagramm einer fehlerhaften Reisepaketanfrage im Rahmen der ereignisgesteuerten Architektur nach dem Choreography Pattern [Eigene Darstellung]

so propagiert dieser die Nachricht, die den Fehler ausgelöst hat, an das `HotelOfferingDLQ` Topic. Die Nachricht wird danach vom „TravelPackageService“ konsumiert, sodass der Status des Reisepaketes auf „ERROR“ gesetzt werden kann. Der fehlerhafte Status des Reisepaketes dient als Indikator dafür, dass eine manuelle Bearbeitung durch einen Mitarbeiter vorgenommen werden muss.

Fehleridentifikation

Der Einsatz von Dead Letter Queues unterstützt dabei, aufgetretene Fehler im Rahmen der Nachrichtenverarbeitung, wie beispielsweise fehlgeschlagene Hotel- oder Flugangebotsanfragen, zu identifizieren. Technische Fehler außerhalb der Nachrichtenverarbeitung können zusätzlich über Log-Dateien der einzelnen Microservices eingesehen und ausgewertet werden. An dieser Stelle bietet es sich an, die Log-Dateien der verschiedenen Services zentral an einer Stelle zu aggregieren. Als Werkzeuge zum Sammeln und Verarbeiten der Log-Dateien empfiehlt sich die Verwendung des sogenannten „ELK-Stack“, der sich aus den Open-Source-Projekten Elasticsearch, Logstash und Kibana zusammensetzt [vgl. Wolff, 2018, S. 246 f.]. Im Rahmen der ereignisgesteuerten Implementierung des Reisebuchungssystems ist eine solche Verarbeitung der Logs jedoch nicht implementiert.

Nachvollziehbarkeit

Um die Nachvollziehbarkeit innerhalb eines ereignisgesteuerten Systems zu gewährleisten, existieren mehrere Möglichkeiten. Der von Bellemare für das Choreography Pattern vorgeschlagene Ansatz ist es, den Prozessfluss anhand der auftretenden Business Events nachzuverfolgen [vgl. Bellemare, 2020, S. 139]. Im Rahmen der Orchestrierung kann sich das Monitoring etwas einfacher gestalten, da der Orchestrierungsservice zentral eine Liste aller Events und dessen Verarbeitungsstatus verwalten und abrufen kann [vgl. Bellemare, 2020, S. 144]. Innerhalb der Beispielimplementierungen ist ein zentraler Dienst namens „MonitoringService“ implementiert, der die Ereignishistorie in chronologischer Reihenfolge pro Systemanfrage darstellt. Abbildung 4.13 zeigt exemplarisch den Ereignisfluss für eine gestartete Reiseanfrage.

13.09.2021 [22:38:25.045] - **TravelPackageRequestReceivedEvent**

13.09.2021 [22:38:32.073] - **HotelOfferingFetchedEvent**

13.09.2021 [22:38:32.096] - **TravelPackageUpdatedEvent**

13.09.2021 [22:38:38.090] - **FlightOfferingFetchedEvent**

13.09.2021 [22:38:38.102] - **TravelPackageUpdatedEvent**

13.09.2021 [22:38:55.680] - **TravelBookingReceivedEvent**

13.09.2021 [22:39:17.527] - **TravelBookingAcceptedEvent**

13.09.2021 [22:39:21.594] - **HotelBookedEvent**

13.09.2021 [22:39:29.660] - **FlightsBookedEvent**

Abbildung 4.13.: Event Historie des Choreografie-basierten „MonitorService“ im Rahmen der ereignisgesteuerten Implementierung [Eigene Darstellung]

Um zu erkennen, welche Ereignisse einer Kundenanfrage an das System zugeordnet sind, wird eine eindeutige Identifikationsnummer (auch „Correlation ID“ genannt) zu Beginn der Reiseanfrage generiert. Die ID wird dann bei allen nachfolgenden Ereignissen in der Payload mitgeschickt, sodass eine Spurenerkennung stattfinden kann (siehe Attribut „traceId“ der Klasse `de.thi.travel.common.messaging.DomainEvent`). Einen solchen Ansatz beschreibt auch Wolff [vgl.

Wolff, 2018, S. 248] für die Zuordnung von Log-Einträgen zu Requests bzw. Kunden im Kontext Microservice-basierter Systeme.

Neben der genannten Möglichkeit zur Darstellung des Ereignisflusses wird im Rahmen der Implementierung außerdem ein Distributed Tracing Tool namens „Zipkin“ [OpenZipkin, 2021] eingesetzt. Zipkin bietet die Möglichkeit über verteiltes Tracing einen Request innerhalb des Systems auf Call Stack Ebene nachzuverfolgen. Das Tool unterstützt damit zur Erkennung von Flaschenhälsen und Services mit langen Verarbeitungszeiten [vgl. Wolff, 2018, S. 249]. Abbildung 4.14 zeigt exemplarisch einen Trace in Zipkin dargestellt für einen Request zur Anfrage eines neuen Reisepaketes. In der Darstellung ist deutlich zu sehen, dass die Anfrage der Flugangebote am meisten Zeit beansprucht hat, weil das Angebot aufgrund eines Fehlers zweimal angefragt werden musste (siehe rote Balken bei „FlightService“ Zeilen).

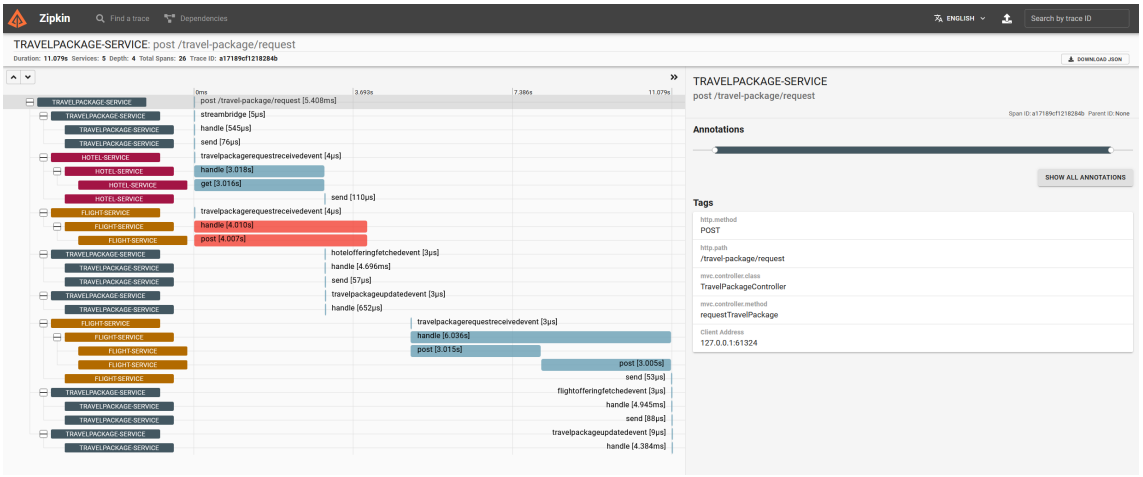

Abbildung 4.14.: Zipkin Trace eines Requests zur Generierung eines neuen Reisepaketes [Eigene Darstellung]

Transaktionalität

Die Transaktionalität wird im Rahmen der ereignisgesteuerten Architektur durch Einsatz des „Saga Patterns“ sichergestellt. Das Saga Pattern ist ein Muster zur Implementierung verteilter Transaktionen. Die verteilte Transaktion besteht dabei aus einer Sequenz mehrerer lokaler Transaktionen, die von den jeweiligen Teilnehmern bzw. Microservices ausgeführt werden. Schlägt eine lokale Transaktion fehl, so wird eine Reihe kompensierender Transaktionen ausgeführt, die das Ziel haben alle vorherigen Änderungen wieder rückgängig zu machen. Beide Architekturvarianten — Choreografie und Orchestrierung — können zur Koordination des Saga Patterns angewendet werden [vgl. Garcia-Molina und Salem, 1987; vgl. Bellemare, 2020, S. 145 ff.; vgl. Microsoft, 2021]. Das Sequenzdiagramm aus Abbildung 4.15 zeigt die Anwendung des Choreografie-basierten Saga Patterns zur Sicherstellung der Konsistenz bei der Buchungsabwicklung. In dem Beispiel schlägt die Kreditkartenabbuchung aufgrund einer abgelaufen Kreditkarte fehl. Da davor bereits das Hotel und die Flüge gebucht wurden, müssen diese Schritte daraufhin durch kompensierende Transaktionen rückgängig gemacht werden.
Im Falle der Implementierung nach dem Orchestration Pattern verlagert sich die Logik zur Steuerung der Kompensationen in den „BookingOrchestratorService“. Dieser ist in dem dargestellten Beispiel im Rahmen eines Flugbuchungsfehlers verantwortlich, die Hotelstornierung und das Senden einer Absagemail anzustoßen, sodass sich das System letztendlich wieder in einem konsistenten Zustand befindet (vgl. Abbildung 4.16).

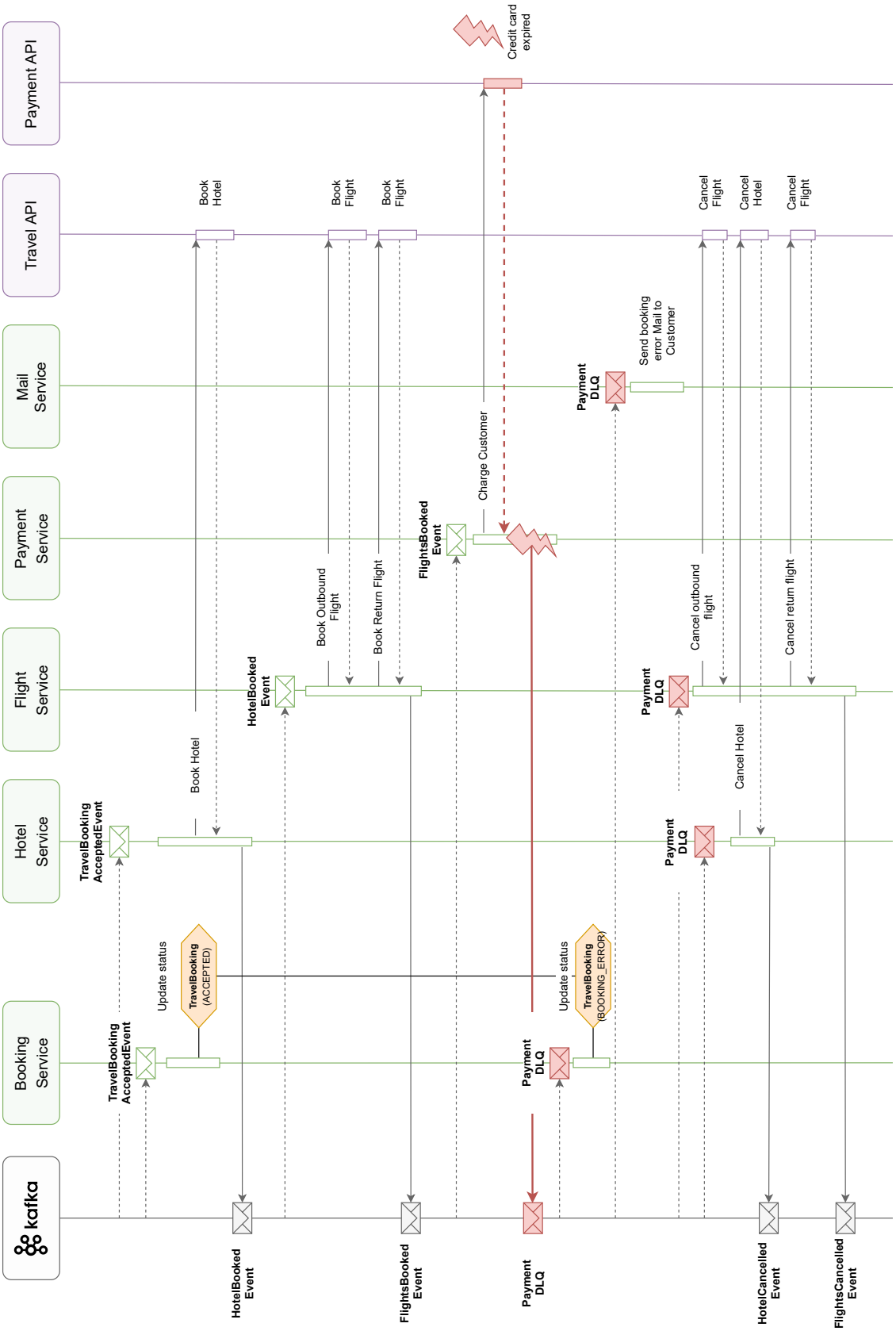

Abbildung 4.15.: Sequenzdiagramm zur Darstellung des Choreografie-basierten Saga Patterns im Rahmen eines Buchungsfehlers [Eigene Darstellung]

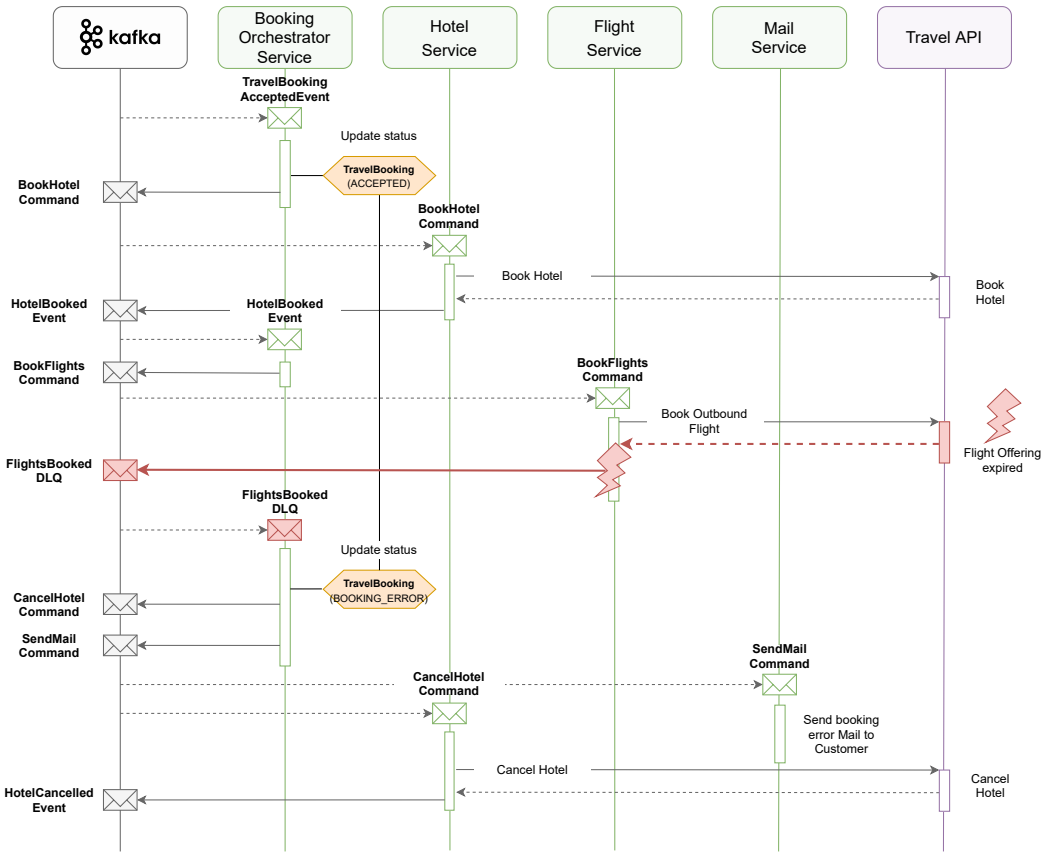

Abbildung 4.16.: Sequenzdiagramm zur Darstellung des orchestrierten Saga Patterns im Rahmen eines Buchungsfehlers [Eigene Darstellung]

Parallelität

Die parallele Ausführung von Aufgabensträngen ist im System implizit durch das Publish-Subscribe Muster der Kafka Topics gegeben. Mehrere Services können sich auf Events subskribieren und dadurch parallel Verarbeitungen anstoßen. Dies findet bei der Angebotsermittlung statt. Hotel- und Flugangebot werden dort parallel angefragt. Entweder durch gemeinsames konsumieren des `RequestTravelPackageEvents` (Choreografie) oder dadurch dass der „TravelPackageOrchestratorService“ im Rahmen der Orchestrierung einen `FetchHotelCommand` und `FetchFlightsCommand` sendet (siehe hierzu Methode `handleTravelPackageRequest` der Java-Klasse `de.thi.eda.orchestration.travelpackage.service.TravelPackageService`).

Zur parallelen bzw. asynchronen Methodenausführung innerhalb eines Event-Driven Microservice wird die Annotation `@Async` des Spring Frameworks verwendet. Die Annotation legt fest, dass ein Methodenaufruf in einem separaten Thread ausgeführt werden soll. Damit das funktioniert ist es notwendig eine Bean vom Typ `java.util.concurrent.Executor` zu konfigurieren, wo unter anderem angegeben wird wie viele Threads gleichzeitig verwendet werden dürfen [vgl. VMware, 2021b]. Für die ereignisgesteuerte Implementierung reicht ein Thread Pool von maximal zwei Threads aus. Die Konfiguration hierzu ist in Listing 4.5 dargestellt.

```
1 @Configuration
2 @EnableAsync
```

```
3  public class AsyncConfiguration {
4      @Bean
5      public Executor taskExecutor() {
6          ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
7          executor.setCorePoolSize(2);
8          executor.setMaxPoolSize(2);
9          executor.setThreadNamePrefix("Parallel-");
10         executor.initialize();
11         return executor;
12     }
13 }
```

Listing 4.5: Konfiguration des Thread Pool Executors zur parallelen Verarbeitung innerhalb der ereignisgesteuerten Architektur

Asynchrone Methodenaufrufe finden im Kontext der EDA-Implementierung beispielsweise innerhalb des „FlightService“ bei der Anfrage eines Hin- und Rückflugs von der Travel API statt (siehe hierzu Klasse `de.thi.eda.choreography.flight.messaging.MessageConsumer`). Die Methode `fetchFlightOffering` des `de.thi.eda.choreography.flight.service.FlightService` ist mit der Annotation `@Async` versehen, damit eine asynchrone Ausführung möglich wird.

Reaktivität

Im Anforderungskapitel (vgl. 4.3.2) wurden zwei Indikatoren für die Reaktivität des Reisebuchungssystems genannt: Die flexible Verarbeitung eingehender Reisewarnmeldungen und der Umgang mit Wartezuständen.

Reisewarnmeldungen werden über Nachrichten vom Travel Warning System periodisch an das Kafka Topic namens `TravelWarningEvent` gesendet (vgl. Kapitel 4.2.3). Im Rahmen der ereignisgesteuerten Implementierung werden diese Nachrichten vom „BookingService“ (bzw. „BookingOrchestratorService“) konsumiert und verarbeitet. Dieser prüft ob die Reisewarnung bestehende Kundenbuchungen betrifft und stößt gegebenenfalls, durch das Senden eines Events oder Commands, eine Benachrichtigung des Kunden an.

Wartezustände treten sowohl nach Erstellung eines neuen Reisepaketes auf als auch nach erfolgter Reisebuchungsabwicklung. Ein erstelltes Reisepaket soll maximal drei Tage gültig sein und muss danach als abgelaufen markiert werden. Außerdem hat ein Kunde bis zu sieben Tage vor Reisebeginn die Möglichkeit, ein bereits gebuchtes Reisepaket zu stornieren. Zur Abwicklung der zeitlichen Intervalle wird hierzu über die Klasse `org.springframework.scheduling.TaskScheduler` des Spring Frameworks eine asynchrone Task innerhalb eines neuen Threads geplant. Diese wird dann zu dem definierten Zeitpunkt ausgeführt. Listing 4.6 zeigt den Programmcode der Choreografie-basierten Implementierung zum Einplanen einer Task, die ein Reisepaket nach Ablauf der definierten Zeitspanne in Zeile 10 als abgelaufen markiert (siehe Klasse `de.thi.eda.choreography.travelpackage.service.TravelPackageService`).

```
1  private void scheduleTravelPackageExpirationTime(UUID travelPackageId) {
2      log.info("Scheduling expiration date time for travel package {}", travelPackageId);
3
4      taskScheduler.schedule(() -> {
5          log.info("Calling expiration date time function for travel package {}", travelPackageId);
6          Optional<TravelPackage> tpOpt = travelPackageRepository.findById(travelPackageId);
7          tpOpt.ifPresent(tp -> {
```

```
8            if (tp.getStatus() == OPEN) {
9                log.info("Travel Package ({}) expired!", travelPackageId);
10               tp.setStatus(EXPIRED);
11               travelPackageRepository.save(tp);
12           }
13       });
14   }, computeTravelPackageExpirationDate());
15 }
16
17 private Date computeTravelPackageExpirationDate() {
18   return new Date(System.currentTimeMillis()
19           + TimeUnit.MINUTES.toMillis(this.expirationTimeInMinutes));
20 }
```

Listing 4.6: Verwendung des Task Schedulers zur zeitlichen Planung und Ausführung von Programmcode zur Markierung abgelaufener Reisepakete

Da eine Wartezeit von drei Tagen für Demonstrationszwecks zu lange wäre, wird diese in den Beispielimplementierungen auf wenige Minuten gesetzt. Das eigentliche Einplanen der Task geschieht in Zeile 4 durch die Methode `schedule()`. Als Methodenparameter wird sowohl die auszuführende Methode als Lambda-Ausdruck als auch der Ausführungszeitpunkt angegeben.

4.7. Process Engine basierte Implementierung

4.7.1. Zielsetzung und Vorgehensweise der Implementierung

Zielsetzung der Implementierung

Zielsetzung der Implementierung ist es, die in Kapitel 4.3 gestellten Prozess- und Systemanforderungen unter Anwendung einer Process Engine basierten Architektur (vgl. Kapitel 2.4.4) umzusetzen. Aktuelle und typische Best Practices Process Engine basierter Architekturen sind dabei zu berücksichtigen, sodass ein repräsentativer Vergleich mit anderen Architekturen möglich wird.

