

**Aufbau einer Microservice-basierten Architektur
zur Speicherung und Ausführung von subjekt-orientierten Prozessen**

**Masterarbeit
zur Erlangung des akademischen Grades eines
„Diplomingenieurs für technisch-wissenschaftliche Berufe“
eingereicht am Master-Studiengang Informationsmanagement**

Verfasser:

Stefan Stani, BSc

Betreuer:

FH-Prof. Mag. Dr. Robert Singer

Graz, 2017

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Masterarbeit selbstständig angefertigt und die mit ihr verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für gutes wissenschaftliches Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die vorliegende Originalarbeit ist in dieser Form zur Erreichung eines akademischen Grades noch keiner anderen Hochschule vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Graz, 6. September 2017

Stefan Stani

Inhaltsverzeichnis

Abbildungsverzeichnis	iv
Tabellenverzeichnis	vi
Abkürzungsverzeichnis	vii
Listings	viii
1 Einleitung	1
1.1 Problemstellung und Motivation	1
1.2 Wissenschaftliche Fragestellung	2
1.3 Zielsetzung	2
1.4 Aufbau der Arbeit	2
2 Definitionen	4
2.1 Subjekt-orientiertes Business Process Management (S-BPM)	4
2.1.1 Verwendung der natürlichen Sprache in S-BPM	5
2.1.2 Modellierung von S-BPM Prozessen	6
2.1.3 Weitere Komponenten von S-BPM	8
2.1.4 Anforderungen an ein Workflow-Management-System	10
2.2 Microservices	11
2.2.1 Definition	11
2.2.2 Grundsätze von Microservices	13
2.2.3 Merkmale von Microservices	15
2.2.4 Application Programming Interface (API) Design	18
2.2.5 Vorteile der Nutzung von Microservices	20
3 Datenbankmodell	22
3.1 Prozessmodell	24
3.2 Prozessinstanz	29
3.3 Integration des Datenbankmodells in den Prototypen	32
3.3.1 Verwendung von Interfaces	33
3.3.2 Verwendung des Builder-Pattern	34
4 Prototyp	35
4.1 Basistechnologien	36
4.1.1 Spring Boot	36
4.1.2 Spring Cloud	39
4.2 Komponenten des Prototypen	42
4.2.1 ConfigurationService	42
4.2.2 ServiceDiscovery	43
4.2.3 Gateway	45
4.2.4 ProcessModelStorage	49
4.2.5 ProcessEngine	52
4.2.6 ExternalCommunicator	62

Inhaltsverzeichnis

4.2.7	Graphical User Interface (GUI)	71
5	Beispielprozesse	72
5.1	Prozessbeschreibung	72
5.2	Beispielprozess 1: Reiseantrag mit Prozesskommunikation	73
5.3	Beispielprozess 2: Reiseantrag mit externem IT-System	83
6	Erweiterungen und Verbesserungen für den Prototypen	87
6.1	Neue Microservices	89
6.1.1	AuthenticationService	89
6.1.2	LogServer	89
6.1.3	LifeCycleService	90
6.2	Vorhandene Microservices	91
6.2.1	ProcessEngine	91
6.2.2	ProcessModelStorage	93
6.2.3	Gateway	94
7	Vergleich mit einem klassischen Workflow-Management-System	95
7.1	Klassisches Workflow-Management-System	95
7.2	Szenarien	97
7.2.1	Skalierbarkeit	97
7.2.2	Erweiterbarkeit	98
7.2.3	Verfügbarkeit	98
7.2.4	Antwortzeiten	99
7.2.5	Deployment	99
7.2.6	Verantwortlichkeiten	100
7.2.7	Flexibilität bei neuen Anforderungen	100
7.2.8	Datenbanktransaktionsmanagement	101
7.2.9	Monitoring	101
7.2.10	Import und Ausführung bestehender Prozessmodelle	102
7.2.11	Standortübergreifende Prozessausführung	103
7.2.12	Unternehmensübergreifende Prozesse	104
7.3	Gesamtbewertung	104
8	Fazit	107
	Literaturverzeichnis	110

Abbildungsverzeichnis

2.1	Geschäftsprozessbeschreibung in natürlicher Sprache	5
2.2	Subject Interaction Diagram (SID) Beispiel	6
2.3	Subject Behavior Diagram (SBD) Beispiel	8
2.4	Subject Interaction Diagram (SID) Beispiel mit externem Subjekt	9
2.5	Vergleich monolithischer Ansatz mit dem Microservice-basierten Ansatz	13
2.6	Responsibility-Vergleich von monolithischer Architektur mit Microservices	14
2.7	Aufbau von monolithischen Anwendungen und Microservices	15
2.8	Verwendung von verschiedenen Technologien in Microservices	17
2.9	Automatisiertes Deployment	17
2.10	Ökosystem von Microservices	18
2.11	Synchrone Kommunikation	19
2.12	Asynchrone Kommunikation	19
3.1	Datenbankmodell	23
3.2	Objekthierarchie des Persistence-Packages	33
3.3	Instanziierung von Objekten des Persistence-Packages mittels Builder-Pattern	34
4.1	Microservice-basierter Prototyp	35
4.2	Struktur eines Spring Boot-Projekts	38
4.3	Architektur einer Spring Boot-Anwendung	39
4.4	Komponenten von Spring Cloud	40
4.5	Architektur des Microservices ConfigurationService	42
4.6	Architektur des Microservices ServiceDiscovery	44
4.7	Anzeige der verfügbaren Services von Eureka	45
4.8	Architektur des Microservices Gateway	45
4.9	Klassendiagramm des Interfaces AuthenticationProvider	47
4.10	Aufbau des Role Based Access Control (RBAC)	48
4.11	Klassendiagramm des Interfaces RBACRetrievalService	49
4.12	Architektur des Microservices ProcessModelStorage	50
4.13	Klassendiagramme der Parser für Prozessmodelle	51
4.14	Sequenzdiagramm zur Veranschaulichung des Imports von Prozessen	52
4.15	Architektur des Microservices ProcessEngine	53
4.16	Aufbau des Aktorenmodells	54
4.17	Verwendung der Mailbox in Akka	55
4.18	Funktionsweise der Komponente Message Dispatcher in Akka	56
4.19	Verwendete Akteure in der ProcessEngine	58
4.20	Klassendiagramm der Klasse TaskManager	58
4.21	Objekthierarchie eines Tasks	59
4.22	Composer der ProcessEngine	60
4.23	Parser der ProcessEngine	61
4.24	Interface zur Implementierung eines Refinements	62
4.25	Architektur des Microservices ExternalCommunicator	63
4.26	Klassendiagramm der Interfaces Composer und DataTypeComposer im ExternalCommunicator	66

Abbildungsverzeichnis

4.27 Klassendiagramm der Interfaces Parser und DataTypeParser im ExternalCommunicator	67
4.28 Klassendiagramm des Interfaces SendPlugin im ExternalCommunicator	67
4.29 Verwendete Akteure im ExternalCommunicator	68
4.30 Sequenzdiagramm zur Veranschaulichung des Sendens von Nachrichten im ExternalCommunicator	69
4.31 Sequenzdiagramm zur Veranschaulichung des Empfangs von Nachrichten im ExternalCommunicator	70
5.1 Subject Interaction Diagram (SID) der Beispielprozesse	73
5.2 Subject Behavior Diagram (SBD) des Subjekts TeamleiterIn in den Beispielprozessen	73
5.3 Subject Behavior Diagram (SBD) der Subjekte MitarbeiterIn und Travel Management im Beispielprozess 1	74
5.4 Login-Bildschirm	75
5.5 Dashboard des Subjekts MitarbeiterIn	75
5.6 Prozess starten	75
5.7 State: Reiseantrag anlegen	76
5.8 State: Reiseantrag senden	76
5.9 Dashboard des Subjekts TeamleiterIn	77
5.10 State: Reiseantrag empfangen	77
5.11 State: Reiseantrag prüfen	78
5.12 State: Reiseantrag annehmen	78
5.13 Dashboard des Subjekts MitarbeiterIn nach dem Empfang der Annahme des Reiseantrags	79
5.14 State: Empfange Reiseantrag Antwort	80
5.15 State: Sende an Travel Management	80
5.16 Dashboard des Subjekts Travel Management	81
5.17 State: Empfange Reisedaten	81
5.18 State: Hotelbuchung durchführen	82
5.19 State: Empfange Hotelbuchung im Beispielprozess 1	82
5.20 Subject Behavior Diagram (SBD) des Subjekts MitarbeiterIn im Beispielprozess 2	83
5.21 Log-Eintrag im ExternalCommunicator bei erfolgreichem Senden	84
5.22 Senden von HTTP-Anfragen mittels Restlet Client	85
5.23 State: Empfange Hotelbuchung im Beispielprozess 2	85
6.1 Verbesserungen für den Microservice-basierten Prototyp	88
6.2 Architektur des Microservices LogServer	90
6.3 Architektur des Microservices LifeCycleService	91
6.4 Balancing Dispatcher für die ProcessEngine	92
6.5 Klassendiagramm des Interfaces ReceiveStrategy	93
7.1 Architektur des klassischen Workflow-Management-Systems	96

Tabellenverzeichnis

3.1	Beschreibung der Attribute der Tabelle ProcessModel	24
3.2	Beschreibung der Attribute der Tabelle SubjectModel	24
3.3	Beschreibung der Attribute der Tabelle ProcessSubjectModelMap	25
3.4	Beschreibung der Attribute der Tabelle SubjectModelRule	25
3.5	Beschreibung der Attribute der Tabelle State	26
3.6	Beschreibung der Attribute der Tabelle Transition	26
3.7	Beschreibung der Attribute der Tabelle BusinessObjectModel	27
3.8	Beschreibung der Attribute der Tabelle BusinessObjectFieldModel	27
3.9	Beschreibung der Attribute der Tabelle StateBusinessObjectModelMap . . .	28
3.10	Beschreibung der Attribute der Tabelle BusinessObjectFieldModelPermission	28
3.11	Beschreibung der Attribute der Tabelle MessageFlow	28
3.12	Beschreibung der Attribute der Tabelle BusinessObjectModelMessageFlowMap	29
3.13	Beschreibung der Attribute der Tabelle ProcessInstance	29
3.14	Beschreibung der Attribute der Tabelle Subject	30
3.15	Beschreibung der Attribute der Tabelle ProcessSubjectInstanceMap	30
3.16	Beschreibung der Attribute der Tabelle SubjectState	31
3.17	Beschreibung der Attribute der Tabelle BusinessObjectInstance	31
3.18	Beschreibung der Attribute der Tabelle BusinessObjectFieldInstance	31
4.1	Beschreibung der Attribute der Tabelle OutboundConfiguration	64
4.2	Beschreibung der Attribute der Tabelle InboundConfiguration	64
4.3	Beschreibung der Attribute der Tabelle MessageProtocol	65
4.4	Beschreibung der Attribute der Tabelle MessageProtocolField	65
7.1	Zuordnung der Module des monolithischen Ansatzes zu Microservices des Prototypen	96
7.2	Gesamtbewertung des Softwarearchitekturvergleichs	106

Abkürzungsverzeichnis

API	Application Programming Interface
BPMN	Business Process Model and Notation
CRUD	Create, Read, Update and Delete
CSV	Comma-separated Values
DDD	Domain-driven Design
EAR	Enterprise Application Archive
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IT	Informationstechnik
JAR	Java Archive
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
JWT	JSON Web Token
LoC	Lines of Code
ORM	Object-relational Mapping
OWL	Web Ontology Language
RBAC	Role Based Access Control
REST	Representational State Transfer
SBD	Subject Behavior Diagram
SID	Subject Interaction Diagram
S-BPM	Subjekt-orientiertes Business Process Management
URI	Uniform Resource Identifier
WAR	Web Application Archives
XML	Extensible Markup Language

Listings

3.1	Builder-Pattern Beispielverwendung	34
4.1	Ausschnitt Gradle-Konfiguration für Spring Boot	38
4.2	bootstrap.properties der ProcessEngine	43
4.3	Zuul-Beispielkonfiguration	46
4.4	Ausschnitt von application.properties des ProcessModelStorage	50
5.1	Nachricht an das externe IT-System	84
5.2	Nachricht vom externen IT-System	84

Kurzfassung

Da die meisten Workflow-Management-Systeme auf monolithischen Softwarearchitekturen mit hoher Komplexität basieren, ist es schwierig diese zu skalieren, zu erweitern und zu warten. Ziel dieser Masterarbeit ist die Konzipierung einer Referenzarchitektur für ein Subjekt-orientiertes Business Process Management (S-BPM) Workflow-Management-System, welches auf modernsten Technologien sowie auf autonomen Microservices basiert und so die Komplexität reduziert.

In dieser Masterarbeit wird eine prototypische Implementierung für das Workflow-Management-System in Java vorgestellt. Darüber hinaus werden die Frameworks Spring Boot und Spring Cloud für die Erstellung der Microservice-basierten Architektur genutzt. Zudem werden Hibernate, MySQL und Akka für die Speicherung und Ausführung von subjekt-orientierten Prozessen eingesetzt.

Die Ergebnisse dieser Masterarbeit zeigen, dass Microservices ein effizientes Architekturenpattern für S-BPM-Workflow-Management-Systeme sind. Im Vergleich zu monolithischen Architekturen sind Microservice-basierte Architekturen einfacher wart-, skalier- und erweiterbar. Des Weiteren wird dargestellt, dass die verschiedenen Services des Workflow-Management-Systems unabhängig voneinander entwickelt, getestet und deployt werden können. Dementsprechend sollten Microservice-basierte Architekturen monolithischen Architekturen vorgezogen werden.

Die wichtigste Schlussfolgerung dieser Masterarbeit ist, dass die Nutzung von Microservices die Entwicklung und Wartbarkeit von komplexer Software deutlich vereinfacht. Darüber hinaus können unterschiedliche Programmiersprachen und Technologien für jedes Microservice eingesetzt werden. Somit ermöglichen Microservices im Vergleich zu monolithischen Architekturen ein hohes Maß an Flexibilität.

Abstract

Since the majority of workflow management systems are based on monolithic software architectures with high complexity it is difficult to scale, extend and maintain them. The purpose of this master thesis is to develop a reference architecture for a subject-oriented business process management (S-BPM) workflow management system that is based on state-of-the-art technologies and independent microservices to reduce complexity.

The aim of this master thesis is to provide a prototypical implementation built on Java. Furthermore, the frameworks Spring Boot and Spring Cloud are utilized for the creation of the microservice-based architecture. Moreover, Hibernate, MySQL and Akka are used for the persistence and execution of subject-oriented processes.

The findings from this master thesis illustrate that microservices are an efficient architecture pattern for S-BPM-workflow management systems. Compared to monolithic architectures it is easier to maintain, scale and extend them. Additionally, it is shown that the different services of the workflow management system can be developed, tested and deployed independently. Thus, the master thesis proves that microservice-based architectures should be preferred to monolithic approaches.

The main conclusion drawn from this master thesis is that the utilization of microservices enables developers to develop and maintain complex software far more simply. In addition, microservices allow the usage of different programming languages and technologies for each microservice. Therefore, microservices generate a high degree of flexibility compared to monolithic architectures.

1 Einleitung

1.1 Problemstellung und Motivation

Heutzutage gibt es keine erfolgreichen Unternehmen mehr, die nicht in irgendeiner Art und Weise Geschäftsprozesse einsetzen, um ihre Ziele zu erreichen. Hierbei kann es sich um einfache Geschäftsprozesse handeln, wo es nur wenige Teilnehmerinnen und Teilnehmer gibt, oder um sehr komplexe Geschäftsprozesse, in welchen sogar hunderte Personen beteiligt sind. Geschäftsprozesse nutzen die Ressourcen eines Unternehmens, um einen gewünschten Output (z.B. Produkt oder Dienstleistung) zu erzielen. Folglich ist es für Unternehmen wichtig, dass diese die Geschäftsprozesse möglichst effektiv und effizient ausführen [20].

Somit basiert der Erfolg eines Unternehmens nicht nur mehr auf den angebotenen Produkten und Dienstleistungen, sondern der Erfolg hängt von der Fähigkeit ab, Geschäftsprozesse flexibel und dynamisch zu designen bzw. anzupassen. Zudem werden die meisten Geschäftsprozesse mittels Informationstechnik (IT)-Unterstützung ausgeführt. Hierfür werden Geschäftsprozesse in sogenannte Workflows transformiert. Eine solche Transformation kann zu inkonsistenten Spezifikationen und somit zum Verlust von Informationen führen [12]. Weiters werden bei diesem Ansatz die beteiligten Personen eines Geschäftsprozesses nicht in den Mittelpunkt gestellt.

Subjekt-orientiertes Business Process Management (S-BPM) ist ein Ansatz, der eine kommunikationsorientierte Sicht der Geschäftsprozesse ermöglicht. D.h. der Fokus liegt nicht mehr auf der strikten Ausführung von Aktivitäten, sondern S-BPM rückt die Interaktion der im Geschäftsprozess involvierten Personen – welche als Subjekte bezeichnet werden – in den Mittelpunkt und bezieht diese auch in den Design-Prozess mit ein [20]. S-BPM bietet somit ein kohärentes Framework für das Management von Geschäftsprozessen [12].

Darüber hinaus benötigen auch subjekt-orientierte Prozesse Softwarearchitekturen, die in der Lage sind, diese Prozesse auszuführen und die IT-Unterstützung von Geschäftsprozessen ermöglichen. Es sind auch bereits Lösungen verfügbar, die die Ausführung von subjekt-orientierten Prozessen bereitstellen. Jedoch setzen diese Lösungen häufig auf monolithische Softwarearchitekturen. Insofern sind solche Systeme schwierig zu warten bzw. zu erweitern. Weiters führt dieser Ansatz zur starren Kopplung von Modulen und folglich sind solche Systeme kaum skalierbar [32].

Um die Vermeidung dieser Probleme zu gewährleisten, setzen viele Unternehmen Microservicearchitekturen ein. Microservices sind kleine, autonome Services, die miteinander interagieren. Anstatt eines großen Software-Monolithen werden die einzelnen Funktionalitäten in mehrere unabhängige Services ausgegliedert. Die Servicegrenzen orientieren sich hierbei an den Geschäftsgrenzen eines Unternehmens. Aufgrund der kleinen Größe der Services sind diese leichter wart- und erweiterbar. Des Weiteren ist es einfacher einzelne Services zu skalieren, da nicht alle Komponenten eines Gesamtsystems bei der Skalierung miteinbezogen werden müssen [32].

Dementsprechend würde eine Softwarearchitektur, welche in der Lage ist, subjekt-orientierte Prozesse auszuführen und dabei auf Microservices basiert, erhebliche Vorteile bei der IT-unterstützten Geschäftsprozessausführung für Unternehmen generieren.

1.2 Wissenschaftliche Fragestellung

Die zentrale Fragestellung dieser Arbeit ist:

Wie kann eine Microservice-basierte Architektur für die Speicherung und Ausführung von subjekt-orientierten Prozessen aufgebaut sein und welche Vor- bzw. Nachteile hat die Microservice-basierte Architektur gegenüber von monolithischen Architekturen?

1.3 Zielsetzung

Im Rahmen dieser Masterarbeit wird eine Softwarearchitektur konzipiert und implementiert, welche die Speicherung und Ausführung von subjekt-orientierten Prozessen ermöglicht. Ziel dieser Arbeit ist demnach eine als Prototyp umgesetzte Microservice-basierte Softwarearchitektur zur Speicherung und Ausführung von subjekt-orientierten Prozessen. Als Programmiersprache kommt Java zur Anwendung und verschiedene Open Source-Bibliotheken werden verwendet. Für den Aufbau und die Umsetzung der Microservices wird Spring Boot und darauf aufbauend Spring Cloud eingesetzt, welche auf Java basieren und die einfache Erstellung von Microservice-basierten Architekturen ermöglichen.

Infolgedessen ist es notwendig zu klären, was S-BPM ist und wie die Modellierung auf Basis von S-BPM funktioniert. Insbesondere die Anforderungen an ein Workflow-Management-System¹ werden hierbei herausgearbeitet. Eine weitere Zielsetzung dieser Arbeit ist, einen Überblick über Microservices zu geben. Hier soll geklärt werden, was darunter im Softwarearchitektur-Kontext verstanden wird und welche Vorteile die Verwendung eines solchen Architekturmusters mit sich bringt.

Diese Arbeit beinhaltet unter anderem eine Dokumentation für den entwickelten Prototypen. Dazu ist es von Bedeutung das Datenbankmodell, welches essentiell für die Gesamtarchitektur ist, zu dokumentieren. Zudem werden alle Microservices des vorgestellten Prototypen sowie deren Technologien vorgestellt und erläutert. Ein weiterer Teil dieser Arbeit ist die Einbindung der Unterstützung von externen Subjekten. Dafür wird ein neues Microservice konzipiert. Ebenso werden potentielle Verbesserungsvorschläge für den entwickelten Prototyp in der Arbeit inkludiert. Schließlich wird ein Softwarearchitekturvergleich aufgestellt, um die Unterschiede zwischen einem klassischen, monolithischen Workflow-Management-System und einem Microservice-basierten System aufzuzeigen.

1.4 Aufbau der Arbeit

In Kapitel 2 wird der Begriff S-BPM definiert und vorgestellt. Zudem werden in diesem Kapitel die Anforderungen an ein Workflow-Management-System, basierend auf S-BPM,

¹ IT-System, zur Modellierung und Ausführung von Geschäftsprozessen [24].

1 Einleitung

herausgearbeitet. Microservices, deren Eigenschaften und Merkmale, sind ebenso Bestandteil dieses Kapitels. In Kapitel 3 wird das Datenbankmodell vorgestellt, welches ein essentieller Bestandteil des Prototypen ist. Der Prototyp selbst wird in Kapitel 4 veranschaulicht. Hierbei werden alle entwickelten Microservices und deren Aufgabenbereiche vorgestellt. In Kapitel 5 wird die Verwendung des Prototypen mithilfe von zwei Beispielprozessen erläutert. Mögliche Verbesserungen und Erweiterungen für den Prototypen sind Bestandteil von Kapitel 6. Der Vergleich mit dem klassischen Workflow-Management-System wird in Kapitel 7 durchgeführt. Am Ende der Arbeit – im Kapitel 8 – folgt das Fazit.

2 Definitionen

Im folgenden Kapitel werden Begriffe, welche essentiell für diese Arbeit sind, definiert und erläutert. Zum einen wird im Abschnitt 2.1 S-BPM definiert und insbesondere die Anforderungen an Workflow-Management-Systeme herausgearbeitet. Zum anderen werden im Abschnitt 2.2 Microservices näher beschrieben, da der in dieser Arbeit vorgestellte Architekturansatz auf Microservices basiert.

2.1 Subjekt-orientiertes Business Process Management (S-BPM)

Für Unternehmen ist es wichtig, Geschäftsprozesse möglichst effektiv und effizient auszuführen, um einen gewünschten Output (z.B. Service, Produkt) zu erzielen. Dies wird durch den Einsatz von Geschäftsprozessmanagement sichergestellt. Hierbei werden verschiedene Methoden und Werkzeuge eingesetzt, welche Identifikation, Kontrolle und Verbesserung der Geschäftsprozesse eines Unternehmens ermöglichen [20]. Laut [17] ist ein Geschäftsprozess eine logische Abfolge von Tätigkeiten. Zudem enthält ein Prozess ein Start-Event (Input) und generiert ein Resultat (Output), mit dem Ziel, Wert für die Kundinnen bzw. Kunden zu schaffen.

Geschäftsprozesse definieren, wie Unternehmen auf bestimmte Geschäftsergebnisse (z.B. Bestellung, Beschwerden, etc.) reagieren. Bei vielen Geschäftsprozessen eines Unternehmens handelt es sich hierbei um wissensintensive Serviceprozesse. Serviceprozesse müssen mit einer hohen Flexibilität ausgeführt werden, weshalb die beteiligten Personen eines Geschäftsprozesses sehr häufig miteinander interagieren müssen. Dementsprechend sind die Prozessbeteiligten meist hochqualifizierte Mitarbeiterinnen bzw. Mitarbeiter eines Unternehmens, welche genau wissen wie die Tätigkeiten bestmöglichst auszuführen sind. Folglich ist es für diese nicht motivationsfördernd, wenn eine zentrale Kontrollstelle alle Arbeitsschritte genau vorgibt [20].

Die meisten Methoden des Geschäftsprozessmanagements stellen die strikte Ausführung von Aktivitäten in den Mittelpunkt. Die beteiligten Personen eines Geschäftsprozesses geraten somit in den Hintergrund. Bei S-BPM jedoch wird die Interaktion der Prozessbeteiligten (in S-BPM werden diese als Subjekte bezeichnet) in den Mittelpunkt gerückt [12]. Des Weiteren ermöglicht S-BPM die Verwendung einer einfach zu verstehenden grafischen Notation (siehe Abschnitt 2.1.2) und basiert auf der natürlichen Sprache (siehe Abschnitt 2.1.1), womit Geschäftsprozesse einfach zu modellieren, evaluieren und modifizieren sind. Die Eigenschaften von S-BPM erlauben die dezentralisierte, selbstorganisierte Gestaltung der Arbeitsmuster [20]. S-BPM bietet somit ein kohärentes Framework für das Management von Geschäftsprozessen, mit dem Fokus auf den Wissensaustausch von jenen Personen, welche bei strategischen, taktischen und operativen Themen involviert sind. Demnach handelt es sich bei S-BPM um einen integrierten Ansatz zur organisatorischen Gestaltung und Entwicklung eines Unternehmens [12].

2.1.1 Verwendung der natürlichen Sprache in S-BPM

Subjekte stehen im Vordergrund der S-BPM-Modellierung, weil sie die aktiven Akteure oder Systeme in einem Unternehmensprozess darstellen. Dieser Fokus auf Subjekte ermöglicht die direkte Ableitung subjekt-orientierter Prozessmodelle von natürlichen Sätzen. Ein natürlicher Satz besteht aus Subjekt, Prädikat und Objekt. Der Vorteil der natürlichen Sprachbeschreibungen ist, dass sie von allen Menschen üblicherweise verstanden werden können. Natürliche Sprachen werden zur Kommunikation zwischen Personen verwendet, wohingegen Sprachen im Kontext von Geschäftsprozessen, zur Beschreibung von Prozessmodellen genutzt werden [12].

Prozessmodelle beschreiben die Aktivitäten, die Kommunikation der Prozessbeteiligten sowie verwendete Daten eines Geschäftsprozesses. Demzufolge wird eine Referenz für alle Prozessbeteiligten geschaffen, da die Aktivitäten bzw. Techniken, welche ausgeführt bzw. verwendet werden sollten, definiert werden. Grundsätzlich sind diese Prozessmodelle für die Prozessbeteiligten wichtig. Jedoch muss auch berücksichtigt werden, dass beispielsweise Softwareentwicklerinnen bzw. Softwareentwickler Anwendungen in Geschäftsprozesse integrieren oder diverse Stakeholder diese Geschäftsprozesse evaluieren. Prozessmodelle sollten daher nicht nur für die Expertinnen bzw. Experten (Personen, die Prozessmodelle erstellen), sondern auch für jene Personen, welche ihre Tätigkeiten auf Basis der Prozessmodelle durchführen, verständlich sein. Daher muss ein Prozessmodell ein gemeinsames Verständnis der Geschäftstätigkeit für die Stakeholder schaffen [12].

Da die natürliche Sprache sofort verständlich für alle Stakeholder ist, wird diese grundsätzlich für die erste Beschreibung von Geschäftsprozessen oder Aktivitäten verwendet. Das Resultat dieser Beschreibungen wird dann häufig mit Diagrammen ergänzt. Wie bereits erwähnt bestehen natürliche Sprachen aus drei semantischen Hauptkomponenten [12]:

- Das Subjekt definiert den Startpunkt einer Handlung.
- Das Prädikat bestimmt die Handlung selbst.
- Das Objekt legt das Ziel der Handlung fest.

Dies erleichtert die Beschreibung von Geschäftsprozessen, da bei Geschäftsprozessen Akteure beschrieben werden, die Handlungen auf bestimmte Objekte durchführen. Abbildung 2.1 veranschaulicht einen Dienstreiseantrag, welcher in die Komponenten Subjekt, Prädikat und Objekt heruntergebrochen wurde.

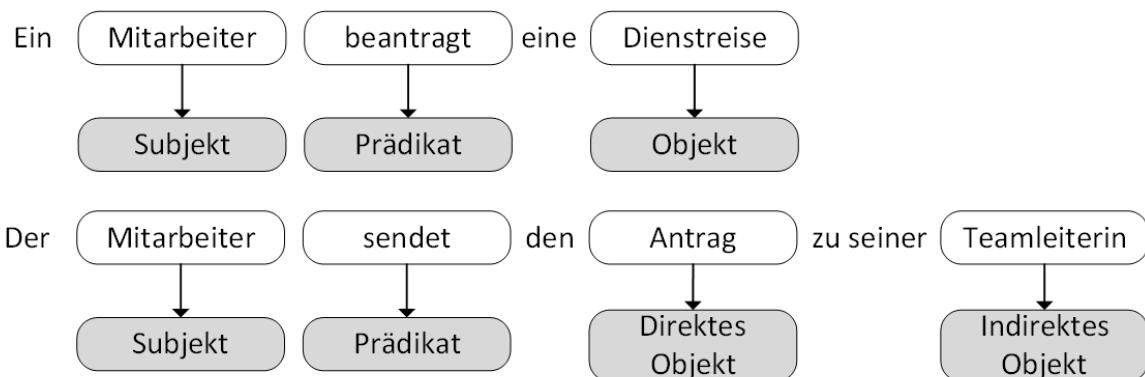


Abbildung 2.1: Geschäftsprozessbeschreibung in natürlicher Sprache (modifiziert übernommen von [12])

2.1.2 Modellierung von S-BPM Prozessen

In diesem Abschnitt werden die Modellierungskomponenten von S-BPM vorgestellt. Für die Modellierung von Geschäftsprozessen in S-BPM werden jedenfalls zwei Diagramme benötigt.

2.1.2.1 Subject Interaction Diagram (SID)

Subjekte sind die zentralen Komponenten von S-BPM. Dementsprechend startet man bei der Modellierung mit der Identifikation der im Geschäftsprozess involvierten Rollen, welche in S-BPM als Subjekte bezeichnet werden. Der Datenaustausch zwischen den Subjekten basiert ausschließlich auf Nachrichten, die ebenso für die Modellierung definiert werden müssen. Beim Senden von Nachrichten werden die benötigten Daten vom Absender an die Empfänger gesendet. Solche Nachrichten können strukturierte Informationen enthalten; so genannte Business Objects, auf welche später noch genauer eingegangen wird. Die eben genannten Komponenten ergeben schlussendlich das SID [12].

Abbildung 2.2 veranschaulicht ein Beispiel für ein SID. Die Subjekte Mitarbeiter und Teamleiterin sind hierbei Bestandteil des Geschäftsprozesses. Der Mitarbeiter hat die Möglichkeit einen Dienstreiseantrag an seine Teamleiterin zu senden. Diese kann wiederum entscheiden, ob der Antrag angenommen bzw. abgelehnt wird. Anschließend wird die Antwort an den Mitarbeiter zurückgesendet [12].

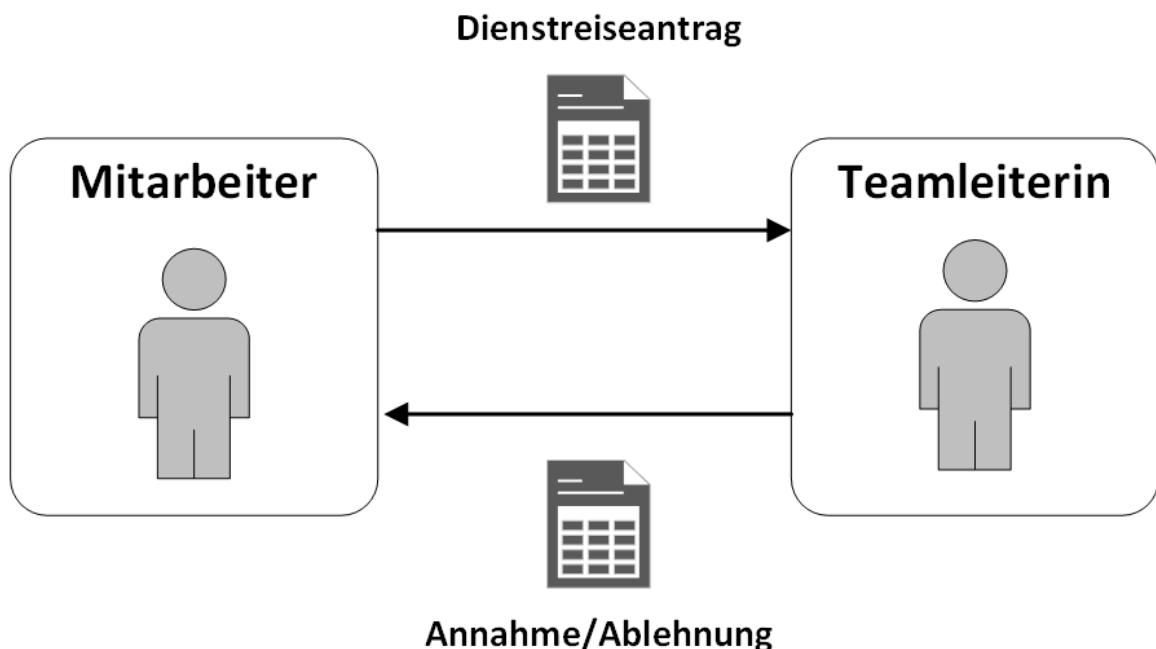


Abbildung 2.2: Subject Interaction Diagram (SID) Beispiel (modifiziert übernommen von [12])

Für den Empfang von Nachrichten wird in S-BPM das Input Pool-Konzept angewandt. Jedem Subjekt ist ein eigener Input Pool zugeordnet. Dieser dient als Nachrichtenpuffer, der Nachrichten vorübergehend speichert. Ein Input Pool kann somit als Posteingang für die jeweiligen Subjekte definiert werden. Für ein Subjekt sind alle Nachrichten im Input Pool sichtbar und das Subjekt kann eine beliebige Nachricht aus diesem Pool auswählen. Des Weiteren wird für einen Input Pool eine Konfiguration (z.B. Input Pool Größe, etc.) festgelegt sowie eine Strategie (z.B. Löschen der ältesten Nachricht im Pool, etc.) definiert [12].

2.1.2.2 Subject Behavior Diagram (SBD)

Das interne Verhalten der Subjekte wird mittels SBD modelliert. Dafür bedarf es nur drei verschiedener Statustypen [12]:

- Funktionen ausführen (Function-State)
- Nachricht senden (Send-State)
- Nachricht empfangen (Receive-State)

Beim Senden wird versucht die Nachricht an ein anderes Subjekt zu senden und diese im Input Pool zu speichern. Wenn der Sendevorgang erfolgreich durchgeführt wurde, wird ein Statusübergang beim Sender ausgeführt. Der Empfangsvorgang wird hingegen in zwei Phasen unterteilt. Im ersten Schritt wird überprüft, ob die erwartete Nachricht bereits empfangen wurde. Wenn dies der Fall ist, dann wird automatisch ein Statusübergang beim Empfänger ausgelöst. Ein Subjekt kann alternativ mehrere unterschiedliche Nachrichten erwarten. In diesem wird geprüft, ob bereits eine dieser Nachrichten verfügbar ist [12].

Des Weiteren müssen Start- bzw. End-States definiert werden, damit der Prozessstart bzw. das Prozessende bekannt sind. Diese müssen für jedes Subjekt festgelegt werden. Erst wenn sich alle Subjekte im End-State befinden, kann ein Geschäftsprozess beendet werden. Geschäftsprozessmanagement unterscheidet zwischen Prozessmodellen und Prozessinstanzen. Subjekt-orientierte Prozessmodelle beschreiben das Verhalten der im Prozess involvierten Personen (Subjekte). Folglich repräsentieren solche Modelle verallgemeinerte Situationen und welche Tasks ausgeführt werden müssen. Im Kontext des Prozessmodells sind Subjekte abstrakte Ressourcen. Eine Prozessinstanz wird erst erstellt, wenn ein Geschäftsprozess – basierend auf einem Modell – gestartet wird. Das Prozessmodell gibt also den Ablauf des auszuführenden Geschäftsprozesses vor. Die Prozessinstanzen enthalten konkrete Daten (z.B. Instanziierung von Business Objects). Zudem werden den Subjekten konkrete Akteure (reale Personen oder Maschinen) zugewiesen [12].

Angelehnt an die bisherigen Beispiele wird in Abbildung 2.3 das interne Verhalten des Subjekts Teamleiterin mittels SBD visualisiert. Nach Empfang des Dienstreiseantrags des Mitarbeiters kann die Teamleiterin diesen annehmen oder ablehnen. Dies wird dann an den Mitarbeiter zurückgesendet. Dieses Beispiel verwendet alle drei verfügbaren Statustypen von S-BPM [12].