Vorgehensweise zur Implementierung

Das Vorgehen zur Implementierung des Reisebuchungssystems nach einer Process Engine basierten Architektur ähnelt dem Vorgehen zur Implementierung der ereignisgesteuerten Architektur: Mittels Domain-Driven Design können die einzelnen Bounded Contexts identifiziert und daraus die Microservices und dessen Domänen-Ereignisse abgeleitet werden. Wie im Grundlagenkapitel genannt, stellen Process Engines die zentrale Komponente zur Implementierung von Prozessflüssen im Rahmen der Architektur dar. Vielmehr können Process Engines ereignisgesteuerte Architekturen nach Rücker ergänzen, indem diese als Orchestrierungswerkzeug fungieren (vgl. Kapitel 2.4.4). Deshalb bietet es sich an die Process Engine basierte Implementierung auf Basis der ereignisgesteuerten Architektur nach dem Orchestration Pattern aufzubauen. Jegliche Steuerungs- und Ablauflogik innerhalb der jeweiligen Microservices wird mittels BPMN-Prozessen durch Process Engines implementiert. Als Process Engine wird die bereits in Kapitel 4.1 genannte Open-Source-Engine von Camunda eingesetzt.

Implementierung von Serviceaufgaben innerhalb der Camunda Process Engine

Spezifische Operationen, wie das Speichern von Geschäftsobjekten, Senden von Ereignissen oder Durchführen logischer Berechnungen, sind essenziell zur Implementierung des Reisebuchungssystems. Bei Ablauf in einer Process Engine muss deshalb die Möglichkeit bestehen, Programmcode ausführen zu können. Hierfür sieht die BPMN-Spezifikation „Serviceaufgaben“ vor, welche durch kleine Zahnräder am obereren linken Rand gekennzeichnet sind. In Camunda gibt es die Option Java-Klassen an Serviceaufgaben zu binden, die das `org.camunda.bpm.engine.delegate.JavaDelegate` Interface implementieren. Bei Eintritt der Serviceaufgabe wir dann die aus dem Interface stammende `execute()` Methode der Klasse aufgerufen. Im Rahmen des Spring Frameworks sollte die Java Klasse außerdem mit `@Component` annotiert sein, damit diese als Spring Bean vom Spring Application Context instantiiert werden kann. Wie in Kapitel 3.2 erläutert wurde, werden BPMN-Modelle als XML-Datei repräsentiert. Zur Definition von Java Delegates erweitert Camunda den BPMN-Standard um einen weiteren XML-Namensraum. Java Delegates können so über das Attribut `camunda:delegateExpression` an Serviceaufgaben konfiguriert werden. Als Wert wird dabei eine Expression nach dem Schema `${className}` angegeben, wobei „className“ den Klassennamen darstellt. Dieser muss in der „camelCase“ Notation geschrieben sein. Eine beispielhafte Konfiguration des Java Delegates zur Hotelbuchung des „HotelService“ ist in Listing 4.7 abgebildet.

```
<bpmn:serviceTask id="Activity_073adbx" name="Hotel buchen"
    camunda:delegateExpression="${bookHotelDelegate}">
```

Listing 4.7: Camunda Servicetask mit angehefteter Delegate Expression

Die zugehörige Java Klasse `de.thi.process.engine.hotel.delegate.booking.BookHotelDelegate` ist in Listing 4.8 abgebildet. Wie in Zeile 3 zu erkennen ist, implementiert die Klasse das `JavaDelegate` Interface. Ab Zeile 8 ist der Code für die `execute()` Methode abgebildet, der bei Eintritt der zugehörigen Serviceaufgabe durch die Process Engine ausgeführt wird.

```
1  @Component
2  public class BookHotelDelegate implements JavaDelegate {
3      @Autowired
4      private HotelService hotelService;
5      @Override
6      public void execute(DelegateExecution execution) throws Exception {
7          BookHotelCommandPayload bookHotelPayload
8                  = (BookHotelCommandPayload) execution.getVariable("data");
9          try {
10             HotelOfferingDTO offering = bookHotelPayload.getTravelBooking()
11                     .getTravelPackage().getHotelOffering();
12             hotelService.bookHotelOffering(offering);
13         } catch (Exception ex) {
14             throw new BpmnError("hotelBookingError");
15         }
16     }
17 }
```

Listing 4.8: Implementierung eines Java Delegates zur Hotelbuchung

4.7.2. Architekturvorstellung

Die Prozess Engine basierte Architektur zur Implementierung des Reisebuchungssystems ist in Abbildung 4.17 dargestellt. Wie zu erkennen existieren analog zur ereignisgesteuerten Archi-

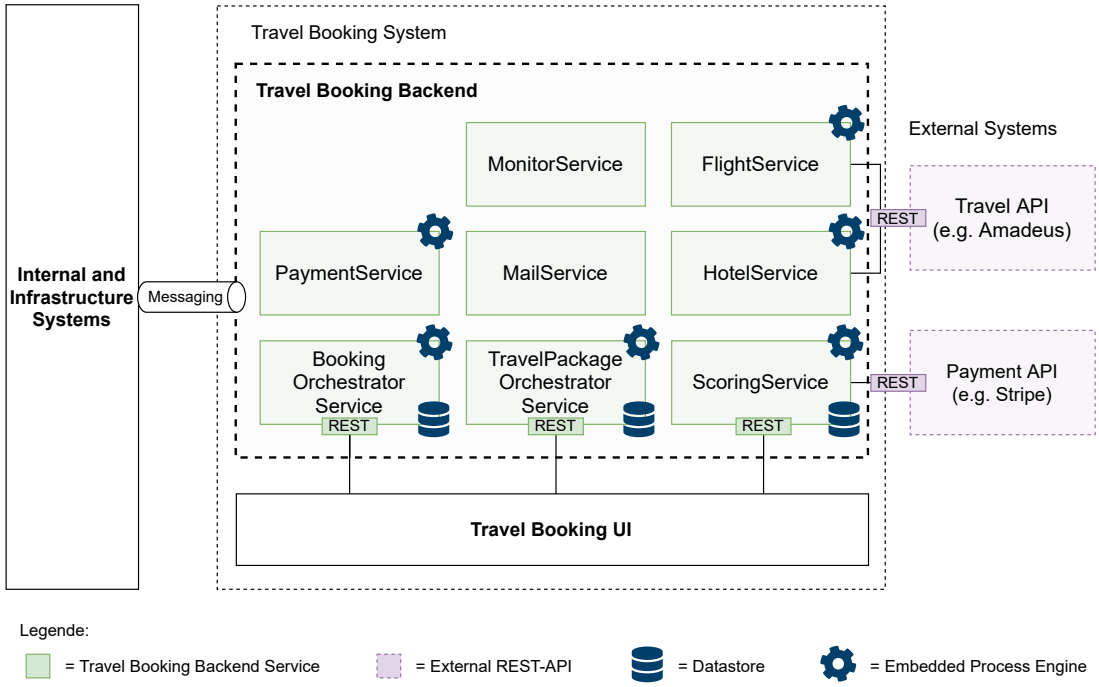

Abbildung 4.17.: Architekturschaubild des Reisebuchungssystems nach einer Process Engine basierten Architektur [Eigene Darstellung]

tektur acht Backend Services, die über Messaging miteinander kommunizieren. Im Vergleich zu den ereignisgesteuerten Architekturen haben die jeweiligen Services, die intern Prozesslogik abbilden, nun jedoch jeweils eine Process Engine eingebettet. Damit werden die für dessen Bounded Context gültigen Prozesse implementiert. Insgesamt neun BPMN-Modelle und eine DMN-Tabelle kommen zum Einsatz. DMN steht für „Decision Model and Notation“ und ist ein von der OMG verwalteter Standard zur Modellierung von Entscheidungen. DMN-Tabellen können genauso wie BPMN von einer Process Engine ausgeführt werden [vgl. The Object Management Group, 2021b; vgl. Freund und Rücker, 2019, S. 177]. Innerhalb der Process Engine basierten Architektur wird DMN zur Abbildung der Entscheidungslogik für die Rabattberechnung bei der Reisepaketgenerierung eingesetzt. Die BPMN-Modelle sind insgesamt über sechs Services verteilt, von denen jeder Service eine eigene Process Engine beinhaltet (symbolisiert durch das Zahnrad am rechten oberen Rand eines Services in Abbildung 4.17). Zur Einfachheit bei der Umsetzung greifen alle Services im Rahmen der Implementierung auf eine zentrale Datenbank zu. Da ein solches Vorgehen die Autonomie eines Microservices verletzt, wird üblicherweise innerhalb realer Anwendungen je Dienst eine eigene Datenbank verwendet.

Die Prozessmodelle der einzelnen Services steuern die Kommunikation und Koordination innerhalb des Systems. Das Versenden und Konsumieren von Commands und Events erfolgt über Serviceaufgaben. Rücker betont hierbei, dass die Orchestrierung nicht zentral erfolgen muss, sondern gezielt lokal von einzelnen Microservices geschehen kann [vgl. Ruecker, 2021, S. 170]. Dies widerspricht dem klassischen Ansatz eines zentralen Orchestrators, der den Gesamtprozess verantwortet und steuert. Damit die Instanzen der verteilten Prozesse ohne Probleme miteinander kommunizieren können, ist es notwendig, diese in Beziehung zueinander zu stellen. Dafür wird im Kontext der Implementierung der sogenannte „Business Key“ zur Korrelation zwischen den Prozessen eingesetzt. Nachrichten können so exakt einer Prozessinstanz zugeordnet werden. Camunda definiert den Business Key als einen domänenspezifischen Identifikator für eine

Prozessinstanz [vgl. Lindhauer, 2018]. Im Bereich des „TravelPackageService“ ist beispielsweise die jeweilige Reisepaket-ID als Business Key gesetzt, während der „BookingOrchestratorService“ die Buchungsnummer verwendet. Listing 4.9 zeigt wie mittels der Camunda Process Engine API in Java eine Nachricht anhand des Business Keys korreliert werden kann. Der Parameter `processVariables` ist optionaler Natur und bietet die Möglichkeit zusätzliche Daten an die Prozessinstanz zu übermitteln.

```
1 runtimeService.correlateMessage(messageName, businessKey, processVariables);
```

Listing 4.9: Beispielcode zur Korrelation einer Nachricht anhand des Business Keys mit der Camunda Process Engine API

4.7.3. Validierung der technischen Systemanforderungen

Nachfolgend wird erläutert inwiefern die in Kapitel 4.3.2 definierten Systemanforderungen durch die Process Engine basierte Implementierung umgesetzt sind.

Integration in bestehende Systemlandschaft

Wie bei der ereignisgesteuerten Architektur dient auch hier das Integration System als Adapter zur Integration der bestehenden Systemlandschaft. Es sind hierfür zwei Camel Routen definiert, die nach Eingang einer Nachricht die Kundendaten in das CRM und Billing System speichern. Der zugehörige Java Code in den folgenden Klassen implementiert:

```
1 de.thi.travel.integrationsystem.pe.SaveCustomerToBillingSystemRoute
2 de.thi.travel.integrationsystem.pe.SaveCustomerToCRMSystemRoute
```

Das Versenden des Commands an das Integration System zur Speicherung der Kundendaten ist in dem fachlichen Prozessmodell des „BookingOrchestratorService“ explizit über Serviceaufgaben modelliert (siehe `src/main/resources/travel-booking-orchestrator.bpmn`).

Integration externer Dienste

Der „FlightService“, „HotelService“ und „ScoringService“ tätigen analog zur EDA Implementierung direkte REST-Aufrufe an die externen Systeme. Die Systemaufrufe sind als Serviceaufgaben modelliert. Die Java Delegates der Serviceaufgaben verwenden auch hier als HTTP-Client die Spring interne Klasse `org.springframework.web.client.RestTemplate`, um die REST-Aufrufe auszuführen.

Technische Resilienz

Die Process Engine basierte Implementierung behandelt technische Fehler auf zweierlei Arten:

- **Fehlerereignisse**: Process Engines auf Basis von BPMN ermöglichen es technische Fehler explizit im Prozessmodell über „Fehlerereignisse“ zu behandeln. Diese werden durch einen Blitz symbolisiert. In Abbildung 4.18 ist das Prozessmodell zur Hotelangebotsanfrage des „HotelService“ gezeigt. Die Serviceaufgabe „Hotel anfragen“ kommuniziert direkt über eine REST-Schnittstelle mit der Travel API. Bei mehrfachem Fehlschlagen der Angebotsanfrage und wenn alle Retries getätigt sind, wird innerhalb des Java Delegates `de.thi.process.engine.hotel.delegate.offering.RequestHotelOfferingDelegate` über das Werfen einer Exception vom Typ `org.camunda.bpm.engine.delegate.BpmnError` das angeheftete Fehlerereignis ausgelöst.

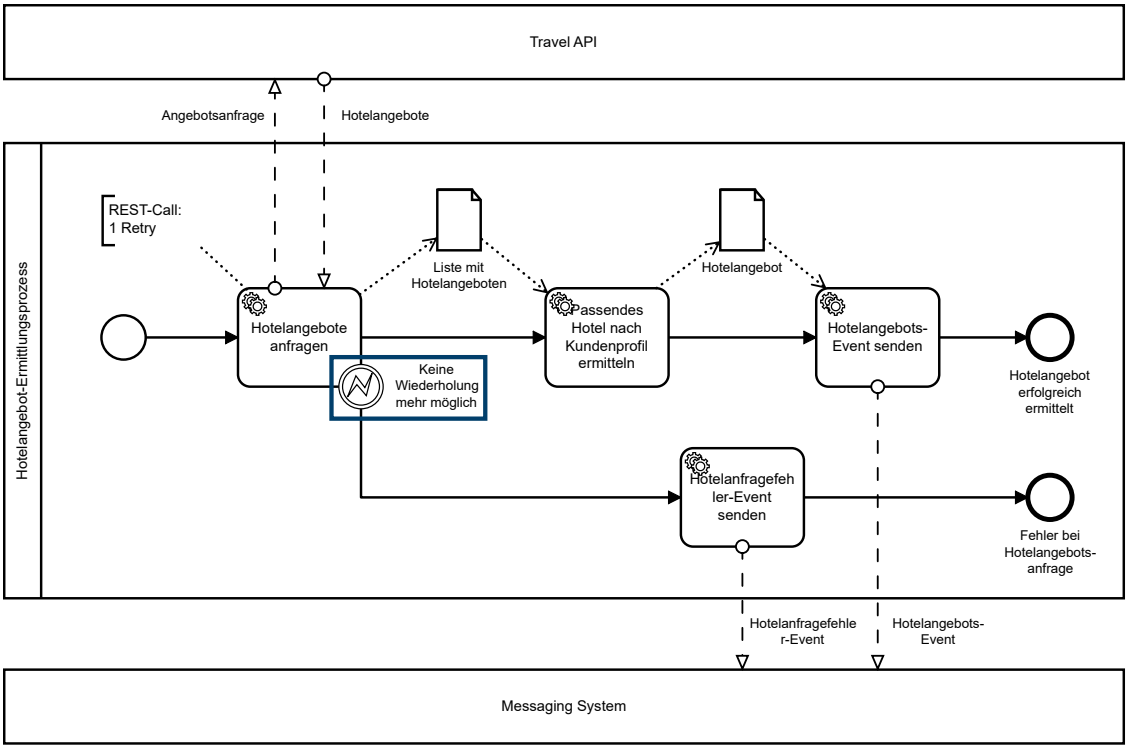

Abbildung 4.18.: BPMN-Prozessmodell zur Hotelangebotsanfrage des „HotelService“ im Rahmen der Process Engine basierten Implementierung [Eigene Darstellung]

- **Retry-Verfahren**: Generell ist es denkbar, Retry-Verfahren innerhalb des BPMN-Modells zu modellieren, jedoch empfiehlt es sich aufgrund des Aufwands und der schlechten Lesbarkeit Retry-Verfahren nicht explizit im Prozess zu modellieren. Camunda bezeichnet dieses Vorgehen auch als Anti-Pattern [vgl. Camunda Services GmbH, 2021a]. Die Camunda Process Engine bietet deshalb standardmäßig Möglichkeiten zur Konfiguration einer Retry-Strategie bei Serviceaufgaben. Die Konfiguration folgt dem ISO 8601 Standard zur Definition von Wiederholungsintervallen und kann bei Camunda in der XML-Definition des BPMN-Modells über den Parameter `camunda:failedJobRetryTimeCycle` gesetzt werden [vgl. Camunda Services GmbH, 2021b]. Um beim Beispiel der Hotelangebotsanfrage zu bleiben (siehe Prozessmodell in Abbildung 4.18), sieht die Konfiguration eines Retries mit maximal zwei Versuchen und einer Verzögerung von drei Sekunden bei der Serviceaufgabe „Hotelangebot anfragen“ wie folgt aus:

```
<bpmn:extensionElements>
  <camunda:failedJobRetryTimeCycle>R2/PT3S</camunda:failedJobRetryTimeCycle>
</bpmn:extensionElements>
```

Neben den Fehlerereignissen und Retry-Verfahren ermöglichen es Process Engines im Fehlerfalle den Zustand eines Prozesses zu einem definierten Prozessschritt zurückzusetzen. Auch trägt zur technischen Resilienz bei, dass zur Laufzeit fehlerhafte Prozessvariablen relativ einfach verändert werden können.

Fachliche Resilienz

Zur Behandlung fachlicher Fehler ist in der BPMN-Spezifikation das Eskalationsereignis vorgesehen. Verwendet wird dieses in der Beispielimplementierung unter anderem bei der Reisepa-

ketgenerierung (siehe Abbildung 4.19). Schlägt hier das Einholen des Hotel- oder Flugangebots fehl, so wird dies über ein Eskalationsereignis an den Elternprozess kommuniziert. Danach kann mit der manuellen Fehlerbehandlung durch einen Mitarbeiter begonnen werden (siehe Ereignis „Fehler bei Angebotsermittlung“ in Prozess `src/main/resources/travel-paclage-orchestrator.bpmn` des „TravelPackageOrchestratorService“).

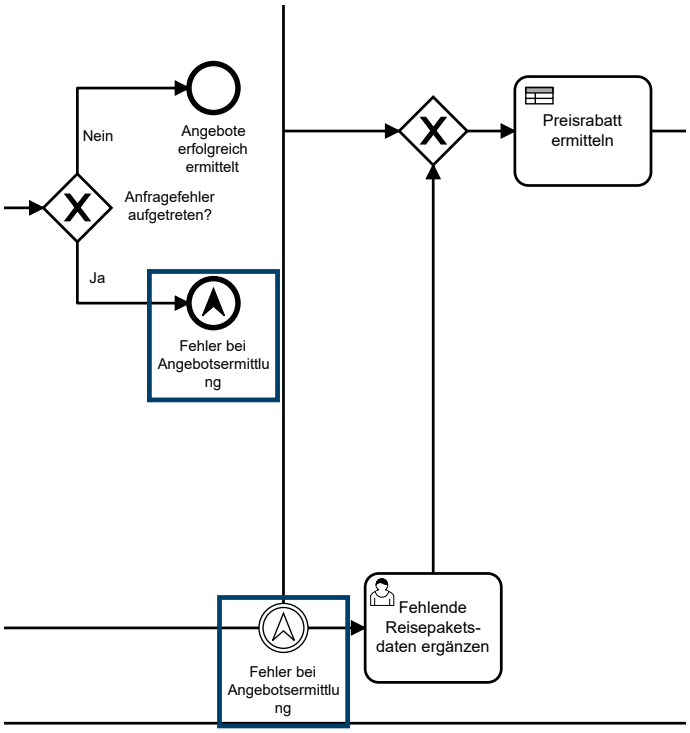

Abbildung 4.19.: Ausschnitt des BPMN-Modells zur Reisepaketgenerierung im Rahmen der Process Engine basierten Implementierung [Eigene Darstellung]

Fehleridentifikation

Die Fehleridentifikation ist durch ähnliche Methoden wie auch bei der ereignisgesteuerten Implementierung eingesetzt wurden gegeben. Der große Vorteil der Verwendung von Process Engines ist es jedoch, dass Fehler die im Rahmen der Ausführung des Prozesses innerhalb der Engine auftreten gespeichert und eingesehen werden können. Mehr hierzu ist in nachfolgendem Punkt zur Nachvollziehbarkeit erläutert.

Nachvollziehbarkeit

BPMN liefert als Prozessnotation bereits eine Modellierungssprache die einen Geschäftsprozess grafisch darstellt. Zur Ausführung der Geschäftsprozesse speichern Process Engines implizit alle prozessbezogenen Daten ab. Dies ermöglicht es in Echtzeit den aktuellen Zustand eines Prozesses oder dessen Pfad innerhalb des Prozessdiagramms nachzuverfolgen. Zum Überwachen und Steuern von Prozessinstanzen bietet Camunda bereits eine Webapplikation namens „Camunda Cockpit“ an, dessen Open-Source-Version im Rahmen der Process Engine basierten Implementierung für das Monitoring verwendet und zentral durch den „MonitorService“ bereitgestellt wird.

Da es im Rahmen des beispielhaften Reisebuchungssystems lästig wäre, für jede eingebettete Process Engine ein eigenes Camunda Cockpit zur Überwachung der lokalen Prozesse bereitzustellen, hat es sich für die Beispielimplementierung angeboten eine zentrale Datenbank zu verwenden, worin die verteilten Process Engines ihre Daten abspeichern. Das Camunda Cockpit des „MonitorService“ kann so auf die zentrale Datenbank zugreifen und alle Prozessinformationen der Microservices gebündelt in einem Cockpit anzeigen. Die richtige Lösung im Rahmen einer realen Implementierung wäre es, die jeweiligen Datenbanken der unterschiedlichen Process Engines an eine zentrale Instanz des Camunda Cockpits anzubinden. Da dieser Ansatz jedoch mit zusätzlichem Konfigurationsaufwand verbunden ist, wurde er innerhalb der Beispielimplementierung nicht umgesetzt.

Abbildung 4.20 zeigt die Ansicht des Camunda Cockpits zur Überwachung des Geschäftsprozesses, der durch den „BookingOrchestratorService“ implementiert ist. Durch die kleinen blauen Kreise an den Aktivitäten wird die Anzahl der aktiven Prozessinstanzen, die sich an diesem konkreten Schritt befinden, signalisiert. So kann der Prozessablauf in Echtzeit nachverfolgt werden. Auch ist im unteren Bereich des Bildes eine Auflistung aller aktiven Prozessinstanzen zur ausgewählten Prozessdefinition dargestellt. Das Camunda Cockpit bietet neben einer detaillierten Ansicht und Möglichkeit zur Änderung aller Prozessvariablen einer konkreten Prozessinstanz noch viele weitere Features zur Steuerung und Überwachung von BPMN Prozessen. Was vor allem zur Anforderung der Nachvollziehbarkeit beiträgt, ist die Visualisierung des Prozessflusses aller aktiven und vergangenen Prozessinstanzen. Historische Prozessdaten können zum Zeitpunkt dieser Arbeit jedoch nur in der Enterprise-Version des Cockpits visuell dargestellt und ausgewertet werden. Über die REST-API der Process Engine ist es aber dennoch möglich diese abzufragen.

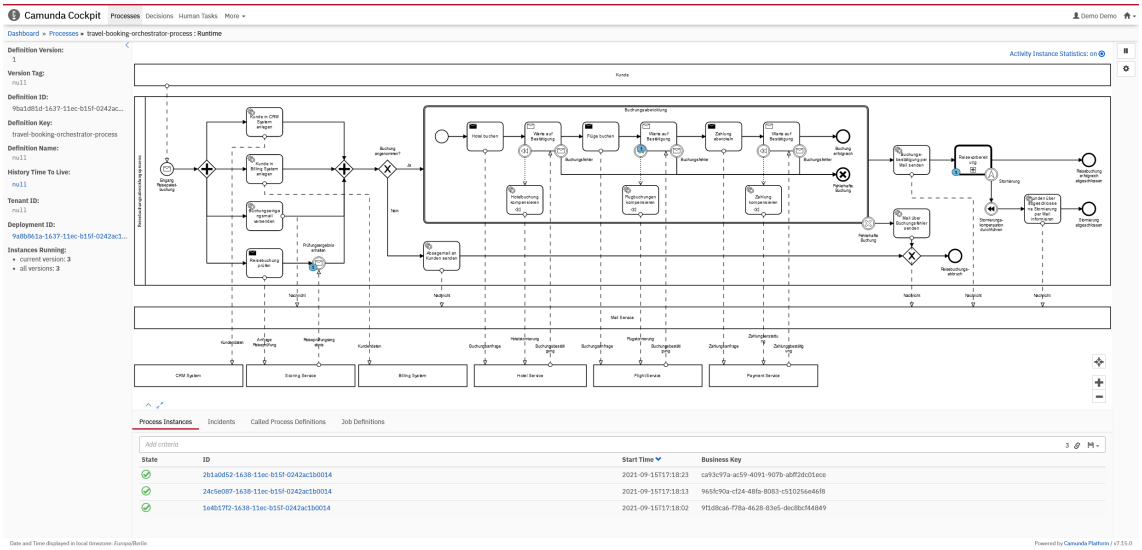

Abbildung 4.20.: Camunda Cockpit Anzeige zur Überwachung des Fachprozesses des „BookingOrchestratorService“ im Rahmen der Process Engine basierten Implementierung [Eigene Darstellung]

Transaktionalität

Zur Durchführung von Transaktionen ist in BPMN der „Transaktionsteilprozess“ spezifiziert. Dieser ist durch eine doppelte Umrandung gekennzeichnet und arbeitet in Kombination mit sogenannten „Kompensationsereignissen“. Kompensationsereignisse werden mit zwei rückwärts

gerichteten Pfeilen symbolisiert und können an Aktivitäten angeheftet werden, die mit einer definierten Kompensationsaufgabe rückgängig gemacht werden sollen [vgl. Freund und Rücker, 2019, 88 f.].

Darstellung 4.21 zeigt den Transaktionsteilprozess zur Buchungsabwicklung innerhalb des „BookingOrchestratorService“. Wie an den Nachrichtenaktivitäten zu erkennen ist, findet die Buchungsabwicklung asynchron statt, da der „BookingOrchestratorService“ mit dem „HotelService“, „FlightService“ und „PaymentService“ über Commands kommuniziert. An den Aktivitäten „Bestätigung erhalten“ sind die Kompensationsereignisse zur Stornierung angeheftet und Nachrichtenereignisse, die mögliche Buchungsfehler behandeln. Bei Eintritt eines Buchungsfehlers wird das Abbruchereignis „Buchung fehlerhaft“ ausgelöst, wodurch die Kompensationsaufgaben angestoßen werden.

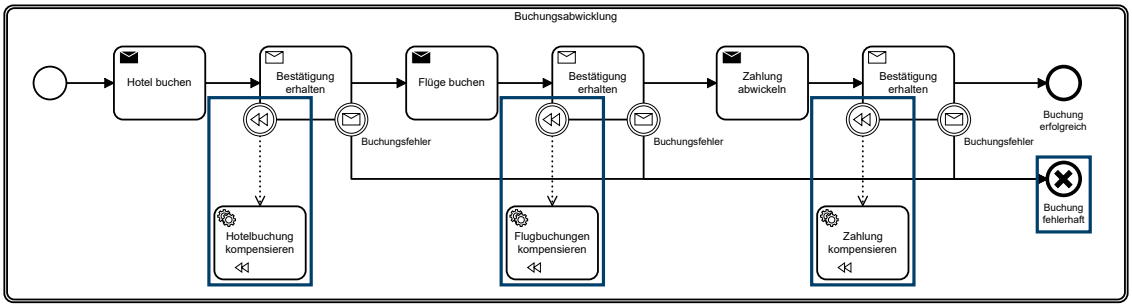

Abbildung 4.21.: Transaktionsteilprozess zur Buchungsabwicklung im Rahmen der Process Engine basierten Implementierung [Eigene Darstellung]

Bei Eintritt des Abbruchereignisses, werden alle Kompensationsaufgaben gestartet, deren zugeordnete Aktivität vor Abbruch erfolgreich abgeschlossen wurde. Anhand des Ablaufs ist zu erkennen, dass Transaktionsteilprozesse durch die Kompensation von Aktivitäten letztendlich auch eine Implementierung des in Kapitel 4.6.4 genannten Saga Patterns sind. Unterschied zur ereignisgesteuerten Implementierung ist jedoch, dass die Process Engine jegliche Logik zur Implementierung und Steuerung des Saga Patterns übernimmt und damit kein zusätzlicher Implementierungsaufwand entsteht.

Parallelität

Parallele Verarbeitungsstränge werden bei der Process Engine basierten Implementierung innerhalb des BPMN-Modells mit einem parallelen Gateway modelliert. Das parallele Gateway ist durch eine Raute mit Plus-Symbol gekennzeichnet und führt alle ausgehenden Pfade gleichzeitig durch die Process Engine aus. Am abschließendem parallelen Gateway wird dann wieder auf das Eintreffen der ausgehenden Pfade gewartet, bis mit dem Prozessfluss fortgefahren wird.

Das BPMN-Modell aus Abbildung 4.22 zeigt den Prozess zur Anfrage der Flugangebote im „FlightService“. Das parallele Gateway teilt den Prozessfluss in zwei Pfade auf, um ein Hin- als auch Rückflugangebot anzufragen.

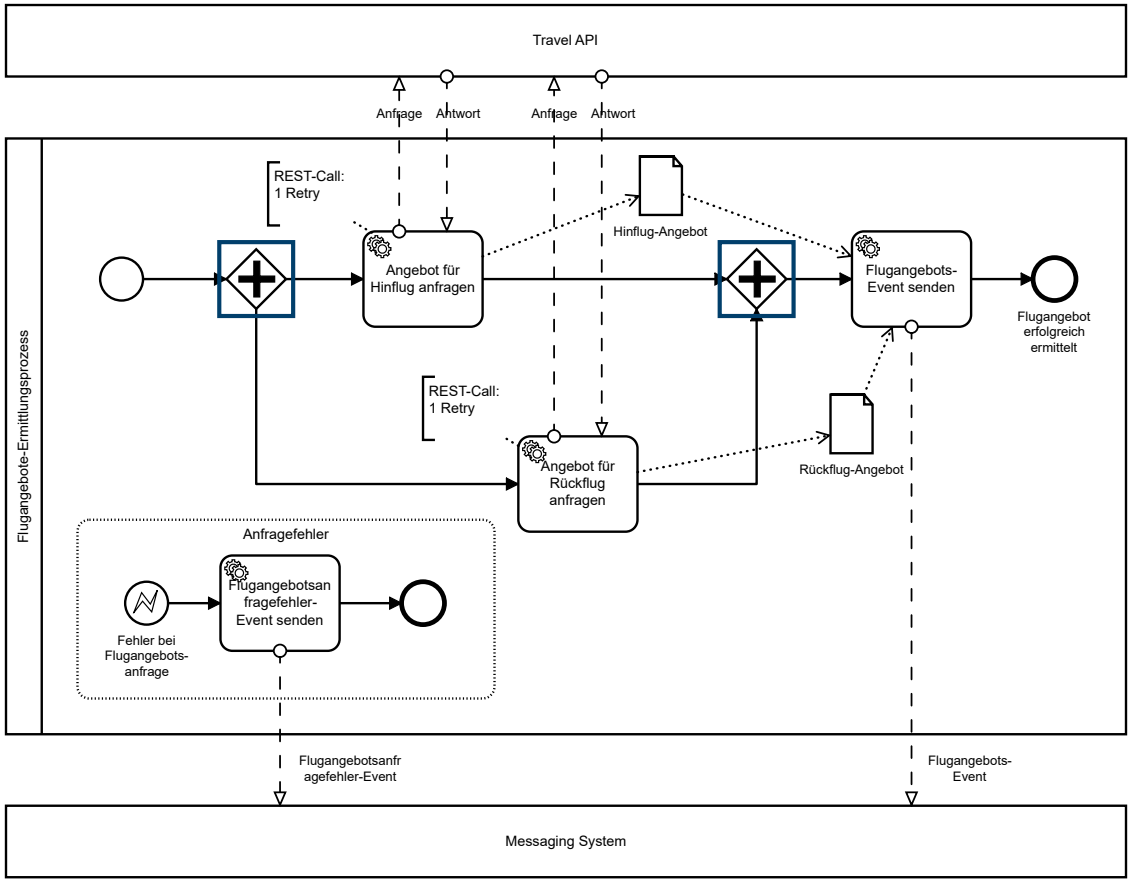

Abbildung 4.22.: Flugangebotsanfrage-Prozess im Rahmen der Process Engine basierten Implementierung [Eigene Darstellung]

Standardmäßig führt die Camunda Process Engine Aktivitäten innerhalb einer Prozessinstanz niemals echt parallel (ohne Multi-Threading) aus, um Inkonsistenzen der Prozessdaten zu vermeiden. Um eine echt parallele Ausführung innerhalb der Prozessinstanz zu ermöglichen, ist es notwendig die einzelnen Serviceaufgaben mit den Attributen `camunda:asyncBefore="true"` und `camunda:exclusive="false"` zu konfigurieren. Zur Synchronisierung der einzelnen Threads setzt die Process Engine auf „optimistisches Locking“. Wenn zwei Aktivitäten gleichzeitig Daten ändern, gewinnt derjenige, der die Daten zuerst schreibt. Alle anderen werfen eine `OptimisticLockingException` und wiederholen die Ausführung. Da eine wiederholte Ausführung der Servicetasks im Falle einer `OptimisticLockingException` ein unerwünschter Seiteneffekt ist (es sollte z. B. nicht zweimal ein Hotel gebucht werden), empfiehlt Camunda das endende parallele Gateway mit `camunda:asyncBefore="true"` zu ergänzen. Durch diese Konfiguration wird im Falle einer `OptimisticLockingException` nicht die Servicetask, sondern das endende Gateway solange wiederholt, bis der Konflikt gelöst ist [vgl. Camunda Services GmbH, 2021d]. Die Gefahr, dass Geschäftslogik gegebenenfalls doppelt ausgeführt wird, kann so umgangen werden.

Reaktivität

Die Reaktivität, gemessen an den Beispielen genannt im Anforderungskapitel 4.3.2, ist in der Process Engine basierten Implementierung wesentlich durch zwei Konstrukte des BPMN-Standards gegeben: Den Ereignis-Teilprozessen und den ereignisbasierten Gateways. Eingehende Ereignisse in Form von Reisewarnmeldungen werden in der Implementierung durch Ereignis-

Teilprozesse behandelt (vgl. Abbildung 4.23). Der Ereignis-Teilprozess ist mit einer gepunkteten Umrandung markiert. Ausgelöst kann dieser nur werden, wenn der umgebene Prozess aktiv ist. Der Ereignis-Teilprozess kann hierbei entweder unterbrechend (Start-Ereignis mit durchgezogener Linie) oder nicht-unterbrechend (Startereignis mit gestrichelter Linie) gestartet werden [vgl. Freund und Rücker, 2019, 90].

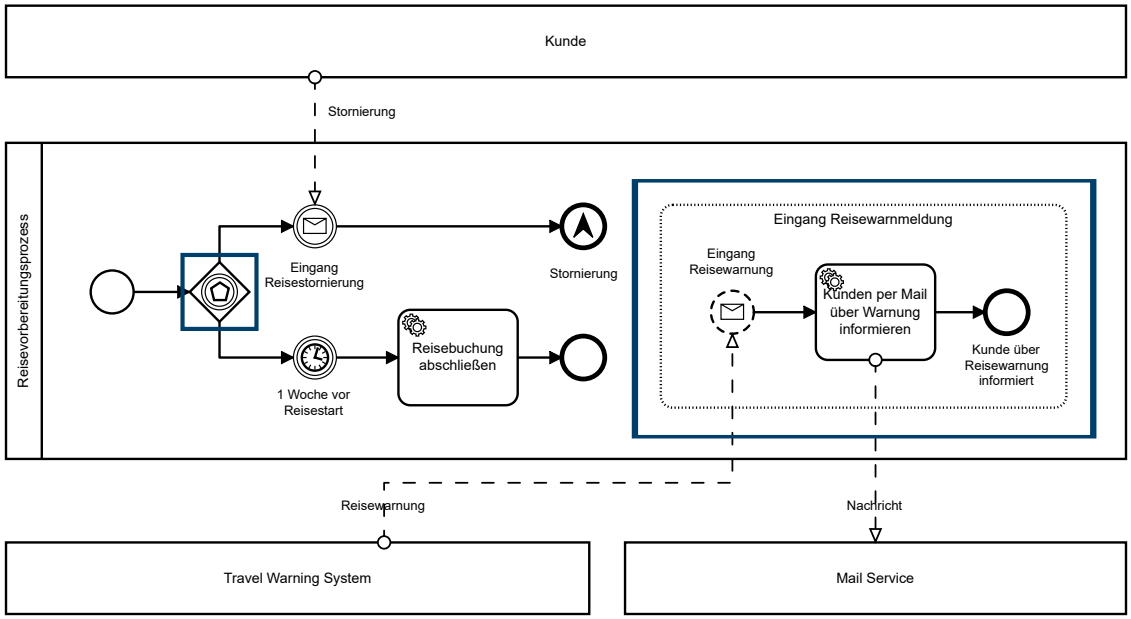

Abbildung 4.23.: BPMN-Modell des Reisevorbereitungsprozesses im Rahmen der Process Engine basierten Implementierung [Eigene Darstellung]

Um sicherzustellen, dass Kunden nur über Reisewarnmeldungen informiert werden, die dessen Reiseziel betreffen, ist es notwendig eine Nachrichtenkorrelation anhand des Reiseziels („destination“) vorzunehmen (siehe Zeile 2 in Listing 4.10). Die Methode `correlateAll()` in Zeile 4 signalisiert hierbei, dass die Reisewarnmeldung an alle aktiven Reisebuchungsprozesse gesendet werden soll.

```
1 runtimeService.createMessageCorrelation("ReceiveTravelWarning")
2         .processInstanceVariableEquals("destination", travelWarning.getCity().toString())
3         .setVariablesLocal(variables)
4         .correlateAll();
```

Listing 4.10: Camunda Nachrichtenkorrelation zur Reisewarnmeldung

Das zweite Konstrukt, das genannt wurde zur Unterstützung der Reaktivität innerhalb des Systems stellen ereignisbasierte Gateways dar. Dieses kommt im Rahmen der Reisevorbereitung zum Einsatz (siehe Raute mit doppeltem Kreis und Fünfeck in Abbildung 4.23). Befindet sich der Prozess an dem ereignisbasierten Gateway, wartet er solange, bis eines der nachfolgenden Ereignisse eintritt. Die Verarbeitung erfolgt danach XOR-basiert, das heißt, nur ein Pfad wird durchlaufen. An dieser Stelle wird entweder eine Woche bis vor Reisestart gewartet um die Reisebuchung abzuschließen oder es trifft eine Stornierung des Kunden ein (symbolisiert durch das Zeit- und Nachrichtenereignis). Trifft eine Stornierung ein, wird die Meldung über ein Eskalationsereignis an den Elternprozess kommuniziert, welcher dann die eigentliche Stornierung der Buchungen durchführt.

4.8. Implementierung nach dem Prozessgesteuerten Ansatz

4.8.1. Zielsetzung und Vorgehensweise der Implementierung

Zielsetzung der Implementierung

Zielsetzung der Implementierung ist es, die in Kapitel 4.3 gestellten Prozess- und Systemanforderungen nach dem Prozessgesteuerten Ansatz (vgl. Kapitel 2.4.5) umzusetzen. Das Reisebuchungssystem soll demnach als prozessgesteuerte Anwendung implementiert werden, wobei entsprechende Best Practices einer prozessgesteuerten Architektur zu berücksichtigen sind.

Vorgehensweise zur Implementierung

Die Vorgehensweise zur Entwicklung einer prozessgesteuerten Anwendung findet, wie im Grundlagen Kapitel 2.4.5 genannt, Top-Down statt. Aus diesem Grund wird zunächst, unabhängig von der bestehenden Systemlandschaft, der fachliche Reisebuchungsprozess in seiner Gesamtheit in BPMN modelliert. Die fachlichen Modelle dienen als Grundlage zur Implementierung der eigentlichen prozessgesteuerten Anwendung mithilfe einer Process Engine. Die Anwendung kapselt dabei jegliche dem Fachprozess zugehörige Funktionalität und verwaltet lokal dessen nötigen Datenstrukturen [vgl. Stiehl, 2013, S. 104]. Im nächsten Schritt wird ausgehend von den fachlichen Anforderungen der PDA ein Servicevertrag definiert. Der Servicevertrag legt fest, welche Daten und Funktionen für die prozessgesteuerte Anwendung von außerhalb erbracht werden. Er stellt damit die Schnittstelle zu der bestehenden Systemlandschaft her. Die technische Umsetzung des Servicevertrags übernimmt letztendlich die Servicevertrag-Implementierungsschicht, dessen Umsetzung im Rahmen des Reisebuchungssystems abschließend in diesem Kapitel beschrieben ist.

4.8.2. Architekturvorstellung

Abbildung 4.24 zeigt das Architekturschaubild zur Implementierung des Reisebuchungssystems nach dem Prozessgesteuerten Ansatz. Grundlegende Komponente dieser Architektur stellt die prozessgesteuerte Anwendung (im Diagramm als „PDA“ abgekürzt) dar. Diese implementiert monolithisch den Reisebuchungsprozess durch den Einsatz der Camunda Process Engine. Das BPMN-Modell des Reisebuchungsprozesses ist in Darstellung 4.25 zu sehen. Hierbei ist anzumerken, dass die Call-Aktivität „Buchungsabwicklung“ in der tatsächlichen Implementierung ein eingebetteter Transaktionsteilprozess ist. Das liegt daran, dass Camunda derzeit keine zugeklappten Teilprozesse und keine nachträgliche Kompensation von Transaktionsteilprozessen in Form einer Camunda Call-Aktivität erlaubt. Da diese Aspekte im BPMN-Standard jedoch vorgesehen sind, wurde zu Zwecken der Übersichtlichkeit der Transaktionsteilprozess zur Buchungsabwicklung in der Abbildung durch eine Call-Aktivität ersetzt.

Das in 4.25 dargestellte Prozessmodell repräsentiert den rein fachlichen und systemunabhängigen Geschäftsprozess. Alle technischen bzw. integrativen Aspekte sind über den Servicevertrag und dessen Implementierungsschicht von dem Fachprozess entkoppelt (siehe Nachrichtenfluss zwischen mittleren und zugeklapptem unteren Pool). Das stellt auch einen Unterschied zu der im vorherigen Kapitel erläuterten Process Engine basierten Implementierung dar. Dort ist der Ende-zu-Ende Geschäftsprozess nach den jeweiligen Bounded Contexts aufgeteilt und es findet keine Trennung zwischen Fach- und Integrationsprozessen statt.

Einzelne Aspekte die nicht zwingend der prozessgesteuerten Anwendung zuzuordnen sind, sind

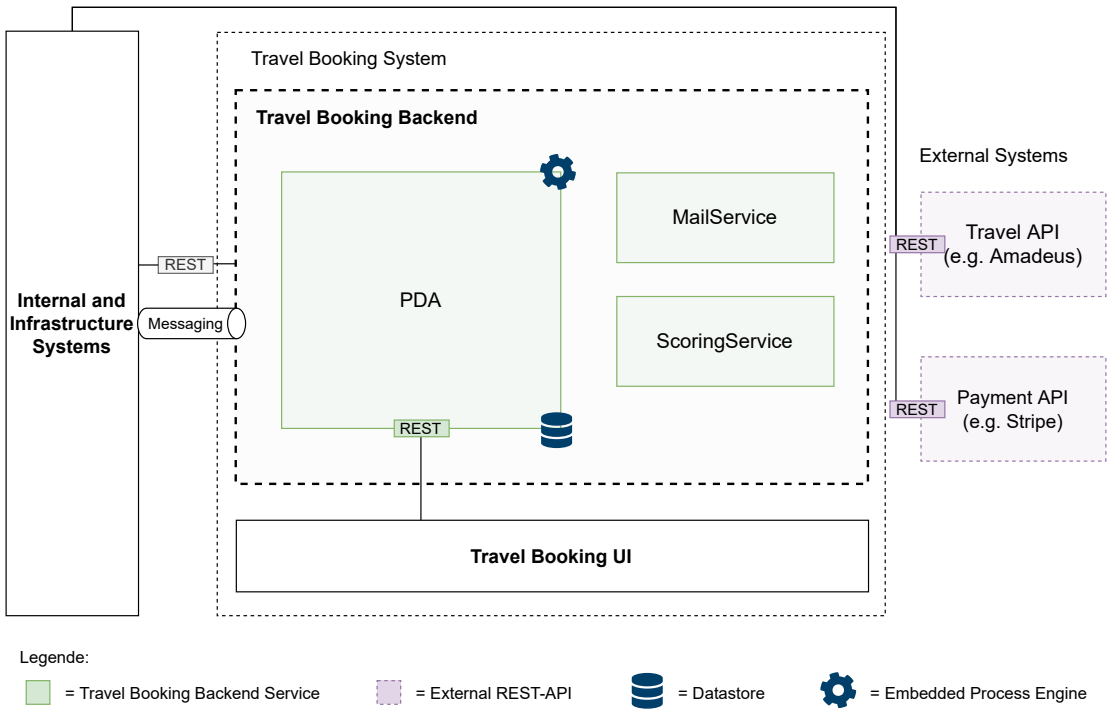

Abbildung 4.24.: Architekturschaubild des Reisebuchungssystems nach der prozessgesteuerten Architektur [Eigene Darstellung]

durch separate Microservices implementiert. Hierzu zählt die Kundenprüfung („ScoringService“) und der „MailService“, der sich um den Mailversand an den Kunden kümmert. Die Kommunikation zwischen PDA und anderen Systemen bzw. Services erfolgt ausschließlich über das Integration System (siehe Verbindung zwischen Travel Booking Backend und „Internal and Infrastructure Systems“ in Abbildung 4.24). Das Integration System verantwortet die Integrationslogik im Rahmen der prozessgesteuerten Architektur zum Zwecke einer besseren Entkopplung und Separation of Concerns. Als Protokolle kommt für den asynchronen Austausch Messaging und für synchrone Aufrufe REST zum Einsatz.