2 Definitionen

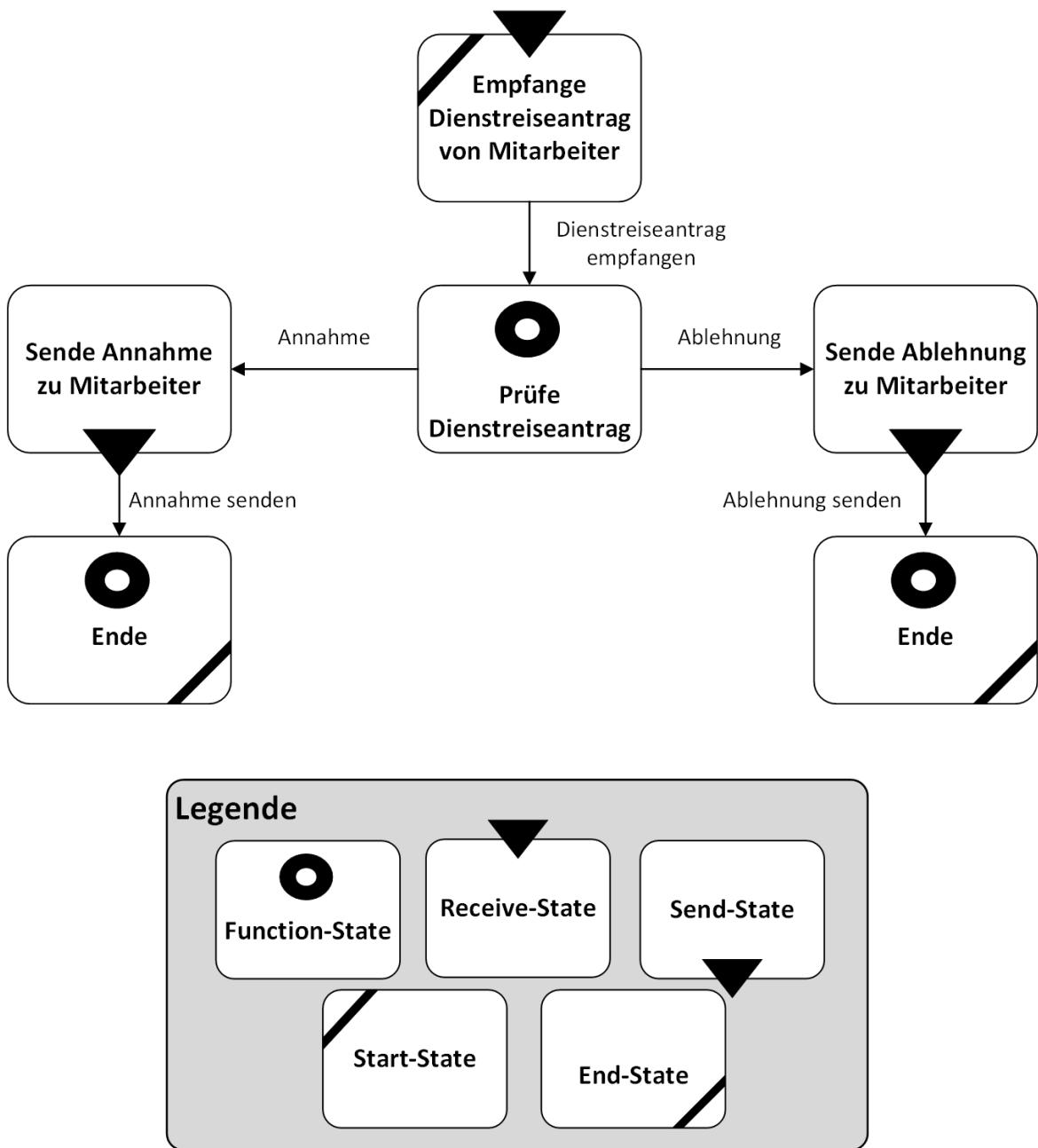


Abbildung 2.3: Subject Behavior Diagram (SBD) Beispiel (modifiziert übernommen von [12])

2.1.3 Weitere Komponenten von S-BPM

Im folgenden Abschnitt werden weitere Komponenten von S-BPM definiert und erklärt. Die Komponenten sind essentiell für das Verständnis der Funktionen in der vorgestellten Softwarearchitektur. Des Weiteren gibt Abschnitt 2.1.3.5 einen kurzen Überblick über jene Komponenten von S-BPM, die nicht Bestandteil der entwickelten Softwarearchitektur sind.

2.1.3.1 Business Objects

Business Objects sind Objekte, die in einem Geschäftsprozess verwendet werden. Weiters sind sie passiv und können daher selbst keine Aktionen oder Handlungen initialisieren. Business Objects werden von den Subjekten im Geschäftsprozess verarbeitet [21]. Die Basisstruktur eines Business Objects enthält: Bezeichnung, Datenstrukturen und Datenelemente. Die

2 Definitionen

Bezeichnung (z.B. Reiseantrag, Urlaubsantrag) wird hierbei vom Geschäftsfall abgeleitet. Business Objects bestehen aus Datenstrukturen. Die Strukturen enthalten einfache Datenelemente (z.B. String, Nummer) oder weitere Datenstrukturen. Die Definition von Zugriffsrechten für Business Objects ermöglicht die Einschränkung von Berechtigungen auf Subjektebene. Hierbei wird zwischen Lesezugriff und Lese/Schreibzugriff unterschieden. D.h. es muss erklärt werden, ob ein Subjekt überhaupt Berechtigungen für ein Business Object benötigt. Ist dies der Fall, dann muss noch determiniert werden, welche Zugriffsrechte das sind [12].

2.1.3.2 Externe Subjekte

In den bisherigen Ausführungen dieser Arbeit war das interne Verhalten der Subjekte stets bekannt. Insofern werden Subjekte, wo das interne Verhalten bekannt ist, als interne Subjekte bezeichnet. Subjekte, deren internes Verhalten nicht bekannt sind, werden als externe Subjekte bezeichnet. Somit können Prozessbestandteile, welche nicht Teil des unternehmensinternen Geschäftsprozesses sind, miteinbezogen werden. Der Austausch zwischen internen und externen Subjekten basiert auf Nachrichten, wobei externen Subjekten kein direkter Zugriff auf Business Objects zur Verfügung steht [12].

Abbildung 2.4 zeigt ein SID, welches ein externes Subjekt inkludiert. In diesem Beispiel sendet der Mitarbeiter die Reiseinformationen an das Travel Management. Es handelt sich hierbei um ein externes Subjekt, da das interne Verhalten nicht bekannt ist. Allerdings muss der Aufbau der Nachrichten sehr wohl bekannt sein, damit ein erfolgreicher Nachrichtenaustausch zwischen internen und externen Subjekten möglich ist [12].

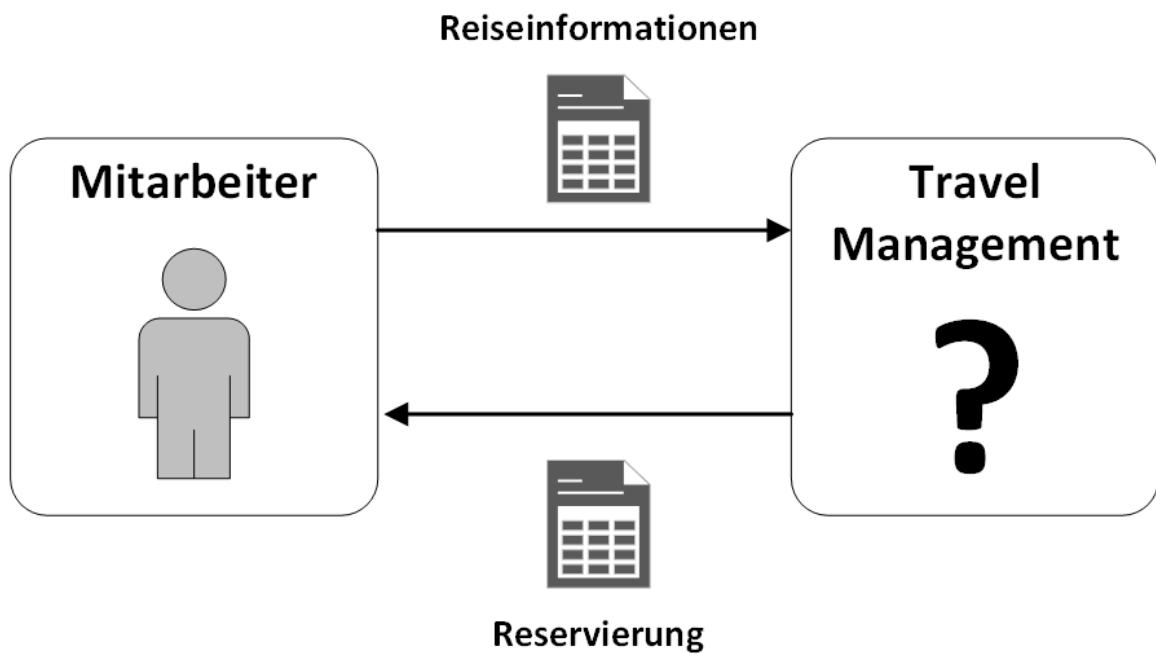


Abbildung 2.4: Subject Interaction Diagram (SID) Beispiel mit externem Subjekt (modifiziert übernommen von [12])

2.1.3.3 Refinements

Refinements sind ein spezieller Function-State, da Benutzerinnen bzw. Benutzer keine direkten Interaktionen (z.B. Hinzufügen eines Wertes für das Business Object) durchführen. Es

handelt sich um hinterlegten Programmcode, der automatisch im Hintergrund ausgeführt wird [20].

2.1.3.4 Timeouts

Grundsätzlich können Timeouts für Send-States oder Receive-States festgelegt werden. Timeouts werden Zeitspannen (z.B. 24 Stunden) zugeordnet. Nach Ablauf der Zeitspanne wird automatisch eine definierte Aktion ausgeführt. Bei einem Send-State wird von einem Subjekt an ein anderes Subjekt eine Nachricht gesendet. Ist ein Timeout definiert worden und innerhalb des Zeitrahmens war kein erfolgreiches Senden möglich, dann wird automatisch ein Statusübergang ausgelöst. Bei einem Receive-State funktionieren Timeouts in ähnlicher Art und Weise. Im Unterschied zum Send-State wird aber gewartet, ob eine bestimmte Nachricht eingegangen ist. Ist dies nicht der Fall, wird ebenso ein Statusübergang durchgeführt [12].

2.1.3.5 Nicht inkludierte Komponenten

In diesem Abschnitt werden weitere Komponenten von S-BPM behandelt, die allerdings nicht im Zuge der Implementierung des Prototypen umgesetzt wurden [12]:

- Multisubjekte: Diese sind ein spezieller Subjekttyp, der ermöglicht, dass mehrere Instanzen eines einzelnen Subjektes in einer Prozessinstanz erstellt werden. Beispielsweise sendet die Teamleiterin eine Nachricht an das Multisubjekt Mitarbeiter. Da es sich um ein Multisubjekt handelt, erhalten alle Mitarbeiter diese Nachricht und für jeden einzelnen Mitarbeiter wird eine eigene Instanz des Multisubjekts erstellt.
- Freedom of Choice: In den bisherigen Ausführungen wurde das interne Verhalten der Subjekte stets als vorgegebene Sequenz von Tätigkeiten modelliert. Bei der Verwendung von Freedom of Choice ist ein Subjekt nicht an die strikte Sequenz der Tätigkeiten gebunden, sondern kann zur Laufzeit selbstständig entscheiden, in welcher Sequenz die Tätigkeiten ausgeführt werden.
- Exception Handling: Ausnahmen und/oder Fehler können ständig in einem Geschäftsprozess auftreten. Mittels Exception Handling wird definiert, wie Subjekte mit solchen Ausnahmen umgehen und darauf reagieren.

2.1.4 Anforderungen an ein Workflow-Management-System

Laut [19] müssen Workflow-Management-Systeme folgende Funktionen zur Verfügung stellen:

- Verschiedene Operationen, um Business Objects zu erstellen/ändern/löschen.
- Es muss festgelegt werden, wie die Business Objects gespeichert werden (z.B. in einer Datenbank).
- Die Unterstützung von unterschiedlichen Datentypen für die Datenelemente der Business Objects.
- Ein Berechtigungskonzept für die Business Objects im Prozess.
- Die Integration von bereits existierenden Anwendungen.
- Verschiedene Schnittstellen, damit Benutzerinnen bzw. Benutzer das Workflow-Management-System benutzen können.
- Funktionen, welche den Austausch von Nachrichten ermöglichen.
- Möglichkeiten, die ein effizientes Monitoring der Prozesse erlauben.

- Ein Zuweisungsmechanismus, damit Subjekten Organisationselemente (z.B. Konto des Active Directory) zugewiesen werden können.
- Parser, die Prozessmodelle in das Workflow-Management-System importieren und ausführbar machen.
- Die Engine muss neustartfähig sein, sodass bereits gestartete Prozessinstanzen fortgesetzt werden können.
- Des Weiteren muss ein Workflow-Management-System skalierbar sein, um eine große Anzahl von Prozessen gleichzeitig zu verarbeiten.
- Auch die Nachvollziehbarkeit von konkreten Prozessinstanzen anhand von Log-Dateien muss gegeben sein.

2.2 Microservices

Grundsätzlich beschreibt eine Softwarearchitektur die Organisation eines Systems. Ein System besteht hierbei aus verschiedenen Strukturen, die wiederum in Komponenten zerlegt werden. Die Schnittstellen dieser Komponenten sowie deren Beziehungen sind ebenso Bestandteil der Softwarearchitektur. Eine effektive Softwarearchitektur hilft dabei Qualitätsanforderungen und -ziele während des gesamten Lebenszyklus der Architektur zu erfüllen. Zudem wird die Verständlichkeit eines Systems erhöht, indem grundlegende Entwurfsentscheidungen und Konzepte nachvollziehbar bleiben. Folglich ist eine effektive, langfristige Softwarearchitektur essentiell für den Erfolg von Softwareprojekten [37].

Die Popularität von Microservice-basierten Softwarearchitekturen ist in letzter Zeit stark gestiegen, da dieser Ansatz die Erreichung von Geschäftsanforderungen erleichtert. Bei Microservices handelt es sich um keine grundlegende Neuerfindung, sondern sie sind das Resultat von vorherigen Softwarearchitekturansätzen. In der Ära der Digitalisierung ist es für Unternehmen von fundamentaler Bedeutung agil auf neue technische Herausforderungen reagieren zu können. Dazu ist das schnelle Adoptieren von Technologien notwendig. Die Zeiten von riesigen Softwaremonolithen sind aufgrund ihrer Inflexibilität vorbei. Microservicearchitekturen ermöglichen hingegen das schnelle, agile Reagieren auf neue Geschäftsanforderungen [36].

2.2.1 Definition

Sam Newman [32] definiert Microservices folgenderweise:

"Microservices are small, autonomous services that work together."

Eine weitere interessante Definition, welche die Servicegrenzen miteinbezieht, wurde von Adrian Cockcroft [7] erstellt:

"Loosely coupled service-oriented architecture with bounded contexts."

Bei Microservices handelt es sich also um mehrere kleine und unabhängige Services, die miteinander interagieren. Die Servicegrenzen orientieren sich hierbei an den Geschäftsgrenzen eines Unternehmens [32]. Laut [41] basiert dieser Ansatz auf der UNIX-Philosophie:

- Ein Service legt den Fokus auf eine Sache.
- Die Services müssen in der Lage sein miteinander zu kommunizieren.
- Die Verwendung einer universellen Schnittstelle.

2 Definitionen

Grundsätzlich sind Microservices ein Modularisierungskonzept. D.h. ein großes Software-System wird in einzelne Module aufgesplittet. Jedes Microservice kann einen eigenen Technologiestack (z.B. Programmiersprache, Basistechnologie) nutzen. Des Weiteren verwendet jedes Microservice einen eigenen Datenbestand (z.B. eigene Datenbank, eigenes Schema in gemeinsamer Datenbank) [41].

Bei Microservices stellt sich natürlich die Frage, wie klein bzw. groß ein einzelnes Service sein darf. Generell gibt es auf diese Frage keine richtige Antwort. [41] schlägt unter anderem vor, dass die Lines of Code (LoC) ein Indikator für die Größe eines Services sein können. Obwohl [41] berücksichtigt hat, dass unterschiedliche Programmiersprachen das Resultat verfälschen, hat er nicht bedacht, dass auch verschiedene Programmierstile einen deutlichen Einfluss auf das Ergebnis haben, wodurch die LoC ungeeignet sind, die Größe eines Microservices zu bestimmen. Weiters empfiehlt [41] die Größe nicht zu klein zu wählen, weil dadurch mehr Infrastruktur benötigt wird. Dieser Punkt erscheint unsinnig, da Infrastrukturkosten in den letzten Jahren extrem gesunken sind. Darüber hinaus schreibt [41], dass die verteilte Kommunikation mit der Anzahl der Microservices zunimmt und somit größere Servicegrößen gewählt werden sollten. Jedoch sollte dies keinen Einfluss haben, weil dadurch Grundprinzipien von Microservices (u.a. Interaktion miteinander) verletzt werden.

[32] empfiehlt die Größe eines Microservices so zu wählen, dass es einem einzelnen Team möglich ist, dieses zu entwickeln und zu warten. Wenn dieses Service zu groß für ein Team wird, dann sollte dieses aufgeteilt werden. Je kleiner ein Service ist, desto unabhängiger ist dieses. Jedoch erhöht dies folglich die Komplexität bei der Interaktion zwischen den verschiedenen Microservices. Dementsprechend sind Unternehmen gefordert ihre Teams so zu wählen, dass diese in der Lage sind, eigenständig die Entwicklung eines Services voranzutreiben. Unternehmen, die nicht dazu im Stande sind, ihre Team so zu strukturieren, werden Microservices auch nie erfolgreich einsetzen können.

In der zweiten Definition wird der Begriff Bounded Context verwendet, weshalb dieser Begriff auch geklärt werden muss. Der Begriff stammt aus dem Domain-driven Design (DDD)-Ansatz und ist somit schon deutlich älter. DDD basiert auf Modelle und besagt, dass jedes Softwaresystem ein Modell der Realität ist. Üblicherweise bilden die meisten großen Softwaresysteme nicht nur ein einziges Modell ab. Mithilfe von Microservices wird versucht, große Komponenten (im DDD-Kontext Modelle) in kleinere aufzutrennen. Dies soll die Komponenten weniger kompliziert machen. Dementsprechend ist der Microserviceansatz dem DDD-Ansatz sehr ähnlich. Das Konzept des Bounded Context wurde eingeführt, um kleine Komponenten zu schaffen. Jede Komponente eines Systems lebt in seinem eigenen Bounded Context. D.h. das Modell einer jeden Komponente wird nur in diesem einen Bounded Context und in keinem anderen verwendet [7]. [32] erweitert diesen Ansatz noch und statuiert, dass ein Bounded Context genau einen autonomen Geschäftsbereich abgrenzt. Folglich ist der Bounded Context-Ansatz ideal für die Erkennung der verschiedenen möglichen unabhängigen Microservices [32].

Abbildung 2.5 macht die Unterschiede der beiden Ansätze noch einmal deutlich. Beim klassischen, monolithischen Ansatz handelt es sich um ein in sich geschlossenes System, das aus verschiedenen voneinander abhängigen Modulen besteht. Beim Microserviceansatz wird dieses große System aufgelöst. Anstatt der Module werden einzelne unabhängige, autonome Services verwendet.

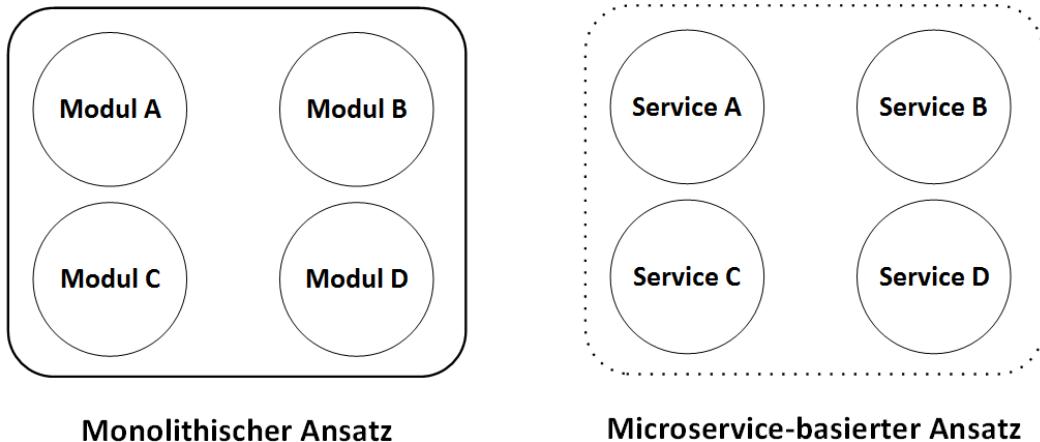


Abbildung 2.5: Vergleich monolithischer Ansatz mit dem Microservice-basierten Ansatz (modifiziert übernommen von [36])

2.2.2 Grundsätze von Microservices

In diesem Abschnitt werden die Grundsätze von Microservices behandelt. Diese Grundsätze sind unbedingt beim Design und bei der Entwicklung von Microservices einzuhalten.

2.2.2.1 Single Responsibility

Dieser Grundsatz legt fest, dass jeder Bestandteil (Klasse, Funktion, Service, etc.) der Software nur für eine Aufgabe verantwortlich ist. Demgemäß sollte kein Bestandteil für zwei Aufgaben verantwortlich sein bzw. keine Bestandteile für ein und dieselbe Aufgabe verantwortlich sein [36].

Abbildung 2.6 zeigt ein E-Commerce-System, das aus den Modulen Bestellung und Produkt besteht. Beim monolithischen Ansatz ist deutlich erkennbar, dass beide Module Bestandteil einer einzigen, großen Anwendung sind. Infolgedessen können Änderungen in einem Modul ungewollte Änderungen im anderen Modul auslösen. Der Microservice-basierte Ansatz sieht zwei autonome Microservices vor. Jedes Service ist hierbei genau einer Aufgabe zugewiesen, wo die Servicegrenzen auf den Geschäftsgrenzen basieren. Somit können Änderungen in einem Service keine ungewollten Änderungen in einem anderen Service herbeiführen [36].

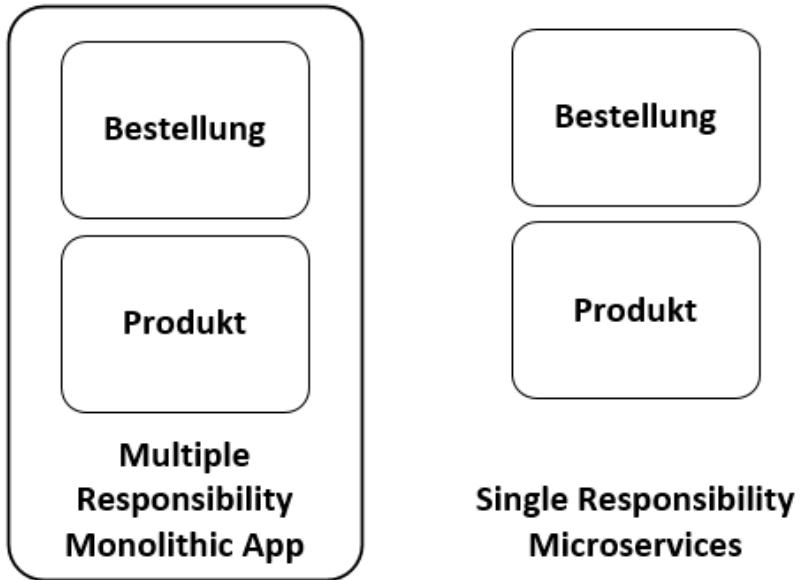


Abbildung 2.6: Responsibility-Vergleich von monolithischer Architektur mit Microservices (modifiziert übernommen von [36])

2.2.2.2 Autonomie

Microservices sind in sich geschlossene, unabhängige, autonome Services, welche für genau eine Aufgabe verantwortlich sind. Ein Microservice beinhaltet alle notwendigen Abhängigkeiten, wie z.B. verwendete Bibliotheken. Darüber hinaus sind auch die Ausführungsumgebungen direkt im Service inkludiert. Dies kann beispielsweise ein Webserver oder sogar ein Container sein. Abbildung 2.7 verdeutlicht die Unterschiede von klassischen, monolithischen Anwendungen und Anwendungen, welche auf Microservices basieren. Bei klassischen Anwendungen wird eine Web Application Archives (WAR)- oder eine Enterprise Application Archive (EAR)-Datei erstellt, die mittels Webserver (z.B. JBoss, WegLogic, etc.) ausgeführt wird. Bei Microservices ist dies nicht der Fall. Hierbei wird nur eine Java Archive (JAR)-Datei – im Microservicekontext auch als Fat-JAR bezeichnet – gebaut, welche alle Komponenten, die zur Ausführung benötigt werden, enthält. Dieses JAR kann als eigenständiger Java-Prozess ausgeführt werden [36].

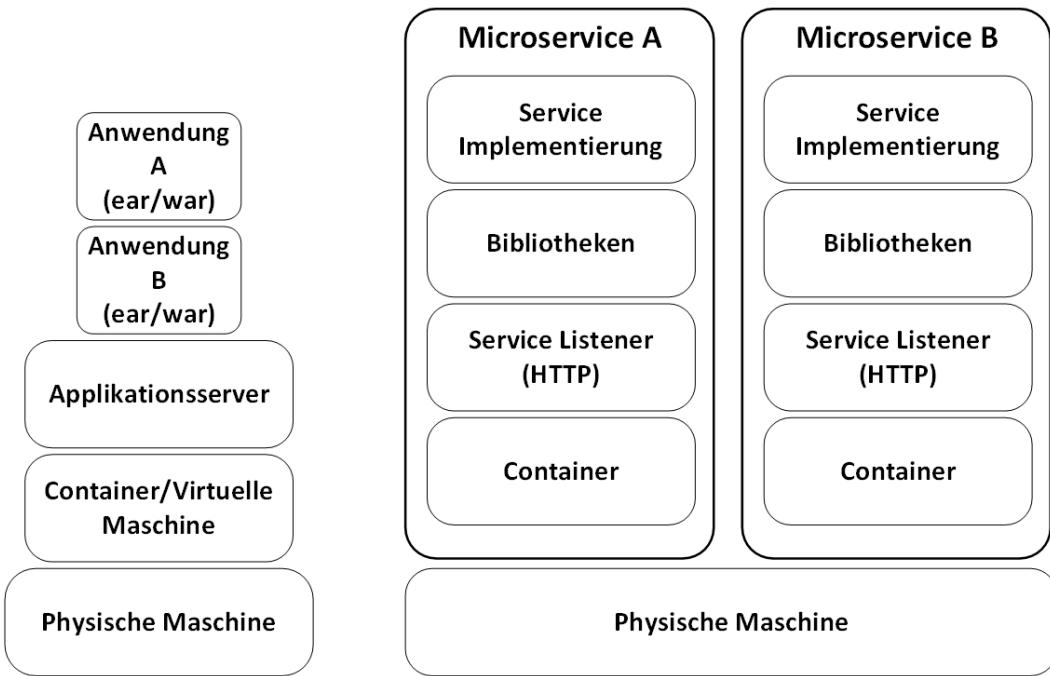


Abbildung 2.7: Aufbau von monolithischen Anwendungen und Microservices (modifiziert übernommen von [36])

2.2.3 Merkmale von Microservices

Bisher gibt es noch keine allgemein akzeptierte Definition für Microservices. Nichtsdestotrotz gibt es einige Merkmale, die bei allen erfolgreichen Microservice-basierten Architekturen zutreffen. Diese Merkmale sind Bestandteil dieses Abschnittes [36].

2.2.3.1 Services sind First-Class-Objekte

In der Welt der Microservices sind Services sogenannte First-Class-Objekte. Microservices stellen Service-Endpunkte als APIs bereit und verbergen folglich sämtliche Realisierungsdetails. Die interne Implementierung, die Architektur und sogar die Technologien (z.B. Programmiersprache, Datenbank, etc.) sind nach außen hin nicht bekannt, da diese hinter der API verborgen sind. Des Weiteren muss auch in den Unternehmen ein Umdenken stattfinden. Bei der Verwendung von Microservices gibt es keine Anwendungsentwicklung mehr, sondern eine Serviceentwicklung. Für viele Unternehmen ist diese kulturelle Änderung häufig schwierig, weil es die bisher bekannten Wege Anwendungen zu bauen, komplett verändert [36].

2.2.3.2 Merkmale von Services in einem Microservice

Die folgenden Merkmale beschreiben die Services [36]:

- **Servicevertrag:** Microservices werden mittels klar definierten Serviceverträgen beschrieben. JavaScript Object Notation (JSON) und Representational State Transfer (REST) sind sehr beliebte Servicekommunikationstechnologien. Es gibt verschiedene Techniken, (z.B. JSON Schema, Swagger, etc.) um Serviceverträge für JSON und REST zu definieren.
- **Lose Kopplung:** Microservices sind unabhängig und infolgedessen lose gekoppelt. In den meisten Fällen erhalten Microservices ein Event als Input und antworten auf diesen

Input mit einem anderen Event. Messaging und Hypertext Transfer Protocol (HTTP) werden häufig zur Interaktion zwischen Microservices verwendet.

- Service-Abstraktion: Beim Microserviceansatz wird nicht nur die Serviceimplementierung vollständig abstrahiert, sondern auch alle verwendeten Bibliotheken und die Ausführungsumgebung (siehe Abbildung 2.7).
- Service-Wiederverwendung: Da Microservices von mobilen Geräten, klassischen Desktopanwendungen, anderen Microservices oder sogar anderen Systemen genutzt werden können, sind sie diese wiederverwendbar.
- Zustandslosigkeit: Microservices sind zustandslos. Wenn es notwendig ist den Status zu speichern, dann sollte dies in einer Datenbank erfolgen.
- Entdeckbarkeit: In einer typischen Microserviceumgebung notifizieren Microservices beispielsweise eine Service Registry, dass das Service nun verfügbar ist. Wird ein Microservice beendet, wird ebenso eine Notifikation ausgesendet.
- Interoperabilität: Microservices nutzen Standards für Protokolle und für den Austausch von Nachrichten. Messaging, HTTP, REST, JSON sind sehr beliebte Technologien hierfür.

2.2.3.3 Microservices sind leichtgewichtig

Wie bereits erwähnt sind Microservices für genau eine Unternehmensaufgabe verantwortlich. Als Resultat davon sind Microservices leichtgewichtiger. Diese Eigenschaft muss auch für alle Technologien, welche in einem Microservice verwendet werden, zutreffen. Insbesondere für Web Container eignen sich z.B. Jetty oder Tomcat besser als WebLogic oder WebSphere, da diese deutlich kleiner und einfacher zu konfigurieren sind [36].

2.2.3.4 Unabhängige interne Technologien

Microservices sind vollständig autonom und verstecken die interne Implementierung hinter definierten APIs. Somit ist es möglich, dass Microservices intern verschiedene Technologien oder Architekturen nutzen. Beispiele dafür sind [36]:

- Microservices benutzen verschiedene Versionen einer Technologie. Ein Microservice verwendet beispielsweise Java 1.7 und ein anderes benutzt bereits die aktuellere Version 1.8.
- Microservices können in verschiedenen Programmiersprachen entwickelt sein.
- Auch für die Speicherung von Daten können unterschiedlichste Technologien (z.B. MySQL oder nur ein interner Cache) eingesetzt werden.

In Abbildung 2.8 werden zwei verschiedene Microservices dargestellt. Für die Suche von Hotels ist die Performance ein wichtiger Faktor, weshalb Erlang und ein interner Cache verwendet werden. Für die Buchung von Hotels wird ein traditionellerer Ansatz mit Java und MySQL gewählt. Die interne Implementierung wird hinter einer API, basierend auf REST und JSON, verborgen [36].

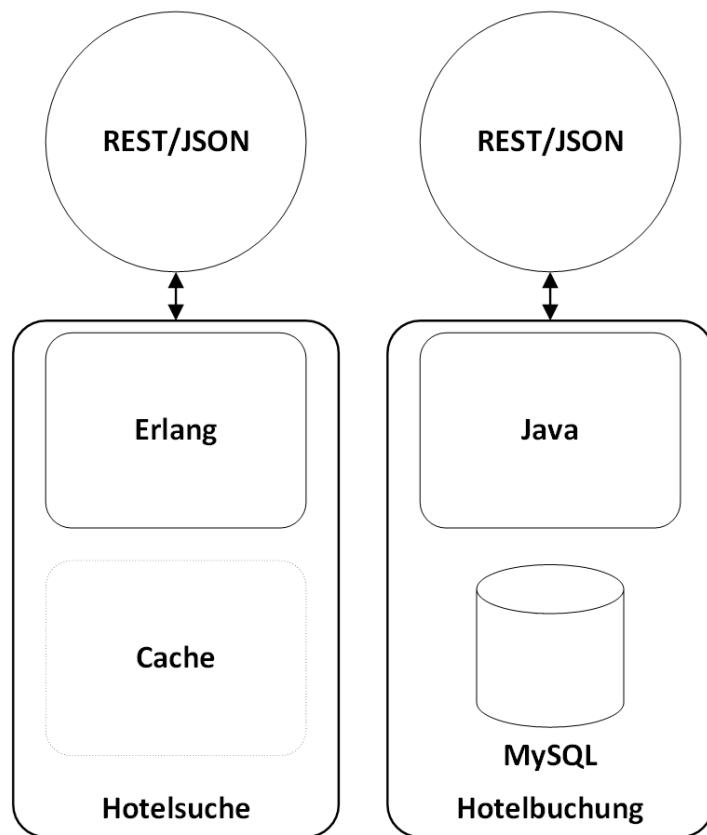


Abbildung 2.8: Verwendung von verschiedenen Technologien in Microservices (modifiziert übernommen von [36])

2.2.3.5 Automatisiertes Deployment

Da der Microserviceansatz monolithische Anwendungen in mehrere kleinere Services splittet, bringt dies neue Herausforderungen beim Deployment mit sich. Dementsprechend ist es schwierig mehrere Microservices zu deployen, wenn das Deployment nicht automatisiert ist. Die Tatsache, dass Ausführungsumgebungen in Microservices inkludiert sind, ist dabei hilfreich. Abbildung 2.9 veranschaulicht einen typischen Deploymentprozess von Microservices. Für die kontinuierliche Integration können beispielsweise Git¹ und Jenkins² eingesetzt werden. Nach diesem Task werden die automatisierten Tests ausgeführt. Wurden die Tests erfolgreich durchgeführt, findet das automatisierte Deployment (z.B. mit Spring Cloud) statt [36].



Abbildung 2.9: Automatisiertes Deployment (modifiziert übernommen von [36])

¹ Flexibles Versionskontrollsystem für Quellcode [28]

² Java-basierte Anwendung für kontinuierliche Integration [10]

2.2.3.6 Ökosystem von Microservices

Die meisten skalierbaren Microservices nutzen ein unterstützendes Ökosystem (siehe Abbildung 2.10). Dieses besteht häufig aus einem Gateway, das die Anfragen an die einzelnen Microservices weiterdelegiert. Zudem registrieren sich die Microservices selbstständig bei einer Service Registry. Weiters ist ein zentrales Logging und Monitoring, u.a. für die Suche von Fehlern, ein essentieller Bestandteil eines typischen Ökosystems [36].