Servicevertrag

Basierend auf dem Top-Down Vorgehen des Prozessgesteuerten Ansatzes werden die fachlichen Anforderungen der prozessgesteuerten Anwendung im ersten Schritt unabhängig von der konkreten Systemlandschaft in Form eines Servicesvertrags definiert. Der Servicevertrag ist rein fachlich motiviert und umfasst nur Datendefinitionen, die auch tatsächlich für die prozessgesteuerte Anwendung von Relevanz sind. Dabei werden sowohl eingehende als auch ausgehende Schnittstellen zwischen prozessgesteuerter Anwendung und der IT-Landschaft betrachtet. Die eigentliche Erbringung des Servicevertrags ist dann Aufgabe der Servicevertrag-Implementierungsschicht [vgl. Stiehl, 2013, S. 28].

Der Servicevertrag vorliegender Implementierung wird beispielhaft anhand der Flugangebotsanfrage erläutert. Ein Ausschnitt des Fachprozesses der prozessgesteuerten Anwendung ist hierzu in Abbildung 4.26 dargestellt. Die Aktivität „Hin-, und Rückflug ermitteln“ sendet eine Anfrage zur Angebotsermittlung an die Servicevertrag-Implementierungsschicht. Als Eingabedaten für

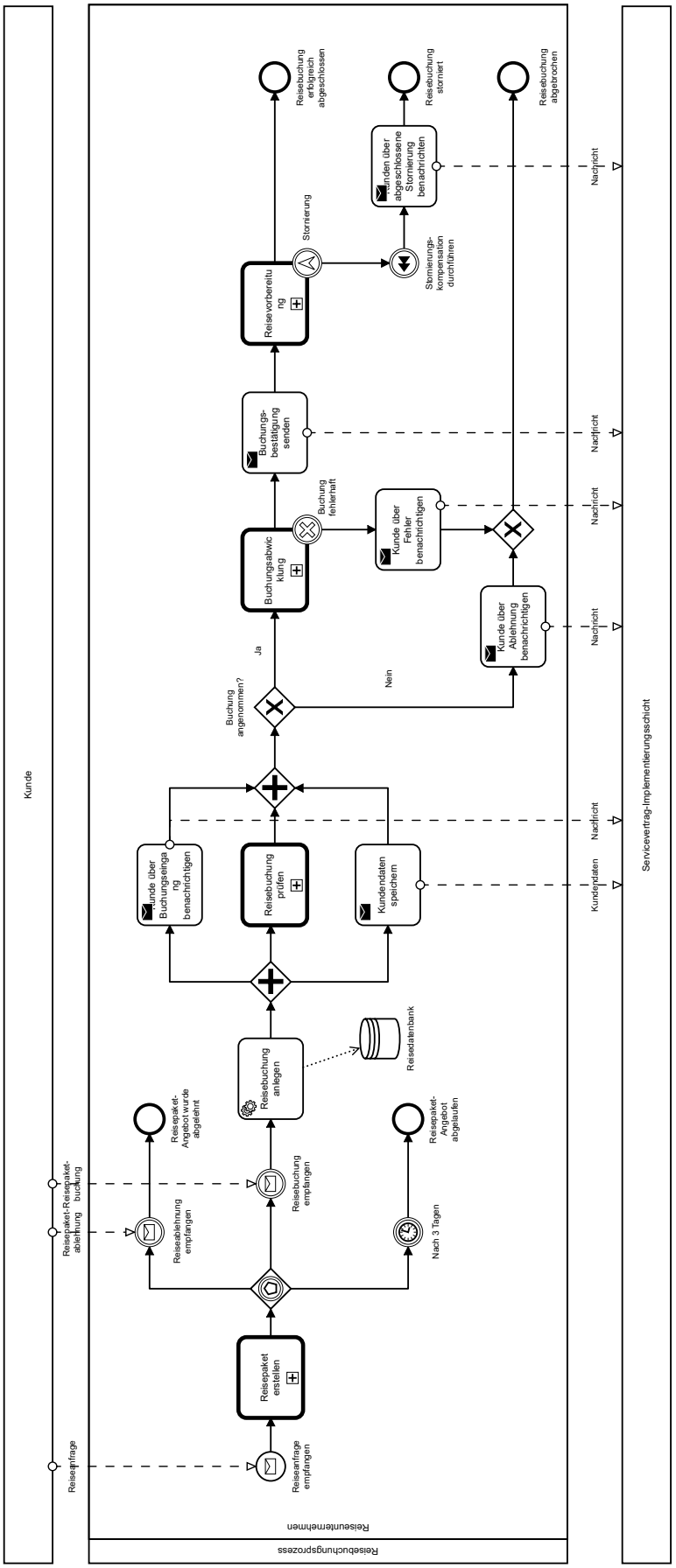

Abbildung 4.25.: BPMN-Modell des Reisebuchungsprozesses im Rahmen der prozessgesteuerten Anwendung [Eigene Darstellung]

die Schnittstelle wird der Abreiseort, das Reiseziel, das Start- und Enddatum und ob es sich um eine Premiumreise handelt angegeben. Die Servicevertrag-Implementierung kann basierend auf diesen Daten ein Hin- und Rückflugangebot ermitteln. Als Antwort erwartet die prozessgesteuerte Anwendung entweder die zwei Flugangebote als Datentyp `FlightOffering` (siehe Datenmodell 4.8) oder einen Anfragefehler. Die Antwort der Servicevertrag-Implementierung erfolgt in diesem Beispiel über das Senden von Nachrichtenereignissen an die prozessgesteuerte Anwendung („Flugangebote erhalten“ und „Flugangebot-Anfragefehler erhalten“). Zur Sicherstellung der Einhaltung des Servicevertrags könnte beispielsweise eine API-Management Plattform (vgl. Kapitel 2.5.4) eingesetzt werden.

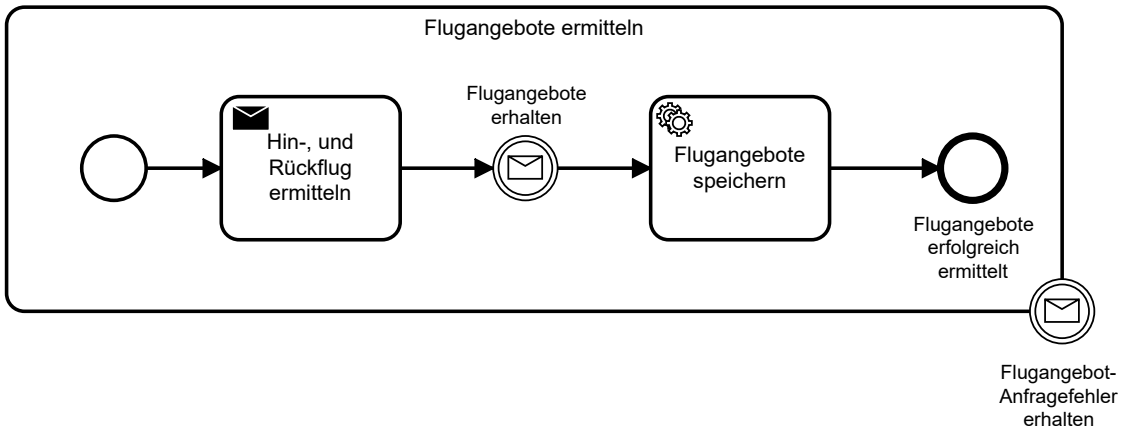

Abbildung 4.26.: Prozessausschnitt zur Flugangebotsanfrage innerhalb der prozessgesteuerten Anwendung [Eigene Darstellung]

Servicevertrag-Implementierungsschicht

Die Aufgabe der Servicevertrag-Implementierungsschicht im Rahmen des Reisebuchungssystems ist es, die definierten Schnittstellen des Servicevertrags zu implementieren und bereitzustellen. Außerdem kümmert sie sich um die Fehlerbehandlung und technische bzw. integrationsbezogene Details zur Anbindung weiterer Systeme (z. B. Routing, Empfängerermittlung oder Datentransformationen). Die Servicevertrag-Implementierungsschicht ist innerhalb der Implementierung über technische Prozessmodelle und dem Integration System realisiert.

Stiehl empfiehlt für eine höhere Flexibilität und bessere Wartbarkeit zustandslose Integrationsaufgaben, wie das Routen oder Mapping von Nachrichten an Backend-Systeme, von den technischen Prozessen zu entkoppeln, sodass Änderungen der Systemlandschaft keine Auswirkungen auf den Integrationsprozess haben [vgl. Stiehl, 2013, S. 122]. Aus diesem Grund wird jegliche Integrationslogik im Rahmen der Implementierung über das Integration System durch insgesamt 14 Camel Routen erbracht. Um jedoch vor allem in technische Abläufe, wie z. B. der parallelen Hin- und Rückflugermittlung zur Implementierung der Flugangebotsschnittstelle, eine bessere Transparenz zu erhalten, sind zusätzlich sechs technische BPMN-Prozessmodelle innerhalb der Servicevertrag-Implementierungsschicht umgesetzt. Die technischen Prozesse kommunizieren hierbei mit den Fachprozessen über Nachrichtenereignisse innerhalb derselben Process Engine. Aus den fachlichen und technischen Prozessmodellen werden die Systeme damit nie direkt, sondern stets über das Integration System entkoppelt angesprochen, sodass jegliche Logik zur Integration und Empfängerermittlung zentral in dem Integration System gebündelt ist. Im Falle einer asynchronen Kommunikation findet der Austausch zwischen technischem Prozess und Integration System über Kafka statt.

Abbildung 4.27 zeigt den technischen Prozess zur Flugangebotsermittlung. Wie zu erkennen kommuniziert dieser über Nachrichtenereignisse mit dem Fachprozess und synchron per Serviceaktivität mit dem Integration System. Das Integration System übernimmt das Routing der Anfrage zur externen Travel API und liefert dessen Antwort zurück an den technischen Prozess der Servicevertrag-Implementierung. Im Vergleich zu der im vorherigen Kapitel behandelten Process Engine basierten Implementierung fällt auf, dass die technischen Details der einzelnen Microservices (z. B. „FlightService“ und „HotelService“) innerhalb der prozessgesteuerten Architektur durch Servicevertrag-Implementierungen in Form technischer Prozesse abgebildet ist. Jegliche Fachlogik der einzelnen Microservices ist extrahiert und zentral in den Fachprozessen der PDA gekapselt.

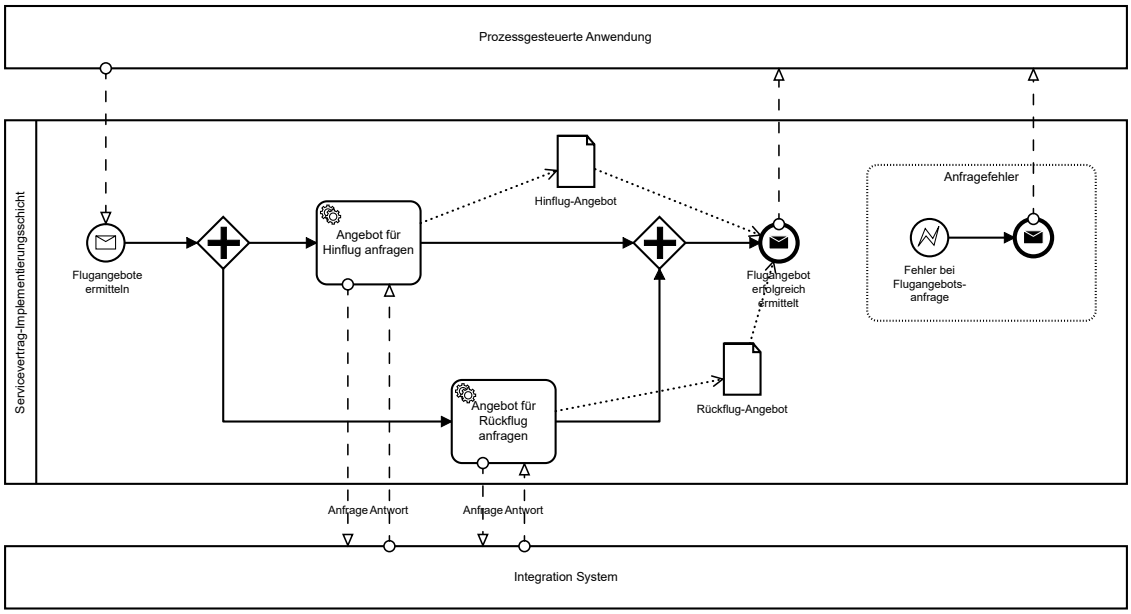

Abbildung 4.27.: Technischer Prozess zur Servicevertrag-Implementierung der Flugangebotsanfrage [Eigene Darstellung]

4.8.3. Validierung der technischen Systemanforderungen

Die Implementierung nach dem Prozessgesteuerten Ansatz soll analog zu den vorherigen Implementierung anhand der technischen Systemanforderungen aus Kapitel 4.3.2 validiert werden.

Integration in bestehende Systemlandschaft

Die Kommunikation mit der bestehenden Systemlandschaft findet im Rahmen der Implementierung über die Servicevertrag-Implementierungsschicht statt. Diese dient als Vermittler zwischen der prozessgesteuerten Anwendung und der hypothetischen Systemlandschaft. Die prozessgesteuerte Anwendung ruft dabei niemals direkt aus den Fachprozessen Systeme auf, sondern stets entkoppelt über technische Prozesse und das Integration System. Da die prozessgesteuerte Architektur eine strikte Verfolgung des Separation of Concerns Prinzips vorsieht, ist jegliche Integrationslogik zur Empfänger- bzw. Schnittstellenermittlung über das Integration System abgewickelt. Die BPMN-Prozesse und das Integration System kommunizieren entweder synchron oder asynchron miteinander. Synchrone Aufrufe werden über Serviceaktivitäten abgewickelt, während asynchrone Aufgaben durch Nachrichtenaktivitäten und -ereignisse modelliert sind.

Als Implementierungsbeispiel kann hier die Reisewarnmeldung angeführt werden. Die Camel Route zur Implementierung des Servicevertrags ist in Listing 4.11 abgebildet. Das Travel Warning System kommuniziert über das Integration System asynchron via Events mit der PDA. Um Kafka-Nachrichten zu konsumieren kommt die Camel-Kafka-Komponente zum Einsatz (Zeile 1). Die Nachricht wird daraufhin in eine Camunda-Message mit spezifischen Attributen und Variablen entsprechend dem Servicevertrag transformiert (Zeile 4-9). Abschließend wird die Nachricht an die bereitgestellte Camunda Messages REST-API gesendet und dabei mit aktiven Reisebuchungsprozessinstanzen korreliert (Zeile 13).

```
1  from("kafka:TravelWarningEvent?brokers=" + this.kafkaBrokerUrl)
2      .convertBodyTo(String.class)
3      .log("Send travel warning to PDA app: ${body}")
4      .setProperty("MessageName", constant("ReceiveTravelWarning"))
5      .setProperty("BusinessKeyVariableName", constant("travelBookingId"))
6      .setProperty("CorrelationKeySource", constant("city"))
7      .setProperty("CorrelationKeyDestination", constant("destination"))
8      .setProperty("All", constant(true))
9      .process(new CamundaMessageProcessor())
10     .setHeader(Exchange.HTTP_METHOD, constant("POST"))
11     .setHeader("Content-Type", constant("application/json"))
12     .log("Camunda API data payload:\n${body}")
13     .to(this.pdaAppUrl + "/engine-rest/message");
```

Listing 4.11: Camel Route zur Erfüllung des Servicevertrag für die Reisewarnmeldung

Integration externer Dienste

Auch die externe Travel API oder Payment API ist über den Servicevertrag von der PDA entkoppelt. In der Beispielimplementierung sind das Integration System und technische Prozesse für die Implementierung zuständig.

Listing 4.12 zeigt beispielhaft die Camel Route zum Buchen eines Hotelangebots über die Travel API (siehe Klasse `de.thi.travel.integrationsystem.pda.booking.BookHotelRoute`). Die Camel Route stellt eine REST-API nach außen zur Verfügung (Zeile 1) und leitet eingehende Anfragen an die Travel API weiter (Zeile 7).

```
1  rest("/booking/hotels").bindingMode(RestBindingMode.off).post()
2      .route()
3      .convertBodyTo(String.class)
4      .log("Book hotel offer ${body}")
5      .setHeader(Exchange.HTTP_METHOD, constant("POST"))
6      .setHeader(Exchange.CONTENT_TYPE, constant("application/json"))
7      .to(this.travelApiUrl + "/booking/hotels?bridgeEndpoint=true")
8      .end();
```

Listing 4.12: Camel Route zur Kommunikation mit der Travel API

Der synchrone Aufruf an die Camel API wird wiederum durch eine Serviceaktivität eines technischen Prozesses der Servicevertrag-Implementierungsschicht angestoßen. Das Ergebnis wird an den Fachprozess der PDA über ein Nachrichtenereignis innerhalb der Process Engine kommuniziert (siehe technisches Prozessmodell zur Hotelbuchung in Abbildung 4.28).

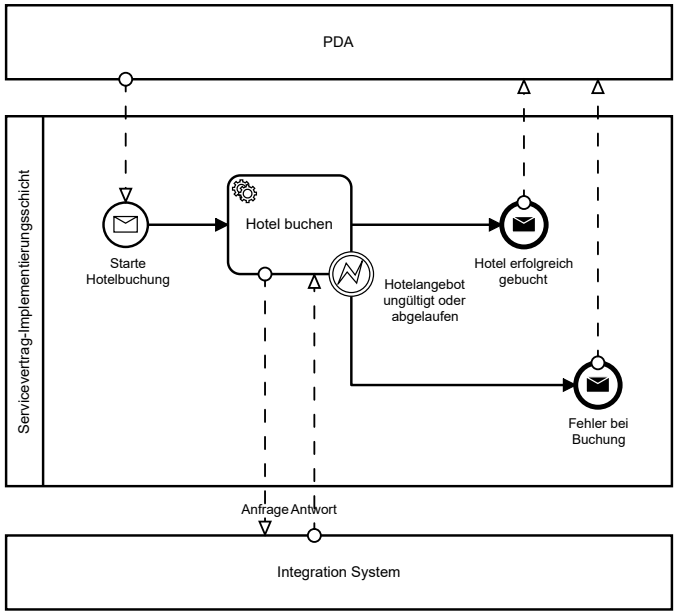

Abbildung 4.28.: Technischer Prozess der Servicevertrag-Implementierung für die Hotelbuchung [Eigene Darstellung]

Technische Resilienz

Aufgrund dem Einsatz einer Process Engine ist die technische Resilienz ähnlich wie bei der Process Engine basierten Implementierung erfüllt. Ausnahme stellen die Retries dar. Diese werden im Rahmen der Implementierung nicht durch die Camunda Process Engine umgesetzt, sondern vom Integration System bei der Verarbeitung berücksichtigt und implementiert. Dafür wird die Java-Klasse `org.apache.camel.builder.DefaultErrorHandlerBuilder` von dem Apache Camel Framework eingesetzt. Über dessen Methoden `maximumRedeliveries()` und `redeliveryDelay()` können die Anzahl der Retries und dessen Zeitspannen dazwischen konfiguriert werden. Eingebunden wird der Error Handler durch Aufruf der `errorHandler()` Methode von der Klasse `org.apache.camel.model.RouteDefinition`. Für die Hotel- und Flugangebotsermittlung sieht die Konfiguration des Error Handler, der im Fehlerfalle eine erneute Verarbeitung nach drei Sekunden startet, wie folgt aus:

```
1 errorHandler(defaultErrorHandler().maximumRedeliveries(1).redeliveryDelay(3000))
```

Neben dem Retry-Verfahren ermöglicht das Integration System durch Apache Camel auch eine einfache Definition von Dead-Letter-Channel, an diese fehlerhafte Nachrichten nach Ablauf der Retries gesendet werden. Dies kann über die Methode `deadLetterChannel()` des genannten `DefaultErrorHandlerBuilder` angegeben werden. Da innerhalb der prozessgesteuerten Architektur jedoch das BPMN-Fehlerereignis zur Behandlung der technischen Fehler eingesetzt wird, ist die Konfiguration eines Dead-Letter-Channels nicht implementiert.

Fachliche Resilienz

Zur Behandlung fachlicher Fehler wird das Eskalationsereignis der BPMN-Spezifikation eingesetzt. Im Unterschied zur bereits behandelten Process Engine basierten Implementierung können innerhalb des Fachprozesses der PDA fachliche Fehler gesamtheitlich behandelt werden. Das liegt daran, dass der Geschäftsprozess nicht wie bei der Process Engine basierten

Architektur nach den jeweiligen Bounded Contexts aufgeteilt ist, sondern als kompletter Ende-zu-Ende-Prozess vorliegt.

Fehleridentifikation

Die Fehleridentifikation ist durch die gleichen Methoden wie bei den vorherigen Implementierungen gegeben. Durch die Trennung von fachlichen und technischen Prozessen und der Kapselung jeglicher Integrationslogik innerhalb des Integration System können auftretende Fehler isoliert und damit einfacher identifiziert werden.

Nachvollziehbarkeit

Generell ermöglicht die prozessgesteuerte Architektur eine gesamtheitliche Nachvollziehbarkeit des Geschäftsprozesses, da dieser Ende-zu-Ende modelliert und implementiert ist. Da bei der Implementierung die Camunda Process Engine zum Einsatz kommt, wird das bereits genannte Camunda Cockpit (vgl. Kapitel 4.7.3) verwendet. Dieses wird zusammen mit der Workflow Engine als Teil der prozessgesteuerten Anwendung mit ausgeliefert, sodass zentral ein Überblick über alle Prozesse gewonnen werden kann.

Transaktionalität

Aufgrund dem Einsatz einer Process Engine ist die Transaktionalität analog zur Process Engine basierten Implementierung durch den Einsatz von Transaktionsteilprozessen erfüllt. In der Implementierung nach dem PDA ist die Transaktionalität zusätzlich auf fachlicher und technischer Ebene entkoppelt.

Parallelität

Aufgrund dem Einsatz einer Process Engine ist die Parallelität analog zur Process Engine basierten Implementierung durch den Einsatz von parallelen Gateways erfüllt. Die Implementierung nach dem Prozessgesteuerten Ansatz trennt jedoch im Vergleich zur Process Engine basierten Implementierung zwischen fachlichen und technischen Prozessen. Durch die Entkopplung der bestehenden Systemlandschaft über das Integration System ist es dadurch möglich, die Servicevertrag-Implementierungsschicht echt parallel zu implementieren. Technische Aufgaben zur Anbindung von Systemen können so effizient abgewickelt werden, wie beispielsweise das Einholen von Hotel- oder Flugangeboten. Der fachliche Geschäftsprozess bleibt dabei unberührt.

Reaktivität

Aufgrund dem Einsatz einer Process Engine ist die Reaktivität analog zur Process Engine basierten Implementierung durch den Einsatz von Ereignis-Teilprozessen und Ereignisbasierten Gateways erfüllt.

5. Architekturvergleich anhand ausgewählter Qualitätsmerkmale

Es wurden nun die einzelnen Implementierungen des Reisebuchungsprozesses nach den jeweiligen Architekturstilen vorgestellt. Dieses Kapitel widmet sich dem Architekturvergleich. Dieser findet auf Grundlage der in Kapitel 2.1.4 vorgestellten Qualitätsmerkmalen gemäß DIN/25010 (siehe Anhang A.1) statt. Die einzelnen Merkmale hängen dabei oft in wechselseitiger Wirkung miteinander zusammen und beeinflussen bzw. unterstützen sich. Für den Architekturvergleich werden den einzelnen Architekturstilen außerdem Abkürzungen zugeordnet, die so auch im Quellcode der Implementierungen verwendet wurden. Diese werden durchgängig für eine bessere Verständlichkeit und Zuordnung innerhalb des Vergleichs verwendet und sind wie folgt definiert:

- EDAC = Ereignisgesteuerte Architektur nach dem Choreography Pattern
- EDAO = Ereignisgesteuerte Architektur nach dem Orchestration Pattern
- PE = Process Engine basierte Architektur
- PDA = Architektur nach dem Prozessgesteuerten Ansatz

5.1. Auswahl der Qualitätsmerkmale

In Kapitel 2.1.4 wurden Architekturqualitätsmerkmale nach quantitativen und qualitativen Aspekten unterteilt. Im Rahmen dieser Arbeit werden die Architekturen überwiegend nach qualitativen Merkmalen bewertet, da ein quantitativer Vergleich je nach Implementierung stark variieren kann. Als Kriterien für den Architekturvergleich werden dabei die Qualitätsmerkmale gemäß DIN/25010 (siehe Anhang A.1) herbeigezogen. Jedes Hauptmerkmal unterteilt sich in weitere Untermerkmale, die es im Rahmen der Analyse zu betrachten gilt.

Die „Funktionale Eignung“, „Effizienz“, „Betreibbarkeit“ und „Sicherheit“ werden als Merkmale nicht berücksichtigt: Die funktionale Eignung ist im Rahmen des Vergleich nicht relevant, da durch die Prozess- und Systemanforderungen (vgl. Kapitel 4.3) sichergestellt ist, dass alle Implementierungen den Reisebuchungsprozess in gleichem Umfang und funktional korrekt abbilden. Die Effizienz geht als Merkmal in Richtung der quantitativen Analyse, die wie vorhin beschrieben kein Teil dieser Arbeit ist. Das Merkmal der Betreibbarkeit (engl.: „usability“) setzt den Fokus auf die Bedienbarkeit bzw. Benutzerinteraktion mit dem System. Da die Interaktion zentral für alle Implementierungen über eine Webanwendung (siehe Travel Booking UI in Kapitel 4.4) gesteuert wird, ist eine Betrachtung dieses Merkmals hinfällig. Auch die Sicherheit als Betrachtungsgegenstand wird ausgeschlossen, da Sicherheitskonzepte innerhalb dieser Arbeit nicht thematisiert und deshalb auch kein Teil der Implementierung sind.

Übrig bleiben als zu untersuchende Qualitätsmerkmale deshalb die „Zuverlässigkeit“, „Kompa-

tibilität“, „Wartbarkeit“ und „Übertragbarkeit“. Im Folgenden werden nun die unterschiedlichen Architekturstile anhand dieser Merkmale und ihren Unterpunkten untersucht und bewertet.

5.2. Merkmal: Zuverlässigkeit

Die Zuverlässigkeit spiegelt die Fähigkeit des Systems wieder, unter diversen Bedingungen die Leistungserbringung aufrechtzuerhalten. Die Architekturimplementierungen werden anhand der Untermerkmale nach DIN/25010 untersucht. Dazu zählt die Verfügbarkeit, Fehlertoleranz und Wiederherstellbarkeit.

Verfügbarkeit

Die Verfügbarkeit des Systems geht inhärent mit der Fähigkeit zur Skalierung einher, weshalb beide Aspekte gemeinsam am Beispiel der prototypischen Implementierungen betrachtet werden. Nach Wolff profitieren vor allem Microservices von einer hohen Verfügbarkeit, da diese horizontal skaliert werden können [vgl. Wolff, 2018, S. 150]. Microservices sind die Basis der im Rahmen dieser Arbeit betrachteten EDAC, EDAO und PE Umsetzungen. Im Falle einer erhöhten Nachfrage an Reisebuchungen, wie es beispielsweise saisonal bedingt sein kann, könnte es zu einer Überlastung bei der Reisepaketgenerierung kommen. Um dem entgegenzuwirken könnten beispielsweise der „TravelPackageService“, „HotelService“ und „FlightService“ unabhängig voneinander einzeln skaliert werden. Die Lastverteilung kann dabei von einem Load Balancer übernommen werden. Ein solches Vorgehen wäre bei der monolithischen Implementierung nach der PDA nicht denkbar, da dort jegliche Fachlogik zur Reisepaketgenerierung zentral in der prozessgesteuerten Anwendung gekapselt ist. Eine explizite Skalierung einzelner Teilprozesse oder Funktionalitäten des Fachprozesses ist deshalb nicht gegeben. Was die PDA allerdings ermöglicht, ist eine Skalierung der Servicevertrag-Implementierungsschicht. Hier können beispielsweise Microservices eingesetzt werden, die die eigentliche Anbindung an die Systemlandschaft hochverfügbar abwickeln.

Ein weiterer Aspekt der im Kontext der Verfügbarkeit bei der Skalierung zu beachten ist, ist die Dauer, die ein einzelner Service zum Starten benötigt. Wie sich bei der Implementierung des Reisebuchungssystems gezeigt hat, benötigen Services die eine Camunda Process Engine eingebettet haben im Durchschnitt länger bis die Applikation einsatzbereit ist. Das liegt vor allem an den umfassenden Funktionen, die eine Process Engine standardmäßig bereitstellt, wobei es natürlich je nach verwendeter Workflow Engine variieren kann. Besonders deutlich ist dieser Effekt innerhalb der PE-Architektur aufgefallen, da dort jeder der Services eine eigene Workflow Engine verwendet. Diesem Kriterium nach ist die EDAC und EDAO in Bezug auf die Startzeit eines Services überlegen.

Fehlertoleranz

Die Fehlertoleranz kann anhand der in Kapitel 4.3.2 gestellten Resilienz Anforderungen untersucht werden. Die ereignisgesteuerten Implementierungen (EDAC, EDAO) setzen in einem Fehlerfalle auf Retry-Verfahren und Dead Letter Queues, um entweder eine erneute Verarbeitung anzustoßen oder den Fehler explizit zu behandeln (siehe Implementierungskapitel 4.6.4). Auch die PE und PDA setzen Retry-Verfahren und Dead Letter Queues ein. Zusätzlich profitieren diese Architekturen jedoch von dem BPMN-Fehlerereignis zur expliziten Behandlung auftretender Fehler. Befindet sich ein Geschäftsprozess in einem fehlerhaften Zustand, ermög-

licht die Process Engine außerdem das Zurücksetzen des Prozesses zu einem vorherigen Schritt oder das Anpassen korrupter Prozessdaten während der Laufzeit (vgl. Kapitel 4.7.3). Ein solcher Funktionsumfang ist nur über zusätzlichen Implementierungsaufwand im Rahmen der EDAC und EDAO realisierbar. Dabei ist jedoch zu beachten, dass im Rahmen der PE Implementierung Änderungen des Prozesszustands immer nur auf den lokalen Geschäftsprozess des jeweiligen Services beschränkt sind. Die PDA liefert im Vergleich dazu eine ganzheitliche Ende-zu-Ende Betrachtung des Reisebuchungsprozesses und damit mehr Flexibilität im Hinblick auf aktive Zustandsänderungen, die sich über den gesamten Prozess erstrecken können.

Die Fehlertoleranz im Hinblick auf die Nichteinhaltung spezifizierter Schnittstellen ist außerdem am besten bei der PDA gegeben. Durch die Entkopplung der prozessgesteuerten Anwendung von externen Systemen über den Servicevertrag, ist die Einhaltung der Schnittstellen immer sichergestellt. Auftretende Fehler werden von der Servicevertrag-Implementierungsschicht behandelt. Der direkte Aufruf externer Systeme aus den einzelnen Microservices im Kontext der EDAC, EDAO und PE Implementierungen führt zu einer engeren Bindung und Kopplung und damit erhöhter Fehleranfälligkeit. Generell empfiehlt sich im Kontext aller Architekturen der Einsatz einer zentralen API-Managementplattform zur Sicherstellung und Einhaltung der Schnittstellenverträge.

Wiederherstellbarkeit

Die Wiederherstellbarkeit ist im Kontext ereignisgesteuerter Architekturen (EDAC, EDAO) eine Herausforderung, da sie nicht inhärent gegeben ist. Lösen lässt sich dieses Problem, indem auf die Replay Funktionalitäten des Event Brokers aufgebaut werden. Diese basieren darauf, dass Event Broker, im Vergleich zu Message Broker, als persistenter Speicher und „Single Source of Truth“ fungieren. Event-Driven Microservices, die während der Verarbeitung ausfallen, müssen also entweder alle aktuellen und historischen Ereignisse über Kafka abrufen und damit ihren Zustand wiederherstellen oder all diese Daten selbstständig in einer Datenbank verwalten. In jedem Fall ist hier bei der EDAC und EDAO mit einem zusätzlichen Implementierungsaufwand zu rechnen. Fällt im Rahmen des Reisebuchungssystem beispielsweise der „Hotel-Service“ während einer Hotelbuchung aus, so kann dieser sobald er wieder verfügbar ist den aktuellen Zustand der Buchungsabwicklung über einen Ereignisstrom von Kafka abfragen. Dabei ist jedoch zu beachten, dass der Prozess immer nur bis zu dem Zustand des zuletzt gesendeten bzw. gespeicherten Ereignisses wiederhergestellt werden kann. Jegliche Ergebnisse aus intern abgeschlossenen Zwischenschritten können unter Umständen — wenn keine sonstigen Mechanismen zur Wiederherstellbarkeit implementiert sind — verloren gehen. Die EDAC Implementierung würde hierbei mit einem höheren Implementierungsaufwand einhergehen als die EDAO Umsetzung, da im Falle des Orchestration Pattern der Orchestrierungsservice jegliche Logik für die Wiederherstellbarkeit des Prozesszustands zentral abbilden kann.

Im Vergleich zur EDAC und EDAO verhalten sich Architekturen, die den Reisebuchungsprozess über eine Workflow Engine abbilden (PE und PDA), anders. Die Process Engine speichert die Zustände der Prozessinstanzen in der Datenbank ab und kann bei Ausfall diese wiederherstellen. Die Funktionalität zur Wiederherstellbarkeit ist damit implizit durch die Process Engine gegeben, wodurch im Gegensatz zu ereignisgesteuerten Architekturen zusätzlicher Implementierungsaufwand und damit das Auftreten möglicher technischer Schulden verhindert werden kann.

5.3. Merkmal: Kompatibilität

Mit der Kompatibilität wird die Fähigkeit zum Informationsaustausch und die Integration der Systeme untereinander untersucht. Untermerkmale sind die Ersetzbarkeit, Koexistenz und Interoperabilität.

Ersetzbarkeit

Die Ersetzbarkeit geht mit dem Prinzip der Modularisierung einher. Modulare Einheiten mit klaren Schnittstellen können einfacher ausgetauscht und eingesetzt werden. Verteilte Systeme auf Basis von Microservices gehen mit dem Vorteil einher, dass individuelle Services unabhängig voneinander ersetzt werden können. Dadurch ist während der Betriebslaufzeit eine schrittweise Ersetzbarkeit auf Serviceebene gegeben. Eine monolithische Applikation hat eine solche Ersetzbarkeit standardmäßig nicht gegeben. Hier besteht zwar die Möglichkeit einzelne Teilfunktionalitäten aus dem System auszulagern, dies ist jedoch mit einer erneuten Bereitstellung der Applikation verbunden.

Koexistenz

Gemäß DIN/25010 beschreibt die Koexistenz die Fähigkeit, mit anderen Systemen in der bestehenden Systemlandschaft ohne beeinflussender Wirkung zu bestehen. Bei Betrachtung der Beispielimplementierungen kann kein nachteiliger Effekt der Implementierungen auf die hypothetische Systemlandschaft festgestellt werden. Basierend auf diesem Untermerkmal können deshalb keine Rückschlüsse auf Architekturunterschiede geschlossen werden.

Interoperabilität

Die Interoperabilität betrachtet die Fähigkeit zur Kooperation des Reisebuchungssystems mit anderen Systemen. Die EDAC, EDAO und PE Implementierung setzen bei der Interoperabilität auf nachrichtenbasierte Kommunikation. Daten werden über Ereignisse zwischen den Systemen ausgetauscht. Synchrone Aufrufe werden durch die Services intern abgewickelt, wie beispielsweise bei der Anfrage des „HotelService“ an die externe Travel API zur Ermittlung der Hotelangebote. Bei der Kommunikation mit den internen Systemen fungiert das Integration System als Adapter, um zwischen Nachrichten-basierter Kommunikation und diversen Schnittstellen — in der Beispielimplementierung SOAP und CSV-Import — zur Anbindung weiterer Systeme zu übersetzen. Innerhalb großer Unternehmen kommen hierbei üblicherweise API-Management-Plattformen zum Einsatz, dessen Aufgabe eine zentrale Anbindung und Bereitstellung aller internen und externen Systemschnittstellen ist. Die PDA Implementierung sieht im Gegensatz dazu eine Entkopplung zwischen prozessgesteuerter Anwendung und der bestehenden Systemlandschaft in Form eines Servicevertrags vor. Jegliche Art der Kooperation mit bestehenden Systemen erfolgt stets über das Integration System, das für die Erfüllung des Servicevertrags zuständig ist.

Im Vergleich zur PDA Implementierung haben Systemlandschaftsänderungen innerhalb der EDAC, EDAO und PE Implementierung direkte Auswirkungen auf die einzelnen Fachprozesse. Die PE Implementierung trennt beispielsweise nicht zwischen technischen und fachlichen Prozessen, sondern spricht externe Systeme direkt aus dem Fachprozess heraus an (siehe Serviceaufgaben zum Speichern der Kundendaten innerhalb des „BookingOrchestratorService“). Kommen neue Systeme hinzu oder müssen bestehende ausgetauscht bzw. verändert werden,

hat das direkte Auswirkungen auf den Fachprozess, was eine Anpassung und erneute Bereitstellung notwendig macht. Mit zunehmender Zeit besteht außerdem die Gefahr, dass das fachliche Modell mit vielen integrationsspezifischen Details belastet wird und dadurch der Kernprozess verloren geht. Die PDA Implementierung verfolgt hier strikt das Prinzip des „Separation of Concerns“ und sieht eine klare Trennung zwischen fachlichen und integrationzentrischen Prozessen vor. Die prozessgesteuerte Anwendung ist über den Servicevertrag von der bestehenden Systemlandschaft entkoppelt. Systemänderungen wirken sich deshalb nicht auf die Fachprozesse aus, sondern erfordern eine Anpassung der Servicevertrag-Implementierungsschicht. Ändert sich beispielsweise die Schnittstelle der externen Travel API, wird die Servicevertrag-Implementierung dieser zur Erbringung der zugehörigen Servicevertrag-Schnittstelle angepasst. Ein weiteres denkbares Szenario wäre, dass Hotelangebote von mehreren externen Diensten bezogen werden sollen. In diesem Fall wird die Servicevertrag-Implementierung um die Anbindung der weiteren Systeme ergänzt. Gegebenenfalls kann diese noch Datentransformationen durchführen, sodass die Liste der Hotelangebote dem Datenformat des Servicevertrags entsprechen. Die eigentliche prozessgesteuerte Anwendung bleibt von der Erweiterung unberührt, da keine Änderungen am Servicevertrag notwendig sind.

5.4. Merkmal: Wartbarkeit

Die Wartbarkeit bezieht sich darauf, wie hoch der Aufwand für Änderungen am System ist. Änderungen entstehen unter anderem durch das Umsetzen neuer Anforderungen, als auch die Anpassung und Verbesserung bestehender Funktionalität. Ein hoher Wartungsaufwand ist ein Indikator für hohe technische Schulden (vgl. Kapitel 2.1.3). Im Rahmen dieser Arbeit steht die Wartbarkeit der Geschäftsprozesse zusätzlich im Fokus. Als Untermerkmale der Wartbarkeit führt DIN/25010 die Modularität, Wiederverwendbarkeit, Analysierbarkeit, Modifizierbarkeit (Änderbarkeit), Stabilität und Prüfbarkeit auf.

Modularität

Das Streben nach Modularität wurde bereits als Architekturprinzip im Grundlagenkapitel vorgestellt (vgl. Kapitel 2.1.2). Für die Wartbarkeit eines Systems spielt diese eine wichtige Rolle: Ist das System modular aufgebaut, so haben Änderungen an Komponenten nur wenig Auswirkungen auf andere Komponenten. Der Änderungsbereich kann so eingedämmt werden.

Das Architekturschaubild der EDAC zeigt, dass das Reisebuchungssystem entsprechend einer fachlichen Schichtung entlang der Bounded Contexts in mehrere Services aufgeteilt ist. Diese agieren als modulare Einheiten und kommunizieren über Events miteinander (siehe Abbildung 4.10). Microservices im Allgemeinen sind für ihr Modularisierungskonzept, ihre Agilität und der einhergehenden losen Kopplung bekannt: Ein Qualitätsmerkmal, welches ereignisgesteuerte Architekturen nach Bruns und Dunkel auch aufweisen [vgl. Bruns und Dunkel, 2010, S. 73]. EDAC weisen zwar eine lose Kopplung auf Serviceebene auf, sind jedoch auf Basis des Ereignisaustauschs indirekt voneinander abhängig. Rücker warnt in diesem Kontext davor, dass der Ereignisaustausch implizit den logischen Ablauf des Geschäftsprozesses durch „Event Chains“ abbildet [vgl. Ruecker, 2021, S. 150 ff.]. Der Prozessfluss wird nicht zentral durch eine Instanz verwaltet, sondern über die Kooperation der Services sichergestellt. Die Services sind deshalb indirekt auf fachlicher Ebene stark miteinander gekoppelt, was Geschäftsprozessänderungen erschwert. Auch Martin Fowler zeigt die Risiken einer solchen Architektur auf: Der Prozessfluss, der durch das Verketten von Ereignissen entsteht, ist nicht explizit sichtbar. Das kann vor al-

lem bei komplexen Systemen zu einer schlechten Wartbarkeit und hohen technischen Schulden führen [vgl. Fowler, 2017]. Dieses Verhalten ist auch im Rahmen der Beispielimplementierung zu erkennen (siehe Merkmal zur Modifizierbarkeit).

Durch die Kapselung der Prozesslogik in zentrale Orchestrierungsdienste — wie es bei der EDAO und PE Implementierung der Fall ist — kann die Modularität im Hinblick auf die Wartung der Geschäftsprozesse weiter verbessert werden. Wie anhand des Reisebuchungssystems zu erkennen ist, ist jegliche Prozesslogik zentral in dem „TravelBookingOrchestrator“ und „TravelPackageOrchestrator“ Service implementiert, wo Veränderungen an dem Prozessablauf vorgenommen werden können. Process Engines unterstützen zusätzlich bei der Modularität zur Trennung der Prozess-, Geschäfts- und Entscheidungslogik. Die Prozesslogik wird durch BPMN Prozesse abgebildet. Die Geschäftslogik ist in Java Delegates gekapselt und jegliche Entscheidungslogik kann nach Bedarf in DMN-Tabellen modelliert sein.

Die monolithische Implementierung des Reisebuchungssystems nach der PDA setzt auf das Prinzip des „Separation of Concerns“. Markant ist dabei die Trennung von Fach- und Integrationsprozessen. Änderungen externer Systeme oder technischer Details wirken sich so nicht auf die fachlichen Prozesse der prozessgesteuerten Anwendung aus. Da die prozessgesteuerte Anwendung als Monolith implementiert ist, ist jedoch auf eine saubere Trennung der fachlichen Module zu achten, die bei dem Einsatz von Microservices implizit gegeben ist. Die Prozessgesteuerte Anwendung profitiert hier analog zur PE Implementierung von der Verwendung von Teilprozessen zur modularen Strukturierung der Geschäftsprozesse.

Wiederverwendbarkeit

Mit der Wiederverwendbarkeit wird betrachtet, inwiefern einzelne Teile des Systems intern wiederverwendet oder für andere Systeme genutzt werden können. Das Merkmal der Wiederverwendbarkeit geht hierbei mit der Modularität einher: Modulare Einheiten des Systems können Schnittstellen nach außen bereitstellen, die wiederum von anderen Systemen genutzt werden können.