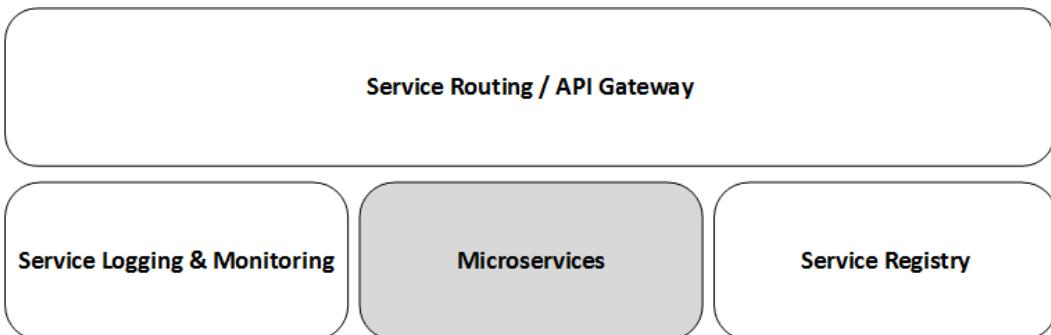


Abbildung 2.10: Ökosystem von Microservices (modifiziert übernommen von [36])

2.2.3.7 Anforderungen an Reactive Microservices

[14] hat noch weitere Anforderungen für Microservices, welche auf dem reaktiven Manifest³ basieren, definiert:

- Antwortbereit: Grundsätzlich gilt es die Antwortzeiten möglichst niedrig zu halten, damit die Erwartungen der Benutzerinnen bzw. Benutzer erfüllt werden. Dies gilt für alle Anfragen zu einem Service sowie auch für die Kommunikation zwischen Services.
- Widerstandsfähig: Gut implementierte Services erfüllen alle Anfragen ohne Ausfälle. Sollte es dennoch zu Ausfällen in Teilkomponenten kommen, dann sollte dies keineswegs zum Ausfall des Gesamtsystems führen. Folglich bleiben die einzelnen Teilkomponenten antwortbereit.
- Elastisch: Services sollten skalierbar und in der Lage sein mit wechselnden Lasten umzugehen. Weiters sollten Services den gleichzeitigen Betrieb auf mehreren unterschiedlichen Maschinen unterstützen.
- Nachrichtenorientiert: Alle bisher genannten Anforderungen können nur erfüllt werden, indem die verschiedenen autonomen Services mittels asynchronen Nachrichten kommunizieren. Um dies zu realisieren ist eine nicht blockierende und asynchrone API notwendig und ermöglicht somit die effiziente Verwendung von Ressourcen.

2.2.4 Application Programming Interface (API) Design

Nicht nur das Design und die Implementierung von Microservices ist entscheidend, sondern auch die Kommunikation. Dies betrifft zum einen die Kommunikation zwischen Microservices und zum anderen die Kommunikation mit anderen Komponenten oder Systemen. Demgemäß sind die Schnittstellen (APIs) der Services von essentieller Bedeutung für die Unabhängigkeit und die lose Kopplung der einzelnen Services [7]. Grundsätzlich wird zwischen synchroner und asynchroner Kommunikation unterschieden [32]. Die Unterschiede werden in den nachfolgenden Abschnitten erläutert.

³ <http://www.reactivemanifesto.org/de>

2.2.4.1 Synchrone Kommunikation

Bei der Verwendung der synchronen Kommunikation wird eine Anfrage zu einem Service gesendet und auf eine Antwort gewartet. Die weitere Ausführung wird somit blockiert. Beispielsweise basiert REST auf diesem Muster [32]. Dieser Ablauf wird in Abbildung 2.11 veranschaulicht.

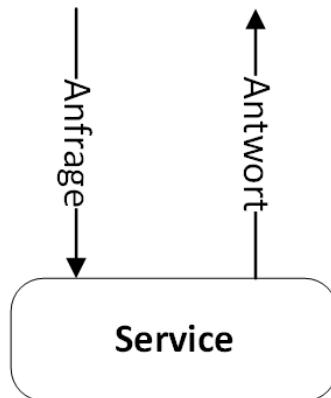


Abbildung 2.11: Synchrone Kommunikation (modifiziert übernommen von [36])

Synchrone Kommunikation ist häufig einfacher zu implementieren, weil nachdem die Antwort empfangen wurde, sofort bekannt ist, ob eine Operation erfolgreich war bzw. Fehler aufgetreten sind [32]. Da die Fehler immer sofort zurückgemeldet werden, befindet sich das System ständig in einem konsistenten Zustand. Darüber hinaus sind keine zusätzlichen Infrastrukturkomponenten (z.B. Message Broker wie Apache ActiveMQ) notwendig. Ein Nachteil ist jedoch, dass der aufrufende Thread blockiert wird, bis die Operation fertiggestellt und eine Antwort erhalten wurde, womit die Skalierbarkeit eines Systems eingeschränkt wird. Synchrone Kommunikation erhöht zudem die Abhängigkeiten von Services, denn das Fehlschlagen eines einzelnen Services führt zum Fehlstart der gesamten Servicekette [36].

2.2.4.2 Asynchrone Kommunikation

Abbildung 2.12 zeigt ein Service, das asynchrone Kommunikation einsetzt. Hierbei werden asynchrone Nachrichten akzeptiert und die Antworten auf diese werden asynchron an andere Komponenten gesendet [36]. Dementsprechend wartet die anfragende Komponente nicht auf die Fertigstellung der Operationen. Asynchrone Kommunikation basiert auf Events. Beim Senden von Events muss nicht bekannt sein, welche Komponenten darauf reagieren bzw. was mit diesem eingefügten Event passiert. Des Weiteren können neue Abonnenten für Events leicht hinzugefügt werden [32].



Abbildung 2.12: Asynchrone Kommunikation (modifiziert übernommen von [36])

Dieser Ansatz ist besonders für langwierige Operationen hervorragend geeignet [32]. Zudem wird die Skalierbarkeit der einzelnen Services erhöht, da Threads Operationen effizienter abarbeiten. Im Gegensatz zum synchronen Ansatz führt ein Ausfall eines einzelnen Services nicht zum Kompletausfall der Servicekette, womit eine lose Kopplung der Services gegeben

ist. Ein Nachteil wiederum ist die Verwendung von zusätzlichen Infrastrukturkomponenten, die zum Austausch von Nachrichten notwendig sind. Dies erhöht die Komplexität und geht mit erweitertem Aufwand für die Konfiguration von diesen Komponenten einher [36].

2.2.5 Vorteile der Nutzung von Microservices

Microservices bringen im Vergleich zu monolithischen Ansätzen einige Vorteile mit sich. Ein großer Vorteil von Microservices ist die Flexibilität, da für jedes Microservice die optimale Technologie und Architektur auf Basis der Anforderungen eingesetzt werden kann. Dies ist nur aufgrund der Unabhängigkeit und Autonomie der Services möglich. Weiters können gleiche Technologien in unterschiedlichen Versionen verwendet werden. Beispielsweise kann ein Service auf Java basieren und ein anderes Service auf Scala. Mit monolithischen Architekturen ist eine solche Flexibilität niemals möglich [36]. Des Weiteren sind keinerlei Implementationsdetails nach außen hin bekannt. Dies betrifft auch den internen Datenspeicher eines Services. Der kontrollierte Zugriff erfolgt somit ausschließlich über APIs. Folglich können interne Änderungen jederzeit vorgenommen oder Services sogar ausgetauscht werden, solange die API unverändert bleibt [32].

Zudem erleichtern Microservices die Entwicklung von innovativen und experimentellen Funktionen. Microservice sind kleiner und einfacher. Somit fällt es Unternehmen leichter mit neuen Prozessen, Algorithmen, etc. zu experimentieren. Mit monolithischen Systemen sind Änderungen oder Neuentwicklungen häufig kompliziert und kostenintensiv. Mit der Verwendung von Microservices können neue Services schnell entwickelt und eingebunden werden. Stellt sich nach ein paar Monaten heraus, dass das neue Service nicht wie erwartet funktioniert, kann es geändert bzw. neu entwickelt werden. Diese Kosten werden deutlich niedriger als beim monolithischen Ansatz sein [36].

Ein weiterer Vorteil der Verwendung von Microservices ist die Skalierbarkeit der einzelnen Services. Es ist davon auszugehen, dass es bei den Services unterschiedliche Anforderungen bezüglich Skalierbarkeit gibt. Bei monolithischen Anwendungen – z.B. als EAR oder WAR bereitgestellt – kann nur die gesamte Anwendung skaliert werden. Microservices können deutlich einfacher und kostengünstiger skaliert werden, da die Services kleiner sind und jedes Service individuell betrachtet werden kann. Microservices sind autonome, unabhängig deploybare Services. Dementsprechend können Services durch andere ähnliche Services ersetzt werden. Monolithische Anwendungen sind von Natur aus hochkohäsiv und infolgedessen schwierig erweiterbar und wartbar. Microservices hingegen können einfach ausgetauscht werden [36].

Üblicherweise wachsen Anwendungen während ihres Einsatzes, indem immer neue Funktionen hinzugefügt werden. Für monolithische Systeme bedeutet dies, dass diese ständig erweitert werden, wobei die Wartbarkeit wegen des Wachstums sinkt. Aufgrund der Unabhängigkeit von Microservices können neue Funktionen in neu hinzugefügte Services bereitgestellt werden. Eine Anpassung der bereits vorhanden Services ist nur geringfügig notwendig. Weiters können veraltete Technologien in Microservices einfacher ausgetauscht werden, da sie im Normalfall lose gekoppelt sind. Technologieänderungen sind sehr schwierige Unterfangen in monolithischen Anwendungen und häufig gleicht dies einer Neuentwicklung. Angesichts der schnellen Änderungen von Technologien können neu entwickelte Monolithen auf veralteten Technologien basieren, bevor diese überhaupt in Betrieb gehen. Um für diese Technologieänderungen gewappnet zu sein, werden häufig

2 Definitionen

Abstraktionsschichten, welche die dahinterliegenden Technologien verbergen, implementiert. Dies führt jedoch zu komplizierteren Architekturen, womit Technologieänderungen schwierig, riskant und teuer sind. Mit Microservices ist das deutlich einfacher umzusetzen. Jedes einzelne Service kann individuell geändert werden oder eine neue Technologie kann für das Service eingeführt werden [36].

Es kann die Notwendigkeit bestehen dasselbe Service in unterschiedlichen Versionen zur gleichen Zeit anzubieten. Beim Wechsel von einer Version zur anderen wird meist eine Ausfallzeit von Kundinnen bzw. Kunden nicht akzeptiert, womit ein Szenario geschaffen wird, wo zumindest ein Service in verschiedenen Versionen, ausgeführt wird. Ein weiteres Szenario ist das Anbieten von Services in einer neuen Version nur für bestimmte Kundinnen bzw. Kunden. Beide Szenarien sind mit monolithischen Ansätzen komplex durchzuführen. Mit Microservices ist dies einfacher, da die notwendigen Ausführungsumgebungen sowie Service Listener im Service inkludiert sind. Demzufolge können Services in verschiedenen Versionen einfach gestartet werden. Einzig das Gateway muss angepasst werden und die Anfragen (z.B. Anfragen einer bestimmten Kundin/eines bestimmten Kunden oder auf Basis der Region) an die richtige Serviceversionen weiterleiten [36].

Idealerweise ist ein Team für ein einzelnes Microservice zuständig. Dies schafft klare Verantwortlichkeiten, weil das Team verantwortlich für die Architektur, die Implementierung, das Testen von Funktionen und das Deployment des Services zuständig ist. Zudem wird verhindert, dass diese genannten Verantwortlichkeiten über viele verschiedene Teams aufgeteilt werden. Der Fokus eines Teams auf ein Microservice, welches genau eine Unternehmensfunktion abdeckt, schafft Expertinnen bzw. Experten für diesen Geschäftsbereich. Besonders wichtige Architekturentscheidungen (z.B. Auswahl eines Message Brokers), die mehrere Teams betreffen, können im sogenannten Architecture Board diskutiert und getroffen werden. Von essentieller Bedeutung ist auch die Schnittstellendefinition der Services, da nur so die korrekte Interaktion der Services sichergestellt werden kann. Hierbei sind ebenso mehrere Teams involviert [32].

3 Datenbankmodell

Abbildung 3.1 veranschaulicht das Datenbankmodell, welches ein essentieller Bestandteil für die in dieser Arbeit vorgestellte prototypische Architektur ist. Grundsätzlich besteht das Datenbankmodell aus zwei wesentlichen Bestandteilen. Zum einen wird das Prozessmodell gespeichert. Im Prozessmodell werden die im Prozess beteiligten Rollen (in S-BPM Subjekte) und deren Verhalten beschrieben. Speziell für S-BPM werden die Interaktionen und der Nachrichtenaustausch der einzelnen Subjekte persistiert. Zudem wird der Aufbau der Business Objects definiert. Zum anderen wird mit diesem Datenbankmodell die Möglichkeit geschaffen, Prozessinstanzen zu speichern. Prozessinstanzen stellen konkrete Ausführungen von Prozessmodellen dar. Dazu wird ein Prozessmodell in eine Prozessinstanz überführt.

In Abschnitt 3.1 werden die Tabellen, welche dem Prozessmodell zugeordnet werden, beschrieben. Abschnitt 3.2 beschreibt hierbei die Tabellen der Prozessinstanz. Beide Abschnitte enthalten Informationen über alle Tabellen, deren Beziehungen untereinander und die Attribute der einzelnen Tabellen. In Abschnitt 3.3 wird die Integration des Datenbankmodells in die vorgestellte Architektur erläutert. Hierfür wird Object-relational Mapping (ORM) mittels Hibernate verwendet, deren Beschreibungen ebenso Bestandteil dieses Abschnittes sind.

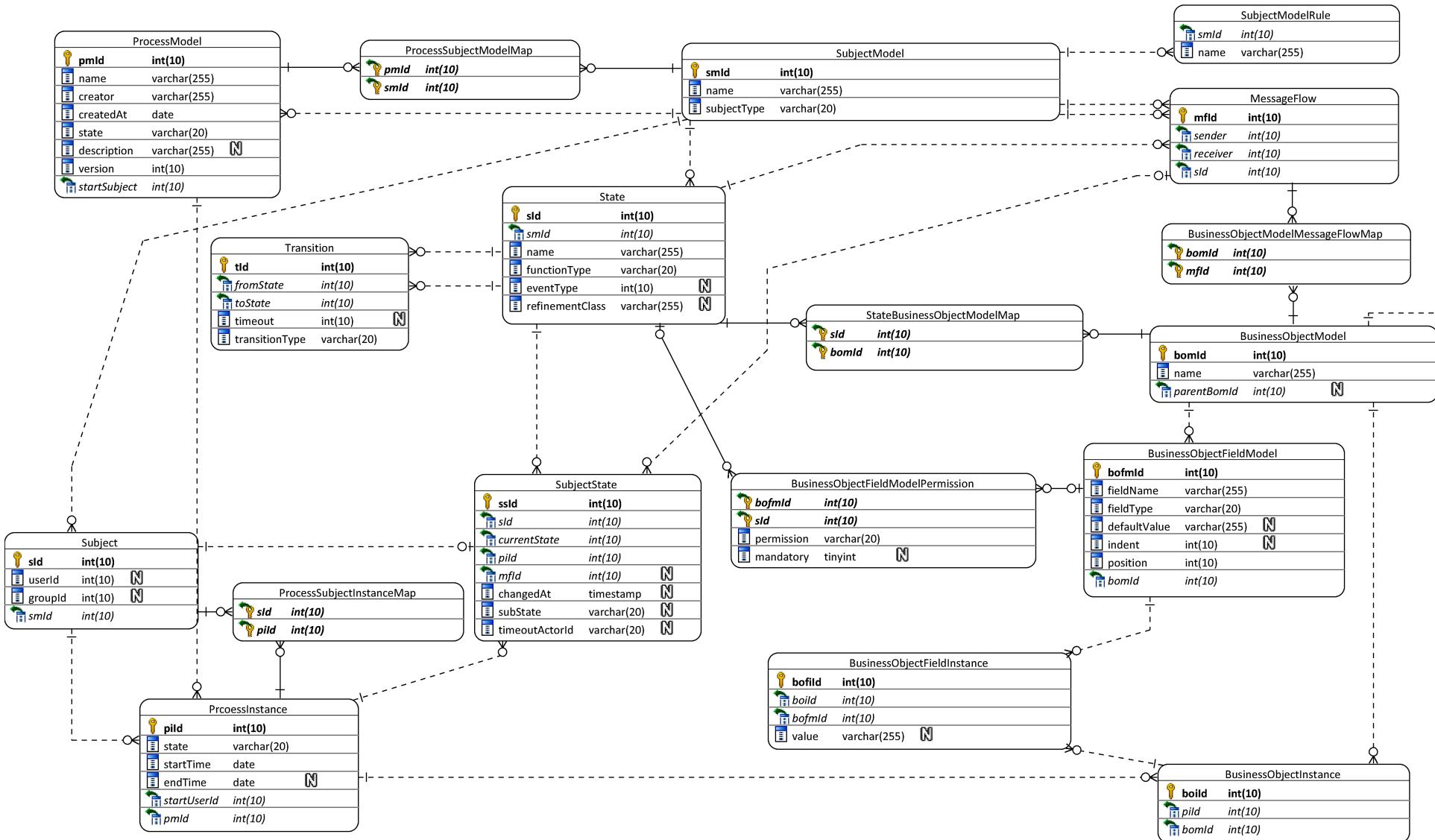


Abbildung 3.1: Datenbankmodell

3.1 Prozessmodell

In diesem Abschnitt werden alle relevanten Tabellen zur Beschreibung eines Prozessmodells erläutert.

ProcessModel

In dieser Tabelle werden Basisinformationen, wie z.B. Name des Prozessmodells, Beschreibung, etc. definiert. Folgende Attribute sind Bestandteile dieser Tabelle:

Attributname	Beschreibung
pmlId	Primärschlüssel dieser Tabelle
name	Attribut zur Speicherung des Namens des Prozessmodells
creator	Persistiert den Namen der Benutzerin/des Benutzers, die/der das Prozessmodell importiert hat
createdAt	Speichert den Zeitpunkt des Imports
state	Enumeration zur Speicherung des Status des Prozessmodells Mögliche Werte: ACTIVE und INACTIVE
description	Ermöglicht eine zusätzliche Beschreibung des Prozessmodells
version	Dient zur Persistierung der derzeitigen Versionsnummer des Prozessmodells. Prozessmodelle sind unveränderbar. Dementsprechend muss bei einer Änderung eines Prozessmodells, dieses neu importiert und mit einer neuen Versionsnummer versehen werden.
startSubject	Legt jenes Subjekt fest, welches den Prozess starten kann

Tabelle 3.1: Beschreibung der Attribute der Tabelle ProcessModel

SubjectModel

Mithilfe dieser Tabelle werden die beteiligten Subjekte eines Prozesses definiert. Ein Subjektmodell kann natürlich Bestandteil von mehreren Prozessmodellen sein.

Attributname	Beschreibung
smId	Primärschlüssel dieser Tabelle
name	Speichert den Namen des Subjektmodells
subjectType	Enumeration, welche festlegt, ob es sich um ein internes oder externes Subjekt handelt Mögliche Werte: INTERNAL, EXTERNAL.

Tabelle 3.2: Beschreibung der Attribute der Tabelle SubjectModel

ProcessSubjectModelMap

Hierbei handelt es sich um eine notwendige Zwischentabelle, mit der die Zuordnung von Subjekt- und Prozessmodellen erfolgt. Demgemäß wird festgelegt, welche Subjektmodelle Bestandteil eines Prozessmodells sind.

Attributname	Beschreibung
pmId	Fremdschlüssel der Tabelle ProcessModel und Primärschlüssel dieser Tabelle
smId	Fremdschlüssel der Tabelle SubjectModel und Primärschlüssel dieser Tabelle

Tabelle 3.3: Beschreibung der Attribute der Tabelle ProcessSubjectModelMap

SubjectModelRule

Diese Tabelle definiert welche Benutzerinnen bzw. Benutzer einer Organisationseinheit einem Subjektmodell zugeordnet werden können. Die Zuordnung basiert hierbei auf Berechtigungen (Rules). Mehr Informationen über dieses Zuordnungsmodell befinden sich im Abschnitt 4.2.3.2.

Attributname	Beschreibung
smId	Fremdschlüssel der Tabelle SubjectModel und Primärschlüssel dieser Tabelle
name	Definiert den Namen der Rule

Tabelle 3.4: Beschreibung der Attribute der Tabelle SubjectModelRule

State

Mithilfe dieser Tabelle wird das interne Verhalten eines Subjekts beschrieben. Dementsprechend wird hier festgelegt, welche Aktivitäten in einem bestimmten Status für ein Subjekt durchzuführen sind. Ein Statusobjekt ist in einem Prozessmodell genau einem Subjektmodell zugeordnet und einem Subjektmodell können beliebig viele Statusobjekte zugeordnet sein. Die Beschreibung der Attribute dieser Tabelle erfolgt in Tabelle 3.5.

Transition

Diese Tabelle legt fest in welcher Reihenfolge das interne Verhalten eines Subjektes abzuarbeiten ist bzw. legt fest, welche Statusübergänge möglich sind. Dies wird ermöglicht, indem State-Paare (fromState und toState) gebildet werden. Die Attribute werden in Tabelle 3.6 näher veranschaulicht.

Attributname	Beschreibung
sId	Primärschlüssel dieser Tabelle
sMId	Fremdschlüssel der Tabelle SubjectModel; mithilfe dieses Attributs wird die Zuordnung zum Subjektmodell festgelegt.
name	Definiert den Namen
functionType	Hiermit wird festgelegt, ob es sich um einen Function-State, Send-State oder Receive-State (siehe Abschnitt 2.1.2.2) handelt. Damit ist für die Process-Engine ersichtlich, wie ein State abzuarbeiten ist. Angesichts der Unterstützung von Refinements (siehe Abschnitt 2.1.3.3) wurde ein spezieller Refinement-State eingeführt, damit für die Process-Engine erkennbar ist, dass hier keine Benutzerinnen- bzw. Benutzerinteraktion notwendig ist. Mögliche Werte: SEND, RECEIVE, FUNCTION, REFINEMENT
eventType	Festlegung, ob es sich um einen Start- bzw. End-State handelt Mögliche Werte: START, END
refinementClass	Handelt es sich um einen Refinement-State, dann wird mithilfe dieses Attributes festgelegt, welche Java-Klasse im Hintergrund ausgeführt wird.

Tabelle 3.5: Beschreibung der Attribute der Tabelle State

Attributname	Beschreibung
tId	Primärschlüssel dieser Tabelle
fromState	Fremdschlüssel der Tabelle State; ausgehend von diesem State werden die möglichen nächsten States definiert.
toState	Fremdschlüssel der Tabelle State; ermöglicht die Definition des nächsten States auf Basis des fromState.
timeout	Ermöglicht die Festlegung eines Zeitrahmens in Sekunden; nach Ablauf dieser Zeit, wird ein Timeout ausgelöst.
transitionType	Definition, ob es sich um einen Standardübergang (wird von der Benutzerin bzw. dem Benutzer ausgeführt) oder einem speziellen Übergang (z.B. in Form eines Timeouts) handelt. Mögliche Werte: NORMAL, AUTO_TIMEOUT

Tabelle 3.6: Beschreibung der Attribute der Tabelle Transition

BusinessObjectModel

Diese Tabelle definiert den Aufbau eines Business Objects (siehe Abschnitt 2.1.3.1). Einem Business Object können beliebig viele Felder zugeordnet sein. Des Weiteren kann ein Business Object weitere Business Objects enthalten.

Attributname	Beschreibung
bomId	Primärschlüssel dieser Tabelle.
name	Definiert den Namen des Business Objects
parentBomId	Legt das übergeordnete Business Object fest; dieses Attribut ist leer, wenn kein übergeordnetes Business Object vorhanden ist.

Tabelle 3.7: Beschreibung der Attribute der Tabelle BusinessObjectModel

BusinessObjectFieldModel

Hiermit werden die Felder eines Business Objects beschrieben. Folglich ist ein Feld genau einem BusinessObjectModel zugeordnet.

Attributname	Beschreibung
bofId	Primärschlüssel der Tabelle
fieldName	Definiert den Namen des Feldes
fieldType	Festlegung des Datentyps des Feldes; dementsprechende Festlegung, ob es sich beispielsweise um einen Text, eine Nummer handelt. Mögliche Werte: STRING, NUMBER, DECIMAL, DATE, TIMESTAMP, BOOLEAN
defaultValue	Speicherung eines Standardwertes für das Feld
indent	Mögliche Einrückung zur Formatierung des Feldes bei der Anzeige
position	Legt die Reihenfolge der Felder fest
bomId	Fremdschlüssel der Tabelle BusinessObjectModel und somit Zuordnung zum Business Object

Tabelle 3.8: Beschreibung der Attribute der Tabelle BusinessObjectFieldModel

StateBusinessObjectModelMap

Mithilfe dieser Tabelle wird ein Business Object für einen Status verfügbar gemacht. Demgemäß erfolgt die Zuordnung von BusinessObjectModel und State.

Attributname	Beschreibung
sId	Fremdschlüssel der Tabelle State und folglich Festlegung des Status
bomId	Fremdschlüssel der Tabelle BusinessObjectModel; hiermit erfolgt die Definition des Business Objects.

Tabelle 3.9: Beschreibung der Attribute der Tabelle StateBusinessObjectModelMap

BusinessObjectFieldModelPermission

Für jedes Feld eines Business Objects können Berechtigungen für einen bestimmten Status (State) deklariert werden. Somit wird definiert, welches Subjekt in welchem Status Änderungen an einem Feld des Business Objects vornehmen darf bzw. ob nur Leserechte oder gar keine Rechte vorhanden sind.

Attributname	Beschreibung
bofId	Fremdschlüssel der Tabelle BusinessObjectFieldModel und Primärschlüssel dieser Tabelle
sId	Fremdschlüssel der Tabelle State und Primärschlüssel dieser Tabelle
permission	Festlegung der Berechtigung für das Feld; grundsätzlich wird unterschieden zwischen Leserechten, Schreibrechten und gar keinen Rechten (d.h. Feld wird gar nicht angezeigt). Mögliche Werte: READ, READ_WRITE, NONE
mandatory	Zusätzliche Definition, ob der Wert des Feldes leer sein darf

Tabelle 3.10: Beschreibung der Attribute der Tabelle BusinessObjectFieldModelPermission

MessageFlow

Handelt es sich beim Status um einen Send-State bzw. Receive-State, wird ein Eintrag in dieser Tabelle erstellt, um den Nachrichtenfluss zwischen Subjekten zu bestimmen.

Attributname	Beschreibung
mFId	Primärschlüssel dieser Tabelle
sender	Fremdschlüssel der Tabelle SubjectModel; damit wird das Sender-subjekt persistiert.
receiver	Fremdschlüssel der Tabelle SubjectModel; hiermit erfolgt die Definition des Empfängers der Nachricht.
sId	Fremdschlüssel der Tabelle State

Tabelle 3.11: Beschreibung der Attribute der Tabelle MessageFlow

BusinessObjectModelMessageFlowMap

Sobald es zum Austausch von Nachrichten zwischen Subjekten kommt, muss der Aufbau der Nachrichten definiert werden. Im Kontext dieser Process-Engine beinhalten Nachrichten Business Objects, womit bei der Interaktion zwischen Subjekten Business Objects gesendet werden. Ein Business Object kann natürlich Bestandteil von mehreren Interaktionen sein.

Attributname	Beschreibung
bomId	Fremdschlüssel der Tabelle BusinessObjectModel und folglich die Zuordnung des Business Objects
mflId	Fremdschlüssel der Tabelle MessageFlow; mithilfe dieses Attributes erfolgt die Deklaration des Nachrichtenflusses

Tabelle 3.12: Beschreibung der Attribute der Tabelle BusinessObjectModelMessageFlowMap

3.2 Prozessinstanz

In diesem Abschnitt werden alle Tabellen, die für die Speicherung und Ausführung einer Prozessinstanz notwendig sind, beschrieben.

ProcessInstance

Nach dem Start eines Prozesses wird ein Eintrag in dieser Tabelle, welche die Basisinformationen einer konkreten Prozessinstanz enthält, erstellt.

Attributname	Beschreibung
pild	Primärschlüssel dieser Tabelle
state	Definiert den aktuellen Status (z.B. gestartet, beendet) der Prozessinstanz; mögliche Werte: ACTIVE, FINISHED, CANCELLED_BY_SYSTEM, CANCELLED_BY_USER
startTime	Speichert den Zeitstempel beim Start der Prozessinstanz
endTime	Speichert den Zeitstempel bei Beendigung der Prozessinstanz
startUser	Fremdschlüssel der Tabelle Subject, um die Benutzerin/den Benutzer, welche/r den Prozess gestartet hat, zu persistieren
pmlId	Fremdschlüssel der Tabelle ProcessModel, damit die Zuordnung zum Prozessmodell bekannt ist

Tabelle 3.13: Beschreibung der Attribute der Tabelle ProcessInstance

Subject

Diese Tabelle dient zur Speicherung eines konkreten Subjekts und deren Zuordnung zu einem Organisationselement (z.B. Benutzerin bzw. Benutzer des Active Directory).

Attributname	Beschreibung
sId	Primärschlüssel dieser Tabelle
userId	Zuordnung der/des Benutzerin/Benutzers einer Organisation
groupId	Zuordnung der Gruppe einer Organisation
smId	Fremdschlüssel der Tabelle SubjectModel, damit das Modell eines Subjektes bekannt ist

Tabelle 3.14: Beschreibung der Attribute der Tabelle Subject

ProcessSubjectInstanceMap

Das ist eine notwendige Zwischentabelle zur Definition der beteiligten Subjekte einer Prozessinstanz.

Attributname	Beschreibung
pId	Fremdschlüssel der Tabelle ProcessInstance und folglich die Definition einer konkreten Prozessinstanz
sId	Fremdschlüssel der Tabelle Subject, damit das beteiligte Subjekt persistiert ist

Tabelle 3.15: Beschreibung der Attribute der Tabelle ProcessSubjectInstanceMap

SubjectState

Diese Tabelle wird verwendet, um den derzeitigen Status eines Subjekts zu speichern. Für alle internen Subjekte eines Prozesses wird ein Eintrag in dieser Tabelle angelegt. Bei Statusänderungen eines Subjektes wird der Eintrag in dieser Tabelle angepasst. Die Attribute werden in Tabelle 3.16 beschrieben.

BusinessObjectInstance

Speichert eine konkrete Instanz eines Business Objects nach Vorgabe des Modells. Diese Instanz eines Business Objects ist genau einer Prozessinstanz zugeordnet. Die genauere Beschreibung erfolgt in Tabelle 3.17.

BusinessObjectFieldInstance

Mithilfe dieser Instanz wird ein Feld eines Business Objectspersistiert. Jedes Feld speichert hierbei den Wert in einem Format, welches abhängig vom definierten Datentyp ist. Die Attribute werden näher in Tabelle 3.18 veranschaulicht.

Attributname	Beschreibung
ssId	Primärschlüssel dieser Tabelle
sId	Fremdschlüssel der Tabelle Subject und somit Zuordnung des Subjektes einer Prozessinstanz
currentState	Fremdschlüssel der Tabelle State; mithilfe dieses Attributes wird der aktuelle Status eines Subjektes im Prozess definiert.
pId	Fremdschlüssel der Tabelle ProcessInstance
mflId	Fremdschlüssel der Tabelle MessageFlow; damit wird festgelegt, welcher Nachrichtenfluss eingetreten ist und welche Nachricht empfangen wurde.
changedAt	Zeitstempel, der die letzte Änderung an diesem Eintrag speichert
subState	Zusätzlich notwendiger Status, für Send-States, Receive-States und Notifikationen an andere Services Mögliche Werte: TO_RECEIVE, RECEIVED, TO_SEND, SENT, NOTIFIED_EC
timeoutActorId	Persistiert den Actor, der ein mögliches Timeout auslöst

Tabelle 3.16: Beschreibung der Attribute der Tabelle SubjectState

Attributname	Beschreibung
boId	Primärschlüssel der Tabelle
pId	Fremdschlüssel der Tabelle ProcessInstance, damit die Zuordnung einer konkreten Prozessinstanz gewährleistet ist
boModelId	Fremdschlüssel der Tabelle BusinessObjectModel, welche den Aufbau einer Instanz vorgibt

Tabelle 3.17: Beschreibung der Attribute der Tabelle BusinessObjectInstance

Attributname	Beschreibung
bofId	Primärschlüssel dieser Tabelle
boId	Fremdschlüssel der Tabelle BusinessObjectInstance
bofModelId	Fremdschlüssel der Tabelle BusinessObjectFieldModel, womit der Aufbau, der Datentyp, etc. vorgegeben werden
value	Attribut zur Speicherung des konkreten Wertes. Die Art und Weise, wie der Wert gespeichert wird, ist abhängig vom Datentyp

Tabelle 3.18: Beschreibung der Attribute der Tabelle BusinessObjectFieldInstance

3.3 Integration des Datenbankmodells in den Prototypen

Wie bereits kurz erwähnt, wird das Object-relational Mapping (ORM) mittels Hibernate realisiert. Damit werden relationale Daten von Datenbanken in Java Objekte gemappt. Hibernate ist dabei eines der meist genutzten Java-Frameworks für das ORM. Dieser Erfolg basiert auf der stetigen Weiterentwicklung von Hibernate. Des Weiteren ist dieses Framework umfassend dokumentiert, modular aufgebaut, beliebig anpassbar und für verschiedenste Datenbankanbieter verfügbar, womit es naheliegend ist, dass Hibernate für den vorgestellten Prototypen eingesetzt wird [9].

Nahezu alle Softwareanwendungen benötigen persistente (nicht flüchtige) Daten. Im Kontext von Java bedeutet dies meist, dass Objekte gemappt und konkrete Objektinstanzen mithilfe von SQL in einer Datenbank gespeichert werden. Für manche Anwendungen mag es ausreichend sein, Datenbankzugriffe mit Java Database Connectivity (JDBC) zu realisieren, jedoch leidet die Codequalität darunter, da der Code aufgrund unzähliger SQL-Befehle direkt im Sourcecode deutlich schwieriger zu lesen ist. Anwendungen, welche in der Lage sind Prozessmodelle zu speichern und diese auszuführen, müssen jedenfalls eine große Anzahl von SQL-Befehlen absetzen. Aus diesem Grund ist der Einsatz eines Frameworks für ORM absolut notwendig. Das Objekt-basierte und relationale Mapping basiert in Hibernate auf Metadaten, die in Java-Klassen mittels Annotationen (z.B. @Entity, @Column, etc.) definiert werden. Dementsprechend wird eine Zuordnung von Klassen einer Anwendung und dem Schema einer Datenbank geschaffen. Im Wesentlichen ermöglicht ORM und somit Hibernate eine reversible Transformation von Daten von einer Darstellung in eine andere Darstellung [9].

Hibernate wurde in der Java-Welt so gut aufgenommen, dass die standardisierte Java Persistence API einige Ideen von Hibernate aufgenommen und spezifiziert hat (diese gelten natürlich auch für Hibernate) [13]:

- Verwendung von Metadaten für das Mapping von persistenten Klassen und Tabellen eines Datenbankschemas
- APIs, welche die einfache Ausführung von Create, Read, Update and Delete (CRUD)-Operationen für persistente Klassen ermöglichen
- Java Persistence Query Language – dem SQL sehr ähnlich ist – zur Spezifikation von Datenbankabfragen

Abbildung 4.1 veranschaulicht die Gesamtarchitektur des Prototypen. In dieser Architektur ist auch das Modul Persistence ersichtlich. Jedoch handelt sich es sich bei diesem Modul um kein Microservice, sondern um ein Java-Package, welches in die Microservices ProcessModelStorage und ProcessEngine inkludiert ist. Dieses Package enthält alle Mappings für das in diesem Kapitel vorgestellte Datenbankmodell (siehe Abbildung 3.1). Bestandteil dieses Java-Packages sind somit Java-Klassen – mit Annotationen versehen – die die Abfrage und Speicherung von Prozessmodellen bzw. Prozessinstanzen ermöglichen.

3.3.1 Verwendung von Interfaces

Ein weiterer Bestandteil dieses Packages sind Interfaces¹ (z.B. für ProcessModel, ProcessInstance, etc.). Die definierten Interfaces geben vor, welche Methoden in den abgeleiteten Klassen jedenfalls enthalten sein müssen. Abbildung 3.2 zeigt beispielhaft den Aufbau der Objekthierarchie. Infolgedessen ist für jede persistente Klasse (in diesem Fall basierend auf Hibernate) ein Interface definiert, in welchem die zu implementierenden Methoden angegeben werden. Diese Art der Umsetzung bringt einige Vorteile, wie die Entkopplung von Spezifikation und Implementierung, mit sich. Weiters ist die interne Implementierung nach außen hin nicht bekannt und die Implementierung kann auch für Tests angepasst werden [27]. Ein weiterer Vorteil ist, dass die Art und Weise wie die Daten (in diesem Prototypen in einer Datenbank) gespeichert werden, leicht geändert werden kann. Beispielsweise können die Daten als JSON-Datei auf einer Festplatte abgelegt werden. Dazu müssen nur neue Implementierungsklassen, basierend auf den vorgegebenen Interfaces, erstellt werden. Da den Microservices die Implementierungsdetails nicht bekannt sind und diese nur die Interfaces des Persistence-Package verwenden, sind keine weiteren Anpassungen notwendig.