Bei der EDAC, EDAO und PE Implementierung erbringen die Event-Driven Microservices über ihre Zusammenarbeit den fachlichen Ende-zu-Ende Geschäftsprozess, der über die einzelnen Services verteilt ist. Entsprechend der Aufteilung des Systems können hierbei Komponenten wiederverwendet werden. Im Rahmen des Reisebuchungssystem kann der „HotelService“ beispielsweise auch für andere Systeme innerhalb des Unternehmens eingesetzt werden, die ein Hotelangebot benötigen. Eine solche Wiederverwendbarkeit setzt bei Event Driven Microservices jedoch eine Anpassung voraus, da gegebenenfalls unerwünschte Folgeaktivitäten durch Events ausgelöst werden. Beispielsweise wenn ein anderes System die Hotelangebotsanfrage mittels Event anstößt und die Nachricht mit dem Hotelangebot zusätzlich vom „TravelPackageService“ oder „TravelPackageOrchestratorService“ im Kontext des Reisebuchungsprozesses konsumiert wird. Zusätzliche Anpassungen oder Erweiterungen des Service sind notwendig, was zu einer Überladung des ursprünglich ersinnten, kompakten Microservices führen kann.

Die EDAC, EDAO und PE Implementierungen zielen durch den Ereignisaustausch auf eine hohe Autonomie und lose Kopplung untereinander ab. Eine Konsequenz dessen ist, dass die Services keine internen Komponenten wiederverwenden und deshalb Datenobjekte und Operationen redundant im System implementieren müssen. Bruns und Dunkel [vgl. Bruns und Dunkel, 2010, S. 74] stellen dabei klar, dass eine solche Redundanz die Wartbarkeit des Systems beeinflusst. Änderungen der redundanten Logik sind an unterschiedlichen Stellen vorzuneh-

men und können dabei zu Inkonsistenzen im System führen. Im Gegensatz dazu ermöglicht es die monolithische Struktur der PDA gleiche Bausteine innerhalb der Applikation an mehreren Stellen wiederzuverwenden. Außerdem können Geschäftsprozesse durch die Verwendung von Teilprozessen modular gestaltet und einfach in unterschiedliche Prozesse eingebettet werden. Auch technische Details der Servicevertrag-Implementierungsschicht sind zur mehrfachen Verwendung zentral im Integration System gebündelt.

Generell empfiehlt es sich für die Anbindung und Verwaltung von Systemschnittellen eine API Management Plattform einzusetzen. Diese kann dazu beitragen Redundanz im System zu reduzieren, indem jegliche Integrations- und Schnittstellenlogik an einer zentralen Stelle gekapselt und bereitgestellt wird. API Management Plattformen können durch die Bündelung der APIs an einer Stelle einen zentralen Anlaufpunkt zur Nutzung der Services bieten.

Analysierbarkeit

Die Analysierbarkeit geht mit der Anforderungen an die Nachvollziehbarkeit und Fehleridentifikation des Reisebuchungsprozesses einher. Im Rahmen der EDAC und EDAO Implementierung ist die Analysierbarkeit eine Herausforderung. Das liegt vor allem an der einhergehenden Komplexität die aus einem verteilten, ereignisgesteuerten System resultiert. Das Nachvollziehen des Prozessflusses ist nur bedingt durch zusätzlichen Implementierungsaufwand möglich. Dies führt zu technischen Schulden, was bei großen komplexen Systemen die Wartbarkeit erschwert. Anhand der Beispielimplementierung ist zu erkennen, dass die EDAC und EDAO Implementierung bei der Systemanalyse auf eine gebündelte Auswertung der Applikationslogs und Verwendung von Distributed Tracing Tools, wie beispielsweise Zipkin, angewiesen ist. Da hier nur eine technische Analysierbarkeit des Systems gegeben ist, besteht für die EDAC und EDAO die Notwendigkeit einen zusätzlichen „MonitoringService“ zu implementieren, der alle auftretenden Geschäftsereignisse konsumiert und den Ereignisverlauf der einzelnen Reisebuchungsanfragen darstellt (siehe hierzu technische Systemanforderung zur Nachvollziehbarkeit in Kapitel 4.6.4). Durch die exemplarische Implementierung des Reisebuchungssystems wird also deutlich, dass eine Analysierbarkeit des Geschäftsprozesses nur erreichbar ist, indem unterschiedlichste Tools und Dienste eingesetzt bzw. eigens hierfür entwickelt werden. Die Analysierbarkeit ist im Rahmen der ereignisgesteuerten Architektur deshalb immer mit einem Mehraufwand in der Umsetzung verbunden.

Die PE und PDA Implementierung unterstützten die Analysierbarkeit generell besser, da diese von der Verwendung einer grafischen Prozessmodellierungssprache und den Analysetools einer Workflow Engine profitieren (siehe Camunda Cockpit) und somit bereits implizit Funktionalitäten zur Analyse bieten. Die Engine speichert automatisch alle Prozesszustände und auftretende Fehler an den Aktivitäten ab, sodass diese schnell und transparent eingesehen und behandelt werden können. Hilfreich ist hierbei auch, dass Fehler explizit Geschäftsprozessaktivitäten zugeordnet werden. So kann im Rahmen der Fehleranalyse das Problem schnell auf einen Prozessschritt innerhalb einer Instanz eingegrenzt werden. Eine solche Analyse ist in ereignisgesteuerten Architekturen mit einem höherem Aufwand verbunden. Die Prozessvisualisierung über BPMN unterstützt dabei, dass auch Domänenexperten den Prozessfluss im System verstehen und analysieren können, ohne dabei technische Log-Dateien oder Traces auswerten zu müssen. Die PE Implementierung geht im Vergleich zu der PDA Implementierung allerdings mit dem Nachteil einher, dass der Ende-zu-Ende Prozess über mehrere Services und Process Engines verteilt ist. Eine Analyse des Geschäftsprozesses in seiner Gesamtheit ist nur bedingt durch zusätzlichen Implementierungsaufwand zur Prozesskonsolidierung erreichbar. Prozessgesteuerte Architekturen betrachten den fachlichen Prozess gesamtheitlich und bieten damit

ohne Mehraufwand die Möglichkeit eine zentralen Monitorings an.

Modifizierbarkeit (Änderbarkeit)

Bei der Modifizierbarkeit bzw. Änderbarkeit wird untersucht wie hoch der Aufwand innerhalb der einzelnen Architekturen ist, Änderungen an dem Reisebuchungsprozess vorzunehmen. Darunter fällt auch der Aspekt der Erweiterbarkeit, um zu prüfen, wie schnell neue Anforderungen an den Fachprozessen umgesetzt werden können.

Zunächst wird betrachtet, welche Auswirkungen eine Änderung des Prozessflusses auf die Beispielimplementierungen hat. Dies wird anhand der Buchungsabwicklung exemplarisch erläutert. Angenommen die Hotel- und Flugbuchung soll erst nach erfolgreichem Zahlungseingang des Kunden vorgenommen werden. Die Zahlungsabwicklung muss also nun an erster Stelle in der Buchungsabwicklung abgearbeitet werden. Abbildung 5.1 zeigt, welche Anpassungen im Rahmen der EDAC Implementierung hierfür notwendig sind. Wie durch die rote

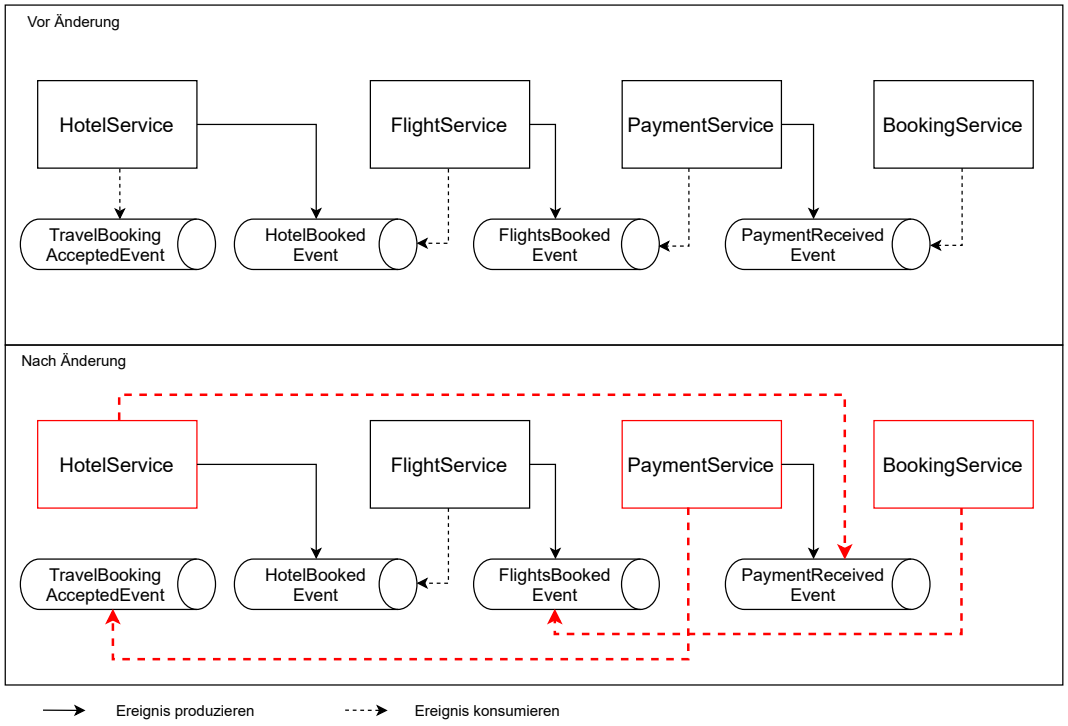

Abbildung 5.1.: Auswirkung einer Prozessänderung innerhalb der Buchungsabwicklung im Rahmen der ereignisgesteuerten Architektur nach dem Choreography Pattern [Eigene Darstellung]

Markierung zu erkennen ist, müssen für diese Änderung insgesamt drei Microservices angepasst werden. Der „HotelService“ konsumiert nun anstatt dem `TravelBookingAcceptedEvent` das `PaymentReceivedEvent`, der „PaymentService“ konsumiert das `TravelBookingAcceptedEvent` und der „BookingService“ erkennt nun den Buchungsabschluss, indem er das `FlightsBookedEvent` abonniert. Zusätzlich kommt weiterer Aufwand hinzu, um in einem Fehlerfalle die Kompensation weiterhin korrekt abzuwickeln. An dieser Stelle ist die bereits erwähnte „Event Chain“ klassischer EDAC festzustellen. Die Services sind auf fachlicher Ebene miteinander gekoppelt.

Im Fall der EDAO und PE Implementierung beschränkt sich der Änderungsbereich der Buchungsabwicklung auf den Orchestrierungsservice („BookingOrchestratorService“). Innerhalb der EDAO setzt die Änderung dabei eine programmatische Anpassung des Orchestrierungsservices voraus. Bei komplexeren Prozessabläufen besteht hier die Gefahr, dass Codeänderungen mit wechselseitigen Wirkungen einhergehen, die auf den ersten Blick im Programmcode nicht sichtbar sind. Beispielsweise wenn mehrere Dienste bestimmte Ereignisse konsumieren. Gleiches gilt für die PE Implementierung. Diese ist der EDAO Implementierung jedoch in dem Aspekt überlegen, dass Prozessanpassungen einfacher und transparenter vorzunehmen sind, da hierfür nur das zugehörige BPMN-Prozessmodell angepasst werden muss. Dies wird deutlich anhand des eben genannten Beispiels der Buchungsabwicklung. Anstatt mehrere Services abändern zu müssen, ist im Rahmen der PE und PDA Implementierung nur eine Verschiebung der Aktivitäten „Zahlung abwickeln“ und „Bestätigung erhalten“ im Prozessmodell notwendig (vgl. Abbildung 5.2).

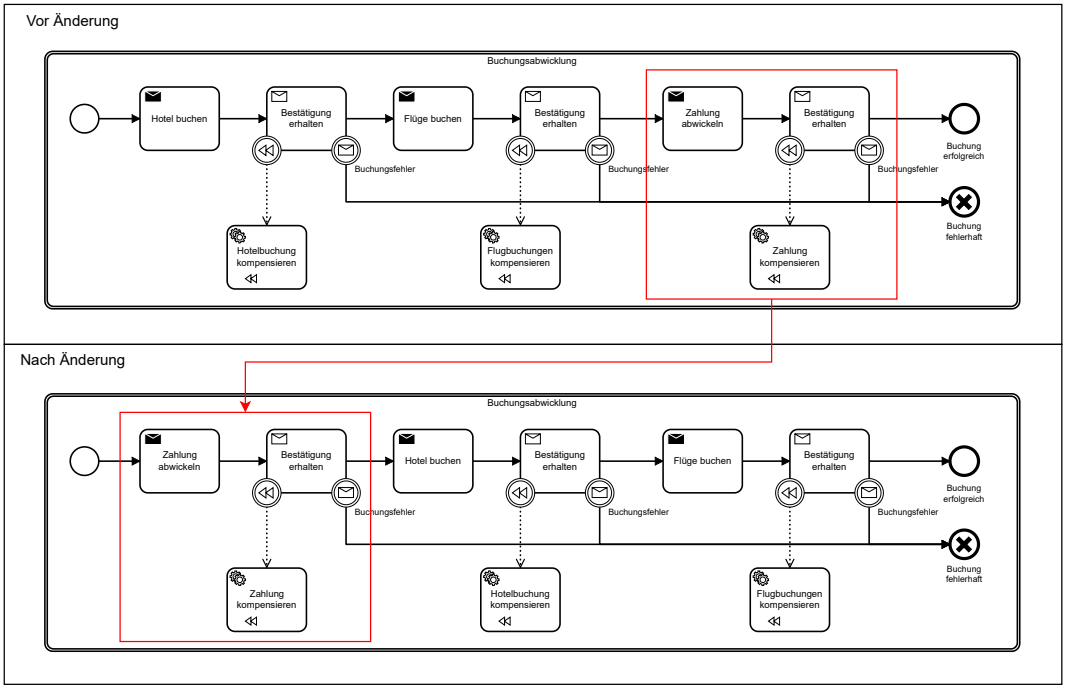

Abbildung 5.2.: Auswirkung einer Prozessänderung innerhalb der Buchungsabwicklung im Rahmen der Process Engine basierten und prozessgesteuerten Architektur [Eigene Darstellung]

Da in der PE Implementierung fachliche und technische Aspekte innerhalb eines Prozessmodells vermischt sind, besteht hier die Gefahr, dass kleine technische Anpassungen immer Änderungen des Gesamtprozesses voraussetzten. Neben dem Fakt, dass durch eine solche Vermischung die Fachlichkeit verloren geht, werden damit auch technische Schulden aufgebaut. Als Beispiel innerhalb der Implementierung kann hier die Speicherung der Kundendaten herangezogen werden. Ist es nötig die Daten zusätzlich aufzubereiten oder in weiteren Systeme abzuspeichern, so muss der Geschäftsprozess angepasst und neu bereitgestellt werden. Die PDA profitiert durch die Verwendung einer Process Engine von denselben Vorteilen wie auch die PE Implementierung. Aufgrund der monolitischen Implementierung des Fachprozesses und der strikten Trennung zwischen Fach- und Integrationsprozessen geht die prozessgesteuerte Architektur jedoch weniger technischen Schulden im Vergleich zu den anderen Architekturstilen ein. Die Architektur eignet sich deshalb vor allem für langlebige IT-Systeme, die einen hohen Wert auf

Flexibilität und Wartbarkeit legen.

Stabilität

Ein stabiles System zeichnet sich dadurch aus, dass bei Änderungen keine Seiteneffekte im System entstehen und nur minimale technische Schulden vorhanden sind. Im Unterpunkt zur Modifizierbarkeit bzw. Änderbarkeit ist bereits gezeigt worden, dass Änderungen des Geschäftsprozesses im Rahmen der EDAC Implementierung dazu führen, dass mehrere Services angepasst werden müssen. Da der Geschäftsprozess hier durch Ereignisketten abgebildet und damit nicht explizit definiert ist, besteht ein höherer Abstimmungsaufwand und die Gefahr, dass Änderungen am Geschäftsprozess unerwartete Auswirkungen haben. Die Fähigkeit, Prozesse in unterschiedlichen Versionen gleichzeitig ausführen zu können, ist auch ein Aspekt, der die Stabilität des Systems beeinflusst. So kann es sinnvoll erscheinen, dass gestartete Prozessflüsse von jeglichen Prozessänderungen unberührt bleiben und nur neue Anfragen den aktuellsten Prozessfluss durchlaufen. Eine solche Versionierung würde innerhalb der EDAC und EDAO Implementierung einen zusätzlichen Programmier- und Wartungsaufwand erzeugen. Die PE und PDA Umsetzung auf der anderen Seite erhält durch die Verwendung der Camunda Engine bereits implizit Funktionalitäten zur Versionierung der Geschäftsprozesse. Dadurch ist eine höhere Stabilität und bessere Aktualisierbarkeit der Geschäftsprozesse gegeben.

Prüfbarkeit

Die Prüfbarkeit betrachtet den Aufwand der zur Sicherstellung der Korrektheit von Funktionen und Änderungen im System notwendig ist. Um Funktionen auf ihre korrekte Ausführung zu validieren und Fehler zu vermeiden, werden üblicherweise Test-Szenarien in Form von beispielsweise Unit- oder Integrationtests definiert. Im Rahmen der exemplarischen Implementierung sind keine Tests implementiert, dennoch können die einzelnen Implementierungen anhand des Testaufwands untersucht werden. Generell ist zu erkennen, dass verteilte Systeme mit einem höheren Testaufwand als monolitische Systeme einhergehen. Neben dem Testen der internen Funktionalität der einzelnen Services über Unit-Tests, ist dort außerdem eine Prüfung der Gesamtfunktionalität, also dem Zusammenspiel mehrerer Microservices zum Erbringen von Geschäftsprozessen, über Integrationtests relevant. Auch nach Richards und Ford [vgl. Ford und Richards, 2020, S. 213 f.] wird die Gesamt-Testbarkeit eines ereignisgesteuerten Systems als schlecht bewertet. Als Grund hierfür wird genannt, dass die Kommunikation über Ereignisse sehr dynamisch und zu Teilen unvorhersehbar stattfinden kann. Konkrete Pfade oder Szenarien sind oftmals nicht fest definiert, da beliebig viele Services auf Events reagieren können. Die Testbarkeit geht bei der EDAC, EDAO und PE Implementierung also tendenziell mit einer höheren Komplexität und eventuell zusätzlichem Implementierungsaufwand einher.

5.5. Merkmal: Übertragbarkeit

Das Merkmal der Übertragbarkeit betrachtet inwiefern sich Software an neue Umgebungen anpassen und übertragen lässt. Sie untergliedert sich nach DIN/25010 in die Anpassbarkeit und Installierbarkeit eines Systems.

Anpassbarkeit

Bei der Anpassbarkeit wird die Fähigkeit des Systems untersucht, sich an unterschiedliche Umgebungen anpassen zu können. Die einzelnen Services der unterschiedlichen Reisebuchungssys-

teme sind mit dem Spring Framework implementiert. Dieses stellt bereits externe Konfigurationsmechanismen in Form von Konfigurationsdateien (src/main/resources/application.yml) bereit, dessen Attribute zu Ausführungsstart gelesen werden. Vor allem im Rahmen der EDAC und EDAO Implementierung werden so Konfigurationsparameter, wie beispielsweise die URLs zu internen oder externen Systemen oder die Gültigkeit eines Reiseangebots in Minuten, definiert. Alle Implementierungen können damit über externe Konfigurationsdateien an unterschiedliche Umgebungen angepasst werden. Architekturen, die eine Process Engine verwenden, haben zusätzlich die Möglichkeit Fachlogik in DMN-Tabellen auszulagern. Vorteil von DMN-Tabellen ist, dass diese im Vergleich zu technischen Konfigurationsdateien zur Laufzeit der Anwendung von Fachexperten geändert werden können. Die PDA Implementierung gewinnt bei der Übertragbarkeit dadurch, dass die eigentliche prozessgesteuerte Anwendung, die jegliche Fachlogik der Geschäftsprozesse kapselt, über die Servicevertrag-Implementierungsschicht von der umgebenen Systemlandschaft entkoppelt ist. Dadurch ist eine Übertragbarkeit der fachlichen Anwendung in unterschiedliche Organisationen oder Bereiche ohne Änderungen möglich. Lediglich die Servicevertrag-Implementierungen müssen an die neue Umgebung zur Anbindung der Systeme angepasst werden. Die EDAC, EDAO und PE Implementierung vermischen die technischen Details zur Anbindung der Systemlandschaft mit den Fachprozessen, weshalb eine Übertragung der Systeme in andere Umgebungen mit einem zusätzlichen Anpassungsaufwand einhergeht.

Installierbarkeit

Die Installierbarkeit ist für die einzelnen Implementierungen des Reisebuchungssystems in gleichem Maße gegeben. Wie in Anhang A.2 erläutert ist, wird das Reisebuchungssystem in Form von Docker-Containern [Docker, Inc., 2021] ausgeliefert. Vorteil der Containerisierung ist, dass das System unabhängig von dem umgebenen Betriebssystem betrieben werden kann, was zu einer hohen Portabilität der Anwendung führt. Die einzige Voraussetzung zum Starten der Reisebuchungssysteme ist eine installierte Docker Version.

5.6. Ergebnisauswertung

Es fand nun ein Architekturvergleich anhand ausgewählter Architekturqualitätsmerkmale gemäß DIN/25010 statt. Am Beispiel des Reisebuchungssystems ist zu erkennen, dass jede der betrachteten Architekturen unterschiedliche Auswirkungen auf die Qualitätsmerkmale und Eigenschaften der Implementierungen haben. Es ist deutlich geworden, dass ereignisgesteuerte Architekturen (EDAC und EDAO) durch Einsatz von Event-Driven Microservices vor allem dann punkten, wenn es um die Verfügbarkeit und Skalierbarkeit des Systems geht. Die Architektur ist dazu ausgelegt, alle der im Grundlagenkapitel genannten „Zwölf-Faktoren“ (vgl. Kapitel 2.4.2) umzusetzen. Dadurch eignet sie sich zur Entwicklung von performanten Cloud-basierten Systemen. Wenn schnelle Startzeiten und einer hoher Durchsatz gewünscht sind, kann diese Architektur deshalb in Betracht gezogen werden. Kritisch zu sehen ist jedoch die Komplexität, die ereignisgesteuerte Architekturen zur Umsetzung von Informationssystemen mit sich bringen. Vor allem die genannten „Event Chains“ im Kontext der EDAC Implementierung sind nicht zu vernachlässigen. Diese führen zu technischen Schulden und erschweren mit zunehmender Größe des Systems die Wartbarkeit. Das Risiko, dass kleine Änderungen große Auswirkungen im System verursachen, nimmt mit Umfang des Systems zu. Entgegengewirkt werden kann dem zumindest schrittweise durch den Einsatz von zentralen Orchestrierungsservices, wie sie im Rahmen der EDAO oder PE Implementierung eingesetzt wurden. Die

Verwaltung und Steuerung einzelner Prozessstränge kann so lokal an einer zentralen Stelle stattfinden, was einen Vorteil im Hinblick auf die Modifizierbarkeit des Prozesses bietet. Ein Schwachpunkt, den sowohl die EDAC als auch EDAO Implementierung aufweist, stellt der genannte Aufwand dar, der zur Analysierbarkeit des Geschäftsprozesses notwendig ist.

Weiterhin hat die Evaluation der Beispielimplementierungen gezeigt, dass die Verwendung einer Process Engine als Architekturkomponente einen Mehrwert zur Unterstützung unterschiedlicher Qualitätsmerkmale mit sich bringt. Sowohl bei der Wiederherstellbarkeit, Fehlertoleranz als auch Wartbarkeit der Geschäftsprozesse ist am Beispiel des Reisebuchungsprozesses zu erkennen, dass ereignisgesteuerte Architekturen nur über einen deutlichen Mehraufwand in der Implementierung den gleichen Erfüllungsgrad wie Process Engine basierte und prozessgesteuerte Architekturen erreichen können. Dabei kommt hinzu, dass jegliche Zusatzimplementierung weitere technische Schulden im System verursachen kann und somit die Stabilität und Wartbarkeit des Systems beeinflusst. Änderungen der Geschäftsprozesse innerhalb des Systems erweisen sich als einfacher umsetzbar, wenn diese über die BPMN-Notation direkt vom Fachbereich vorgenommen werden. Unternehmen können dadurch flexibler und schneller auf sich ändernde Kundenanforderungen reagieren. In Aspekten der Wartbarkeit und Kompatibilität des Systems ist auf die prozessgesteuerte Architektur zu verweisen. Durch die Entkopplung der Fach- und Integrationsprozesse hat die Beispielimplementierung gezeigt, dass eine höhere Flexibilität und Resilienz gegeben ist. Veränderungen an der bestehenden Systemlandschaft beeinflussen den eigentlichen Fachprozess nicht, während im Rahmen der anderen Architekturen Anpassungen je nach Komplexität und Umfang notwendig sind. Durch die gesamtheitliche Betrachtung der Geschäftsprozesse in der PDA ist außerdem eine bessere Wartbarkeit und Transparenz dieser gegeben. Da die Implementierung nach dem Prozessgesteuerten Ansatz generell mit einem größerem Overhead als beispielsweise die ereignisgesteuerte Umsetzung einhergeht, ist die prozessgesteuerte Architektur vor allem für die Entwicklung langlebiger Informationssysteme innerhalb von Unternehmen zu empfehlen.

Letztendlich bleibt festzuhalten, dass ereignisgesteuerte Architekturen den Architekturen die eine Process Engine zur Implementierung von Geschäftsprozessen verwenden in einigen Qualitätsmerkmalen unterlegen sind. Vor allem für die Entwicklung kritischer Kerngeschäftsprozesse sind die Funktionen die Process Engines standardmäßig bereitstellen essenziell, während ereignisgesteuerte Architekturen diese nur über zusätzlichen Implementierungsaufwand erfüllen können. Process Engine basierte Architekturen können herangezogen werden, wenn eine klare Trennung der Verantwortlichkeiten bzw. Teams durch den Einsatz von Microservices erfolgen soll. Aufgrund der Vermischung zwischen technischen und fachlichen Prozessen geht die Architektur jedoch mit der Gefahr einher, über einen längeren Zeitraum technische Schulden aufzubauen, die die Wartbarkeit des Systems erschweren. Durch die klare Trennung der Zuständigkeiten im Rahmen der Implementierung nach dem Prozessgesteuerten Ansatz liefert dieser Architekturstil eine bessere Wartbarkeit des Systems. Auch hier besteht dabei die Möglichkeit einzelne Aspekte in Microservices auszulagern und als Backend-System innerhalb der PDA anzubinden. So kann auch eine gewissen Skalierung und Flexibilität geschaffen werden, ohne dabei die Kontrolle über den Ende-zu-Ende Prozess zu verlieren. Als Erweiterung der Architekturen erscheint eine API-Management Plattform als sinnvoll. Diese kann zentral die Anbindung an interne und externe Services und Systeme verwalten und im Rahmen der PDA beispielsweise auch die Einhaltung des Servicevertrags sicherstellen.

6. Fazit der Ergebnisse des Architekturvergleichs und Ausblick auf weiterführende Untersuchungsmöglichkeiten

Die vorliegende Arbeit befasst sich mit dem Vergleich von Architekturqualitätsmerkmalen basierend auf Ereignissen, Process Engines und dem Prozessgesteuerten Ansatz. Der Vergleich hat dabei auf Grundlage eines beispielhaften Reisebuchungsprozesses stattgefunden. Hierfür wurden fachliche und technische Anforderungen an den Geschäftsprozess und das System definiert. Die korrekte Erfüllung der Anforderungen galt als Bedingung zur Implementierung des Reisebuchungssystems entsprechend der unterschiedlichen Architekturmuster, sodass eine Vergleichbarkeit im Rahmen der Evaluation gegeben war. Es wurde erläutert, auf welche Art und Weise die Anforderungen innerhalb der Architekturstile umgesetzt werden konnten. Die fertigen Implementierungen des Reisebuchungsprozesses sind daraufhin anhand unterschiedlicher Qualitätsmerkmale gemäß DIN/25010 untersucht und verglichen worden.

Der Vergleich hat gezeigt, dass jede Architektur unterschiedliche Auswirkungen auf die Qualitätsmerkmale und Eigenschaften des Reisebuchungssystems hat. Der Einsatz einer Process Engine im Rahmen der Process Engine basierten und prozessgesteuerten Architektur hat dabei im Vergleich zu den ereignisgesteuerten Architekturen zu einem deutlichen Mehrwert im Hinblick auf die Wartbarkeit und Zuverlässigkeit des Reisebuchungssystems geführt. Die ereignisgesteuerten Architekturen haben durch ihre Leichtgewichtigkeit und guten Skalierbarkeit gepunktet. Es ist jedoch aufgefallen, dass die Choreografie-basierte Implementierung nur wenig Kontrolle über den Ablauffluss bietet und auf Ebene des Kommunikationsaustausch implizit eine hohe Kopplung aufweist, die zu technischen Schulden führt und die Änderbarkeit des Geschäftsprozesses erschwert. Die Implementierung nach dem Prozessgesteuerten Ansatz ist den anderen Architekturen durch ihren monolithischen Ansatz und der strikten Trennung von Fach- und Integrationsprozessen über den Servicevertrag vor allem in den Aspekten der Wartbarkeit und Kompatibilität überlegen. Zwar geht die monolithische Implementierung tendenziell mit einer geringeren Skalierbarkeit und gegebenenfalls einem höherem Abstimmungsaufwand im Team einher, dennoch besteht die Möglichkeit, durch den Einsatz von Cloud-basierten Process Engines und der Auslagerung von Teilfunktionalitäten in Funktionen oder Services die Skalierbarkeit zu erhöhen. Auch ist deutlich geworden, dass API Management Plattformen innerhalb der Implementierung Anwendung finden können, um den Zugang und die Einhaltung der API-Schnittstellen der einzelnen Services zu verwalten und übergreifende Funktionen zum Monitoring oder der Sicherheit zentral abzubilden. Als Fazit aus dem Architekturvergleich kann deshalb der klassische Leitspruch „*it depends*“ herangezogen werden. Jede der verschiedenen Implementierungen unterstützt einzelne Qualitätsmerkmale besser als andere. Die Architektur sollte so gewählt werden, dass sie die gewünschten Eigenschaften, die das System erfüllen soll, bestmöglich unterstützt. Aufgefallen ist, dass die Implementierung nach dem Prozessgesteuerten Ansatz jedoch über die meisten Merkmale hinweg überzeugt hat und die Verwendung dieser Architektur deshalb zur Implementierung von Geschäftsprozessen im Kontext langlebiger Infor-

mationssysteme empfohlen wird. Unternehmen sind damit in der Lage, in Zeiten der digitalen Transformation flexibel auf Änderungen in der Prozesslandschaft reagieren zu können, ohne langfristig hohe technische Schulden einzugehen. An dieser Stelle möchte auch noch darauf hingewiesen werden, dass es zwar verlockend erscheint, Systeme zunächst auf Basis einer ereignisgesteuerten Architektur aufzubauen, da diese aufgrund ihrer Leichtgewichtigkeit schnelle Anfangsresultate in der Entwicklung liefert. Der Aufwand, der jedoch zu Beginn der Entwicklung eingespart wird, muss mit zunehmendem Wachstum des Systems durch den erschwerten Wartungsaufwand zurückbezahlt werden. Vor allem die im Vergleich genannten Event Chains aus Abbildung 5.1 können die Änderbarkeit mit zunehmenden Wachstum stark einschränken. Deshalb empfiehlt es sich bereits anfangs klar zu definieren, welche Rolle das zu entwickelnde Systems in Zukunft einnehmen soll, damit eine optimale Architektur hierfür ausgewählt werden kann.

Der Architekturvergleich dieser Arbeit hat sich nur auf einige ausgewählte qualitative Merkmale gestützt. Eine Untersuchung der Architekturstile auf weitere Merkmale, die auch quantitativer Natur sind, wie z. B. Aspekte der Performance, Testbarkeit oder Sicherheit sind mögliche Bereiche weiterführender Arbeiten. Auch die Untersuchung Cloud-basierter Ansätze zur Implementierung von Geschäftsprozessen wurde im Rahmen dieser Arbeit nur kurz angeschnitten und wäre ein mögliches Themengebiet für weitere Untersuchungen. Abschließend bleibt festzuhalten, dass es in der heutigen Zeit mehrere Möglichkeiten gibt, Geschäftsprozesse digital abzubilden. Unternehmen haben abzuwägen, welche Softwarearchitektur sie bestmöglich dabei unterstützt, um auch in Zukunft flexibel auf sich ändernde Kundenanforderungen und disruptive Innovationen reagieren zu können.

Literaturverzeichnis

[Abts und Mülder 2017] Abts, Dietmar ; Mülder, Wilhelm: *Grundkurs Wirtschaftsinformatik: Eine kompakte und praxisorientierte Einführung*. 9., korr. u. erw. Aufl. 2017. Wiesbaden : Springer Fachmedien Wiesbaden and Imprint: Springer Vieweg, 2017. – ISBN 9783658163792

[Amadeus IT Group 2021] Amadeus IT Group: *Amadeus for Developers: Self-Service APIs*. 2021. – URL `https://developers.amadeus.com/self-service`. – Zugriffsdatum: 14.07.2021

[Amazon Web Services 2021] Amazon Web Services: *AWS Step Functions*. 2021. – URL `https://aws.amazon.com/de/step-functions/`. – Zugriffsdatum: 20.06.2021

[Anandan u. a. 2021] Anandan, Sabby ; Bogoevici, Marius ; Fisher, Mark ; Gopinathan, Ilayaperumal ; Heckler, Mark ; Hillert, Gunnar ; Pollack, Mark ; Peralta, Patrick ; Renfro, Glenn ; Risberg, Thomas ; Syer, Dave ; Turanski, David ; Valkealahti, Janne ; Klein, Benjamin ; Carvalho, Vinicius ; Russell, Gary ; Zhurakousky, Oleg ; Bryant, Jay ; Chacko, Soby ; Sibilio, Domenico: *Spring Cloud Stream Reference Documentation*. 2021. – URL `https://docs.spring.io/spring-cloud-stream/docs/3.1.3/reference/html/spring-cloud-stream.html`. – Zugriffsdatum: 03.09.2021

[Apache Software Foundation 2021a] Apache Software Foundation: *Apache ActiveMQ*. 2021. – URL `https://activemq.apache.org/`. – Zugriffsdatum: 08.07.2021

[Apache Software Foundation 2021b] Apache Software Foundation: *Apache Camel*. 2021. – URL `https://camel.apache.org/`. – Zugriffsdatum: 04.12.2021

[Apache Software Foundation 2021c] Apache Software Foundation: *Apache Kafka*. 2021. – URL `https://kafka.apache.org/`. – Zugriffsdatum: 08.07.2021

[Appelfeller und Feldmann 2018] Appelfeller, Wieland ; Feldmann, Carsten: *Die digitale Transformation des Unternehmens: Systematischer Leitfaden mit zehn Elementen zur Strukturierung und Reifegradmessung*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2018. – ISBN 9783662540619

[Backstage Project Authors 2021] Backstage Project Authors: *Backstage*. 2021. – URL `https://backstage.io/`. – Zugriffsdatum: 30.11.2021

[Balzert 2009] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3. Auflage. Heidelberg : Spektrum Akademischer Verlag, 2009 (Lehrbücher der Informatik). – URL `http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10361884`. – ISBN 9783827422477

[Balzert 2011] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. 3. Auflage. Heidelberg : Spektrum Akademischer Verlag, 2011 (Lehrbücher der Informatik). – URL `http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10494391`. – ISBN 9783827422460

[Bass u. a. 2013] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software architecture in practice*. Third edition. Upper Saddle River, NJ : Addison-Wesley, 2013 (SEI series in software engineering). – URL http://proquest.safaribooksonline.com/0321815734. – ISBN 9780132942799

[Becker u. a. 2012] Becker, Jörg (Hrsg.) ; Kugeler, Martin (Hrsg.) ; Rosemann, Michael (Hrsg.): *Prozessmanagement: Ein Leitfaden zur prozessorientierten Organisationsgestaltung*. 7., korr. und erw. Aufl. Berlin [u.a.] : Springer Gabler, 2012. – ISBN 3642338437

[Bellemare 2020] Bellemare, Adam: *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. First Edition. Sebastopol : O'Reilly Media, Inc., 2020. – ISBN 9781492057895

[Bengel 2014] Bengel, Günther: *Grundkurs Verteilte Systeme: Grundlagen und Praxis des Client-Server und Distributed Computing*. 4. Aufl. 2014. Wiesbaden : Springer Fachmedien Wiesbaden, 2014. – ISBN 9783834821508

[Bonér u. a. 2014] Bonér, Jonas ; Farley, Dave ; Kuhn Roland ; Thompson, Martin: *The Reactive Manifesto*. 2014. – URL https://www.reactivemanifesto.org/. – Zugriffsdatum: 27.06.2021

[Bootstrap Team 2021] Bootstrap Team: *Bootstrap*. 2021. – URL https://getbootstrap.com/. – Zugriffsdatum: 05.12.2021

[Brandolini 2021] Brandolini, Alberto: *Introducing EventStorming: An act of Deliberate Collective Learning*. http://leanpub.com/introducing_eventstorming, 2021. – URL http://leanpub.com/introducing_eventstorming. – Zugriffsdatum: 26.06.2021

[Bruns und Dunkel 2010] Bruns, Ralf ; Dunkel, Jürgen: *Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010 (Xpert.press). – URL http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10391830. – ISBN 9783642024399

[Camunda Services GmbH 2021a] Camunda Services GmbH: *Camunda*. 2021. – URL https://camunda.com/. – Zugriffsdatum: 04.12.2021

[Camunda Services GmbH 2021b] Camunda Services GmbH: *Camunda Best Practices: Avoiding to Model Retry Behavior*. 2021. – URL https://camunda.com/best-practices/operating-camunda/#_avoiding_to_model_retry_behavior. – Zugriffsdatum: 15.09.2021

[Camunda Services GmbH 2021c] Camunda Services GmbH: *Camunda Docs: The Job Executor - Retry Time Cycle Configuration*. 2021. – URL https://docs.camunda.org/manual/7.15/user-guide/process-engine/the-job-executor/#retry-time-cycle-configuration. – Zugriffsdatum: 15.09.2021

[Camunda Services GmbH 2021d] Camunda Services GmbH: *The Job Executor: Concurrent Job Execution*. 2021. – URL https://docs.camunda.org/manual/7.15/user-guide/process-engine/the-job-executor/#concurrent-job-execution. – Zugriffsdatum: 18.09.2021

[Casella u. a. 2017] Casella, Karen ; Avery, Phillipa ; Reta, Robert ; Breuer, Joseph: *Scaling Event Sourcing for Netflix Downloads, Episode 1*. 2017. – URL https://netflixtechblog.com/

scaling-event-sourcing-for-netflix-downloads-episode-1-6bc1595c5595. – Zugriffsdatum: 22.06.2021

[Cunningham 1993] Cunningham, Ward: The WyCash portfolio management system. In: *ACM SIGPLAN OOPS Messenger* 4 (1993), Nr. 2, S. 29–30. – ISSN 1055-6400

[De 2017] De, Brajesh: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. 1st edition. Berkeley, CA : Apress, 2017. – ISBN 9781484213056

[Dijkstra 1982a] Dijkstra, Edsger W.: On the Role of Scientific Thought. In: Dijkstra, Edsger W. (Hrsg.): *Selected writings on computing*. New York : Springer, 1982, S. 60–66. – ISBN 978-1-4612-5697-7

[Dijkstra 1982b] Dijkstra, Edsger W. (Hrsg.): *Selected writings on computing: A personal perspective*. New York : Springer, 1982. – ISBN 978-1-4612-5697-7

[Docker, Inc. 2021] Docker, Inc.: *Docker*. 2021. – URL https://www.docker.com/. – Zugriffsdatum: 11.12.2021

[Dunkel u. a. 2008] Dunkel, Jürgen ; Eberhart, Andreas ; Fischer, Stefan ; Kleiner, Carsten ; Koschel, Arne: *Systemarchitekturen für Verteilte Anwendungen: Client-Server, Multi-Tier, SOA, Event-Driven Architectures, P2P, Grid, Web 2.0*. München : Carl Hanser Verlag GmbH & Co. KG, 2008. – ISBN 9783446417458

[Escoffier und Finnigan 2021] Escoffier, Clement ; Finnigan, Ken: *Reactive Systems in Java*. Sebastopol : O'Reilly Media Incorporated, 2021. – URL https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=6802150. – ISBN 9781492091677

[Evans 2004] Evans, Eric: *Domain-driven design: Tackling complexity in the heart of software*. 1. print. Boston : Addison-Wesley, 2004. – ISBN 9780321125217

[Evans 2015] Evans, Eric: *Domain-driven design reference: Definitions and patterns summaries*. 2015. – ISBN 9781457501197

[Facebook Inc. 2021] Facebook Inc.: *React: A JavaScript library for building user interfaces*. 2021. – URL https://reactjs.org/. – Zugriffsdatum: 04.12.2021

[Ford und Richards 2020] Ford, Neal ; Richards, Mark: *Handbuch moderner Softwarearchitektur: Ein technischer Ansatz. Ein umfassende Handbuch zu Patterns, typischen Architekturmerkmalen und Best Practices*. 1. Auflage. Heidelberg : dpunkt, 2020. – ISBN 9783960091493

[Fowler 2005] Fowler, Martin: *Domain Event*. 2005. – URL https://martinfowler.com/eaaDev/DomainEvent.html. – Zugriffsdatum: 20.06.2021

[Fowler 2014] Fowler, Martin: *TechnicalDebtQuadrant*. 2014. – URL https://martinfowler.com/bliki/TechnicalDebtQuadrant.html. – Zugriffsdatum: 14.06.2021

[Fowler 2015] Fowler, Martin: *Patterns of enterprise application architecture*. 21. print. Boston : Addison-Wesley, 2015 (The Addison-Wesley signature series). – ISBN 0321127420

[Fowler 2017] Fowler, Martin: *What do you mean by "Event-Driven"?* 2017. – URL https://martinfowler.com/articles/201701-event-driven.html. – Zugriffsdatum: 28.06.2021

[Fowler 2019] Fowler, Martin: *TechnicalDebt*. 2019. – URL https://martinfowler.com/bliki/TechnicalDebt.html. – Zugriffsdatum: 13.07.2021

[Fowler und Lewis 2014] Fowler, Martin ; Lewis, James: *Microservices: a definition of this new architectural term*. 2014. – URL https://martinfowler.com/articles/microservices.html. – Zugriffsdatum: 27.06.2021

[Freund und Rücker 2019] Freund, Jakob ; Rücker, Bernd: *Praxishandbuch BPMN: Mit Einführung in DMN*. 6., aktualisierte Auflage. München : Hanser, 2019. – ISBN 9783446461123

[Garcia-Molina und Salem 1987] Garcia-Molina, Hector ; Salem, Kenneth: Sagas. In: *ACM SIGMOD Record* 16 (1987), Nr. 3, S. 249–259. – ISSN 0163-5808

[Gharbi u. a. 2018] Gharbi, Mahbouba ; Koschel, Arne ; Rausch, Andreas ; Starke, Gernot: *Basiswissen für Softwarearchitekten: Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture - Foundation Level*. 3., überarbeitete und aktualisierte Auflage. Heidelberg : dpunkt.verlag, 2018. – ISBN 9783960883326

[Google 2021] Google: *Apigee API Management*. 2021. – URL https://cloud.google.com/apigee. – Zugriffsdatum: 30.06.2021

[gRPC Authors 2021] gRPC Authors: *gRPC*. 2021. – URL https://grpc.io/. – Zugriffsdatum: 26.06.2021

[Hedtstück 2020] Hedtstück, Ulrich: *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*. 2. Aufl. 2020. Berlin, Heidelberg : Springer Berlin Heidelberg, 2020. – URL http://nbn-resolving.org/urn:nbn:de:bsz:31-epflicht-1728569. – ISBN 9783662615768

[Hello2Morrow 2021] Hello2Morrow: *Sotograph Product Family*. 2021. – URL https://www.hello2morrow.com/products/sotograph/sotograph. – Zugriffsdatum: 21.06.2021

[Hofer 2021] Hofer, Stefan: *Domain Storytelling*. 2021. – URL https://domainstorytelling.org/. – Zugriffsdatum: 26.06.2021

[Hohpe und Woolf 2015] Hohpe, Gregor ; Woolf, Bobby: *Enterprise integration patterns: Designing, building and deploying messaging solutions*. 19th print. Boston : Addison-Wesley, 2015 (The Addison-Wesley signature series). – ISBN 9780321200686

[IBM 2021] IBM: *IBM API Connect*. 2021. – URL https://www.ibm.com/de-de/cloud/api-connect. – Zugriffsdatum: 30.06.2021

[Kruchten 1995] Kruchten, P. B.: The 4+1 View Model of architecture. In: *IEEE Software* 12 (1995), Nr. 6, S. 42–50. – ISSN 0740-7459