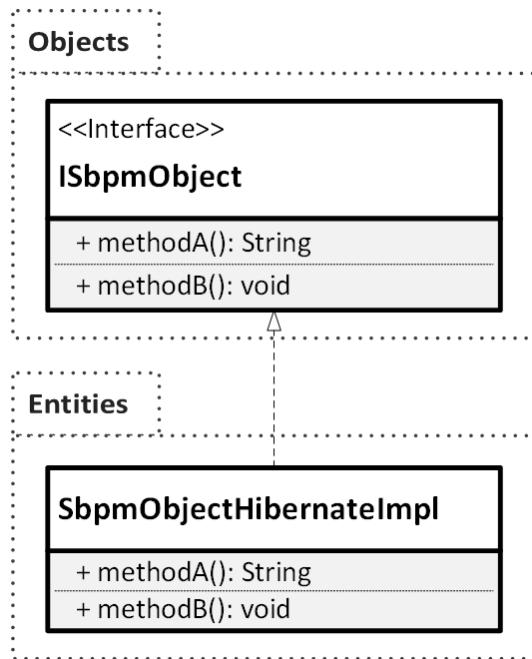


Abbildung 3.2: Objekthierarchie des Persistence-Packages

Eine weitere Besonderheit dieses Packages ist die Unveränderbarkeit (Read-Only Interfaces und Immutable Objects) jener Klassen, die zur Speicherung eines Prozessmodells verwendet werden. Dementsprechend können keine Änderungen an den Objekten vorgenommen werden und nur lesender Zugriff ist möglich. Dies führt dazu, dass bei Änderungen an einem Prozessmodell, dieses Modell neu importiert werden muss. Dafür wurde auch das Attribut `version` (siehe Tabelle 3.1) definiert. Bei Klassen der Prozessinstanz ist diese Unveränderbarkeit nicht gegeben, da hier interne Änderungen (z.B. Statusübergänge) notwendig sind.

¹ Interfaces sind Verträge zwischen Interfaces und implementierenden Klassen, die sicherstellen, dass vorgegebene Methoden implementiert werden [27].

3.3.2 Verwendung des Builder-Pattern

Üblicherweise werden neue Instanzen von Klassen per Konstruktor erstellt. Sobald eine große Anzahl an Parametern für die Instanziierung von Objekten notwendig ist, wird dies für die Entwicklerin bzw. den Entwickler unübersichtlich und folglich fehleranfällig. Darüber hinaus muss unbedingt sichergestellt werden, dass Objekte und deren notwendigen Parameter vollständig und korrekt instanziert sind, damit inkonsistente Objekte vermieden werden. Aus diesen Gründen wird für die Instanziierung das Builder-Pattern eingesetzt. Bei der Verwendung des Builder-Pattern werden Entwicklerinnen bzw. Entwickler bei der Instanziierung unterstützt, indem für jeden Parameter eine Methode bereit gestellt wird. Jede dieser Methoden liefert wiederum die Builder-Instanz zurück. Des Weiteren überprüft der Builder, ob alle Parameter vollständig und korrekt definiert wurden, womit inkonsistente Zustände verhindert werden. Erst bei Aufruf der Methode `build` wird eine Instanz des Objekts erstellt [11].

Abbildung 3.3 veranschaulicht die Anwendung des Builder-Pattern. Ein generisches Interface definiert, dass Builder jedenfalls die Methode `build` implementieren müssen. Mithilfe des generischen Typs `<T>` wird der Rückgabetyp festgelegt. Im angegebenen Beispiel ist der Rückgabetyp mit `ISbpmObject`, welches in Abbildung 3.2 eingeführt wurde, definiert. Zudem wird in Listing 3.1 die beispielhafte Erstellung eines `ProcessModel`-Objekts mittels Builder-Pattern gezeigt.

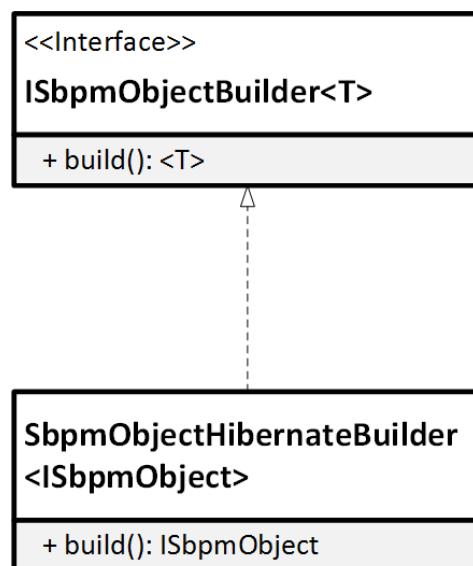


Abbildung 3.3: Instanziierung von Objekten des Persistence-Packages mittels Builder-Pattern

Listing 3.1: Builder-Pattern Beispielverwendung

```

1 final ProcessModel pm = new ProcessModelBuilder()
2     .name("Vacation Request")
3     .description("This is a vaction request!")
4     .state(ProcessModelState.ACTIVE)
5     .version(1.1F)
6     .build();
  
```

4 Prototyp

In diesem Kapitel wird eine prototypische, Microservice-basierte Architektur zur Speicherung und Ausführung von subjekt-orientierten Prozessen vorgestellt. In Abbildung 4.1 ist diese Architektur und die inkludierten Microservices veranschaulicht. Grundsätzlich basiert die Architektur auf sieben verschiedenen Microservices, welche Bestandteil der nachfolgenden Abschnitte sind (einzig beim dargestellten Modul Persistence handelt es sich um ein Java-Package, wie im Abschnitt 3.3 bereits erläutert wurde):

- GUI
- ConfigurationService
- Gateway
- ServiceDiscovery
- ExternalCommunicator
- ProcessModelStorage
- ProcessEngine

All diese Services nutzen Spring Boot und darauf aufbauend Spring Cloud als Basistechnologien (genauere Details sind im Abschnitt 4.1 zu finden). Des Weiteren wird REST, JSON und HTTP zur Kommunikation zwischen den Microservices eingesetzt. Die dauerhafte Speicherung der Daten erfolgt mittels MySQL.

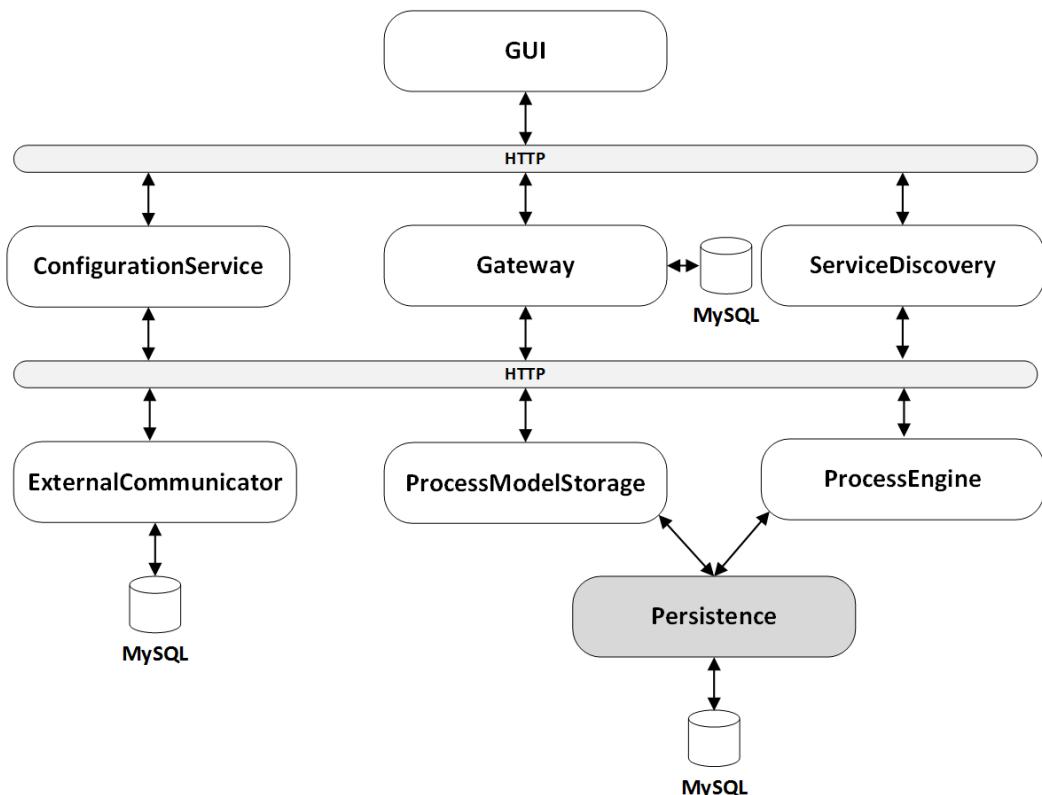


Abbildung 4.1: Microservice-basierter Prototyp

4.1 Basistechnologien

Die genutzten Basistechnologien des Prototypen sind Bestandteil dieses Abschnittes. Im Abschnitt 4.1.1 wird Spring Boot vorgestellt. Darauf aufbauend wird im Abschnitt 4.1.2 Spring Cloud definiert und erläutert. Mithilfe dieser Technologien wurde die Microservice-basierte Architektur des Prototypen aufgebaut.

4.1.1 Spring Boot

Das Spring-Framework ist eines der bekanntesten Frameworks der Java-Welt und wird vor allem für Enterprise-Anwendungen von vielen Unternehmen eingesetzt. Es wird ständig aktiv weiterentwickelt und stellt eine solide Basisarchitektur – basierend auf Design-Patterns – für Anwendungen zur Verfügung. Entwicklerinnen bzw. Entwickler können sich folglich leichter auf die Geschäftslogik konzentrieren, weil die grundlegende Basisarchitektur bereits von Spring bereitgestellt wird [23].

Spring Boot ermöglicht die Erstellung von eigenständigen, einfach ausführbaren Spring-Anwendungen. D.h. im Vergleich zum klassischen Spring-Framework ist in Spring Boot zusätzlich die Ausführungsumgebung (Tomcat, Jetty oder Undertow) direkt inkludiert. Demnach können alle Funktionen von Spring auch in Spring Boot genutzt werden. Des Weiteren bietet Spring Boot eine Basiskonfiguration an, in welche andere hilfreiche Bibliotheken eingebunden sind. Somit ist der Einstieg für Entwicklerinnen bzw. Entwickler recht einfach [40].

Die Verwendung von Spring Boot bringt einige Vorteile mit sich. Aufgrund der Verfügbarkeit von verschiedensten Startermodulen ist beispielsweise die Einbindung von Datenbanken (z.B. MySQL, MongoDB, etc.) mittels Hibernate oder das Veröffentlichen von REST-Schnittstellen unkompliziert möglich. Ein Nachteil des klassischen Spring-Frameworks ist die häufig komplexe Konfiguration. In Spring Boot ist dies nicht mehr der Fall, da die meisten Komponenten bereits automatisch konfiguriert werden und folglich die Komplexität der Konfiguration deutlich gesenkt wird. Beim produktiven Einsatz einer Anwendung ist es für viele Unternehmen essentiell, dass diese über bestimmte Metriken (z.B. über die Performance) verfügen und dementsprechend reagieren können. Das Modul Spring Boot Actuator stellt diese Metriken automatisch und ohne großen Konfigurationsaufwand bereit. Traditionelle Anwendungen stellen eine EAR- oder WAR-Datei zur Verfügung, welche mittels Webserver ausgeführt werden müssen. Mittels Spring Boot wird nur ein einzelnes JAR generiert, das die Ausführungsumgebung inkludiert. Infolgedessen ist das Deployment natürlich deutlich einfacher [34]. Des Weiteren erlaubt Spring Boot die Definition und die Verwendung von Profilen. Somit können beispielsweise verschiedene Profile für den Produktiv- und den Testbetrieb eingesetzt werden, womit das Deployment wiederum erleichtert wird. Ein weiterer wichtiger Bestandteil jeder Anwendung ist der Einsatz eines Log-Frameworks. Spring Boot bringt ein solches Framework bereits automatisch mit und ermöglicht zahlreiche Konfigurationsmöglichkeiten für dieses [23].

Darüber hinaus bietet das klassische Spring-Framework bereits mehrere Vorteile, welche ebenso für Spring Boot zutreffen. Ein Vorteil von Spring ist das effiziente Transaktionsmanagement, das vor allem die Komplexität bei Anwendungen mit Datenbankzugriff senkt. Zudem können die vorhanden Spring-Projekte (z.B. Spring Data für den Zugriff auf Datenbanken, Spring Security für die Sicherheit von Anwendungen, etc.) gleichermaßen in Spring Boot

eingesetzt werden [34].

Aufgrund der Einfachheit und den zahlreichen Funktionen, sowie der umfassenden Konfigurationsmöglichkeiten von Spring Boot, wurde es als Basis-Framework für die Microservices des Prototypen ausgewählt. Insbesondere die einfache Integration von MySQL-Datenbanken mittels Hibernate, welche essentiell für die Speicherung und Ausführung von subjekt-orientierten Prozessen ist, ist ein wichtiger Punkt, der für die Verwendung von Spring Boot spricht. Zudem ermöglicht die umfangreiche Dokumentation die schnelle Einbindung von Funktionen. Weiters muss bei der Ausführung des Prototypen keine zusätzliche Ausführungsumgebung installiert werden, da diese bereits im JAR eingebunden ist. Folglich ist das Testen und das Deployment der einzelnen Services deutlich vereinfacht.

4.1.1.1 Struktur eines Spring Boot-Projekts

In diesem Abschnitt wird die Struktur eines Spring Boot-Projekts, welche in Abbildung 4.2 veranschaulicht ist, beschrieben. Die gezeigte Struktur ist bei allen Spring Boot-Projekten ident. Jedes Spring Boot-Projekt enthält einen Ordner namens `src`. Der genannte Ordner besteht wiederum aus den zwei Ordnern `main` und `test`. Die gesamte Geschäftslogik wird im Ordner `main` unter `java` abgelegt. Konfigurationsdateien werden unter `resources` gespeichert. Das Bereitstellen eines Frameworks, das testbar ist, ist eines der Hauptziele von Spring. Infolgedessen werden diese im Ordner `test` gespeichert, damit die Übersichtlichkeit erhöht wird [23].

Zudem ist die Datei `build.gradle` ebenso ein essentieller Bestandteil eines Spring Boot-Projekts. Gradle benötigt diese Datei, um das Projekt zu bauen und folglich das JAR zu generieren. In `build.gradle` werden alle notwendigen Konfigurationen und externen Abhängigkeiten eines Projekts definiert. Bei Gradle handelt es sich also um ein Build-Werkzeug, womit Builds automatisiert werden können. Dies ist vor allem für das automatisierte Deployment von Microservices von immenser Bedeutung. Des Weiteren verwaltet Gradle die Abhängigkeiten eines Projekts (z.B. externe Bibliotheken oder externe Projekte), indem diese automatisch bezogen und in der angegebenen Version eingebunden werden. Im Vergleich zu anderen Build-Werkzeugen, sind Gradle-Skripte deutlich unkomplizierter, da sie auf Groovy basieren. Darüber hinaus wird die parallele Ausführung von Builds unterstützt. Die Folge davon ist, dass Gradle gegenüber anderen Spring Boot-Build-Werkzeugen, wie z.B. Maven, präferiert werden sollte [31].

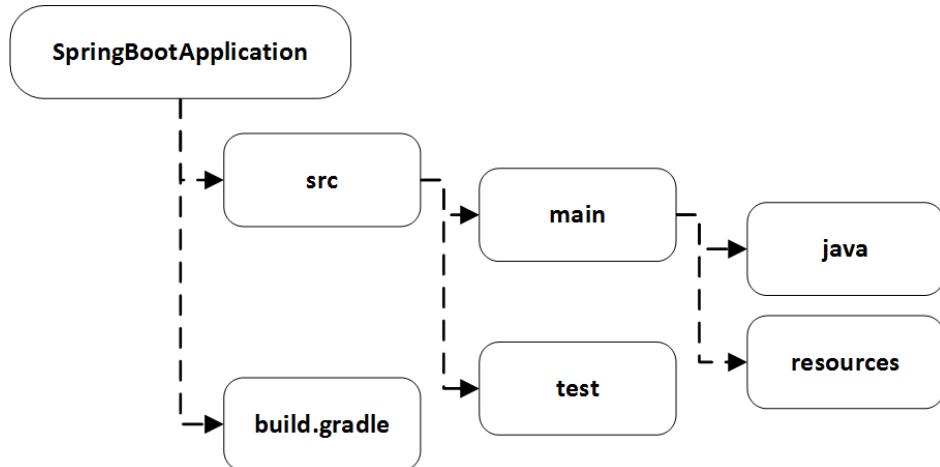


Abbildung 4.2: Struktur eines Spring Boot-Projekts (modifiziert übernommen von [39])

In Listing 4.1 wird der Ausschnitt einer solchen Gradle-Konfiguration gezeigt, die für die Generierung einer Spring Boot-Anwendung notwendig ist. In Zeile 3 wird hierbei die Version von Spring Boot angegeben. Zeile 7 veranschaulicht die Einbindung des Spring Boot-Plugins. Die Zeilen 11-13 definieren den Namen und die Version der JAR-Datei, welche mittels Gradle erstellt wird. Weiters werden in der Konfiguration die Abhängigkeiten der Spring Boot-Anwendung festgelegt. Dementsprechend werden anhand dieses Beispiels Module für den Datenbankzugriff sowie für REST-Schnittstellen Bestandteil des Projekts [39].

Listing 4.1: Ausschnitt Gradle-Konfiguration für Spring Boot

```

1 buildscript {
2     ext {
3         springBootVersion = '1.5.2.RELEASE'
4     }
5
6     dependencies {
7         classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
8     }
9 }
10
11 jar {
12     baseName = 'processengine'
13     version = '0.0.1-SNAPSHOT'
14 }
15
16 dependencies {
17     compile 'org.springframework.boot:spring-boot-starter-data-rest'
18     compile 'org.springframework.boot:spring-boot-starter-data-jpa'
19     runtime 'mysql:mysql-connector-java'
20 }
  
```

4.1.1.2 Architektur einer Spring Boot-Anwendung

In Abbildung 4.3 wird die Architektur einer Spring Boot-Anwendung verdeutlicht. Infolgedessen ist diese auch die Basis für sämtliche Microservices des vorgestellten Prototypen.

Die Kommunikation nach außen hin wird im Controller implementiert. Die definierte API wird mittels REST und JSON bereitgestellt. Des Weiteren wird HTTP als Kommunikationsprotokoll eingesetzt. Dies gilt zum einen für alle externen Anfragen und zum anderen für Anfragen anderer Microservices. Mithilfe der Java-Bibliothek Jackson¹ werden Java-Objekte in JSON-Objekte bzw. JSON-Objekte in Java-Objekte transformiert. Dementsprechend ist die Deklaration der Objekte für die API einfach umsetzbar. Im Controller selbst ist keine Geschäftslogik enthalten. Die Geschäftslogik ist Bestandteil der Komponenten ServiceInterface und ServiceImpl. Bei der Komponente ServiceInterface handelt es sich um ein Interface, in dem der Servicevertrag und somit die notwendigen Funktionen des Services definiert werden. Die Implementierung selbst erfolgt in ServiceImpl, welches von ServiceInterface ableitet. Im Controller ist nur das ServiceInterface eingebunden und die Implementierung dahinter ist unbekannt. Dies hat den Vorteil, dass die Implementierung beispielsweise für Tests speziell angepasst werden kann. Darüber hinaus können spezifische Bedingungen (z.B. in Konfigurationsdateien) hinterlegt werden. Abhängig von diesen Bedingungen wird dann die entsprechende Implementierung geladen. Eine weitere Besonderheit dieser Architektur ist, dass die gesamten Datenbankzugriffe ausschließlich über Repository-Komponenten erfolgen. Diese werden in der ServiceImpl eingebunden und bieten verschiedenste Funktionen, wie z.B. das Lesen oder Speichern von Daten, an. Als Ausführungsumgebung für alle Microservices des Prototypen, welche im JAR direkt inkludiert ist, wird Tomcat – ein leichtgewichtiger Webserver – verwendet.

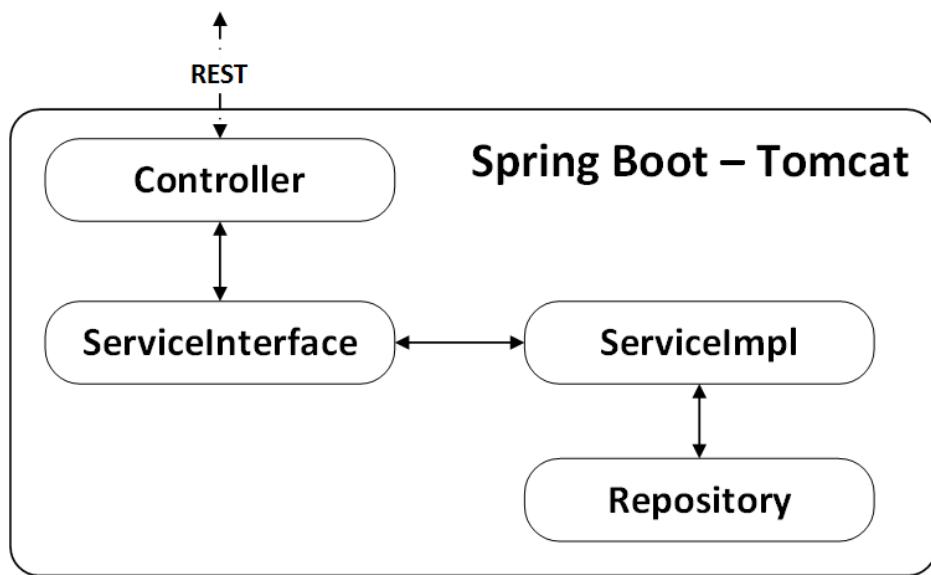


Abbildung 4.3: Architektur einer Spring Boot-Anwendung (modifiziert übernommen von [18])

4.1.2 Spring Cloud

Wie bereits erwähnt wird für den Prototypen zusätzlich das Framework Spring Cloud eingesetzt. Spring Cloud basiert hierbei auf Spring Boot und bietet weitere für Microservicearchitekturen nützliche Funktionen (z.B. zentrales Konfigurationsmanagement, Service Registry, Routing, etc.) an. Die Entwicklung und Koordination von mehreren Microservices ist durchaus komplex und bestimmte Funktionen (z.B. Kommunikation mit anderen Microservices) werden in allen Microservices benötigt. Dementsprechend würde dies zu

¹ <https://github.com/FasterXML/jackson>

4 Prototyp

Boilerplate-Code in allen Microservices führen, da bestimmte Funktionen immer und immer wieder implementiert werden müssten. Spring Cloud bietet einige dieser Funktionen bereits an, womit die Komplexität der einzelnen Microservices deutlich gesenkt wird [2].

Trotz des Namens ist Spring Cloud keine Cloud-Lösung. Nichtsdestotrotz werden Konfigurationen angeboten, um das Deployment von Spring Cloud-Anwendung in die Cloud zu ermöglichen. Derzeit werden die Cloud-Anbieter Cloud Foundry, AWS sowie Heroku unterstützt. Neue Anbieter werden ständig neu integriert. Spring Cloud-Anwendungen können auch auf normalen Desktop-Computern ausgeführt werden. Vor allem für Entwicklerinnen bzw. Entwickler ist dies angenehmer, da Spring Cloud-Anwendungen somit lokal testbar sind [36].

In allen Spring-Frameworks wird das Softwaredesign-Paradigma „Konvention vor Konfiguration“² verfolgt. D.h. Spring Cloud bietet ein Standardset an Konfigurationen an und nur wenn eine Konfigurationseinstellung nicht passend ist, muss diese angepasst werden. Dies vereinfacht die Arbeit von Entwicklerinnen bzw. Entwicklern und schnelle Erfolge sind leicht zu erzielen. Weiters sind die komplexen Funktionen von Spring Cloud leicht über die Konfigurationsdateien konfigurierbar [36].

Viele der Microservicekomponenten von Spring-Cloud stammen vom Netflix Open Source Software Center. Netflix ist einer der Pioniere und Vorreiter bei Microservice-basierten Architekturen. Um mit einer riesigen Anzahl von skalierbaren Microservices zurechtzukommen, hat Netflix eine Reihe von Werkzeugen und Techniken entwickelt, welche das Management der Microservices ermöglichen. Glücklicherweise hat Netflix diese Komponenten als Open-Source-Software zur Verfügung gestellt. Diese Komponenten wurden umfangreich getestet und sind im Produktivbetrieb erprobt. Infolgedessen wurden diese auch in Spring Cloud inkludiert. Zusätzlich wurde noch eine Abstraktionsschicht zwischen Netflix-Komponenten und Spring Cloud eingeführt, damit die Verwendung der Netflix-Komponenten einfacher ist [36].



Abbildung 4.4: Komponenten von Spring Cloud (modifiziert übernommen von [36])

Abbildung 4.4 zeigt einen Ausschnitt der vorhandenen Komponenten von Spring Cloud [36]:

- **Distributed Configuration:** Die Verwaltung von Konfigurationen ist in Microservice-basierten Umgebungen deutlich schwieriger zu bewerkstelligen, als mit monolithischen Ansätzen. Des Weiteren muss die Deklaration von verschiedenen Profilen (z.B. Test, Entwicklung, Produktion) möglich sein. Insofern ist es wichtig, dass Konfigurationen zentral verwaltet werden können. Spring Cloud Config ist jene Komponente von Spring Cloud, die genau dies ermöglicht.
- **Service Registry & Discovery:** Diese Komponente von Spring Cloud bietet Microservices die Möglichkeit sich an einer zentralen Stelle – der Service Registry & Discovery – zu

² Ziel dieses Paradigmas ist die Reduzierung der Komplexität von Konfigurationsdateien, indem Standardkonfigurationseinstellungen bei Nichtkonfiguration verwendet werden [26].

registrieren. Nach der Registrierung ist das Microservice für alle anderen Services verfügbar. Folglich können die anderen Services eine Service-Anfrage an die Service Registry senden und erhalten als Antwort Informationen über das angefragte Service (z.B. Uniform Resource Identifier (URI)). Spring Cloud unterstützt unter anderem die Projekte Eureka sowie ZooKeeper als Service Registry.

- Distributed Messaging: Die Komponente Spring Cloud Stream ist eine Abstraktions-schicht, die eine leichte Einbindung von Messaging-Lösungen wie Kafka oder RabbitMQ bereitstellt.
- Routing: Dies ist Bestandteil der API-Gateway-Komponente. Grundsätzlich werden damit Anfragen an die korrekten Microservices weitergeleitet. Spring Cloud inkludiert hierbei Zuul – eine leichtgewichtige Gateway-Komponente – welche verschiedenste Routing- und Filteroptionen zur Verfügung stellt.
- Load Balancing: Diese Software-basierte Komponente leitet die Anfragen an verfügbare Microservices, basierend auf Load Balancing Algorithmen, weiter. Hierfür wird bei Spring Cloud das Projekt Ribbon genutzt. Der Vorteil der Verwendung von Ribbon ist die einfache Integration mit Zuul zur Weiterleitung der Anfragen.
- Big Data Support: Big Data ist definitiv eine Herausforderung für viele Unternehmen. Auch hierfür bietet Spring Cloud bereits eine Komponente an, um Big Data zu integrieren. Die Projekte Spring Cloud Stream und Spring Cloud Data Flow erleichtern die Integration und den Umgang mit Big Data.
- Distributed Tracing: Vor allem für Operationen, die mehrere verschiedene Microservices nutzen, ist das Logging schwierig umzusetzen. Die Komponente Spring Cloud Sleuth schafft hierbei Abhilfe, damit eine nachvollziehbare Analyse der Log-Dateien möglich ist.
- Service to Service Calls: Die Kommunikation zwischen verschiedenen Microservices ist eine der häufigsten Anforderungen und essentiell für Microservice-basierte Architekturen. Das Projekt Spring Cloud Feign erlaubt die einfache Einbindung von REST-Schnittstellen basierend auf Java-Objekten. Die Nutzung von HTTP-Frameworks ist somit nicht notwendig.
- Cloud Support: Spring Cloud bietet eine Reihe von Funktionen, welche die Integration und das Deployment für verschiedene Cloud-Anbieter (z.B. Cloud Foundry, AWS) bewerkstelligen.
- Security: Die Sicherheit von Anwendungen ist natürlich immer eine fundamentale Anforderung. Vor allem bei der Verwendung von verteilten Systemen, ist dies ein schwierig umzusetzender Punkt. Die Komponente Spring Cloud Security unterstützt die Absicherung von verteilten Systemen, indem externe Authentifizierungssysteme inkludiert werden können. Des Weiteren ist die Verwendung von Single Sign On möglich.

Aufgrund der Tatsache, dass typische Ökosysteme von Microservices (siehe Abbildung 2.10) leicht mit Spring Cloud aufgebaut werden können, da diese Komponenten bereits alle vorhanden sind, wird Spring Cloud zusätzlich zu Spring Boot für den Prototypen verwendet. Des Weiteren ist die ausgezeichnete Integration von Spring Cloud in Spring Boot ebenso entscheidend. Aufgrund der unzähligen weiteren Komponenten von Spring Cloud ist der Prototyp jederzeit erweiterbar, womit neue Funktionen leicht eingefügt werden können. Ein weiterer Grund, der ebenso für den Einsatz von Spring Cloud spricht, ist das Unternehmen Netflix, welches die Entwicklung von neuen bzw. die Weiterentwicklung vorhandener Komponenten ständig vorantreibt. Es gilt als unumstritten in der Softwareentwicklungs-welt, dass Netflix

definitiv einer der größten Wissensträger von Microservice-basierten Architekturen ist und somit ausgezeichnete Komponenten bereitstellt. Des Weiteren sind die Komponenten sehr umfangreich dokumentiert und somit schnell einsetzbar. Insofern ist Spring Cloud ideal für den Prototypen, da die Integration von Spring Cloud-Komponenten ohne großen Aufwand möglich ist.

4.2 Komponenten des Prototypen

In diesem Abschnitt werden die einzelnen Komponenten (Microservices) des vorgestellten Prototypen, welche bereits in Abbildung 4.1 dargestellt wurden, erklärt und erläutert.

4.2.1 ConfigurationService

Für Microservice-basierte Architekturen ist es von essentieller Bedeutung, dass eine zentrale Stelle definiert ist, wo alle Konfigurationsdateien für alle Microservices abgelegt und für diese abrufbar sind. Ohne diesen Ansatz müssten die Konfigurationsdateien in den Microservices selbst gespeichert werden. Dementsprechend ist die Durchführung von Konfigurationsänderungen und das Deployment der Microservices schwierig durchzuführen. Des Weiteren können Konfigurationen mit dem Ansatz des zentralen Konfigurationsservers während der Laufzeit eines Microservices geändert werden.

Für das ConfigurationService wird Spring Cloud Config verwendet. Spring Cloud Config stellt ein Service zur Verfügung, dass die Verwaltung von Konfigurationsdateien in verteilten Systemen ermöglicht. Folglich wird eine zentrale Stelle deklariert, in der alle Konfigurationen der Microservices gespeichert sind. Aufgrund der Verwendung von Technologien wie REST, HTTP sowie JSON kann Spring Cloud Config nicht nur von Spring Cloud-Anwendungen genutzt werden, sondern auch von Anwendungen, welche auf anderen Frameworks basieren oder sogar andere Programmiersprachen nutzen [3]. In Abbildung 4.5 wird die Architektur des ConfigurationService vorgestellt (zur Veranschaulichung des ConfigurationService wurden nicht alle Microservices des Prototypen in die Grafik übernommen).

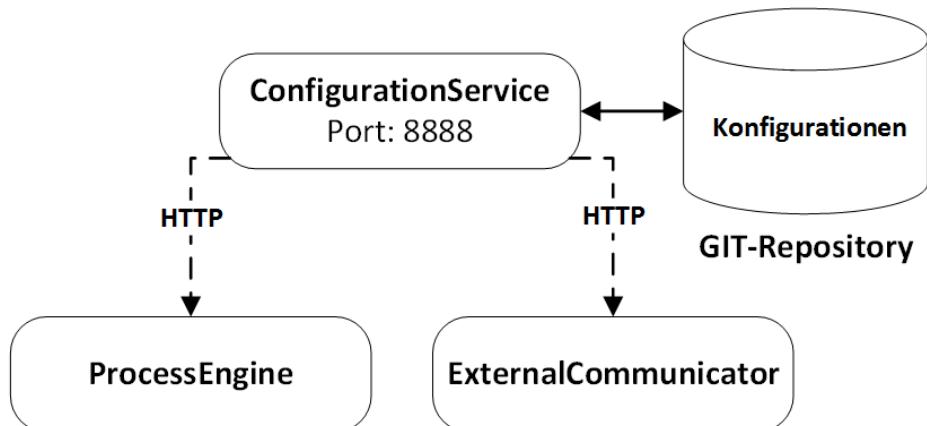


Abbildung 4.5: Architektur des Microservices ConfigurationService (modifiziert übernommen von [36])

Standardmäßig unterstützt Spring Cloud Config die Einbindung eines GIT-Repositories, in welchem die Konfigurationsdateien abgelegt werden. Die Verwendung von GIT ermöglicht

die Nachvollziehbarkeit von Änderungen bei Konfigurationen. Für den Prototypen sind die gesamten Konfigurationsdateien unter <https://github.com/stefanstaniAIM/IPPR2016-config> zu finden. Die Dateinamen der Konfigurationsdateien müssen einem definierten Namensschema folgen [3]:

- {application}: Namensangabe der Applikation (festgelegt mittels `spring.application.name`)
- {profile}: Deklaration des aktuellen Profils der Anwendung (z.B. Test, Produktion, etc.)
- {label}: Versionierung der Anwendung (nützlich für die Ausführung von Microservices in verschiedenen Versionen)

Beim Start der Microservices beziehen diese ihre Konfigurationen vom ConfigurationService. Weiters kann eine POST-HTTP-Anfrage an die Microservices gesendet werden. Bei Erhalt einer solcher Anfrage laden diese die Konfigurationen neu vom ConfigurationService. Jedoch müssen gewisse Konfigurationen (z.B. Name der Anwendung) in den Microservices selbst vorgenommen werden. Ansonsten ist nicht bekannt, welche Konfigurationsdateien aufgrund des Namensschemas vom ConfigurationService bezogen werden sollen. Listing 4.2 zeigt die Konfigurationsdatei `bootstrap.properties` der ProcessEngine, in welcher der Name der Anwendung sowie der URI des ConfigurationService angegeben wird. Weitere Konfigurationen sind zur Einbindung des ConfigurationService nicht notwendig.

Listing 4.2: `bootstrap.properties` der ProcessEngine

```
1 spring.application.name=process-engine  
2 spring.cloud.config.uri=http://127.0.0.1:8888
```

Spring Cloud Config bietet noch zusätzliche Funktionen an, die aber im Zuge der Entwicklung des Prototypen nicht eingebunden wurden. Zum einen wird die Verschlüsselung und Entschlüsselung von Konfigurationsdateien unterstützt. D.h. beim Senden der JSON-Datei (enthält die Konfigurationsparameter) an die Microservices werden diese nicht im Klartext übertragen, sondern verschlüsselt. Bei den Microservices selbst werden diese entschlüsselt. Zum anderen werden Authentifizierungsmechanismen unterstützt. Infolgedessen müssen jene Microservices, die Konfigurationen von der Spring Cloud Config beziehen möchten, in Besitz eines Passworts sein [3].

4.2.2 ServiceDiscovery

Service Registry & Discovery ist eine der zentralen Komponenten eines typischen Ökosystems von Microservices. Vor allem bei einer großen Menge von Microservices, deren Instanzen je nach Auslastung gestartet oder gestoppt werden, wäre es schwierig, alle URIs für die Microservices vorzukonfigurieren. Des Weiteren sind die URIs vor dem Start einer Serviceinstanz nicht immer bekannt und bei jeder Änderung müssten die Konfigurationen angepasst werden. Infolgedessen ist eine manuelle Konfiguration der Microservices in skalierbaren Umgebungen unmöglich und muss automatisiert werden. Die geforderte Automatisierung kann erreicht werden, indem sich Microservices beim Start selbstständig an einer zentralen Stelle – der Service Registry & Discovery – eintragen. Zudem werden die Services entfernt, wenn diese nicht mehr verfügbar sind. Die Service Registry & Discovery enthält die aktuellen Informationen über die Verfügbarkeit der Services sowie Metadaten (z.B. URI, Port, etc.) über diese. Die Microservices können Anfragen an die

Service Registry & Discovery senden, um Informationen über andere Services zu erhalten [36].

Die hier vorgestellte ServiceDiscovery nutzt hierbei die Spring Cloud-Komponente Eureka, die von Netflix entwickelt wurde. Die leichte Einbindung von Eureka in Spring-basierten Anwendungen ist definitiv ein Vorteil. Darüber hinaus bringt Eureka alle geforderten Funktionalitäten einer Service Registry & Discovery mit, weshalb Eureka in die prototypische Architektur inkludiert wurde [36].

Abbildung 4.6 stellt die Architektur und Funktionsweise der ServiceDiscovery dar. Grundsätzlich unterscheidet Eureka zwischen Server und Client. Die Komponente Server agiert hierbei als Service Registry & Discovery. Dementsprechend registrieren sich die Microservices – in diesem Kontext als Clients bezeichnet – beim Start am Server. Diese Registrierung beinhaltet üblicherweise den Namen des Microservice und die URI. Zusätzlich wird der Eureka Client verwendet, um Informationen über andere Services abzurufen. Nach erfolgreicher Registrierung sendet ein Client alle 30 Sekunden einen Ping an den Server. Dies dient als Bestätigung des Clients, dass dieser noch verfügbar und erreichbar ist. Empfängt der Server keinen Ping eines Clients mehr, so wird dieser Client aus der Service-Liste entfernt. Eureka Clients rufen diese Service-Liste selbstständig ab und speichern diese in ihrem Cache. Folglich sind die Informationen über andere verfügbare Services für die Clients verfügbar. Ein Update des Caches wird alle 30 Sekunden durchgeführt. Die Kommunikation von Server und Clients basiert wiederum auf HTTP, REST sowie JSON, womit Eureka auch für Anwendungen, welche nicht auf Spring Cloud basieren, nutzbar ist [36].

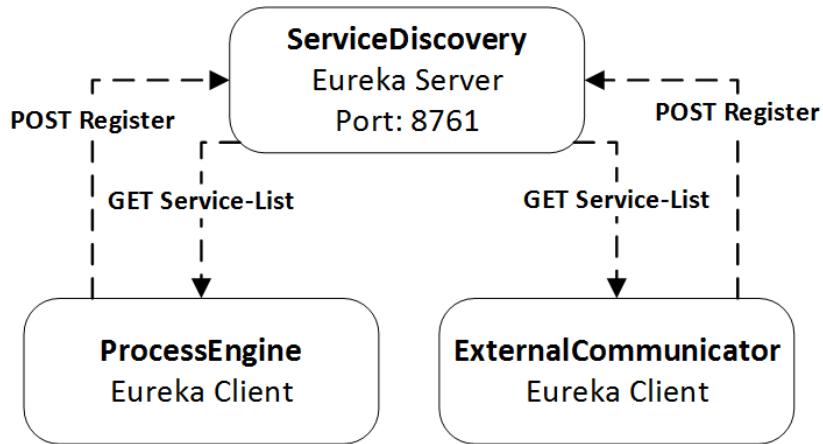


Abbildung 4.6: Architektur des Microservices ServiceDiscovery (modifiziert übernommen von [36])

In Abbildung 4.7 werden die Informationen, welche Eureka bereitstellt, veranschaulicht. Die Microservices des Prototypen werden nach erfolgter Registrierung in die Service-Liste der ServiceDiscovery aufgenommen und sind nun für andere Services verfügbar. Hierbei ist ersichtlich, dass der Name und die zugehörige URI gespeichert werden. Zudem besteht die Möglichkeit der Definition von Zonen, wobei jede Serviceinstanz einer Zone zugeordnet werden kann. Bei Anfragen werden zuerst jene Microservices, welche in der selben Zone des anfragenden Microservices liegen, zurückgeliefert.

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EXTERNAL-COMMUNICATOR	n/a (1)	(1)	UP (1) - 192.168.1.105:external-communicator:0
GATEWAY	n/a (1)	(1)	UP (1) - 192.168.1.105:gateway:10000
PROCESS-ENGINE	n/a (1)	(1)	UP (1) - 192.168.1.105:process-engine:0
PROCESS-MODEL-STORAGE	n/a (1)	(1)	UP (1) - 192.168.1.105:process-model-storage:0

Abbildung 4.7: Anzeige der verfügbaren Services von Eureka

4.2.3 Gateway

In diesem Abschnitt wird das Microservice Gateway erläutert (siehe Abbildung 4.8). Grund-sätzlich besteht dieses Service aus zwei Hauptkomponenten. Die Zuul Gateway API dient zur Weiterleitung aller Anfragen. Alle externen Anfragen (z.B. vom Microservice GUI) werden an das Gateway gesendet. Hier wird dann entschieden, an welches Microservice (z.B. ProcessEngine) die Anfrage weitergeleitet wird. Genaue Details dazu befinden sich im Abschnitt 4.2.3.1. Die zweite Hauptkomponente ist das AuthenticationService, welches die Authentifizierung aller Anfragen sicherstellt. Die Beschreibung dieser Komponente ist Bestandteil des Abschnitts 4.2.3.2. Des Weiteren nutzt das Gateway REST Controller bzw. Feign Client zur Kommunikation – basierend auf REST, JSON sowie HTTP – mit den anderen Microservices.

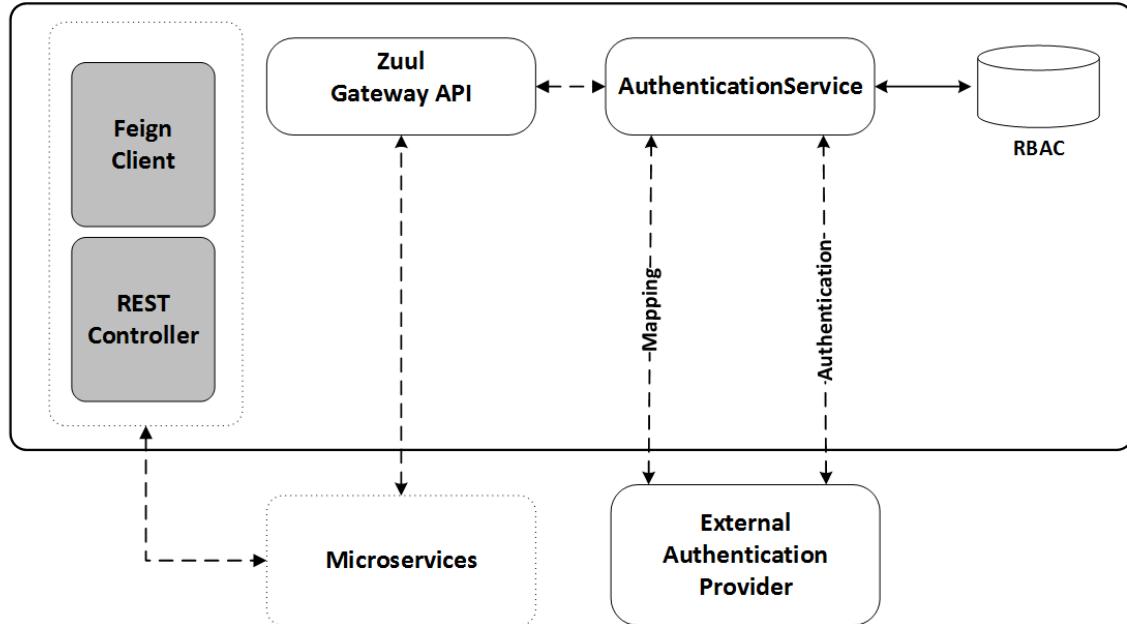


Abbildung 4.8: Architektur des Microservices Gateway

4.2.3.1 Zuul Gateway API

Wie bereits erwähnt, wird die Spring Cloud-Komponente Zuul zur Weiterleitung der Anfragen eingesetzt. In den meisten Microservice-basierten Architekturen werden die APIs der internen Microservices nach außen hin nicht veröffentlicht. Folglich werden nur die notwendigen

APIs der Microservices veröffentlicht. Weiters muss die korrekte Authentifizierung jeder Anfrage sichergestellt werden. Würden diese Anfragen nicht alle an das Gateway gesendet werden, müsste jedes einzelne Microservice eigene Authentifizierungsmechanismen bereitstellen. Einen weiteren Vorteil dieses Aufbaus zeigt folgendes Beispiel: Das Microservice GUI kommuniziert zum einen mit dem Service ProcessModelStorage und zum anderen mit dem Service ProcessEngine. Wäre kein zentrales Gateway vorhanden, müsste das Service GUI vor jeder Anfrage die ServiceDiscovery befragen. Bei Verwendung eines Gateways ist dies nicht der Fall, da die Anfrage einfach an dieses gesendet wird, welches die Anfrage richtig weiterleitet.

In Listing 4.3 wird eine Beispielkonfiguration von Zuul gezeigt. Zeile 4 legt fest, dass alle Anfragen, welche /ec/ zu Beginn des Pfades in der URI beinhalten, an das Microservice ExternalCommunicator weitergeleitet werden. Infolgedessen wird beispielsweise eine Anfrage <http://127.0.0.1:10000/api/ec/upload> mittels Gateway an den ExternalCommunicator weitergereicht. Natürlich gibt es noch weit mehr Konfigurationsmöglichkeiten (z.B. Definition von Zonen), jedoch wurden diese im Zuge der Implementierung des Prototypen nicht benötigt.

Ein weiterer Vorteil von Zuul ist die nahtlose Integration der Spring Cloud-Komponenten Eureka (für Service Registry & Discovery) und Ribbon (für Load Balancing). Nach dem Empfang einer Anfrage und der Zuordnung eines passenden Pfades, wird die serviceId (siehe Zeile 5 im Listing 4.3) an den Eureka Server gesendet. Der Eureka Server prüft anschließend in der Service-Liste, ob eine verfügbare Serviceinstanz für die serviceId vorhanden ist [36]. Sind mehrere verschiedene Serviceinstanzen für die serviceId verfügbar, wird ein Load Balancing der Serviceinstanzen mit Ribbon ausgeführt. Ribbon ist ein Software-basierter Load Balancer, der ebenso von Netflix entwickelt wurde. Standardmäßig wird ein einfacher Round Robin-Algorithmus eingesetzt, welcher die Anfragen auf die vorhanden Serviceinstanzen aufteilt. Weitere Algorithmen (z.B. basierend auf Zonen) sind vorhanden, jedoch sind die Standardeinstellungen für den Prototypen ausreichend [2].

Listing 4.3: Zuul-Beispielkonfiguration

```
1 zuul:
2   routes:
3     external-communicator:
4       path: /ec/**
5       serviceId: external-communicator
```

4.2.3.2 AuthenticationService

In diesem Abschnitt wird das AuthenticationService beschrieben, welches die Authentifizierung aller externen Anfragen regelt. Folglich ist dies Bestandteil des Services Gateway. Grundsätzlich ermöglicht das AuthenticationService die Anbindung an externe Authentifizierungssysteme, wie z.B. Active Directory. Dementsprechend handelt es sich beim AuthenticationService um kein eigenständiges Authentifizierungssystem, da Authentifizierungsanfragen an externe Systeme weiterdelegiert werden.

In Abbildung 4.9 wird das Interface AuthenticationProvider veranschaulicht. Um externe Authentifizierungssysteme einzubinden, muss eine Implementierungsklasse,

welche von AuthenticationProvider ableitet, vorhanden sein. In dieser Implementierungsklasse erfolgt dann die konkrete Anbindung an externe Systeme. Das Interface AuthenticationProvider enthält nur die Methode authenticateUser, welche die Authentifizierung durchführt. Als Parameter werden username und password benötigt. Diese Methode liefert ein Optional vom Typ User zurück. Enthält dieses Optional einen User, dann war die Authentifizierung erfolgreich. Ist das Optional leer, dann war dies nicht der Fall.

Die Festlegung der aktiven Implementierungsklasse von AuthenticationProvider erfolgt mithilfe des Konfigurationsparameters rbac.system.service in application.properties. Zusätzlich sind spezielle Bedingungen hinterlegt, die auf Basis des Konfigurationsparameters, die korrekte Implementierungsklasse laden. Alle Komponenten, welche Authentifizierungsfunktionen benötigen, nutzen ausschließlich das Interface AuthenticationProvider, womit die konkrete Implementierung nicht bekannt ist. Das Interface delegiert alle Anfragen an die Implementierungsklasse weiter. Mit diesem flexiblen Ansatz können beliebige externe Authentifizierungssysteme – sofern dies technisch möglich ist – inkludiert werden.

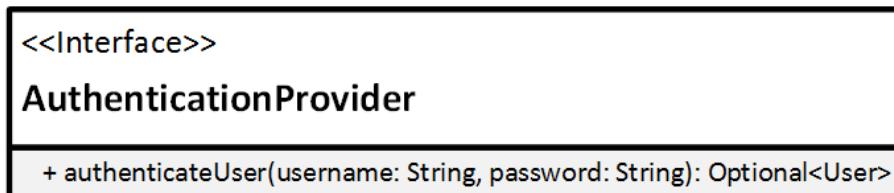


Abbildung 4.9: Klassendiagramm des Interfaces AuthenticationProvider

Nach Einbindung des externen Authentifizierungssystems, muss sichergestellt werden, dass die externen APIs abgesichert werden und folglich keine Anfragen ohne gültige Authentifizierung akzeptiert werden. Hierfür wird JSON Web Token (JWT) eingesetzt, welches hervorragend mit REST sowie HTTP harmoniert. Nach erfolgreichem Login sendet das Gateway einen generierten JWT an das aufrufende Microservice zurück. Ein solcher Token besteht aus drei Teilen [1]:

- Header: Dieser enthält den Token-Typ und den Hash-Algorithmus.
- Payload: Beispielsweise der Benutzername und andere Metadaten können Bestandteil dieser Komponente sein.
- Signature: Diese wird generiert, indem der Header, Payload sowie ein Passwort im angegebenen Hash-Algorithmus eingesetzt werden.

Für den JWT werden die drei angegebenen Komponenten mittels Punkt (.) separiert und sind somit einfach zu parsen. Nach Erhalt eines solchen Tokens speichern die Clients diesen meist als Cookie ab. Bei jeder Anfrage eines Clients muss der JWT im Authorization-Header der HTTP-Anfrage mitgesendet werden. Der Server – in diesem Fall das Gateway – prüft, ob der empfangene JWT valide ist. Ist dies der Fall, dann wird der Zugriff erlaubt. Aufgrund der Leichtgewichtigkeit von JWT und der einfachen Einbindung in HTTP-Anfragen, ist JWT einfach in den Prototypen integrierbar [1].

Das AuthenticationService basiert auf Role Based Access Control (RBAC), das sich zur Vergabe von Berechtigungen vor allem im Windows-Umfeld großer Beliebtheit erfreut. Dieses Prinzip, welches in Abbildung 4.10 dargestellt wird, setzt sich aus drei Komponenten zusammen. Den Benutzerinnen bzw. Benutzer (User) werden Gruppen (Role) zugeordnet.

Die Gruppen können beliebig viele Benutzerinnen bzw. Benutzer enthalten. Einer Gruppe können wiederum beliebig viele Berechtigungen (Rule) zugeordnet werden [16].



Abbildung 4.10: Aufbau des Role Based Access Control (RBAC) (modifiziert übernommen von [16])

Wie im Abschnitt 3.1 bereits kurz erwähnt wurde, basiert die Zuordnung von Subjektmodellen und Organisationselementen auf vergebene Berechtigungen. Einem Subjektmodell können mehrere Berechtigungen zugeordnet werden. Demgemäß kann eine Benutzerin/ein Benutzer die Rolle eines Subjekts nur dann übernehmen, wenn die geforderte Berechtigung vorhanden ist. Beispielsweise ist dem Subjektmodell „Mitarbeiterin“ die Berechtigung „Mitarbeiterinnen-Berechtigung“ zugeordnet. Die Benutzerin „BenutzerinA“ ist Bestandteil der Gruppe „Mitarbeiterinnen-Gruppe“, welche wiederum über die „Mitarbeiterinnen-Berechtigung“ verfügt. Insofern ist es der „BenutzerinA“ erlaubt, als „Mitarbeiterin“ zu agieren.

Das AuthenticationService ist so konzipiert, dass nur Authentifizierungsanfragen an das externe Authentifizierungssystem weiterdelegiert werden. Um dies zu ermöglichen, werden die Benutzerinnen bzw. Benutzer, deren Gruppen sowie die Berechtigungen in das Gateway übernommen. Allerdings werden nur wenige und wirklich notwendige Attribute (z.B. ID der Benutzerin bzw. des Benutzers) gespeichert. Dafür wurde ein eigenes Datenbankmodell entworfen und wiederum mittels Hibernate gemappt. In diesen Tabellen werden die erforderlichen Informationen persistiert. Ist eine Benutzerin/ein Benutzer bereits erfolgreich eingeloggt – ein gültiger JWT ist also verfügbar – dann muss beispielsweise für die Abfrage der Gruppenzugehörigkeit, keine Anfrage an das externe Authentifizierungssystem gesendet werden, da diese Informationen im Gateway gespeichert sind.

Damit die Daten des externen Authentifizierungssystems übernommen werden können, wurde das Interface RBACRetrievalService (siehe Abbildung 4.11) eingeführt. Dieses Interface enthält eine einzelne Methode getSystemUsers(), welche eine Map mit den Benutzerinnen bzw. Benutzern (CacheUser) zurückliefert. Das Objekt CacheUser enthält zudem die zugeordneten Gruppen und deren Berechtigungen. Diese erhaltenen Daten werden dann in die Datenbank (RBAC-Schema) gespeichert bzw. bereits vorhandene Daten werden upgedatet. Ansonsten ist das Konzept gleich wie beim bereits vorgestellten AuthenticationProvider. Dementsprechend gibt es für jedes externe Authentifizierungssystem eine konkrete Implementierungsklasse, wo die tatsächliche Übernahme und Speicherung der Daten durchgeführt wird. Die Steuerung der aktiven Implementierungsklasse erfolgt wiederum über den Konfigurationsparameter rbac.system.service. Für den Prototypen wird derzeit nur die Einbindung von Comma-separated Values (CSV)-Dateien ermöglicht, in welchen die Benutzerinnen bzw. Benutzer deklariert werden. Dennoch können andere externe Authentifizierungssysteme, aufgrund des flexiblen Aufbaus basierend auf Interfaces, leicht eingebunden werden.



Abbildung 4.11: Klassendiagramm des Interfaces RBACRetrievalService

4.2.4 ProcessModelStorage

Die bisher vorgestellten Microservices sind für Aufgabenbereiche verantwortlich, die den Aufbau eines typischen Ökosystems von Microservices ermöglichen. Mit der Einführung des Microservices **ProcessModelStorage** wird der Fokus nun auf die Integration von S-BPM gerichtet. Die Hauptaufgabe dieses Services ist die Speicherung und Bereitstellung von subjekt-orientierten Prozessen. Demnach stellt das **ProcessModelStorage** Funktionen bereit, welche die Speicherung von Prozessmodellen ermöglichen. Diese Modelle müssen in einer vorgegebenen Art und Weise persistiert werden, damit das Microservice **ProcessEngine** in der Lage ist, diese Prozessmodelle in Prozessinstanzen zu überführen. Demgemäß sind Prozessinstanzen kein Bestandteil vom **ProcessModelStorage**.

In Abbildung 4.12 ist die interne Architektur des **ProcessModelStorage** dargestellt. Eine der Service-Komponenten ist der Parser, der externe Prozessmodelle, welche auf einem bestimmten Format, wie z.B. Web Ontology Language (OWL), basieren, parst. Die zweite Komponente Import ist für den Import und folglich der Speicherung der geparssten Prozessmodelle zuständig. Alle Datenbankobjekte befinden sich im Java-Package **Persistence** (siehe Abschnitt 3.3), welches im **ProcessModelStorage** importiert wurde.

Wie in Abbildung 4.1 ersichtlich ist, ist dieses Package auch Bestandteil der **ProcessEngine**. Dementsprechend nutzen zwei Microservices dasselbe Datenbankschema. Grundsätzlich sollte jedes Microservice über einen eigenen Datenspeicher verfügen, womit hier eine essentielle Anforderung an Microservice-basierte Architekturen nicht erfüllt wird. Es ist zwar dennoch eine Trennung gegeben, da nur das **ProcessModelStorage** schreibenden Zugriff auf alle notwendigen Tabellen zur Beschreibung eines Prozessmodells hat. Die **ProcessEngine** hat nur Leserechte für diese Prozessmodell-Tabellen, jedoch Schreibrechte für alle Tabellen der Prozessinstanz. Für den vorgestellten Prototypen ist dies im ersten Schritt auch definitiv ausreichend. Dennoch sollte dies auf lange Sicht geändert werden und folglich sollten beide Microservices über einen eigenen Datenspeicher verfügen. Insofern würde die **ProcessEngine** erst bei Start einer Prozessinstanz das notwendige Prozessmodell direkt vom **ProcessModelStorage** beziehen.

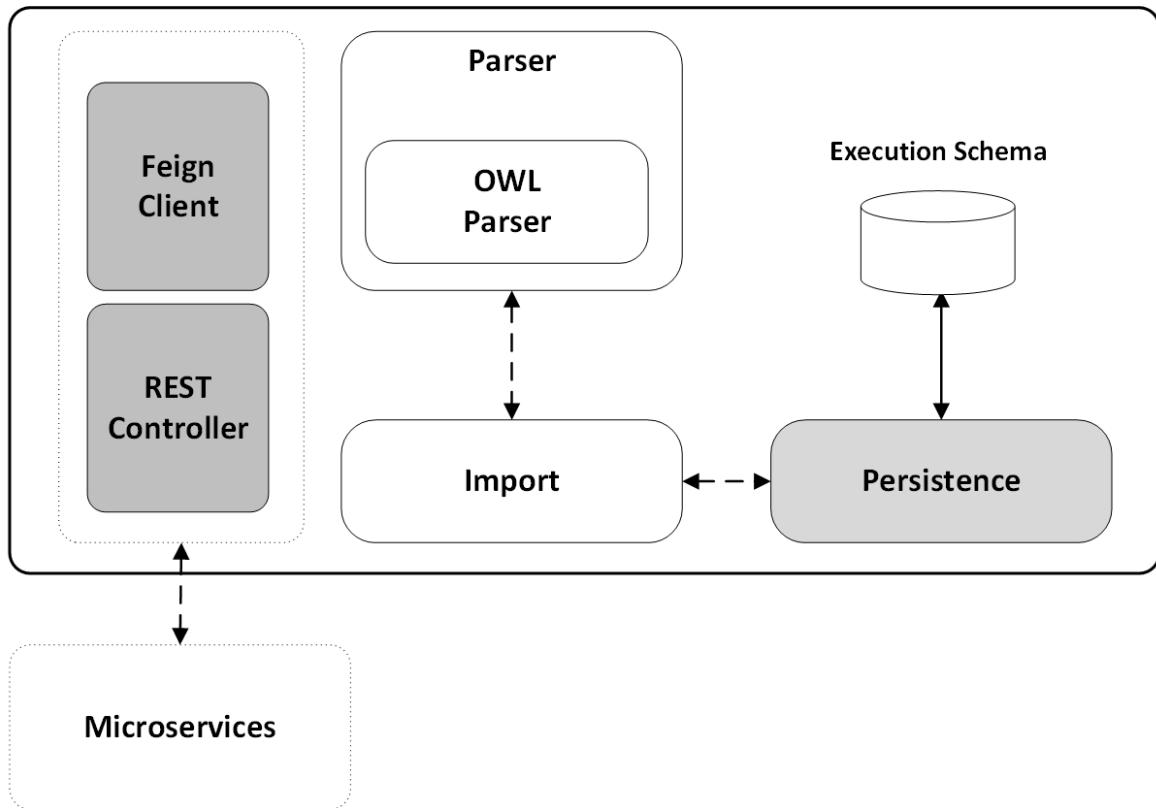


Abbildung 4.12: Architektur des Microservices ProcessModelStorage

Die Tabellen für das Datenbankschema werden automatisch von Hibernate erstellt. Dies wird nur durchgeführt, wenn die Konfigurationsparameter von Listing 4.4 definiert sind. Mittels Zeile 1 wird deklariert, dass die gemappten Datenbankobjekte als Tabellen im Datenbankschema angelegt werden. Zeile 2 stellt sicher, dass ein mögliches vorhandenes Schema nur upgedatet wird. D.h., dass geprüft wird, ob Änderungen der Datenbankobjekte getätigt worden sind und dementsprechend wird die Datenbank angepasst. Diese Konfigurationsparameter wurden in dieser Art und Weise nur im ProcessModelStorage definiert. Infolgedessen muss bei Änderungen der Datenbankobjekte, dieses Microservice neu gestartet werden, damit die Änderungen auch in das Datenbankschema übernommen werden.

Listing 4.4: Ausschnitt von application.properties des ProcessModelStorage

```

1 spring.jpa.generate-ddl=true
2 spring.jpa.hibernate.ddl-auto=update

```

4.2.4.1 Parser

Einer der wichtigsten Komponenten des ProcessModelStorage ist der Parser, dessen Aufgabe es ist, erstellte Prozessmodelle so zu parsen, dass diese importiert werden können. Dazu wurde ein Interface definiert, das die dafür notwendige Methode vorgibt. In Abbildung 4.13 ist dieses ersichtlich. Die Methode parseFile nutzt hierbei den generischen Typ `<I>` für den Eingabeparameter (`input`). Demgemäß wird sichergestellt, dass die Implementierungsklassen den Typ des Parameters selbst bestimmen können, womit dieser Ansatz flexibel anpassbar ist. Ein Objekt des Typs `ProcessModelDTO` wird als Resultat dieser Methode zurückgeliefert. Dieses enthält alle notwendigen Daten eines Prozessmodells.

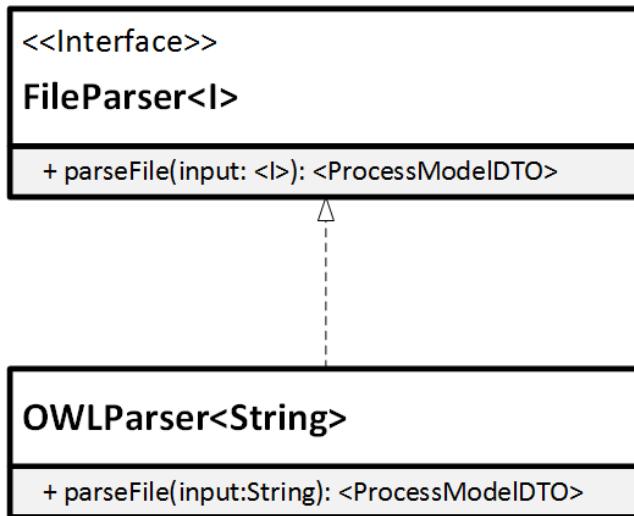


Abbildung 4.13: Klassendiagramme der Parser für Prozessmodelle

Zudem ist in Abbildung 4.13 die konkrete Implementierungsklasse für das Parsen von OWL-Dateien veranschaulicht. Als Datentyp des Eingabeparameters wird hierbei *String* definiert, da der Inhalt der OWL-Dateien als *String* bereitgestellt wird. Microsoft Visio kann beispielsweise verwendet werden, um subjekt-orientierte Prozesse zu modellieren. Diese können als OWL-Datei exportiert werden. Die Struktur der OWL-Dateien basiert auf standard-passont [15]. Nur Dateien, welche dieser Spezifikation folgen, können somit geparsst werden. Für das Parsen der OWL-Dateien wird Apache Jena³ – ein Java-basiertes Framework für Semantic Web – eingesetzt. Neue Parser, zur Unterstützung von anderen Dateiformaten und Strukturen, können leicht hinzugefügt werden, indem neue Implementierungsklassen, welche von **FileParser** ableiten, hinzugefügt werden.

4.2.4.2 Import

Damit neue Prozessmodelle eingefügt werden, sind zwei Schritte notwendig. In Abbildung 4.14 sind diese dargestellt. Das Microservice GUI stellt den Benutzerinnen bzw. Benutzern eine Schnittstelle zur Verfügung, womit diese die Prozessmodelle (gespeichert als OWL sowie gemäß den standard-passont-Spezifikationen) uploaden können. Diese Dateien werden an das **ProcessModelStorage** weitergeleitet. Dieses wählt nun den passenden **FileParser** aus und beginnt diese zu parsen. In den OWL-Dateien werden unter anderem die Subjekte, das interne Verhalten, etc. deklariert. Jedoch werden für den korrekten Import eines Prozessmodells noch zusätzliche Informationen benötigt. Infolgedessen wird nach dem Parsen dieses Resultat an die GUI zurückgesendet, wo die Benutzerinnen bzw. die Benutzer noch einige Einstellungen vornehmen müssen:

- Definition der Versionsnummer und optionale Beschreibung des Prozessmodells
- Deklaration der Felder eines Business Objects und die Angabe der Datentypen für diese
- Vergabe der Subjekt-Berechtigungen für die einzelnen Felder
- Zuordnung von Organisationselementen (in diesem Prototypen basierend auf Berechtigungen wie im Abschnitt 4.2.3.2 erläutert wurde) für die einzelnen Subjekte

³ <https://jena.apache.org/>

Nachdem die Benutzerinnen bzw. die Benutzer diese Einstellungen vorgenommen haben, wird dies wieder an das Service ProcessModelStorage gesendet. Nun wird der eigentliche Import des Prozessmodells gestartet und alle Komponenten werden in die Datenbank persistiert, womit das Prozessmodell nun für die ProcessEngine verfügbar ist. Für die Persistierung der Komponenten werden wiederum die bereitgestellten Builder des Java-Packages Persistence eingesetzt. Nach Abschluss des Imports wird noch zurückgemeldet, ob dieser erfolgreich war bzw. ob Fehler auftraten.

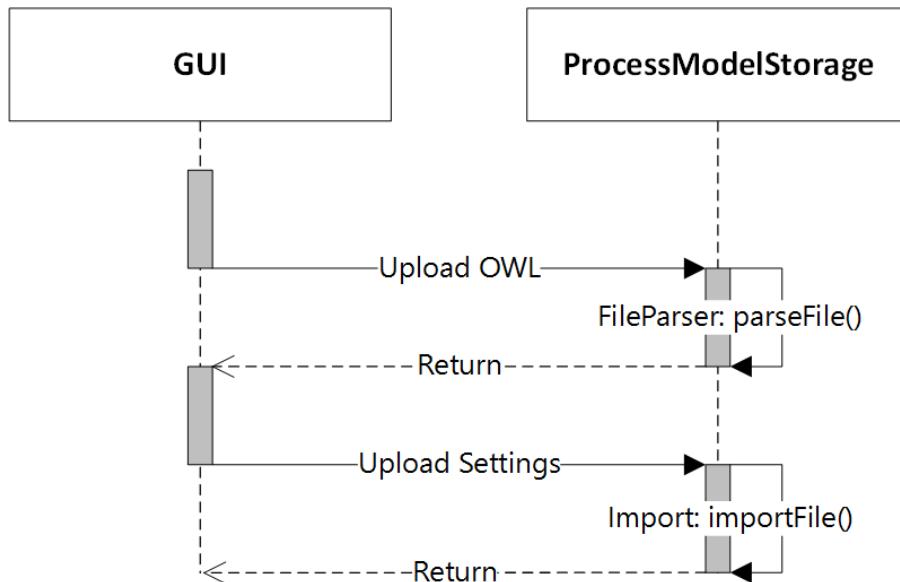


Abbildung 4.14: Sequenzdiagramm zur Veranschaulichung des Imports von Prozessen

4.2.5 ProcessEngine

Der Aufgabenbereich des Microservices ProcessEngine liegt in der Ausführung von subjekt-orientierten Prozessen. Dementsprechend werden die Prozessmodelle, welche mittels ProcessModelStorage importiert werden, in Prozessinstanzen überführt, womit diese für die Benutzerinnen bzw. Benutzer ausführbar sind. In Abbildung 4.15 wird die interne Architektur der ProcessEngine dargestellt. Grundsätzlich wird für die ExecutionEngine, die die Prozessinstanzen steuert, das Akka-Framework⁴ verwendet. Akka wird im Abschnitt 4.2.5.1 näher definiert. Zudem wird im Abschnitt 4.2.5.2 erläutert, wie Akka in der ProcessEngine eingesetzt wird. Ein weiterer Bestandteil der ProcessEngine ist das Java-Package Persistence, welches die Datenbankobjekte und deren Builder enthält. Für die Kommunikation mit anderen Services werden die Komponenten REST Controller bzw. Feign Client eingesetzt.

⁴ <http://akka.io/>

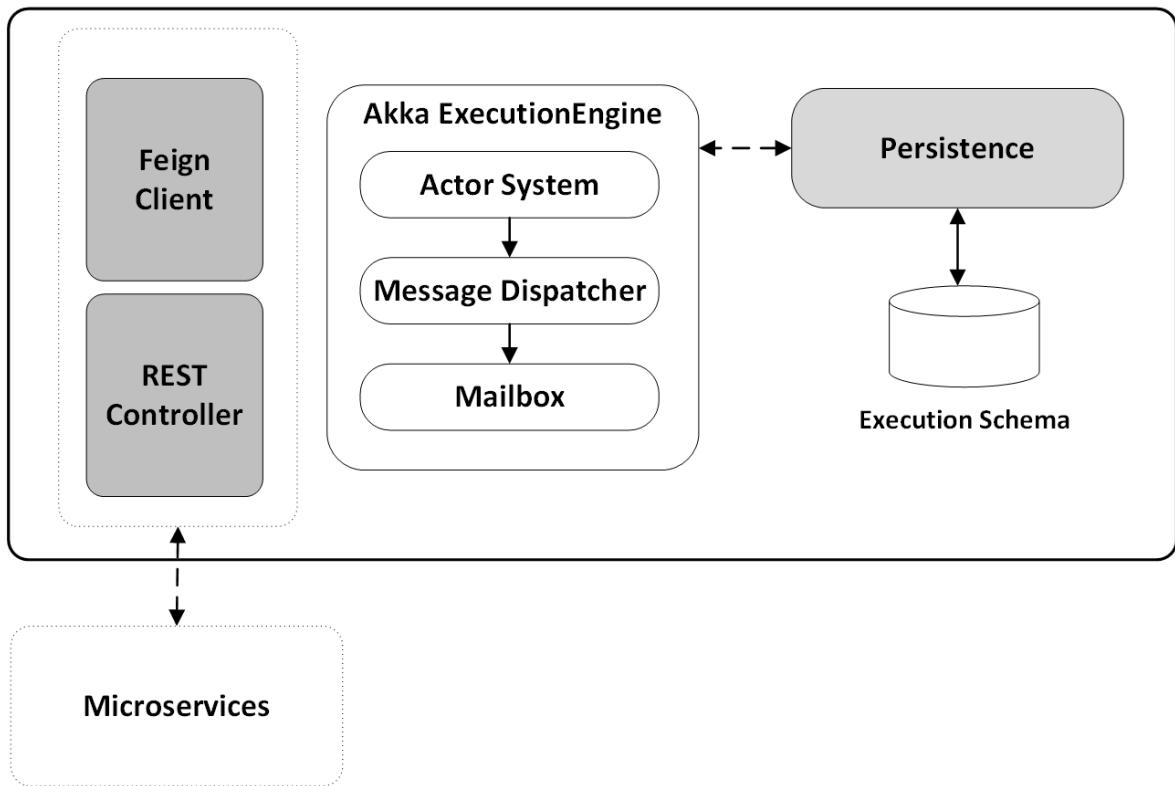


Abbildung 4.15: Architektur des Microservices ProcessEngine

4.2.5.1 Akka

Akka, welches für Java und Scala verfügbar ist, ermöglicht die Integration des Aktorenmodells. Mithilfe dieses Modells werden die Probleme, die bei der Verwendung von Threads oder Locks entstehen (z.B. deutlich komplexerer und fehleranfälligerer Code), vermieden. Im Aktorenmodell sind alle Entitäten als unabhängige Komponenten modelliert, die nur auf empfangene Nachrichten reagieren. Infolgedessen gibt es keinen gemeinsamen Zustand der Aktoren. In Abbildung 4.16 wird der Aufbau des Aktorenmodells veranschaulicht. Der interne Zustand von Aktoren kann nur geändert werden, wenn diese eine spezielle Nachricht empfangen. Des Weiteren agieren Aktoren asynchron [22].

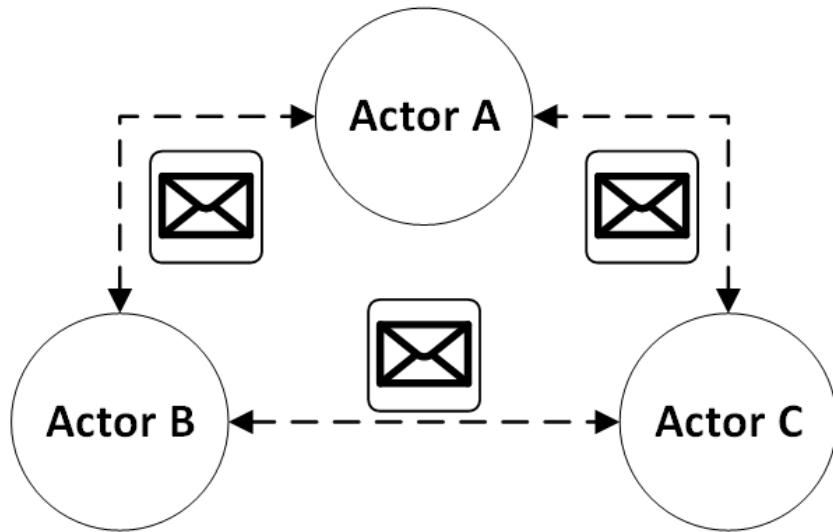


Abbildung 4.16: Aufbau des Aktorenmodells (modifiziert übernommen von [22])

Laut [22] basiert das Aktorenmodell auf folgenden Eigenschaften:

- Nachrichten, als Immutable Objects definiert, damit diese nicht veränderbar sind, werden zur Kommunikation zwischen Aktoren verwendet. Nur der Aktor selbst ist in der Lage den internen Zustand zu verändern, womit Aktoren den internen Zustand kontrollieren. Jegliche Informationen über den Zustand eines Aktors werden nur per Nachricht an andere Akteure veröffentlicht.
- Jeder Aktor besitzt einen eigenen Pool, in welchem alle eingehenden Nachrichten gespeichert werden. Die Nachrichten werden dem Pool entnommen und vom Aktor entsprechend verarbeitet (z.B. Änderung des internen Zustands, Weiterleitung der Nachricht an andere Akteure, Instanziierung von neuen Akten, etc.).
- Nachrichten zwischen Akten werden asynchron ausgetauscht. D.h., dass Sender nicht warten, bis die Nachricht vom Empfänger entgegengenommen wird. Folglich setzt der Sender seine Ausführungslogik sofort nach dem Senden fort und wird nicht blockiert.
- Die Kommunikation zwischen Sender und Empfänger ist entkoppelt und asynchron. Infolgedessen wird jeder Aktor üblicherweise in einem eigenen Thread ausgeführt. Aufgrund der Verwendung von separaten Threads und der Tatsache, dass Akten keine gemeinsamen Zustände nutzen, wird mit dem Aktorenmodell ein verteiltes und skalierbares System bereitgestellt.

Grundsätzlich basiert der Aufbau einer Akka-Anwendung auf Akten. Dementsprechend stellt Akka Komponenten zur Verfügung, um das Aktorenmodell umzusetzen. Jeder Aktor hat einen privaten Zustand, der nicht öffentlich zugänglich ist und bestimmt somit selbstständig das interne Verhalten. Beim Start einer Akka-Anwendung wird zuerst das Actor System initialisiert. Diesem werden die sogenannten Top-Level-Akten zugeordnet, welche weitere Akten erstellen können. Somit wird in Akka eine Hierarchie von Akten aufgebaut. Die Nachrichten selbst werden allerdings nie direkt an die Akten gesendet, da das Actor System eine ActorRef zurückliefert, an welche die Nachrichten gesendet werden. Folglich stellt die ActorRef einen Proxy für die jeweiligen Akten bereit. Des Weiteren besitzt jeder Akka-Aktor einen Namen, der in der zugeordneten Hierarchieebene eindeutig sein muss [35].

Jedem Aktor ist eine Mailbox zugeordnet, in welcher die empfangenen Nachrichten temporär gespeichert werden. Dieses Prinzip ist in Abbildung 4.17 veranschaulicht. Nachrichten werden

nie an den Aktor selbst, sondern an die ActorRef gesendet, welche die Nachrichten in der Mailbox des Aktors speichert. Nachdem der Aktor eine Nachricht zur Verarbeitung entgegennimmt, wird diese aus der Mailbox entfernt [35]. Für die Mailbox von Akka sind verschiedenste Implementierungen verfügbar. Zum einen besteht die Möglichkeit ein Limit an Nachrichten für die jeweilige Mailbox festzulegen. Zum anderen können Nachrichten priorisiert werden. Folglich werden Nachrichten mit höherer Priorität gegenüber Nachrichten niedrigerer Priorität bevorzugt und zuerst vom Aktor verarbeitet [22].

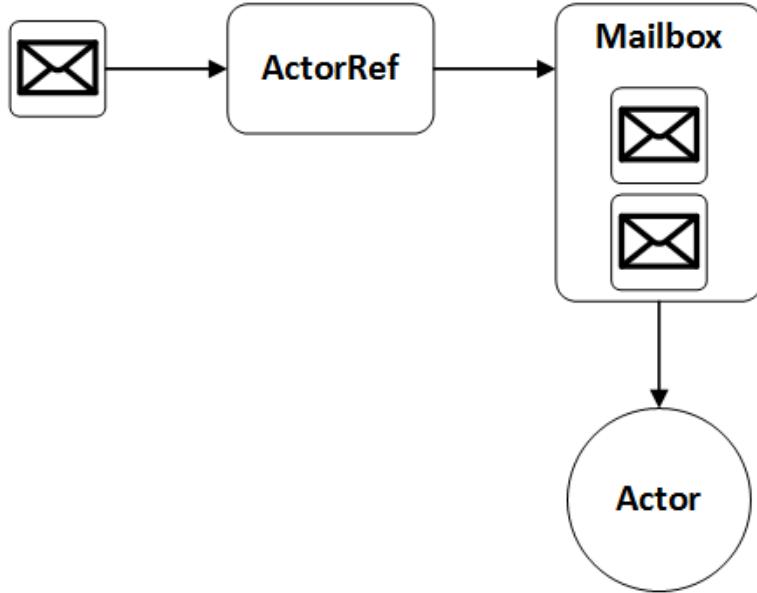


Abbildung 4.17: Verwendung der Mailbox in Akka (modifiziert übernommen von [35])

Eine weitere Komponente der Akka ExecutionEngine ist der Message Dispatcher. Der Message Dispatcher kontrolliert und koordiniert die Zuordnung der Aktoren (inklusive deren Mailbox) an die untergeordneten Threads. Mithilfe dieser Threads werden die Nachrichten von den Aktoren verarbeitet. Dementsprechend startet die Verarbeitung der Nachrichten erst, nachdem dem Aktor ein Thread vom Message Dispatcher zugewiesen wurde. Diese Funktionsweise wird in Abbildung 4.18 dargestellt. Somit stellt der Message Dispatcher sicher, dass die verfügbaren Ressourcen optimiert werden und die Verarbeitung der Nachrichten schnellstmöglich erfolgt. Der Message Dispatcher wird hierbei in eigenem Thread ausgeführt, wohingegen die Threads, die zur Nachrichtenverarbeitung dienen, aus einem separaten Thread-Pool stammen. Akka bietet verschiedene Strategien für den Message Dispatcher an, die allerdings nicht Bestandteil dieser Arbeit sind [22]. Für den Prototypen wurden die Einstellungen für den Message Dispatcher sowie für die Mailbox mit den Standardwerten von Akka belassen.

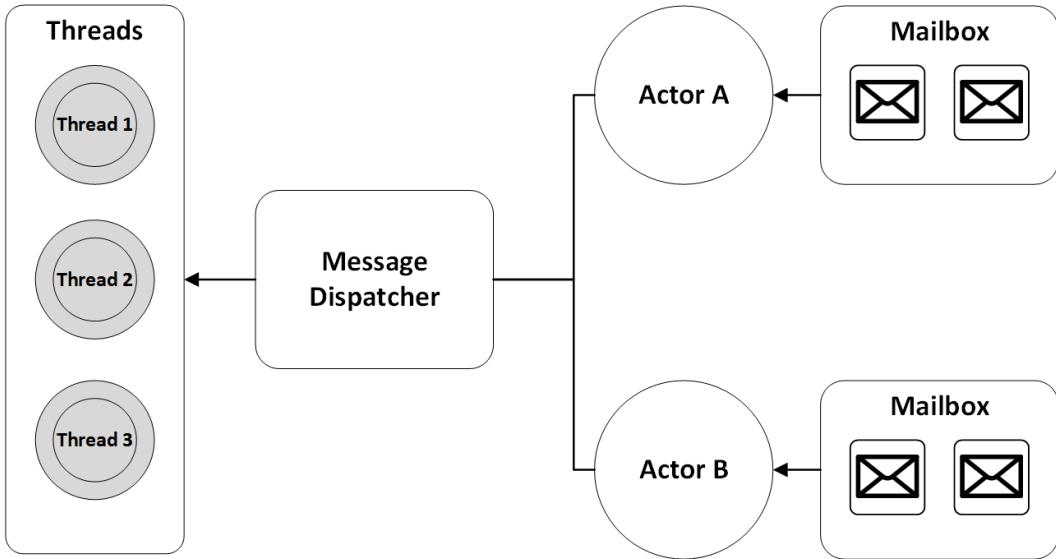


Abbildung 4.18: Funktionsweise der Komponente Message Dispatcher in Akka (modifiziert übernommen von [22])

4.2.5.2 Verwendung von Akka in der ProcessEngine

In Abbildung 4.19 werden die verwendeten Aktoren der ProcessEngine veranschaulicht. Mithilfe dieser Aktoren und deren Austausch von Nachrichten untereinander wird die Ausführung der subjekt-orientierten Prozesse ermöglicht. Der Aufbau wurde so gewählt, dass die Einhaltung des Single Responsibility-Prinzips sichergestellt wird. Dieses Prinzip – das auch bei Microservicearchitekturen angewandt wird – legt fest, dass die jeweiligen Bestandteile der Software nur für eine Aufgabe verantwortlich sind. Folglich führt die Anwendung dieses Prinzips dazu, dass die Aktoren nur eine Aufgabe erfüllen, womit die einzelnen Klassen kürzer und verständlicher sind. Des Weiteren wird das Supervisor-Prinzip in diesem Prototypen genutzt. Dementsprechend werden die Aktoren in Hierarchien (z.B. UserSupervisorActor mit dem untergeordneten UserActor) gegliedert. Grundsätzlich bestimmen Supervisor den Ablauf, wenn einzelne untergeordnete Aktoren abstürzen. Beispielsweise kann ein abgestürzter Aktor vom übergeordneten Supervisor neu gestartet werden [5].

Die folgenden Aktoren sind Bestandteil der ProcessEngine:

- **ProcessSupervisorActor:** Diese Aktor ist verantwortlich für den Start und das Beenden von Prozessinstanzen. Für jede neue Prozessinstanz instanziert dieser Aktor einen neuen ProcessActor. Weiters stellt der ProcessSupervisorActor sicher, dass die aktiven Prozessinstanzen bei Neustart der ProcessEngine wieder hochfahren werden. Nachrichten an einen konkreten ProcessActor werden vom ProcessSupervisorActor an diesen weitergeleitet.
- **ProcessActor:** Informationen zum aktuellen Status der Prozessinstanz werden vom ProcessActor zur Verfügung gestellt. D.h. der Aktor gibt an, ob ein Prozess gestartet, beendet oder abgebrochen wurde. Wie bereits erwähnt, ist eine eindeutige Namensgebung für Aktoren derselben Hierarchieebene notwendig. Demgemäß setzt sich der Name aus „ProcessActor-“ und der eindeutigen Datenbank-ID der Prozessinstanz (`piId`) zusammen.
- **UserSupervisorActor:** Für Benutzerinnen bzw. Benutzer, welche Bestandteil einer Prozessinstanz sind, legt der UserSupervisorActor einen UserActor an. Zudem wer-

den jene Nachrichten, die für UserActor bestimmt sind, vom UserSupervisorActor an diese weitergeleitet. Ebenso stellt dieser Aktor sicher, dass die Benutzerinnen bzw. Benutzer nach Neustart der ProcessEngine wieder hochfahren werden.

- **UserActor**: Der Name des Aktors wird aus „UserActor-“ und der userId (aus der Tabelle User vom Microservice Gateway entnommen) zusammengesetzt. Somit verfügen Benutzerinnen bzw. Benutzer, welche Subjekt einer Prozessinstanz sind, über einen solchen UserActor. Aufgrund der Namenskonvention für diesen Aktor wird sichergestellt, dass es pro Benutzerin/Benutzer immer nur einen UserActor gibt. D.h., dass auch wenn Benutzerinnen/Benutzer als Subjekte in mehreren Prozessinstanzen agieren, nur ein UserActor für diese existiert. Dieser Aktor bestimmt, welche Schritte (z.B. Bestätigung oder Ablehnung eines Urlaubsantrages) in einer Prozessinstanz für die Benutzerin bzw. dem Benutzer als nächstes auszuführen sind. Dies basiert auf dem sogenannten StateObject, das im Abschnitt 4.2.5.4 näher erläutert wird. Ebenso ist das StateObject essentiell für die internen Statusübergänge in einer Prozessinstanz.
- **TaskActor**: Für die Akteure ProcessSupervisorActor, ProcessActor, UserSupervisorActor sowie UserActor wurde die Möglichkeit geschaffen, Tasks an den TaskActor auszulagern. Folglich können komplexe Operationen (z.B. Statusübergänge) direkt im TaskActor ausgeführt werden. Die aufrufenden Akteure werden somit in ihrer Ausführung nicht blockiert, da der TaskActor die übergebenen Tasks autonom ausführt. Der TaskActor basiert auf dem TaskManager, der im Abschnitt 4.2.5.3 dargestellt wird.
- **TimeoutSchedulerActor**: Ist für einen State ein Timeout festgelegt, dann wird automatisch ein TimeoutSchedulerActor gestartet. In diesem Aktor wird der in Akka verfügbare Scheduler⁵ ausgeführt. Diesem Scheduler wird die Zeit bis zum Eintritt des Timeouts als Parameter übergeben. Nach Ablauf dieser übergebenen Zeit, wird eine Nachricht an den betroffenen UserActor gesendet, der das Timeout anschließend behandelt und einen Statusübergang auslöst.
- **AnalysisActor**: Grundsätzlich führt der AnalysisActor SQL-Abfragen aus, um aktuelle Kennzahlen (z.B. Anzahl der aktiven Prozesse, Anzahl der beendeten Prozesse, etc.) bereitzustellen.

Das Microservice ProcessEngine basiert auf Akka, weil einzelne Konzepte von S-BPM und Akka ähnlich sind. Beispielsweise basieren S-BPM und auch Akka auf den Austausch von Nachrichten. Weiters kann das Input-Pool-Konzept von S-BPM mit dem Mailbox-Konzept von Akka verglichen werden. Beide Konzepte dienen als Nachrichteneingang und als temporärer Nachrichtenpuffer für das Subjekt bzw. den Aktor. Zudem weisen Microservices und Akka Analogien in den Merkmalen auf. Zum einen sind Microservices und Akka auf den Aufbau von skalierbaren System ausgerichtet. Zum anderen basieren beide auf Events (in Akka als Nachrichten bezeichnet), welche die asynchrone Kommunikation ermöglichen. Insofern lässt sich ein Akteurenmodell, wie Akka, hervorragend in Microservicearchitekturen integrieren. Vor allem für das Microservice ProcessEngine ist die Skalierbarkeit eine wichtige Anforderung, da dieses in der Lage sein muss, mit einer beliebig großen Anzahl von Prozessen umzugehen. Aus diesem Grund basiert das Microservice ProcessEngine des vorgestellten Prototypen auf Akka.

⁵ <http://doc.akka.io/docs/akka/current/java/scheduler.html>

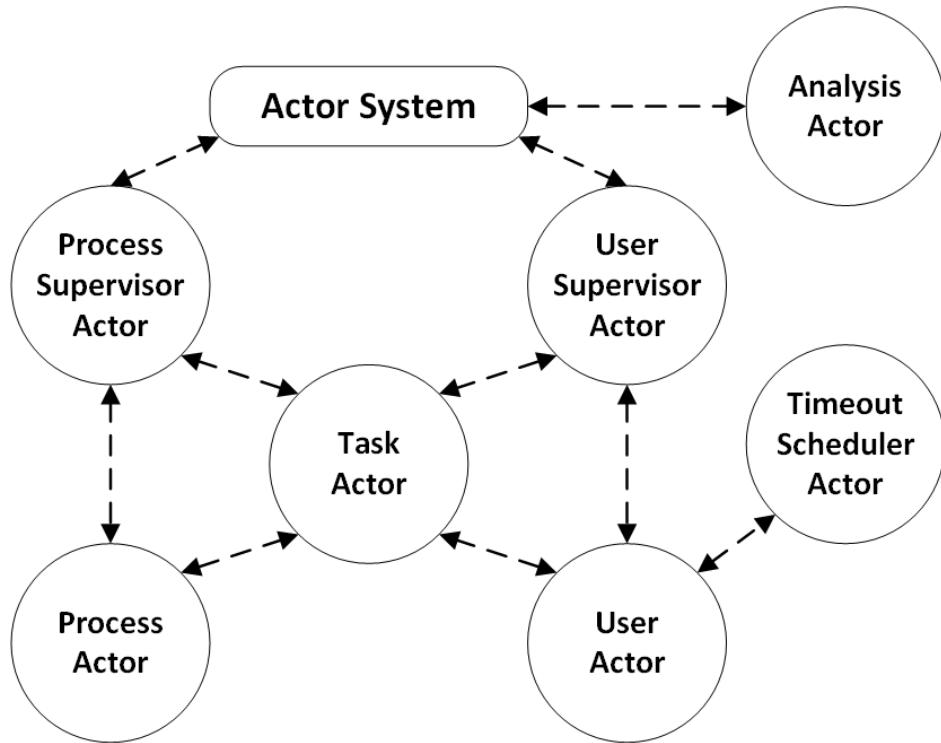


Abbildung 4.19: Verwendete Aktoren in der ProcessEngine

4.2.5.3 TaskManager

Der TaskManager ist jene Komponente, die die Ausführung von Tasks ermöglicht. Tasks sind komplexe Operationen, welche im Hintergrund ausgeführt werden, damit die Ausführungslogik nicht blockiert wird. Aufgrund der Auslagerung von Operationen wird das Single Responsibility-Prinzip umgesetzt, da jeder Task genau eine Aufgabe erfüllt. In Abbildung 4.20 ist das Klassendiagramm der Klasse TaskManager veranschaulicht. Grundsätzlich werden zwei verschiedene Methoden für den Start von Tasks bereitgestellt. Für beide Methoden ist der Parameter `task` vom Typ `TaskAllocation` notwendig. Beim Typ `TaskAllocation` handelt es sich um eine Enumeration, in der alle Tasks mit einem eindeutigen Namen registriert werden müssen. Dementsprechend gibt der Parameter `task` jenen Task an, der ausgeführt werden soll. Der zweite Parameter `message` vom Typ `Object` stellt die Nachricht dar, welche im Task verarbeitet wird. Optional kann ein generischer TaskCallback als Parameter übergeben werden. Somit wird der Aufrufer notifiziert, wenn die Ausführung des Tasks abgeschlossen wurde. Zudem enthält die Notifikation von TaskCallback ein Resultat des Tasks. Da der Aufrufer den Typ des Resultats selbstständig festlegen kann, ist der generische Typ `<T>` notwendig.

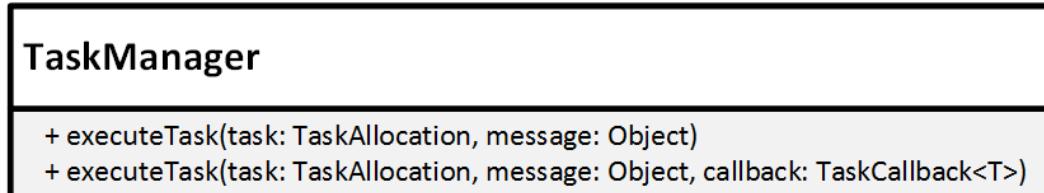


Abbildung 4.20: Klassendiagramm der Klasse TaskManager

Für die Erstellung eines Tasks muss dieser definierte Interfaces implementieren, damit der TaskManager in der Lage ist, diesen Task zu starten. Diese Hierarchie ist in Abbildung 4.21 dargestellt. Das Top-Level-Objekt ist das Interface Task, welches den generischen Typ <I> nutzt, um den Typ der Nachricht, der zur Ausführung erwartet wird, zu bestimmen. Grundsätzlich müssen drei Methoden zur Erstellung eines ausführbaren Tasks implementiert werden. Die Methode canHandle wird vor der Ausführung eines Tasks aufgerufen, um zu prüfen, ob die empfangene Nachricht verarbeitet werden kann. Beispielsweise kann überprüft werden, ob der Parameter einer Nachricht den notwendigen Wert aufweist. In der Methode execute findet die tatsächliche Ausführung des Tasks statt. D.h. mithilfe dieser Methode wird die Geschäftslogik eines Tasks definiert. Die Methode registerCallback ermöglicht die Deklaration einer Notifikation bei Beendigung des Tasks.

Die abstrakte Klasse AbstractTask wird vom Interface Task abgeleitet. Konkrete Tasks (z.B. zur Ausführung eines Statusübergangs) müssen von der Klasse AbstractTask abgeleitet werden. Diese Klasse stellt Konstruktoren zur Initialisierung und diverse Hilfsmethoden für die Tasks bereit. Die Methoden callback können in den Tasks aufgerufen werden, um eine Notifikation an den Aufrufer zu senden. Bei der Klasse AbstractTask handelt es sich um einen Aktor. Infolgedessen werden Tasks mithilfe von Akka und somit als autonomer Thread verwaltet und ausgeführt, womit eine optimale Auslastung der verfügbaren Ressourcen gegeben ist.

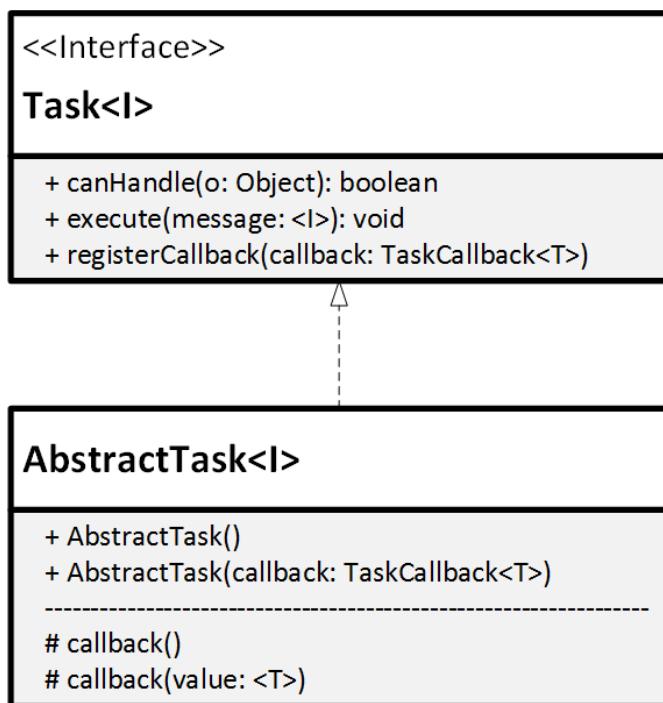


Abbildung 4.21: Objekthierarchie eines Tasks

4.2.5.4 StateObject

Das StateObject bestimmt die nächsten Schritte einer Benutzerin bzw. eines Benutzers in einer Prozessinstanz. Insofern sind folgende Komponenten Bestandteil dieses Objekts:

- Wie im Abschnitt 3.1 bereits erläutert wurde, können einem State beliebig viele Business Objects zugeordnet werden. Das StateObject liefert die zugeordneten Business Objects und deren Felder (inklusive Namen, Datentyp, Berechtigung, Standardwert

des Feldes, etc.) zurück. Wenn bereits eine konkrete Instanz eines Feldes des Business Objects existiert, dann ist dieser Wert ebenso Bestandteil des Objekts. Folglich müssen diese Informationen im Microservice GUI verarbeitet und korrekt angezeigt werden, damit der Benutzerin bzw. dem Benutzer die notwendigen Eingabemasken angezeigt werden.

- Zudem enthält das StateObject die möglichen nächsten States, welche den Benutzerinnen bzw. Benutzern angezeigt werden. Diese können anschließend einen State wählen und bestimmen somit die Fortsetzung des Prozesses.
- Handelt es sich beim derzeitigen State um einen Send-State, dann muss dem Subjekt, das die Nachricht empfangen soll, einer Benutzerin bzw. einem Benutzer (User) zugewiesen werden. Ist eine solche Zuweisung für ein Subjekt notwendig, dann ist diese Information Bestandteil des StateObjects.

Für das StateObject werden sogenannte Composer verwendet, damit diese die Werte der Felder der Business Objects in ein Format transformieren, das im Microservice GUI korrekt dargestellt wird. Hierbei müssen die verschiedenen möglichen Datentypen unterstützt werden. Die Werte der Felder – persistiert in der Tabelle BusinessObjectFieldInstance (siehe Abschnitt 3.2) – werden in der Datenbank als String gespeichert. Für die Transformierung der Datenbankwerte müssen zwei Interfaces, welche in Abbildung 4.22 veranschaulicht sind, implementiert werden. Das Interface DbComposer muss implementiert werden, um Datenbankwerte in Java-Objekte zu transformieren. Mithilfe des generischen Typs <O> wird hierbei der Typ des Java-Objekts festgelegt. Die Methode canCompose prüft, ob eine erfolgreiche Transformation möglich ist. Mittels der Methode compose wird die Transformation durchgeführt. Nun stehen der ProcessEngine die Werte der Felder als Java-Objekte zur Verfügung und beliebige Operationen können damit durchgeführt werden. Im nächsten Schritt müssen die Java-Objekte in JSON-Objekte transformiert werden, da die Kommunikation zwischen den verschiedenen Microservices auf JSON basiert. Hierfür wird das Interface JsonComposer bereitgestellt. Die Methode compose ist die einzige Methode des Interfaces. Der generische Typ <I> wird zur Festlegung des Java-Objekt-Typs verwendet und der Rückgabetyp ist mit String definiert.

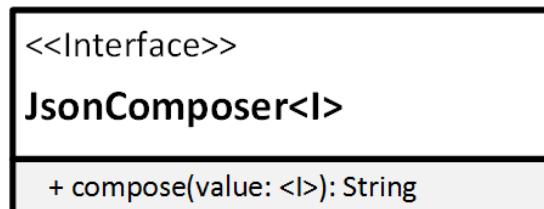
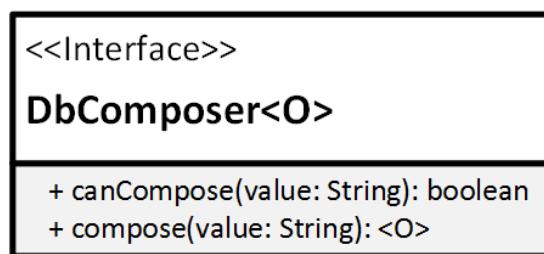


Abbildung 4.22: Composer der ProcessEngine

Ein modifiziertes StateObject wird für den internen Statusübergang eines Subjekts in einer Prozessinstanz verwendet. Demgemäß wird das Attribut currentState der Tabelle

SubjectState (siehe Abschnitt 3.2) angepasst. Folgende Komponenten sind Bestandteil dieses Objekts:

- Eine Komponente dieses Objekts gibt den nächsten State eines Subjekts im Prozess an.
- Die Felder (inklusive deren Werte) der Business Objects werden mithilfe des modifizierten StateObjects vom Microservice ProcessEngine verarbeitet. D.h. es werden Einträge in den Tabellen BusinessObjectInstance bzw. BusinessObjectFieldInstance – wenn diese noch nicht vorhanden sind; ansonsten werden die bereits vorhandenen Einträge verwendet – angelegt, welche die tatsächlichen Werte der Felder der Business Objects enthalten.
- Die dritte Komponente des Objekts beinhaltet die Benutzerinnen- bzw. Benutzerzuweisung eines Subjekts. Dies ist allerdings nur der Fall, wenn es sich beim derzeitigen State um einen Send-State handelt und die Zuweisung noch nicht erfolgt ist. Dazu wird das Attribut userId der Tabelle Subject verwendet.

Für die Transformierung der von der Benutzerin/dem Benutzer eingegebenen Werten, werden Parser eingesetzt. Die Interfaces, die das ermöglichen, werden in Abbildung 4.23 dargestellt. Das Interface JsonParser wird verwendet, um die JSON-Objekte in Java-Objekte zu transformieren. Für die Deklaration des Java-Objekt-Typs wird der generische Typ <O> bereitgestellt. Das Interface enthält zwei Methoden. Zur Prüfung, ob eine Transformation möglich ist, wird die Methode canParse verwendet. Die Transformation wird in der Methode parse durchgeführt. Nach dieser Transformation können die Werte der Felder als Java-Objekte in der ProcessEngine verwendet werden. Wie bereits erwähnt, werden die Werte der Felder als String in der Datenbank gespeichert. D.h. die Java-Objekte müssen transformiert werden, damit diese in der Datenbank gespeichert werden können. Dafür muss das Interface DbParser implementiert werden. Der generische Typ <I> bestimmt hierbei den Typ des Java-Objekts. Die Methode parse führt die Transformierung durch und liefert demgemäß einen String zurück.

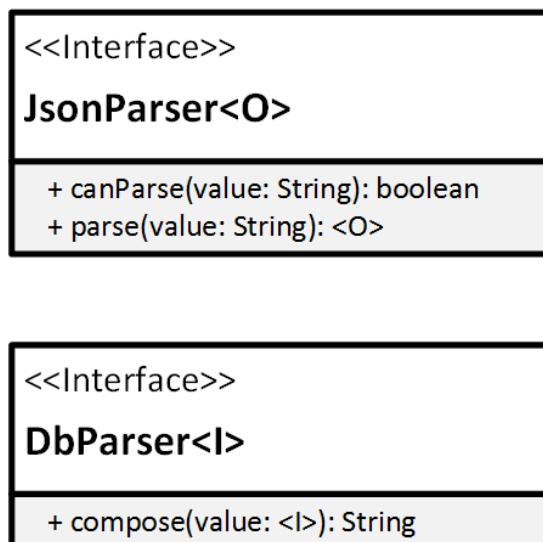


Abbildung 4.23: Parser der ProcessEngine

Wie oben angeführt, wird das StateObject genutzt, um den internen Zustand eines Subjekts zu modifizieren. Handelt es sich beim nächsten State um einen speziellen Refinement-State, dann wird von der ProcessEngine automatisch erkannt, dass hier keine Benutzerinnen- bzw.

Benutzerinteraktion notwendig ist (siehe Abschnitt 2.1.3.3). Insofern ist die ProcessEngine in der Lage hinterlegten Programmcode automatisch zu laden und auszuführen. Die Deklaration der auszuführenden Java-Klasse erfolgt mithilfe des Attributs refinementClass der Tabelle State. Vor der Ausführung des Refinements wird dieses Attribut von der ProcessEngine eingelesen und die angegebene Java-Klasse wird mittels Reflection⁶ geladen. Die angegebene Java-Klasse muss hierbei das Interface Refinement, das in Abbildung 4.24 dargestellt ist, implementieren. Ansonsten ist die ProcessEngine nicht in der Lage, das Refinement auszuführen. Die Methode execute enthält die eigentliche Geschäftslogik des Refinements. Als Parameter wird eine Map der Typen String und BusinessObject übergeben. Folglich ist es möglich, auf die zugeordneten Business Objects sowie deren Felder zuzugreifen. Der Rückgabewert von execute ist ebenso eine Map, in der die modifizierten Business Objects gespeichert werden. Diese Business Objects werden anschließend dauerhaftpersistiert. Die zweite Methode getNextStateId wird zur Definition des nachfolgenden States eingesetzt. D.h. nach erfolgreicher Ausführung des Refinements wird dieser angegebene State für das Subjekt festgelegt. Somit ist eine dynamische Festlegung des nachfolgenden States möglich.

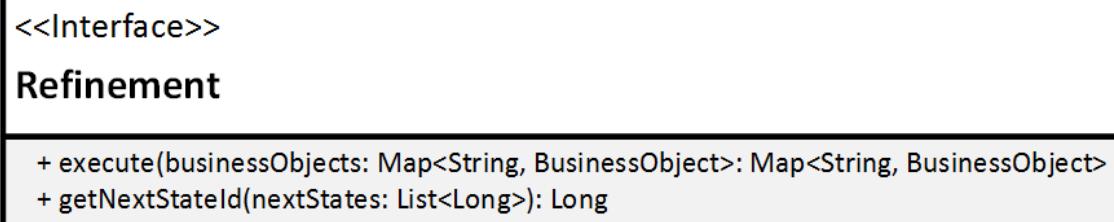


Abbildung 4.24: Interface zur Implementierung eines Refinements

4.2.6 ExternalCommunicator

Das Microservice ExternalCommunicator ermöglicht die Integration von externen Subjekten. Externe Subjekte sind Subjekte, deren internes Verhalten nicht bekannt ist und wurden bereits im Abschnitt 2.1.3.2 erläutert. In der Datenbank wird dem Attribut subjectType der Tabelle SubjectModel der Wert EXTERNAL zugewiesen, damit ein Subjektmodell als externes Subjekt gekennzeichnet wird. Infolgedessen gibt es für dieses Subjektmodell keine Einträge in der Tabelle State, weil das interne Verhalten unbekannt ist und somit nicht modelliert wird. Grundsätzlich wird mit dem ExternalCommunicator die Integration von externen Subjekten geschaffen, indem Anfragen an diese gesendet bzw. Anfragen von diesen empfangen und verarbeitet werden.

In Abbildung 4.25 ist die Architektur des Microservices ExternalCommunicator veranschaulicht. Für die Definition des Nachrichtenaufbaus, welche vom ExternalCommunicator gesendet bzw. empfangen werden können, werden Konfigurationen für die Nachrichten (InboundConfiguration und OutboundConfiguration) verwendet. Der genaue Aufbau dieses Ansatzes wird im Abschnitt 4.2.6.1 näher erläutert. Zudem werden sogenannte Plugins (ReceivePlugin bzw. SendPlugin) eingesetzt, die Funktionen zum Empfangen bzw. Senden von Nachrichten bereitstellen. Der Plugin-Mechanismus wird im Abschnitt 4.2.6.2 definiert. Des Weiteren wird ein Execution Framework, basierend auf Akka, bereitgestellt, das die Konfigurationen und Plugins verwendet und folglich das eigentliche Senden bzw. Empfangen von Nachrichten ermöglicht. Dieses ist Bestandteil des Abschnitts 4.2.6.3.

⁶ Reflection ermöglicht das Laden einer Java-Klasse während der Laufzeit einer Anwendung [30].

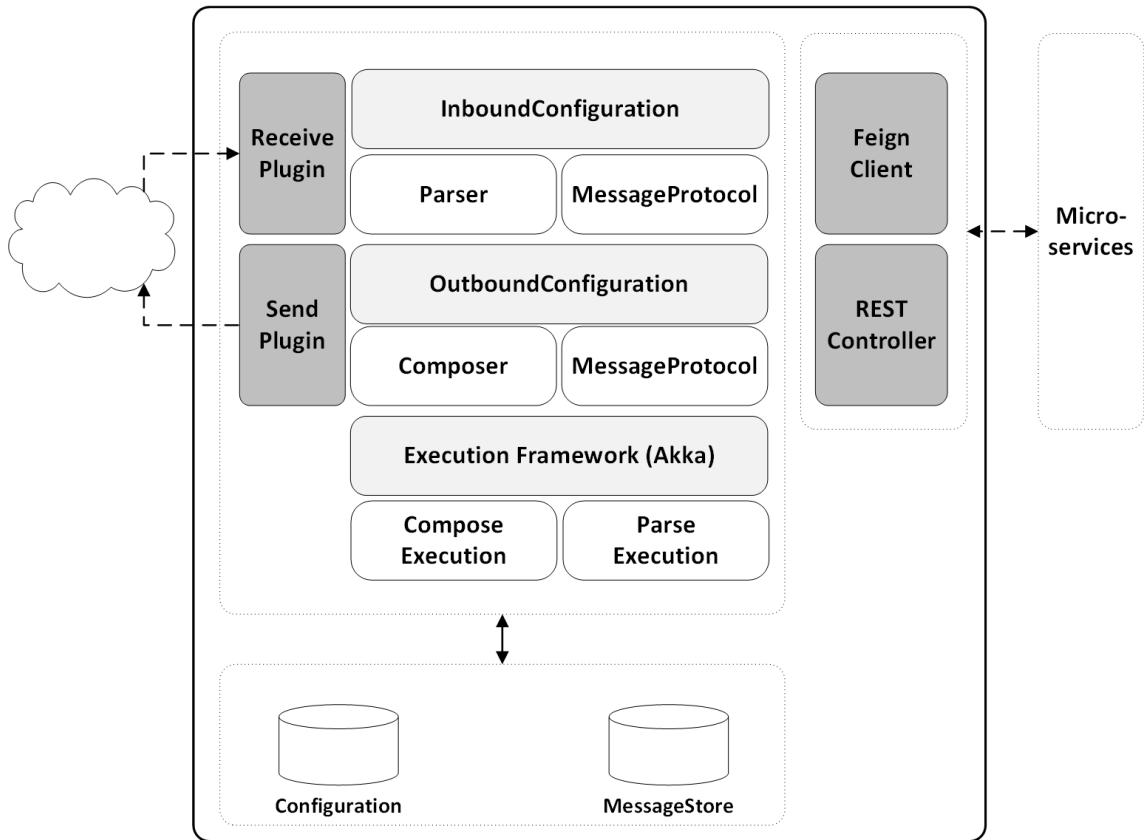


Abbildung 4.25: Architektur des Microservices ExternalCommunicator

4.2.6.1 Konfigurationen

Dem ExternalCommunicator muss der Aufbau der Nachrichten zur korrekten Verarbeitung bekannt sein. Die Komponente OutboundConfiguration wird zur Definition von ausgehenden Nachrichten bereitgestellt. Um den Aufbau von Nachrichten, die vom ExternalCommunicator empfangen werden, zu deklarieren, wird die Komponente InboundConfiguration eingesetzt. Alle Konfigurationen werden hierbei im Datenbankschema Configuration abgelegt. Dementsprechend muss für jedes externe Subjekt, das Nachrichten an ein externes IT-System sendet, ein ConfigurationAssignment angelegt werden. Mit diesem erfolgt die Verknüpfung der Konfiguration mit einem MessageFlow (siehe Abschnitt 3.1). Für externe Subjekte mit hinterlegtem Prozessmodell, welche ebenso in der ProcessEngine ausgeführt werden, ist dies nicht in dieser Art und Weise notwendig, da Standardkonfigurationen verwendet werden. Nur das hinterlegte Prozessmodell des externen Subjekts muss deklariert werden.

In den Tabellen 4.1 und 4.2 werden die Attribute, die zur Konfiguration einer Nachricht im ExternalCommunicator notwendig sind, beschrieben. Der grundsätzliche Aufbau von OutboundConfiguration und InboundConfiguration ist ident. Für die OutboundConfiguration werden Composer (siehe Abbildung 4.26) verwendet, damit Nachrichten beispielsweise in Extensible Markup Language (XML) oder JSON transformiert werden. Weiters besteht die Möglichkeit DataTypeComposer festzulegen, um anzugeben, wie die einzelnen Felder einer Nachricht transformiert werden. Der Datentyp eines Feldes bestimmt hierbei welcher DataTypeComposer verwendet wird. Für die Komponente InboundConfiguration wird ebenso dieses Prinzip angewandt, allerdings werden Parser (siehe Abbildung 4.27) und DataTypeParser für die Transformierung der Nachrichten und

deren Felder eingesetzt.

Sowohl OutboundConfiguration als auch InboundConfiguration nutzen die Komponente MessageProtocol (siehe Tabelle 4.3), welche den Aufbau einer Nachricht definiert. Jede Nachricht kann beliebig viele Felder (siehe Tabelle 4.4) enthalten. Das MessageProtocol dient als Basis für die Transformierung einer externen Nachricht in ein Business Object bzw. eines Business Objects in eine externe Nachricht, da es den genauen Aufbau vorgibt. Infolgedessen bezieht sich das Attribut internalName, das Bestandteil von MessageProtocol und MessageProtocolField ist, auf das Business Object und deren Felder. Das Attribut externalName deklariert wie Business Objects und deren Felder in den externen Nachrichten bezeichnet werden.

Attributname	Beschreibung
id	Primärschlüssel dieser Tabelle
name	Definiert den Namen der Konfiguration
messageProtocol	Deklariert das eingesetzte MessageProtocol
composerClass	Definiert die Klasse, die verwendet wird, um die Nachricht zu transformieren
sendPlugin	Die Nachricht wird mit dem angegebenen Plugin versendet.
dataTypeComposer	Die einzelnen Felder – basierend auf deren Datentypen – werden damit transformiert.
configuration	Liste, die die Speicherung von zusätzlichen Konfigurationsparametern (z.B. URI) ermöglicht

Tabelle 4.1: Beschreibung der Attribute der Tabelle OutboundConfiguration

Attributname	Beschreibung
id	Primärschlüssel dieser Tabelle
name	Definiert den Namen der Konfiguration
messageProtocol	Deklariert das eingesetzte MessageProtocol
parserClass	Definiert die Klasse, die verwendet wird, um die Nachricht zu transformieren
dataTypeParser	Die einzelnen Felder – basierend auf deren Datentypen – werden damit transformiert.
configuration	Liste, die die Speicherung von zusätzlichen Konfigurationsparametern (z.B. URI) ermöglicht

Tabelle 4.2: Beschreibung der Attribute der Tabelle InboundConfiguration

Attributname	Beschreibung
id	Primärschlüssel dieser Tabelle
internalName	Definiert den internen Namen der Nachricht
externalName	Dieses Attribut deklariert den externen Namen der Nachricht.
parent	Legt das übergeordnete MessageProtocol fest; dieses Attribut ist leer, wenn kein übergeordnetes MessageProtocol vorhanden ist.

Tabelle 4.3: Beschreibung der Attribute der Tabelle MessageProtocol

Attributname	Beschreibung
id	Primärschlüssel dieser Tabelle
messageProtocol	Fremdschlüssel der Tabelle MessageProtocol
internalName	Gibt den internen Name des Feldes an
externalName	Definiert den externen Namen des Feldes
dataType	Attribut, das den Datentyp des Feldes angibt
mandatory	Legt fest, ob der Wert des Feldes leer sein darf
defaultValue	Definition eines Standardwertes für das Feld

Tabelle 4.4: Beschreibung der Attribute der Tabelle MessageProtocolField

In Abbildung 4.26 werden die Interfaces Composer und DataTypeComposer dargestellt. Mithilfe der Methode `compose` wird die Transformierung in externe Nachrichten (z.B. basierend auf JSON oder XML) durchgeführt. Für diese Methode müssen folgende Parameter übergeben werden:

- `transferId`: Spezieller Identifikator, der die Eindeutigkeit einer Nachricht sicherstellt, damit die Nachricht einer konkreten Prozessinstanz zugeordnet werden kann. Dieser setzt sich aus der `piId`, der `sId` und der `mfId`, welche bereits im Abschnitt 3 erläutert wurden, zusammen. Die `transferId` ist Bestandteil aller ausgehenden und auch eingehenden Nachrichten.
- `data`: Hierbei handelt es sich um ein Objekt des Typs `InternalData`. Alle empfangenen Business Objects der `ProcessEngine` werden als `InternalData`-Objekt persistiert. Dies gilt auch für alle Nachrichten von externen Subjekten. Diese Objekte werden in JSON-Objekte transformiert und im Datenbankschema `MessageStore` abgelegt. `InternalData` ermöglicht die einheitliche Speicherung von internen Nachrichten und deren Felder im `ExternalCommunicator`, welche mittels `Parser` bzw. `Composer` transformiert werden.
- `protocol`: Wie schon dargelegt, gibt dieses Objekt den Aufbau einer Nachricht vor.
- `config`: Diese Map enthält zusätzliche Konfigurationsparameter, welche im `Composer` genutzt werden können.

Mithilfe der zweiten Methode `getDescription` kann eine textuelle Beschreibung für den `Composer` deklariert werden. Im Prototypen sind derzeit Implementierungen für JSON und XML verfügbar. Alle konkreten Implementierungen müssen vom Interface `Composer`

abgeleitet sein.

Das Interface `DataTypeComposer` gibt an, wie die einzelnen Felder transformiert werden und werden im Composer angewandt. Dies basiert auf den Datentypen der Felder. Das Interface `DataTypeComposer` stellt die Methode `compose` bereit. Der Parameter `input` vom generischen Typ `<I>` – zur flexiblen Deklaration von Datentypen – ist jener Parameter, der in ein String-Objekt transformiert wird. Um beispielsweise eine Transformation für `Integer` durchzuführen, muss eine konkrete Implementierungsklasse angelegt werden, die `<I>` als `Integer` definiert und von `DataTypeComposer` ableitet.

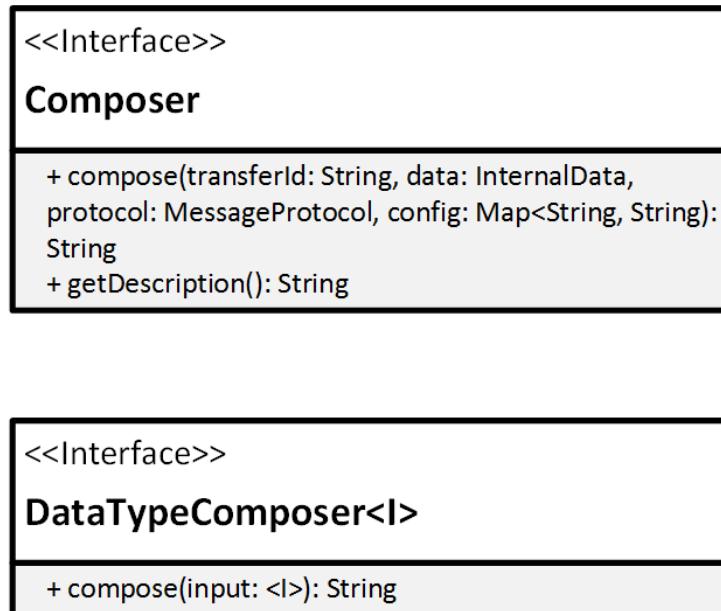


Abbildung 4.26: Klassendiagramm der Interfaces `Composer` und `DataTypeComposer` im ExternalCommunicator

Mittels des Interfaces `Parser`, das in Abbildung 4.27 veranschaulicht ist, werden externe Nachrichten (z.B. basierend auf JSON oder XML) in interne Nachrichten transformiert. Dazu wird die Methode `parse` verwendet. Diese Methode benötigt folgende Parameter:

- `input`: Dieser Parameter enthält die externe Nachricht, welche transformiert und an die `ProcessEngine` weitergeleitet wird.
- `protocol`: Parameter, der den Aufbau einer Nachricht vorgibt.
- `config`: Auch das Interface `Parser` kann verschiedene Konfigurationsparameter im Transformationsprozess nutzen.

Der Rückgabewert der Methode `parse` ist ein `ParseResult`. Dieses spezielle Objekt enthält zum einen die `transferId` der empfangenen Nachricht und zum anderen ein Objekt des Typs `InternalData`. In der derzeitigen Version des Prototypen wird die Transformation von JSON- und XML-Objekten unterstützt. Demgemäß werden diese empfangenen Objekte transformiert und an das Microservice `ProcessEngine` weitergeleitet. Zur Beschreibung einer konkreten Implementierung wird die Methode `getDescription` bereitgestellt.

Für die Transformation der einzelnen Felder einer Nachricht wird das Interface `DataTypeParser` genutzt. Die Methode `parse` führt diesen Transformationsprozess auf Basis des deklarierten Datentypen durch. Infolgedessen wird mittels des generischen Typs

<0> der Rückgabewert der Methode bestimmt. Um eine Transformation für den Typen LocalDateTime zu ermöglichen, muss eine konkrete Implementierungsklasse, welche von DataTypeParser ableitet, implementiert werden, wobei der generische Typ <0> mit LocalDateTime definiert wird.

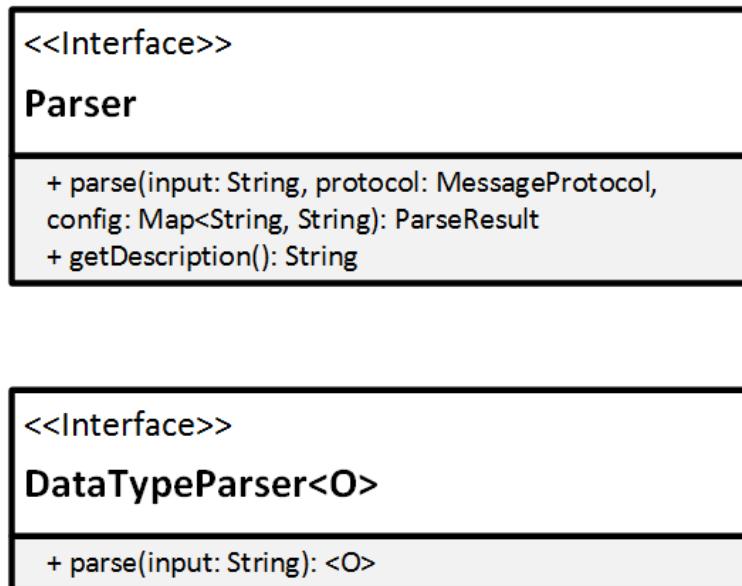


Abbildung 4.27: Klassendiagramm der Interfaces Parser und DataTypeParser im ExternalCommunicator

4.2.6.2 Plugins

Für das Senden bzw. Empfangen von Nachrichten werden sogenannte Plugins verwendet, die diesen Mechanismus bereitstellen. Plugins, zum Senden von Nachrichten, müssen vom Interface SendPlugin ableiten. Dieses Interface wird in Abbildung 4.28 dargestellt und enthält die Methode send. Der Parameter body definiert hierbei den Inhalt der Nachricht. Des Weiteren enthält der Parameter config zusätzlich verwendbare Konfigurationsparameter. Nachdem das gewünschte Plugin geladen wurde, wird die send-Methode aufgerufen, die dann die übergebene Nachricht an ein externes Subjekt sendet.

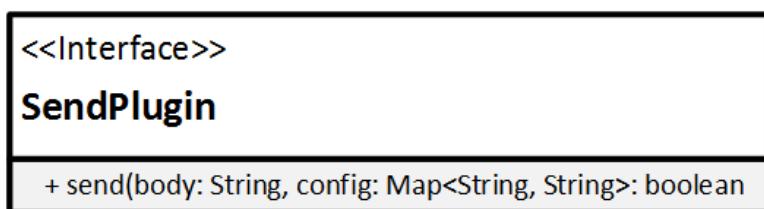


Abbildung 4.28: Klassendiagramm des Interfaces SendPlugin im ExternalCommunicator

Für das Empfangen von Nachrichten steht kein explizites Interface zur Verfügung, da andere Anforderungen gegeben sind. Das SendPlugin wird geladen und das Senden wird durchgeführt. Anschließend wird die Instanz des Plugins gelöscht. Das ReceivePlugin hingegen,

muss ständig aktiv sein, damit die externen Nachrichten angenommen werden können. Insofern werden hierfür die von Spring Boot verfügbaren REST-Controller genutzt, welche die Deklaration von REST-APIs ermöglichen. Diese sind nicht mit Interfaces kompatibel, weshalb kein explizites Interface bereitgestellt wurde.

4.2.6.3 Verwendung von Akka im ExternalCommunicator

Wie bereits erwähnt, wird für das Execution Framework Akka eingesetzt. Die Grundlagen von Akka wurden bereits im Abschnitt 4.2.5.1 erläutert. Das Execution Framework steuert das Senden bzw. das Empfangen von Nachrichten, indem diese transformiert und an externe Subjekte bzw. dem Microservice ProcessEngine weitergeleitet werden. Das Execution Framework nutzt hierbei die bereits vorgestellten Komponenten Parser und Composer für die Transformation der Nachrichten sowie Plugins für das Senden bzw. Empfangen der Nachrichten.

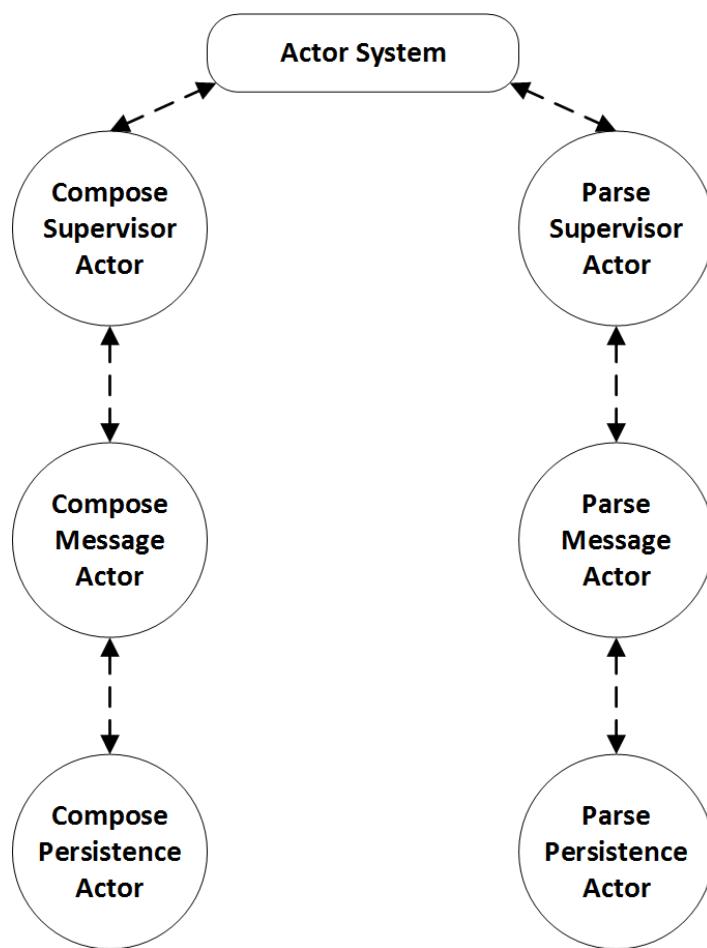


Abbildung 4.29: Verwendete Akteure im ExternalCommunicator

Die verwendeten Akteure des Microservices ExternalCommunicator sind in Abbildung 4.29 veranschaulicht. Wie in der ProcessEngine wird das Single Responsibility-Prinzip angewandt. Zudem wird das Supervisor-Prinzip eingesetzt, womit eine hierarchische Gliederung der Akteure gegeben ist. Folgende Akteure sind Bestandteil des Microservices ExternalCommunicator:

- ComposeSupervisorActor: Jede Anfrage der ProcessEngine wird zuerst von diesem Aktor bearbeitet, indem ein konkreter ComposeMessageActor erstellt wird.

Dementsprechend leitet der ComposeSupervisorActor alle Nachrichten an den ComposeMessageActor weiter.

- ComposeMessageActor: Dieser Aktor ist für die Transformierung der Business Objects in externe Nachrichten zuständig. Des Weiteren sendet dieser Aktor die transformierten Nachrichten durch den Einsatz von Plugins an externe Subjekte. Der ComposeMessageActor greift hierbei nie direkt auf die Datenbank zu.
- ComposePersistenceActor: Alle Datenbankzugriffe werden mit dem ComposePersistenceActor durchgeführt. Demgemäß sendet der ComposeMessageActor eine Nachricht an diesen Aktoren, die bestimmt welche Datenbankoperation auszuführen ist.
- ParseSupervisorActor: Empfangene Nachrichten von externen Subjekten werden vom ParseSupervisorActor verarbeitet. Dieser erstellt einen ParseMessageActor und delegiert die Nachricht an diese Instanz weiter.
- ParseMessageActor: Dieser Aktor transformiert die externe Nachrichten in Business Objects und leitet die Business Objects an das Microservice ProcessEngine weiter. Auch dem ParseMessageActor ist es nicht möglich Datenbankoperationen durchzuführen.
- ParsePersistenceActor: Der ParsePersistenceActor führt diverse Datenbankoperationen für den ParseMessageActor durch.

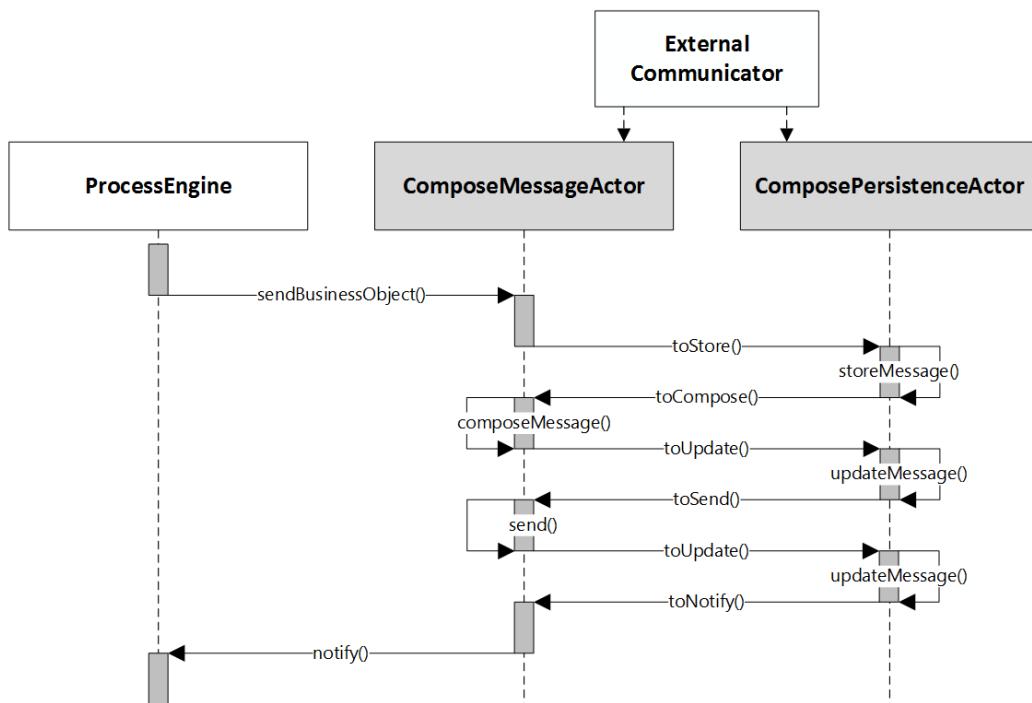


Abbildung 4.30: Sequenzdiagramm zur Veranschaulichung des Sendens von Nachrichten im ExternalCommunicator

In Abbildung 4.30 wird das Senden einer Nachricht mittels Sequenzdiagramm veranschaulicht. Aus Übersichtsgründen ist der ComposeSupervisorActor nicht inkludiert. Dieser ist für die Weiterleitung der Nachrichten an den ComposeMessageActor zuständig. Das Microservice ProcessEngine sendet die zu sendenden Business Objects an das Microservice ExternalCommunicator, wenn es sich beim Empfänger um ein externes Subjekt handelt. Diese Business Objects werden an den ComposeMessageActor weitergeleitet, welcher diese an den ComposePersistenceActor sendet. Der ComposePersistenceActor erstellt ein

Message-Objekt, das in der Datenbank gespeichert wird. Dieses Message-Objekt enthält die transferId, die interne Darstellung (InternalData) sowie die externe Darstellung der Nachricht. Zudem ist dem Message-Objekt ein Status (Mögliche Werte: TO_COMPOSE, COMPOSED, SENT) zugewiesen. In diesem Fall wird der Status mit TO_COMPOSE deklariert.

Nach der Persistierung des Message-Objekts wird der ComposeMessageActor notifiziert. Dieser lädt den zugewiesenen Composer und transformiert die interne Nachricht in eine externe Nachricht (z.B. basierend auf JSON oder XML). Anschließend wird diese externe Nachricht an den ComposePersistenceActor gesendet, damit dieser das Message-Objekt aktualisiert. Zudem wird der Status mit COMPOSED deklariert und der ComposeMessageActor wird abermals notifiziert. Der ComposeMessageActor lädt nun das passende SendPlugin. Mithilfe von diesem wird die externe Nachricht an das externe Subjekt gesendet. Nun wird der Status des Message-Objekts mittels ComposePersistenceActor in SENT geändert. Im letzten Schritt wird das Microservice ProcessEngine notifiziert, dass das Senden erfolgreich war.

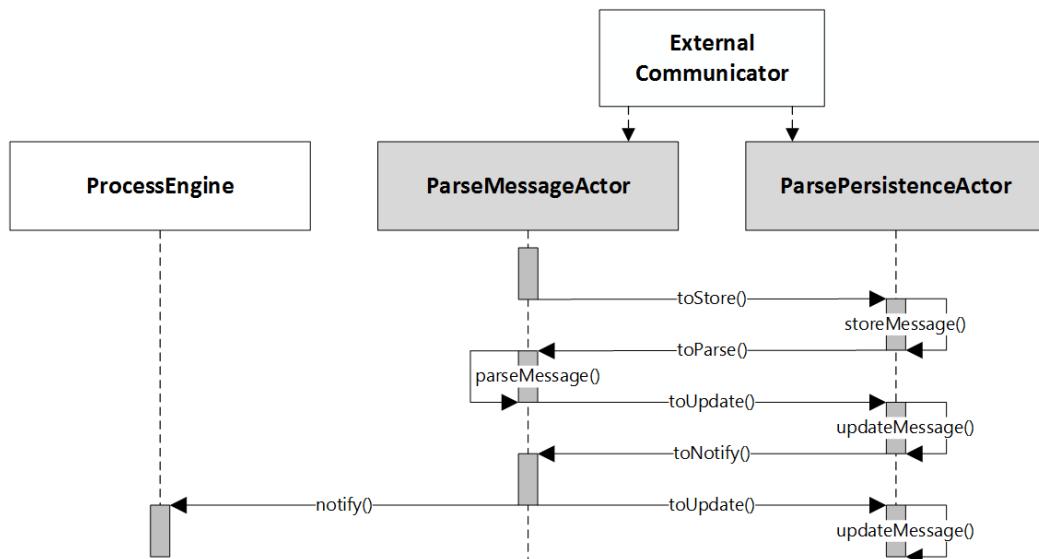


Abbildung 4.31: Sequenzdiagramm zur Veranschaulichung des Empfangs von Nachrichten im ExternalCommunicator

In Abbildung 4.31 wird das Sequenzdiagramm für den Empfang von Nachrichten dargestellt. Das ReceivePlugin, welches die externen Nachrichten an das Execution Framework weiterleitet, ist nicht Bestandteil dieser Abbildung. Zudem ist der ParseSupervisorActor ebenso nicht veranschaulicht, dessen Aufgabe die Weiterleitung der Nachrichten an den ParseMessageActor ist. Wie bereits erwähnt leitet das ReceivePlugin die externen Nachrichten an das Execution Framework weiter. Im ersten Schritt notifiziert der ParseMessageActor den ParsePersistenceActor, welcher ein Message-Objekt mit dem Status TO_PARSE (Mögliche Werte: TO_PARSE, PARSED, RECEIVED) erstellt. Zudem wird die externe Darstellung der Nachricht gespeichert.

Danach erfolgt die Notifizierung des Aktors ParseMessageActor, der die Transformierung der externen Nachricht mittels konfigurierten Parser durchführt. Das Resultat der Transformierung – die interne Nachricht – wird an den ParsePersistenceActor gesendet, welcher das Message-Objekt dementsprechend anpasst. Des Weiteren wird der Status in PARSED geändert und der ParseMessageActor wird wiederum benachrichtigt. Der ParseMessageActor sendet die interne Nachricht (Business Object) an das Microservice

ProcessEngine. Im letzten Schritt wird der Status des Message-Objekts in RECEIVED geupdated, womit der Empfangsprozess abgeschlossen ist.

4.2.7 Graphical User Interface (GUI)

Das Microservice GUI stellt den Benutzerinnen bzw. Benutzern eine grafische Oberfläche bereit, damit diese die Komponenten des Prototypen bedienen können. Demgemäß werden unter anderem Menüs bzw. Dialoge für den Import von Prozessmodellen sowie für die Ausführung von Prozessinstanzen bereitgestellt. Auf den genauen Aufbau der GUI wird in dieser Arbeit jedoch nicht eingegangen. Die GUI basiert auf Angular⁷ v2 (mittlerweile ist bereits Angular v4 verfügbar) und TypeScript⁸. Diese Frameworks eignen sich sehr gut für den Prototypen, da Technologien wie z.B. REST und JSON bereits standardmäßig unterstützt werden. Folglich ist die Einbindung von Anfragen an die anderen Microservices leicht umzusetzen. Die GUI sendet alle Anfragen an das Microservice Gateway, das die Anfragen an die internen Microservices weiterleitet.

⁷ <https://angular.io/>

⁸ <https://www.typescriptlang.org/>

5 Beispielprozesse

In diesem Kapitel werden zwei Beispielprozesse behandelt, um die Funktionsweise des Prototypen – insbesondere des Microservices ExternalCommunicator – zu veranschaulichen. Die Installationsanleitung des Prototypen sowie die verfügbaren Benutzerinnen bzw. Benutzer sind auf <https://github.com/stefanstanAIM/IPPR2016> zu finden, weshalb in dieser Arbeit nicht näher darauf eingegangen wird. Jedoch sind die folgenden Schritte zum Einfügen der beiden Beispielprozesse notwendig:

- Der Konfigurationsparameter `ippr.insert-examples.enabled` der Konfigurationsdatei `application.properties` im Microservice ProcessModelStorage muss mit `true` deklariert werden, damit die Beispielprozesse eingefügt werden.
- Zudem muss für den Konfigurationsparameter `spring.jpa.hibernate.ddl-auto` der vorher genannten Konfigurationsdatei der Wert `create` definiert werden. Dies löscht das Datenbankschema und erstellt dieses anschließend neu mit den Beispielprozessen. Dies ist notwendig, da ansonsten die Fremdschlüsselzuordnung nicht korrekt ist.
- Das Microservice ConfigurationService darf nicht gestartet werden. Andernfalls werden die Konfigurationsdateien von diesem bezogen und die lokalen Änderungen werden nicht übernommen.
- Starten der übrigen Microservices nach Anleitung. Beim Start des Microservices ProcessModelStorage werden die Beispielprozesse automatisch eingefügt.

5.1 Prozessbeschreibung

Bei den Beispielprozessen handelt es sich um einen Reiseantrag, den Mitarbeiterinnen/Mitarbeiter an ihre Teamleiterin/ihren Teamleiter senden. Bei Annahme dieses Antrags wird dieser an das Travel Management (externes Subjekt) weitergeleitet, welches ein Hotel sucht und bucht. Diese Hotelbuchung wird an die Mitarbeiterin/den Mitarbeiter gesendet. Wird der Reiseantrag hingegen abgelehnt, ist der Prozess zu Ende. In Abbildung 5.1 wird das SID der Beispielprozesse dargestellt. Dieses ist für beide Beispielprozesse ident.

In Abbildung 5.2 wird das SBD – also das interne Verhalten – des Subjekts TeamleiterIn dargestellt. Dieses ist für beide Beispielprozesse gleich. Nachdem dieses Subjekt einen Reiseantrag vom Subjekt MitarbeiterIn empfangen hat, kann das Subjekt diesen Reiseantrag ansehen und prüfen. Anschließend kann der Reiseantrag angenommen bzw. abgelehnt werden. Diese Annahme bzw. Ablehnung wird als Nachricht an das Subjekt MitarbeiterIn gesendet, womit der Prozess für das Subjekt TeamleiterIn beendet ist. Das SBD des Subjekts MitarbeiterIn wird in den Abschnitten 5.2 und 5.3 veranschaulicht, da das Verhalten der Beispielprozesse hier unterschiedlich ist.

5 Beispielprozesse

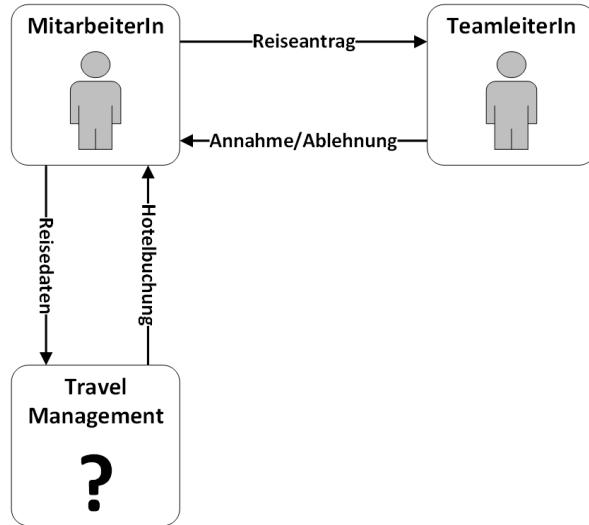


Abbildung 5.1: Subject Interaction Diagram (SID) der Beispielprozesse

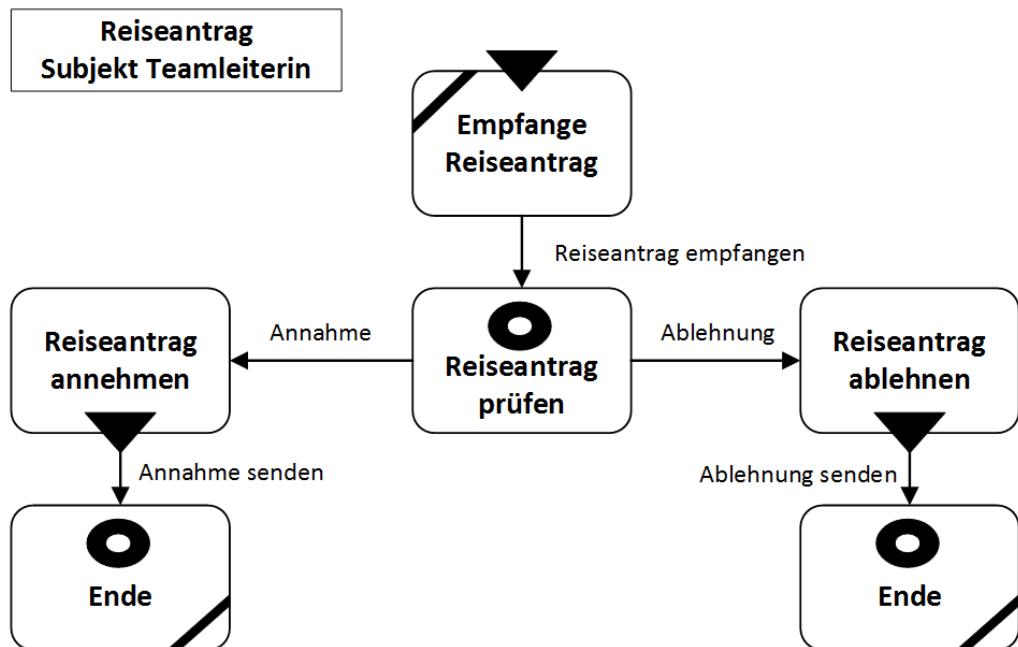


Abbildung 5.2: Subject Behavior Diagram (SBD) des Subjekts TeamleiterIn in den Beispielprozessen

5.2 Beispielprozess 1: Reiseantrag mit Prozesskommunikation

In diesem Abschnitt wird der Beispielprozess 1 erläutert. Abbildung 5.3 zeigt das interne Verhalten des Subjekts MitarbeiterIn. Dieses Subjekt kann einen Reiseantrag anlegen und diesen an das Subjekt TeamleiterIn senden. Anschließend wird auf die Antwort des Subjekts TeamleiterIn gewartet. Wurde der Reiseantrag abgelehnt, dann ist der Prozess abgeschlossen. Wurde der Reiseantrag angenommen, dann wird dieser an das externe Subjekt Travel Management gesendet. Bei diesem externen Subjekt ist ein Prozess hinterlegt. Dementsprechend können die Prozesse Reiseantragsprozess und Hotelbuchungsprozess per Nachrichten interagieren

5 Beispielprozesse

und folglich wird ein Prozessnetzwerk aufgebaut. Das SBD des Subjekts Travel Management ist ebenso in Abbildung 5.3 veranschaulicht. In diesem Beispielprozess sendet das Subjekt MitarbeiterIn die Reisedaten an das Subjekt Travel Management, womit die Reisedaten nun im Hotelbuchungsprozess verfügbar sind. Im Hotelbuchungsprozess kann das Subjekt Travel Management die Hotelbuchung durchführen und an das Subjekt MitarbeiterIn senden. Somit ist der Hotelbuchungsprozess beendet und die Hotelbuchung steht im Reiseantragsprozess zur Verfügung.

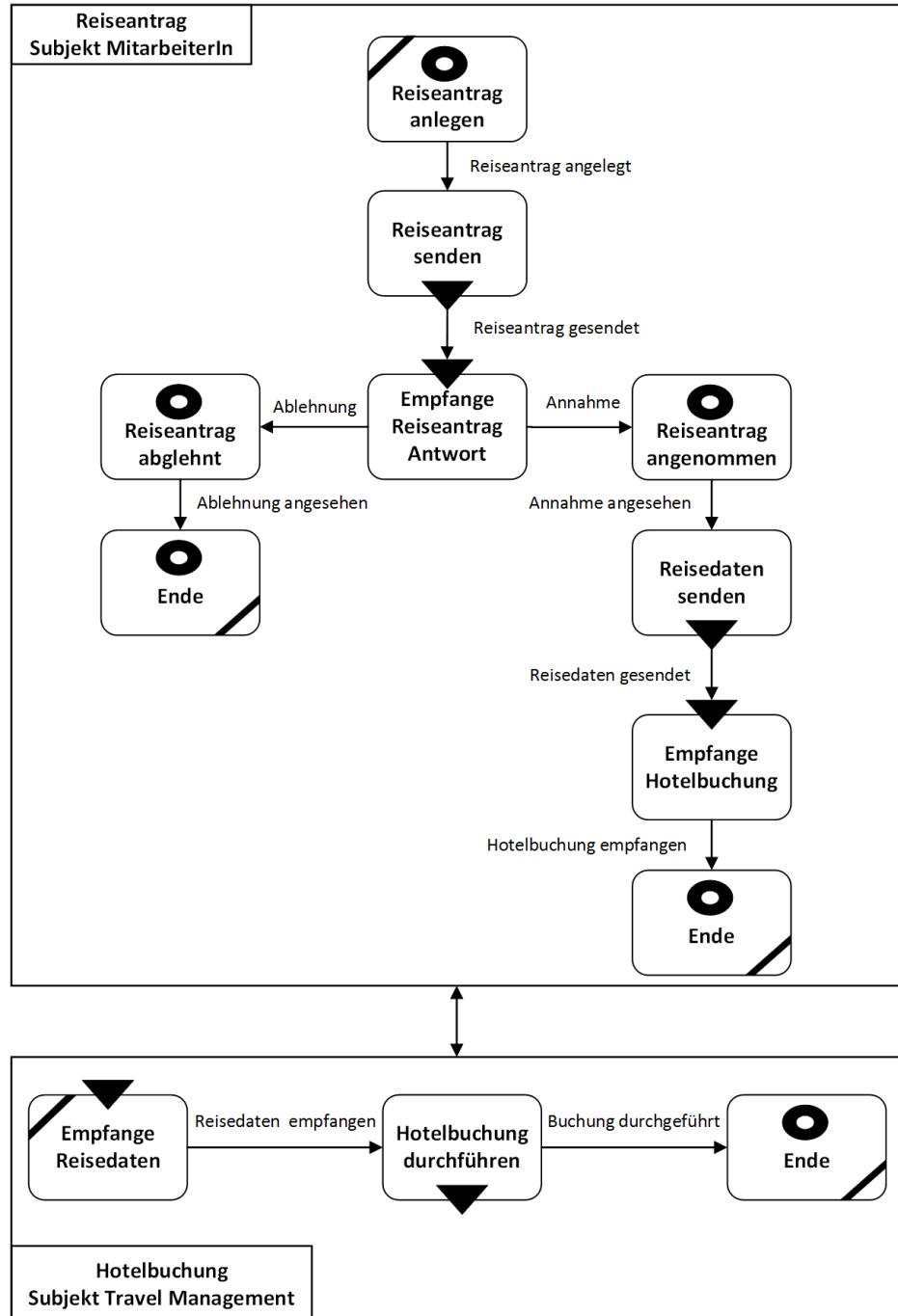


Abbildung 5.3: Subject Behavior Diagram (SBD) der Subjekte MitarbeiterIn und Travel Management im Beispielprozess 1

Nachfolgend erfolgt eine Beschreibung, wie der Beispielprozess 1 mit dem Prototypen auszuführen ist. Nach Aufruf des Microservices GUI erscheint der Login-Bildschirm

5 Beispielprozesse

(siehe Abbildung 5.4), wo *Username* und *Passwort* eingegeben werden können. Um diesen Beispielprozess durchzuführen kann u.a. der Benutzer *stefan* mit dem Passwort *1234* verwendet werden.

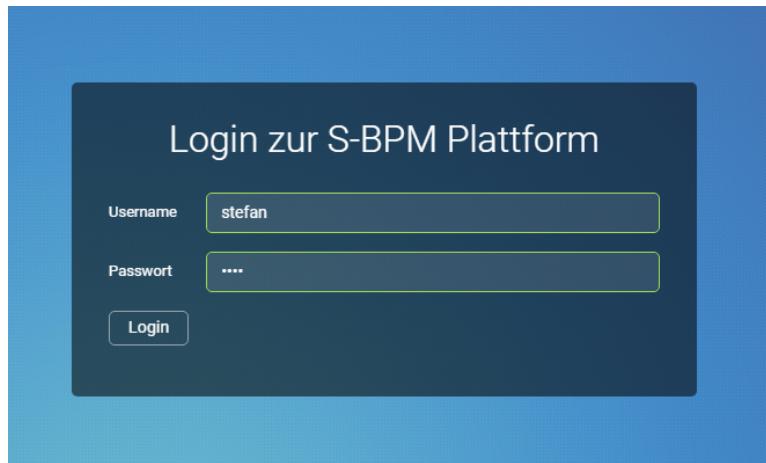


Abbildung 5.4: Login-Bildschirm

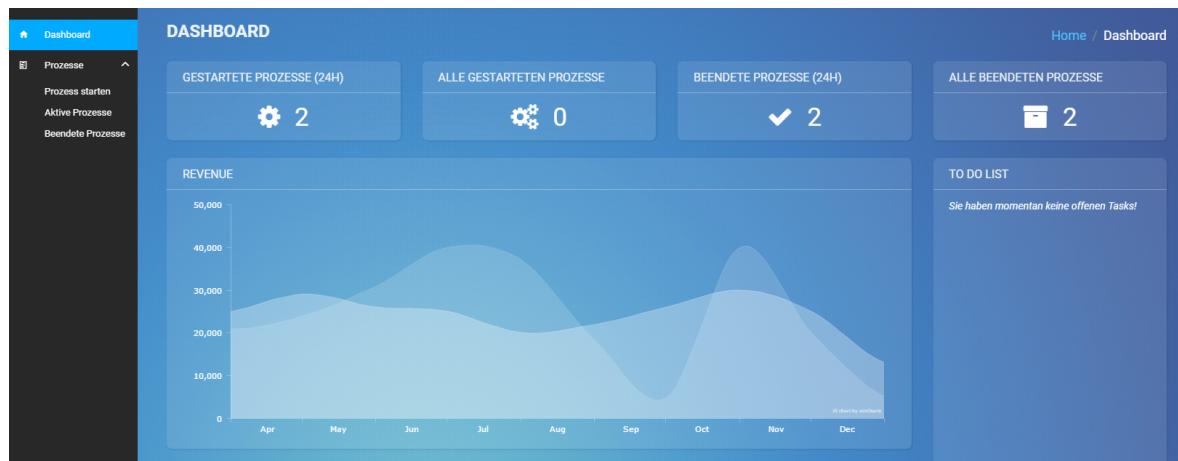


Abbildung 5.5: Dashboard des Subjekts MitarbeiterIn

War das Login erfolgreich wird das Dashboard, das in Abbildung 5.5 veranschaulicht wird, angezeigt, in welchem einige Kennzahlen sowie eine Liste mit offenen Tasks angezeigt wird. Um einen Prozess zu starten, muss links der Menüpunkt *Prozess starten* ausgewählt werden.

PROZESS STARTEN				
#	Name	Beschreibung	Erstelltdatum	Aktion
1	Reiseantrag mit externem IT-System	Beispielprozess 2	20.06.2017 20:01:07	<button>Start</button>
3	Reiseantrag mit Prozesskommunikation	Beispielprozess 1	20.06.2017 20:01:08	<button>Start</button>

Abbildung 5.6: Prozess starten

Nun werden all jene Prozesse angezeigt (siehe Abbildung 5.6), welche die/der derzeit eingeloggte Benutzerin/Benutzer starten kann. Für diesen Beispielprozess muss der Prozess *Reiseantrag mit Prozesskommunikation* ausgewählt werden. Damit die Prozessinstanz angelegt wird, muss *Start* gedrückt werden.

5 Beispielprozesse

Business Objects
State: Reiseantrag anlegen

REISEANTRAG

Von
21.07.2017

Bis
25.07.2017

Ort
Florenz

Mögliche nächste Schritte
Reiseantrag senden

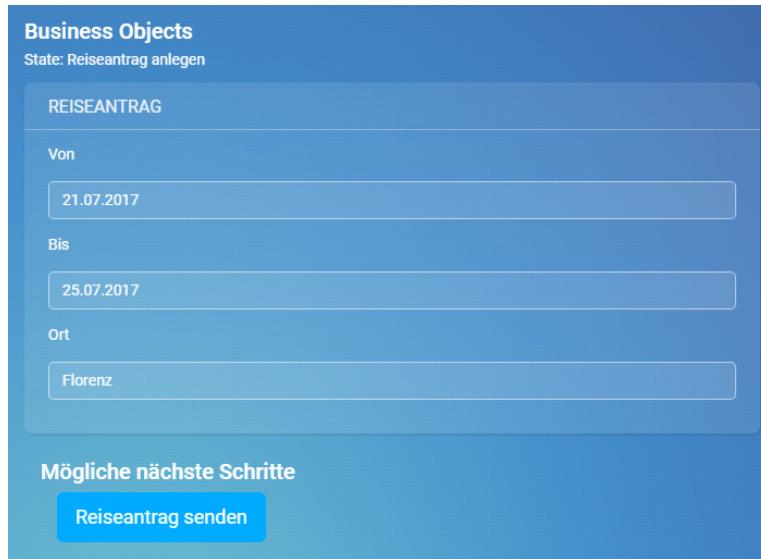


Abbildung 5.7: State: Reiseantrag anlegen

Nach dem Start der Prozessinstanz befinden sich alle beteiligten Subjekte in ihrem Start-State. Demgemäß besteht nun die Möglichkeit einen Reiseantrag anzulegen, wie in Abbildung 5.7 veranschaulicht wird. Dazu müssen Werte für die Felder des Business Objects *Reiseantrag* deklariert werden. Um in den nächsten State zu wechseln, muss *Reiseantrag senden* ausgewählt werden. In diesem Fall gibt es nur einen möglichen nächsten State. Wären hier mehrere vorhanden, würde diese auch hier angezeigt werden.

Business Objects
State: Reiseantrag senden

REISEANTRAG

Von
21.07.2017

Bis
25.07.2017

Ort
Florenz

Zuweisungen
TeamleiterIn: BOSS_RULE
Singer Robert

Mögliche nächste Schritte
Empfange Reiseantrag Antwort

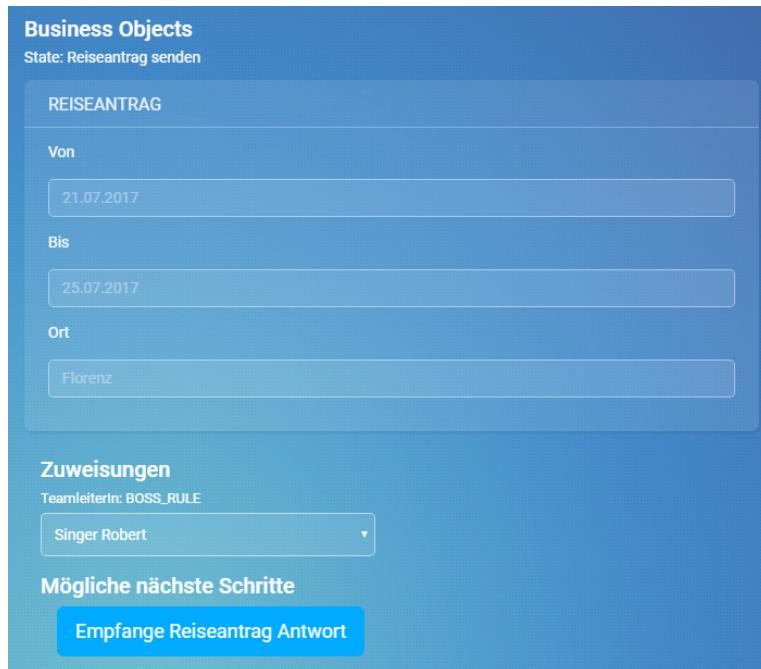


Abbildung 5.8: State: Reiseantrag senden

Im nächsten State, der in Abbildung 5.8 dargestellt wird, wird das vorher erstellte Business Object nochmals angezeigt. Da es sich hier um einen Send-State handelt, muss ein Benutzer/eine Benutzerin ausgewählt werden, welche/r die Rolle des Subjekts TeamleiterIn übernehmen kann. In diesem Fall steht nur der Benutzer *Singer Robert* bereit. Nachdem *Empfange Reiseantrag Antwort* geklickt wurde, wird das Business Object an den Benutzer

5 Beispielprozesse

robert gesendet. War dieser Sendevorgang erfolgreich, wird angezeigt, dass man sich derzeit im Wartezustand befindet. Nun ist ein Benutzerwechsel notwendig. Insofern ist ein Login mit dem Benutzer *robert* und dem Passwort 1234 erforderlich.

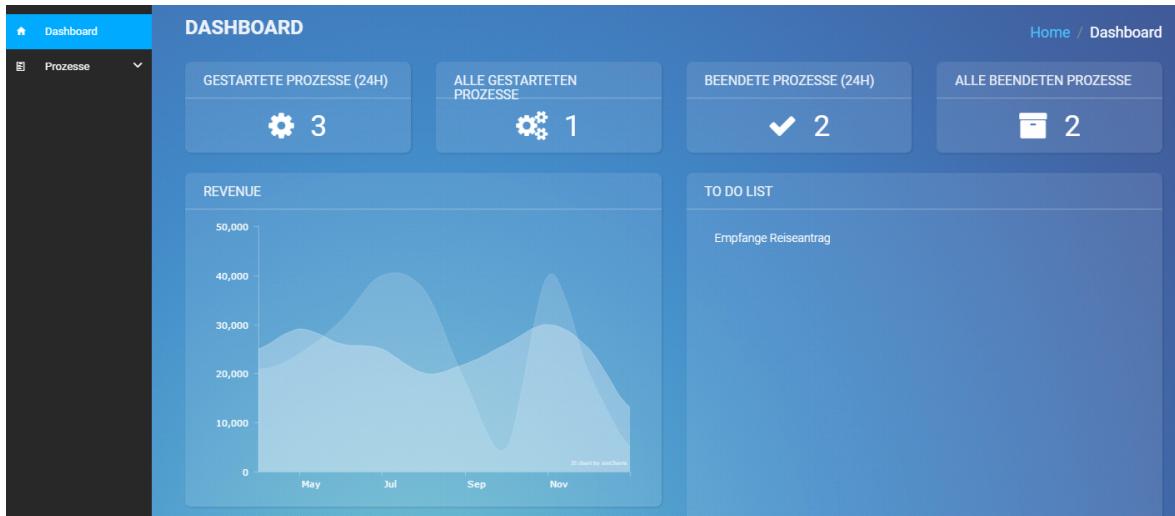


Abbildung 5.9: Dashboard des Subjekts TeamleiterIn

Nach erfolgreichem Login mit dem Benutzer *robert* wird wieder das Dashboard angezeigt (siehe Abbildung 5.9). In der Liste mit offenen Tasks ist ersichtlich, dass ein Reiseantrag empfangen wurde. Dieser offene Task muss nun ausgewählt werden.

The screenshot shows a 'Business Objects' screen. The title is 'REISEANTRAG' and the state is 'Empfange Reiseantrag'. It contains fields for 'Von' (21.07.2017), 'Bis' (25.07.2017), and 'Ort' (Florenz). Below the form is a section titled 'Mögliche nächste Schritte' with a blue button labeled 'Reiseantrag prüfen'.

Abbildung 5.10: State: Reiseantrag empfangen

Nun wird der empfangene Reiseantrag angezeigt (siehe Abbildung 5.10). Dementsprechend ist das empfangene Business Object und dessen Felder veranschaulicht. Im nächsten State kann der Antrag genehmigt oder abgelehnt werden. Dazu muss *Reisenantrag prüfen* gedrückt werden.

5 Beispielprozesse

Business Objects
State: Reiseantrag prüfen

REISEANTRAG

Von
21.07.2017

Bis
25.07.2017

Ort
Florenz

Mögliche nächste Schritte

Reiseantrag annehmen Reiseantrag ablehnen

Abbildung 5.11: State: Reiseantrag prüfen

Im State *Reiseantrag prüfen* kann der Reiseantrag angenommen bzw. abgelehnt werden. Dies ist in Abbildung 5.11 veranschaulicht. Für dieses Beispiel wird dieser akzeptiert, womit *Reiseantrag annehmen* ausgewählt werden muss.

Business Objects
State: Reiseantrag annehmen

REISEANTRAGANNAHME

Information

Mögliche nächste Schritte

Ende

Abbildung 5.12: State: Reiseantrag annehmen

Für die Annahme eines Reiseantrages kann noch eine zusätzliche Information hinterlegt werden (siehe Abbildung 5.12). Beim State *Reiseantrag annehmen* handelt es sich um einen Send-State. Somit wird die Annahme an das Subjekt MitarbeiterIn – in diesem Fall der Benutzer *stefan* – gesendet. Nach Klicken von *Ende* ist der Prozess für das Subjekt TeamleiterIn beendet. Um den Prozess fortzusetzen, ist ein Login des Benutzers *stefan* notwendig.

5 Beispielprozesse

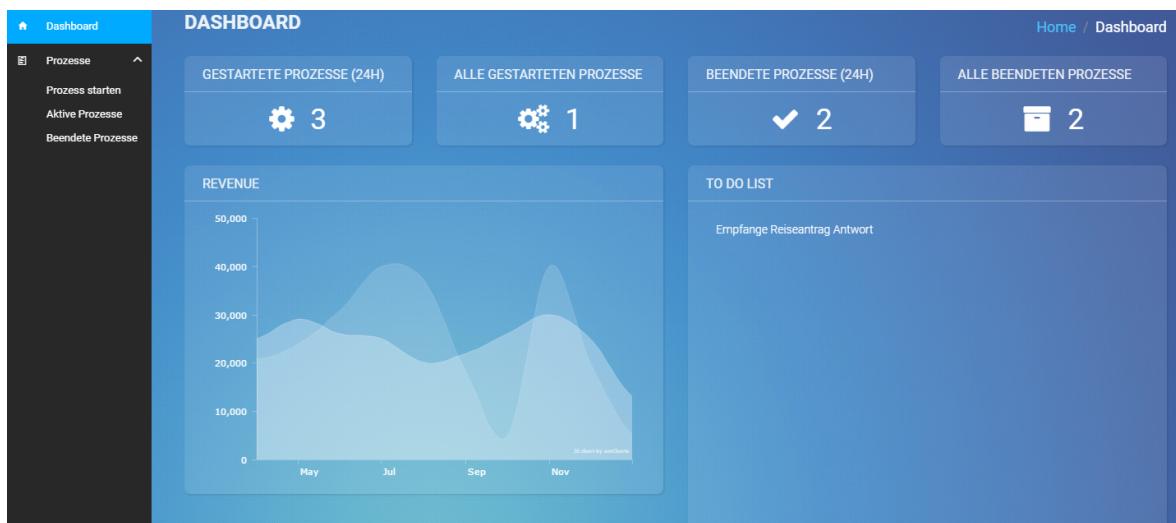


Abbildung 5.13: Dashboard des Subjekts MitarbeiterIn nach dem Empfang der Annahme des Reiseantrags

Nach dem Login mit dem Benutzer *stefan* ist in dessen Liste mit offenen Task ersichtlich, dass dieser eine Antwort für den Reiseantrag erhalten hat. Dieser Task muss wieder ausgewählt werden. Dies wird in Abbildung 5.13 dargestellt.

5 Beispielprozesse

Business Objects
State: Empfange Reiseantrag Antwort

REISEANTRAG

Von
21.07.2017

Bis
25.07.2017

Ort
Florenz

REISEANTRAGANNAHME

Information

Mögliche nächste Schritte

Reiseantrag angenommen

The screenshot shows a user interface for managing travel requests. At the top, it displays the state 'Empfange Reiseantrag Antwort'. Below this, there are two main sections: 'REISEANTRAG' and 'REISEANTRAGANNAHME'. The 'REISEANTRAG' section contains fields for 'Von' (from) set to '21.07.2017', 'Bis' (until) set to '25.07.2017', and 'Ort' (location) set to 'Florenz'. The 'REISEANTRAGANNAHME' section contains a single 'Information' field which is currently empty. At the bottom, a section titled 'Mögliche nächste Schritte' (possible next steps) contains a button labeled 'Reiseantrag angenommen' (travel request accepted). The entire interface has a blue-themed design.

Abbildung 5.14: State: Empfange Reiseantrag Antwort

In diesem State wird nochmal der eingereichte Reiseantrag dargestellt. Zudem ist erkennbar, dass der Reiseantrag angenommen wurde, da als nächster State nur *Reiseantrag angenommen* vorhanden ist (siehe Abbildung 5.14). Wäre dies nicht der Fall, dann würde als nächster State nur *Reiseantrag abgelehnt* auswählbar sein. Der nächste vorhandene State muss nun ausgewählt werden. Im nächsten State *Reiseantrag angenommen* werden dieselben Information nochmals veranschaulicht, weshalb dieser hier nicht explizit dargestellt wird. In diesem hat man die Möglichkeit den State *Sende an Travel Management* auszuwählen. Dies ist für dieses Beispiel auch notwendig.

Zuweisungen
Travel Management: TRAVEL_RULE

TUI Marlene

Mögliche nächste Schritte

Empfangene Hotelbuchung

The screenshot shows a user interface for assigning tasks. It starts with a section titled 'Zuweisungen' (Assignments) under 'Travel Management: TRAVEL_RULE', which lists 'TUI Marlene'. Below this, there is another section titled 'Mögliche nächste Schritte' (possible next steps), which contains a button labeled 'Empfangene Hotelbuchung' (Received hotel booking). The interface has a blue-themed design.

Abbildung 5.15: State: Sende an Travel Management

In Abbildung 5.15 handelt es sich um einen Send-State, der die Reisedaten an das externe

5 Beispielprozesse

Subjekt Travel Management sendet. Wie bereits dargelegt, ist der Prozess Hotelbuchung für dieses externe Subjekt hinterlegt. Infolgedessen werden die Reisedaten in den Hotelbuchungsprozess übernommen und es besteht ein Prozessnetzwerk zwischen Reiseantragsprozess und Hotelbuchungsprozess. Des Weiteren muss eine Benutzerin/ein Benutzer für das Subjekt Travel Management deklariert werden. Nach Auswahl von *Empfange Hotelbuchung* wird die Nachricht an den Hotelbuchungsprozess gesendet und das Subjekt MitarbeiterIn geht in den Wartezustand über. Für diesen Beispielprozess ist nun ein Login mit der Benutzerin *marlene* (Passwort: 1234) notwendig.

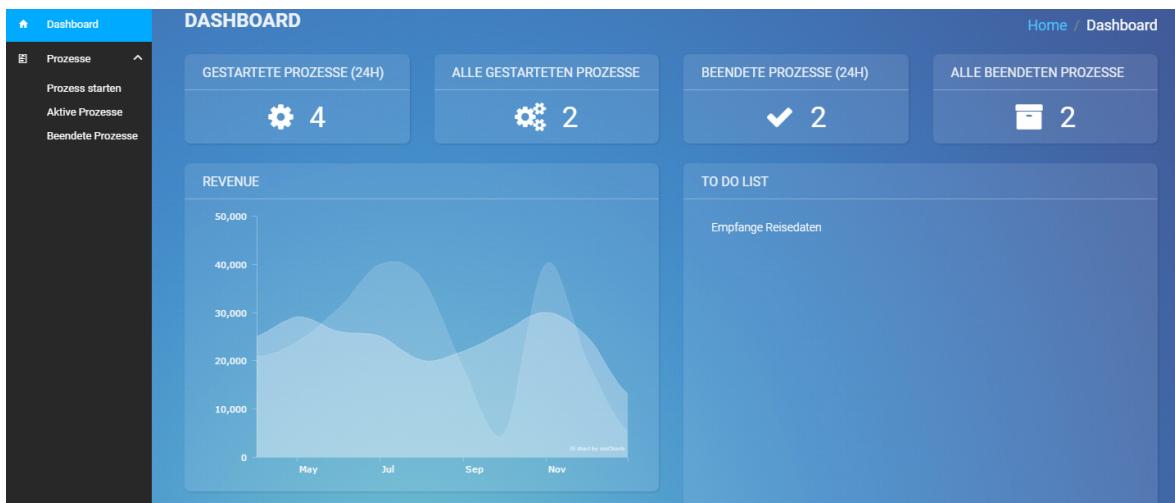


Abbildung 5.16: Dashboard des Subjekts Travel Management

Nach erfolgtem Login ist in der Liste mit Tasks ersichtlich, dass neue Reisedaten eingegangen sind (siehe Abbildung 5.16). Es handelt sich hier allerdings um eine komplett eigenständige Prozessinstanz des Hotelbuchungsprozesses. Dieser offene Task muss nun ausgewählt werden.

The screenshot shows the 'REISEANTRAG' (Travel Request) state. The top header reads 'Business Objects' and 'State: Empfange Reisedaten'. The main form fields are: 'Von' (From) with value '21.07.2017', 'Bis' (Until) with value '25.07.2017', and 'Ort' (Location) with value 'Florenz'. Below the form is a section titled 'Mögliche nächste Schritte' (Possible next steps) containing a blue button labeled 'Hotelbuchung durchführen' (Perform hotel booking).

Abbildung 5.17: State: Empfange Reisedaten

In diesem State wird der akzeptierte Reiseantrag angezeigt, damit eine Hotelbuchung durchgeführt werden kann (siehe Abbildung 5.17). Infolgedessen muss *Hotelbuchung durchführen*

5 Beispielprozesse

ausgewählt werden.

Business Objects
State: Hotelbuchung durchführen

HOTELBUCHUNG

Name

Sterne

Mögliche nächste Schritte

Ende

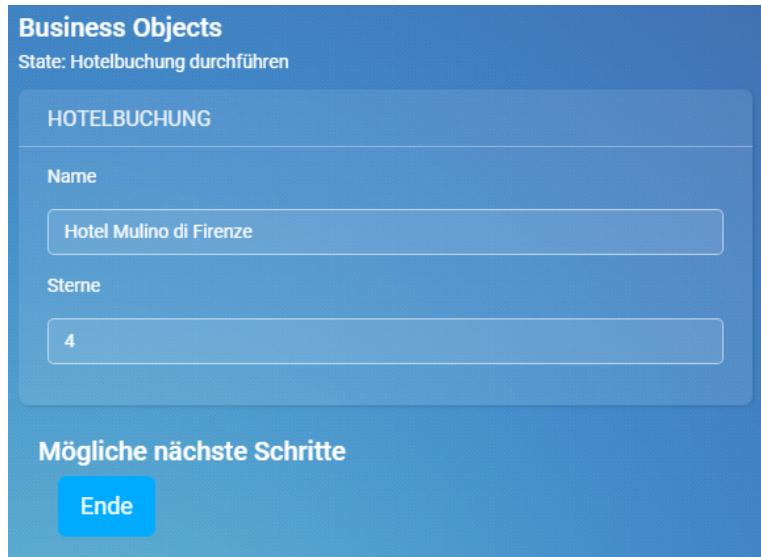


Abbildung 5.18: State: Hotelbuchung durchführen

Anschließend können die Daten des Hotels eingegeben werden. Dies wird in Abbildung 5.18 gezeigt. Hierbei handelt es sich um einen Send-State. Demgemäß wird diese Buchung als Nachricht an den Reiseantragsprozess gesendet. Der Prozess Hotelbuchung ist für das Subjekt Travel Management nun zu Ende. Im nächsten Schritt ist ein Login mit dem Benutzer *stefan* notwendig, um die empfangene Hotelbuchung anzusehen. In der Liste der offenen Tasks wird nun angezeigt, dass Reisedaten empfangen wurden. Dies wird hier jedoch nicht zusätzlich veranschaulicht.

DASHBOARD [Home](#) / [Dashboard](#)

Business Objects
State: Empfange Hotelbuchung

HOTELBUCHUNG

Name

Sterne

Mögliche nächste Schritte

Ende

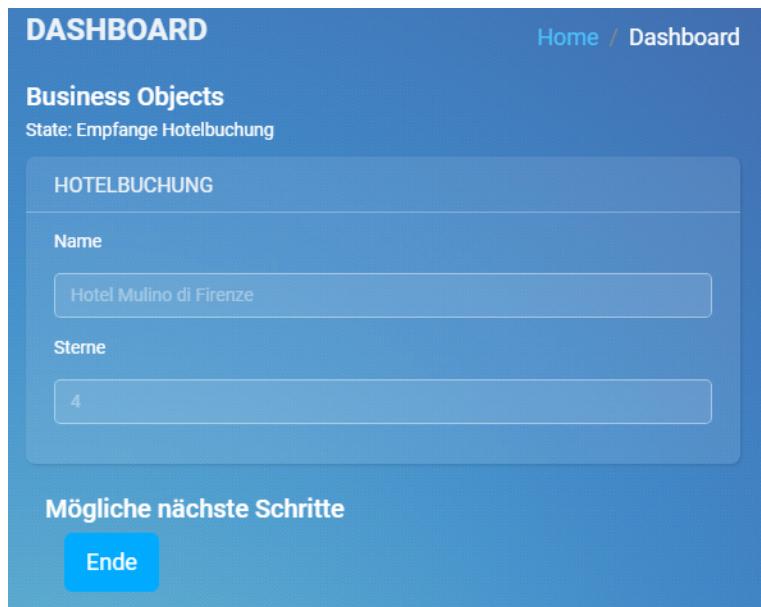


Abbildung 5.19: State: Empfange Hotelbuchung im Beispielprozess 1

Die Hotelbuchung ist nun auch im Reiseantragsprozess verfügbar, wie in Abbildung 5.19 dargestellt wird. Mit der Auswahl von *Ende* ist der Prozess nun vollständig abgeschlossen.

5.3 Beispielprozess 2: Reiseantrag mit externem IT-System

In Abbildung 5.20 wird das SBD des Subjekts MitarbeiterIn vom Beispielprozess 2 veranschaulicht. Der grundsätzliche Ablauf ist gleich wie im Beispielprozess 1. Im Vergleich zum Beispielprozess 1 ist beim externen Subjekt Travel Management kein Prozess hinterlegt, sondern ein externes IT-System. Die Konfigurationseinstellungen für das IT-System werden für diesen Beispielprozess automatisch angelegt. Die Nachrichten an das externe Subjekt bzw. vom externen Subjekt werden an das externe IT-System gesendet bzw. von diesem empfangen. In diesem Beispielprozess werden die Reisedaten an das externe IT-System gesendet und von diesem verarbeitet. Nach der Verarbeitung sendet das externe IT-System die Hotelbuchung zurück und folglich wird diese im Reiseantragsprozess gespeichert und steht dem Subjekt MitarbeiterIn zur Verfügung.

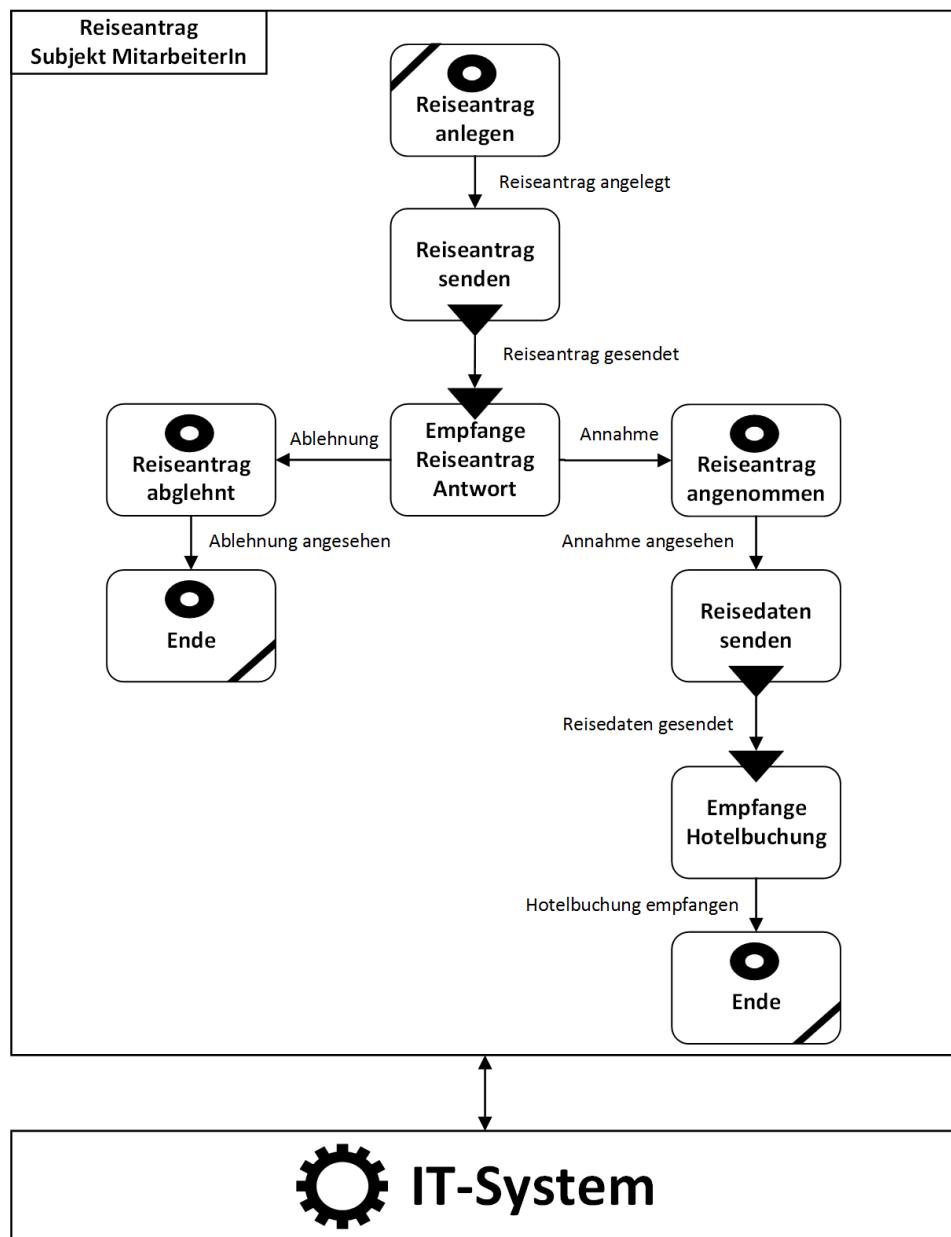


Abbildung 5.20: Subject Behavior Diagram (SBD) des Subjekts MitarbeiterIn im Beispielprozess 2

Für diesen Beispielprozess werden die Schritte für das Anlegen und der Annahme des

Reiseantrages nicht erneut veranschaulicht. Diese können dem Abschnitt 5.2 entnommen werden. Für den Beispielprozess 2 werden dieselben Werte für die Felder der Business Objects des Beispielprozesses 1 verwendet. Einzig beim Starten des Prozesses muss *Reiseantrag mit externem IT-System* ausgewählt werden. Des Weiteren muss beim State *Sende an Travel Management* keine Benutzerin bzw. kein Benutzer deklariert werden, da für das externe Subjekt kein Prozess hinterlegt wurde.

Beim Senden der Reisedaten an das Subjekt Travel Management werden diese an das externe IT-System gesendet. Hierbei wird nur über definierte APIs kommuniziert. Für diesen Beispielprozess wurden die Konfigurationen so deklariert, dass die Reisedaten an `http://localhost:10000/ec/test` gesendet werden. Folglich kann im Log des Microservices ExternalCommunicator geprüft werden, ob dieser Sendevorgang erfolgreich war. Bei erfolgreichem Sendevorgang muss ein Log-Eintrag, wie in Abbildung 5.21, vorhanden sein. Das Subjekt MitarbeiterIn befindet sich nun im State *Empfange Hotelbuchung* und wartet somit auf die Hotelbuchung.

```
2017-06-20 20:39:19,409 DEBUG 11196 --- [o-auto-1-exec-2] a.f.i.c.ExternalCommunicatorApplication : Received [<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<TravelRequest From="21.07.2017" Location="Florenz" TRANSFER-ID="1-1-7" To="25.07.2017"/>
]
```

Abbildung 5.21: Log-Eintrag im ExternalCommunicator bei erfolgreichem Senden

Hierbei werden die Reisedaten in eine externe Nachricht – in diesem Fall auf Basis von XML – transformiert. Dies wird in Listing 5.1 dargestellt. Weiters ist erwähnenswert, dass die Felder des Business Objects in dieser externen Nachricht unterschiedlich benannt werden (z.B. Ort \leftrightarrow Location). Dieses Verhalten basiert auf den hinterlegten Konfigurationen für diesen Beispielprozess. Die möglichen Konfigurationen wurden bereits im Abschnitt 4.2.6.1 erläutert. Im Beispielprozess 1 waren diese Konfigurationen nicht notwendig, da für die Kommunikation zwischen Prozessen der ProcessEngine Standardkonfigurationen verwendet werden.

Listing 5.1: Nachricht an das externe IT-System

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <TravelRequest From="21.07.2017" Location="Florenz" TRANSFER-ID="1-1-7"
To="25.07.2017"/>
```

Das externe IT-System kann nun eine Hotelbuchung als externe Nachricht an den ExternalCommunicator senden, der diese dann verarbeitet. Hierfür muss diese an `localhost:10000/ec/json/booking` gesendet werden. Alle notwendigen Konfigurationen für diesen Beispielprozess wurden bereits automatisch angelegt. Der Inhalt dieser Nachricht ist in Listing 5.2 veranschaulicht. Wie ersichtlich ist, basiert diese auf JSON, da hier andere Konfigurationseinstellungen vorgenommen wurden, um die Flexibilität zu verdeutlichen. Wichtig ist, dass die *TRANSFER-ID* den selben Wert wie in Listing 5.1 hat, damit die Zuordnung der Nachricht an