[Kruchten u. a. 2012] Kruchten, Philippe ; Nord, Robert L. ; Ozkaya, Ipek: Technical Debt: From Metaphor to Theory and Practice. In: *IEEE Software* 29 (2012), Nr. 6, S. 18–21. – ISSN 0740-7459

[Lilienthal 2020] Lilienthal, Carola: *Langlebige Softwarearchitekturen: Technische Schulden analysieren, begrenzen und abbauen*. 3., überarbeitete und erweiterte Auflage. Heidelberg : dpunkt.verlag, 2020

[Lindhauer 2018] Lindhauer, Thorben: *How to Use Business Keys?* 2018. – URL https://camunda.com/blog/2018/10/business-key/. – Zugriffsdatum: 20.09.2021

[Martin 2018] Martin, Robert C.: *Clean architecture: A craftsman's guide to software structure and design*. Boston : Prentice Hall, 2018 (Robert C. Martin series). – URL http://proquest.tech.safaribooksonline.de/9780134494272. – ISBN 013449427X

[Microsoft 2021] Microsoft: *Verteilte Saga-Transaktionen*. 2021. – URL https://docs.microsoft.com/de-de/azure/architecture/reference-architectures/saga/saga. – Zugriffsdatum: 11.09.2021

[Müller 2021] Müller, Thomas: *H2 Database Engine*. 2021. – URL https://www.h2database.com/html/main.html. – Zugriffsdatum: 05.12.2021

[Narkhede u. a. 2017] Narkhede, Neha ; Shapira, Gwen ; Palino, Todd: *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. First Edition. Sebastopol : O'Reilly Media, Inc., 2017. – ISBN 9781491936139

[Netflix Technology Blog 2016] Netflix Technology Blog: *Netflix Conductor: A microservices orchestrator*. 2016. – URL https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40. – Zugriffsdatum: 04.07.2021

[Newman 2015] Newman, Sam: *Microservices (mitp Professional): Konzeption und Design*. 1., 2015. Frechen : mitp, 2015. – URL http://gbv.eblib.com/patron/FullRecord.aspx?p=2089872. – ISBN 9783958450837

[OpenAPI Initiative 2021] OpenAPI Initiative: *OpenAPI Specification*. 2021. – URL https://spec.openapis.org/oas/v3.1.0. – Zugriffsdatum: 30.07.2021

[OpenZipkin 2021] OpenZipkin: *Zipkin*. 2021. – URL https://zipkin.io/. – Zugriffsdatum: 05.12.2021

[Parnas 1972] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In: *Communications of the ACM* 15 (1972), Nr. 12, S. 1053–1058. – ISSN 0001-0782

[Patton 2015] Patton, Jeff: *User Story Mapping: Die Technik für besseres Nutzerverständnis in der agilen Produktentwicklung*. 1. Aufl. CA 95472 : O'Reilly Media, 2015. – ISBN 9783958750685

[Qusay H. 2004] Qusay H., Mahmoud: *Getting Started with Java Message Service (JMS)*. 2004. – URL https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html. – Zugriffsdatum: 08.07.2021

[Ruecker 2021] Ruecker, Bernd: *Practical process automation: Orchestration and integration in microservices and cloud native architectures*. First edition. Beijing and Boston : O'Reilly, 2021. – ISBN 149206145X

[Scheer 2002a] Scheer, August-Wilhelm: *ARIS - vom Geschäftsprozeß zum Anwendungssystem*. 4., durchges. Aufl. Berlin and Heidelberg and New York and Barcelona and Hongkong and London and Mailand and Paris and Tokio : Springer, 2002. – ISBN 3540658238

[Scheer 2002b] Scheer, August-Wilhelm (Hrsg.): *ARIS in der Praxis: Gestaltung, Implementierung und Optimierung von Geschäftsprozessen ; mit 2 Tabellen*. Softcover reprint of the hardcover 1st ed. 2002. Berlin and Heidelberg and New York and Barcelona and Hongkong and London and Mailand and Paris and Tokio : Springer, 2002. – ISBN 9783642627590

[Starke 2020] Starke, Gernot: *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. 9., überarbeitete Auflage. München : Hanser, Carl, 2020. – ISBN 9783446463769

[Stiehl 2013] Stiehl, Volker: *Prozessgesteuerte Anwendungen entwickeln und ausführen mit BPMN: Wie flexible Anwendungsarchitekturen wirklich erreicht werden können*. 1. Auflage. Heidelberg : dpunkt.verlag, 2013. – ISBN 3864900077

[Stiehl 2021] Stiehl, Volker: *Der Prozessgesteuerte Ansatz: Was ist der „Prozessgesteuerte Ansatz“?* 2021. – URL `https://volkerstiehl.de/was-ist-pda/`. – Zugriffsdatum: 05.07.2021

[Stopford 2018] Stopford, Ben: *Designing Event-Driven Systems*. First Edition. Sebastopol : O'Reilly Media, Inc., 2018. – ISBN 9781492038245

[Stripe 2021] Stripe: *Stripe: API Reference*. 2021. – URL `https://stripe.com/docs/api`. – Zugriffsdatum: 14.07.2021

[The GraphQL Foundation 2021] The GraphQL Foundation: *GraphQL*. 2021. – URL `https://graphql.org/`. – Zugriffsdatum: 12.07.2021

[The Object Management Group 2021a] The Object Management Group: *Business Process Model And Notation*. 2021. – URL `https://www.omg.org/spec/BPMN/`. – Zugriffsdatum: 07.11.2021

[The Object Management Group 2021b] The Object Management Group: *Decision Model and Notation*. 2021. – URL `https://www.omg.org/dmn/`. – Zugriffsdatum: 06.12.2021

[The PostgreSQL Global Development Group 2021] The PostgreSQL Global Development Group: *PostgreSQL*. 2021. – URL `https://www.postgresql.org/`. – Zugriffsdatum: 05.12.2021

[Tilkov u. a. 2015] Tilkov, Stefan ; Eigenbrodt, Martin ; Schreier, Silvia ; Wolf, Oliver: *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. 3., aktualisierte und erw. Aufl. Heidelberg : dpunkt-Verl., 2015. – URL `http://gbv.eblib.com/patron/FullRecord.aspx?p=2046366`. – ISBN 3864901200

[Vernon 2013] Vernon, Vaughn: *Implementing domain-driven design*. 1st ed. Upper Saddle River, NJ and Mexico City : Addison-Wesley, 2013 (DDD Series). – ISBN 0321834577

[Vernon 2017] Vernon, Vaughn: *Domain-Driven Design kompakt*. 1. Auflage. Heidelberg : dpunkt.verlag, 2017. – URL `https://ebookcentral.proquest.com/lib/gbv/detail.action?docID=5102179`. – ISBN 9783960881780

[VMware 2021a] VMware, Inc: *Spring*. 2021. – URL `https://spring.io/`. – Zugriffsdatum: 04.12.2021

[VMware 2021b] VMware, Inc: *Spring Guides: Creating Asynchronous Methods*. 2021. – URL `https://spring.io/guides/gs/async-method/`. – Zugriffsdatum: 16.09.2021

[Wiggins 2017] Wiggins, Adam: *The Twelve-Factor App*. 2017. – URL `https://12factor.net`. – Zugriffsdatum: 27.11.2021

[Wolff 2018] Wolff, Eberhard: *Microservices: Grundlagen flexibler Softwarearchitekturen*. 2., aktualisierte Auflage. Heidelberg : dpunkt.verlag, 2018. – URL `https://ebookcentral.proquest.com/lib/subhh/detail.action?docID=5476756`. – ISBN 9783960884132

[Wolff 2019] Wolff, Eberhard: Warum Domain-driven Design? Fachlich sinnvoll schneiden. In: *Javamagazin* (2019), Nr. 9.2019. – URL `https://kiosk.entwickler.de/java-magazin/java-magazin-9-2019/fachlich-sinnvoll-schneiden/`

A. Anhang

A.1. Qualitätsmerkmale von Software gemäß DIN/ISO 25010 nach Gernot Starke

Merkmal	Beschreibung
Funktionale Eignung	**Vorhandensein von Funktionen, die explizite oder implizite Anforderungen erfüllen**
- Angemessenheit	Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen, z. B. die benötigte Genauigkeit von berechneten Werten
- Genauigkeit	Liefert das System Ergebnisse mit dem erforderlichen Grad an Präzision?
Zuverlässigkeit	**Fähigkeit, das Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum aufrechtzuerhalten**
- Verfügbarkeit	Grad, in dem eine Softwarekomponente einsatzbereit und bei Bedarf verfügbar ist
- Fehlertoleranz	Fähigkeit, ein spezifiziertes Leistungsniveau bei Softwarefehlern oder Nichteinhaltung einer spezifizierten Schnittstelle zu bewahren
- Wiederherstellbarkeit	Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und direkt betroffene Daten wiederzugewinnen
Effizienz (engl.: performance efficiency)	**Verhältnis zwischen dem Leistungsniveau und dem Umfang der eingesetzten Ressourcen (etwa: Speicher, CPU)**
- Zeitverhalten	Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung
- Verbrauchsverhalten	Anzahl und Dauer der benötigten Betriebsmittel für die Erfüllung der Funktionen
Betreibbarkeit	**Grad, in dem das System verstanden, gelernt, benutzt und für Benutzer attraktiv ist**
- Erkennbarkeit der Angemessenheit	Grad, in dem das System es Benutzern ermöglicht zu erkennen, ob es für ihre Bedürfnisse geeignet ist
- Erlernbarkeit	Aufwand für Benutzer, die Anwendung des Systems zu erlernen
- Einfachheit der Benutzung	Grad, in dem das Softwareprodukt es den Benutzern leicht macht, es zu bedienen und zu kontrollieren
- Hilfsbereitschaft	Grad, in dem das System Hilfe bereitstellt, wenn nötig
- Attraktivität	Grad, in dem das System attraktiv für Benutzer ist
- Zugänglichkeit	Grad der Bedienbarkeit für Benutzer mit bestimmten Einschränkungen

Merkmal	Beschreibung
Sicherheit	**Schutz vor versehentlichem oder böswilligem Zugriff, Nutzung, Änderung, Zerstörung oder Offenlegung**
- Vertraulichkeit	Grad, in dem das System Schutz vor unbefugter Offenlegung von Daten oder Informationen bietet
- Integrität	Grad, in dem Genauigkeit und Vollständigkeit von Daten gewährleistet ist
- Nicht-Abstreitbarkeit	Grad, in dem Handlungen nachweislich stattgefunden haben, so dass sie später nicht zurückgewiesen werden können
- Verantwortlichkeit	Grad, in dem die Handlungen einer Entität (Person oder System) eindeutig auf diese Entität zurückgeführt werden können
- Authentizität (Glaubwürdigkeit)	Grad, in dem die Identität eines Subjekts oder einer Ressource als die beanspruchte nachgewiesen werden kann
Kompatibilität	**Grad, zu dem ein System Informationen mit anderen Systemen austauschen kann und/oder seine erforderlichen Funktionen ausführen kann, während es dieselbe Hardware- oder Softwareumgebung teilt**
- Ersetzbarkeit	Fähigkeit, das System anstelle eines anderen für den gleichen Zweck in der gleichen Umgebung zu verwenden
- Koexistenz	Fähigkeit, mit anderer Software in einer gemeinsamen Umgebung ohne nachteilige Auswirkungen zu koexistieren
- Interoperabilität	Fähigkeit, mit vorgegebenen Systemen zusammenzuwirken. Hierunter fällt auch die Einbettung in die Betriebsinfrastruktur
Wartbarkeit	**Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist. Änderungen: Korrekturen, Verbesserungen oder Anpassungen an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen**
- Modularität	Grad, zu dem ein System oder Computerprogramm aus einzelnen Komponenten besteht, so dass eine Änderung einer Komponente nur minimalen Einfluss auf andere Komponenten hat
- Wiederverwendbarkeit	Grad, zu dem Systemteile für mehr als ein System oder zur Entwicklung weiterer Systeme genutzt werden können
- Analysierbarkeit	Aufwand, um Mängel oder Ursachen von Versagen zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen
- Modifizierbarkeit (Änderbarkeit)	Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassung an Umgebungsänderungen
- Stabilität	Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen
- Prüfbarkeit	Aufwand, der zur Prüfung der geänderten Software notwendig ist
Übertragbarkeit	**Eignung der Software, von einer Umgebung in eine andere übertragen zu werden. Umgebung kann organisatorische Umgebung, Hardware oder Softwareumgebung einschließen**
- Anpassbarkeit	Software an verschiedene festgelegte Umgebungen anpassen
- Installierbarkeit	Grad, in dem das System in einer bestimmten Umgebung erfolgreich installiert und deinstalliert werden kann

Tabelle A.1.: Qualitätsmerkmale von Software gemäß DIN/ISO 25010 nach Gernot Starke [Starke, 2020, S. 41 ff.]

A.2. Anleitung zum Starten des Reisebuchungssystems

Der Programmcode des praktischen Anteils dieser Arbeit ist beiliegend unter dem Ordner `implementation` auffindbar. Die Datei `implementation/README.md` liefert eine kurze **Startanleitung**. Die implementierten Dienste und Services werden über Docker Container bereitgestellt. Im Verzeichnis `implementation/docker` sind alle für den Start notwendigen Docker Compose Dateien abgelegt. Zum Starten des Reisebuchungssystems sollte zunächst sichergestellt sein, dass alle notwendigen Tools in einer kompatiblen Version auf dem Rechner installiert sind. Tabelle A.2 zeigt eine Übersicht der benötigten Software und der im Rahmen der Implementierung getesteten Versionen. Außerdem sollte zum Start des Systems sichergestellt werden, dass Docker auf ausreichend Ressourcen des Rechners zugreifen kann.

Software	Version
Apache Maven	3.6.3
Docker	20.10.7

Tabelle A.2.: Auflistung der verwendeten Softwareversionen für die Implementierung [Eigene Darstellung]

Zum Erstellen der Docker Container ist es notwendig den Java Code zu compilieren und zu bauen. Dazu wird der Befehl `mvn clean install` im Wurzelverzeichnis ausgeführt. Außerdem muss zur internen Kommunikation der Container ein Docker Netzwerk mit dem Namen „travel-booking-system“ über `docker network create travel-booking-system` erstellt werden. Das System kann über einzelne `.env` Dateien innerhalb des Docker-Verzeichnisses konfiguriert werden. Ein wichtiger Konfigurationsparameter, der vor Ausführung gesetzt werden sollte, ist die Variable `CRM_FILE_MOUNT_DIRECTORY` in der Datei `docker/.env`. Dieser Parameter definiert den Pfad zum Verzeichnis für den Export und Import der CSV-Dateien im Rahmen der Kundendatenspeicherung des CRM-Systems.

A.2.1. Starten der Travel Booking UI

Zum Starten des Travel Booking UI wird der Befehl `docker compose up` in dem Verzeichnis `implementation/docker/travel-booking-ui` ausgeführt. Nach erfolgreicher Ausführung sollte die Webapplikation im Browser unter `localhost:8080` erreichbar sein.

A.2.2. Starten der Infrastruktur und bestehenden Systemlandschaft

Bevor das jeweilige Travel Booking Backend gestartet werden kann, sollten zunächst alle Infrastruktur Systeme sowie die internen und externen Applikationen der hypothetischen bestehenden Systemlandschaft (siehe Übersicht in Abbildung 4.1) aktiv sein. Zunächst wird hierzu in das Verzeichnis `implementation/docker` navigiert, worin anschließend zum Starten der Systemlandschaft folgende Anweisung ausgeführt wird:

```
1 docker compose -f docker-compose-infrastructure.yml -f docker-compose-internal-systems.yml
      -f docker-compose-external-systems.yml up
```

Die Infrastruktur Dienste bestehen aus: Kafka als Message Broker, Kafdrop als UI für Kafka und Zipkin als Distributed-Tracing-Tool.

A.2.3. Starten des Travel Booking Backend

Die Implementierungen des Travel Booking Backend sind entsprechend der Architekturstile als Unterordner im Verzeichnis `implementation/docker` aufgeführt (vgl. Abbildung A.1). Jeder Ordner umfasst eine `docker-compose.yml` und `.env` Konfigurationsdatei. Zum Starten eines gewünschten Travel Booking Backend wird in das Wunschverzeichnis navigiert und der Befehl `docker compose up` ausgeführt.

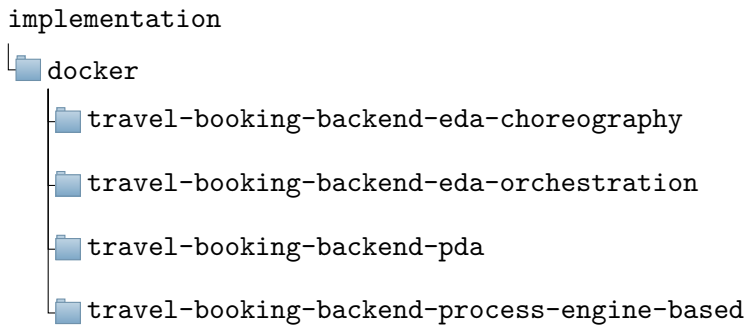

Abbildung A.1.: Docker Container Ordnerstruktur des Travel Booking Backend [Eigene Darstellung]

A.3. OpenAPI Spezifikationen

A.3.1. Spezifikation der Travel API

```yaml
openapi: 3.0.1
info:
  title: Travel API
  description: Mock API service to provide an travel api
  version: v1
paths:
  /offers/flights:
    post:
      tags:
      - flight-api-controller
      summary: Get a single flight offering for a specific date from departure to
        destination. Also provides option to return only premium flights
      operationId: findFlight
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/FlightOfferingRequest'
        required: true
      responses:
        "500":
          description: Processing error
          content:
            application/json:
              schema:
                type: object
        "200":
          description: Found flight offering
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/FlightOfferingDTO'
  /cancellation/hotels:
    post:
      tags:
      - hotel-api-controller
      summary: Cancel a hotel booking
      operationId: cancelHotelBooking
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/HotelOfferingDTO'
        required: true

```

```
    responses:
      "200":
        description: Successfully canceled hotel booking
        content:
          application/json: {}
/cancellation/flights:
  post:
    tags:
    - flight-api-controller
    summary: Cancel a flight booking
    operationId: cancelFlightBooking
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/FlightOfferingDTO'
      required: true
    responses:
      "200":
        description: Successfully canceled flight booking
        content:
          application/json: {}
/booking/hotels:
  post:
    tags:
    - hotel-api-controller
    summary: Book a hotel offering
    operationId: bookHotel
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/HotelOfferingDTO'
      required: true
    responses:
      "500":
        description: Hotel offering expired
        content:
          '*/*':
            schema:
              type: object
      "200":
        description: Successfully booked hotel offering
        content:
          application/json: {}
/booking/flights:
  post:
    tags:
    - flight-api-controller
    summary: Book a flight offering
    operationId: bookFlight
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/FlightOfferingDTO'
      required: true
```

```
    responses:
      "200":
        description: Successfully booked flight offering
        content:
          application/json: {}
      "500":
        description: Flight offering expired
        content:
          '*/*':
            schema:
              type: object
  /offers/hotels/{city}:
    get:
      tags:
      - hotel-api-controller
      summary: Get a list of hotel offerings for a specific city
      operationId: findHotels
      parameters:
      - name: city
        in: path
        required: true
        schema:
          type: string
          enum:
          - Barcelona
          - Muenchen
          - London
          - Rom
          - Sydney
      responses:
        "200":
          description: Found hotel offerings
          content:
            application/json:
              schema:
                type: string
        "500":
          description: Processing error
          content:
            application/json:
              schema:
                type: object
components:
  schemas:
    FlightOfferingRequest:
      type: object
      properties:
        departure:
          type: string
          enum:
          - Barcelona
          - Muenchen
          - London
          - Rom
          - Sydney

```

```
      destination:
        type: string
        enum:
        - Barcelona
        - Muenchen
        - London
        - Rom
        - Sydney
      premium:
        type: boolean
      date:
        type: string
        format: date-time
  FlightInformationDTO:
    type: object
    properties:
      iataCode:
        type: string
        enum:
        - MUC
        - BCN
        - FCO
        - LHR
        - SYD
        - ABC
      terminal:
        type: string
      at:
        type: string
        format: date-time
  FlightOfferingDTO:
    type: object
    properties:
      id:
        type: string
        format: uuid
      airline:
        type: string
        enum:
        - LUFTHANSA
        - RYANAIR
        - EASYJET
      flightClass:
        type: string
        enum:
        - FIRST_CLASS
        - BUSINESS_CLASS
        - ECONOMY_CLASS
      durationInMinutes:
        type: integer
        format: int32
      numberOfStops:
        type: integer
        format: int32

```

```
      price:
        type: number
        format: double
      arrival:
        $ref: '#/components/schemas/FlightInformationDTO'
      departure:
        $ref: '#/components/schemas/FlightInformationDTO'
  HotelOfferingDTO:
    type: object
    properties:
      id:
        type: string
        format: uuid
      name:
        type: string
      imageUrl:
        type: string
      rating:
        type: integer
        format: int32
      basePrice:
        type: integer
        format: int32
      city:
        type: string
        enum:
        - Barcelona
        - Muenchen
        - London
        - Rom
        - Sydney
      address:
        type: string
```

Abbildung A.2.: Spezifikation der Travel API [Eigene Darstellung]

A.3.2. Spezifikation der Payment API

```yaml
openapi: 3.0.1
info:
  title: Payment API
  description: Mock API service to provide an payment api
  version: v1
servers:
- url: http://localhost:8096
  description: Generated server url
paths:
  /payments/refund:
    post:
      tags:
      - payment-api-controller
      summary: Refunds the given amount on the credit card
      operationId: refundToCreditCard
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/RefundToCreditCardRequest'
        required: true
      responses:
        "200":
          description: Successfully refunded amount
          content:
            application/json: {}
  /payments/charge:
    post:
      tags:
      - payment-api-controller
      summary: Charges the credit card with the given amount
      operationId: chargeCreditCard
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ChargeCreditCardRequest'
        required: true
      responses:
        "500":
          description: Credit card expired
          content:
            '*/*':
              schema:
                type: object
        "200":
          description: Successfully charged credit card
          content:
            application/json: {}

```

```yaml
components:
  schemas:
    RefundToCreditCardRequest:
      type: object
      properties:
        creditCardNumber:
          type: integer
          format: int64
        amount:
          type: number
          format: double
    ChargeCreditCardRequest:
      type: object
      properties:
        creditCardNumber:
          type: integer
          format: int64
        amount:
          type: number
          format: double
```

Abbildung A.3.: Spezifikation der Payment API [Eigene Darstellung]

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: 16.12.2021 Unterschrift: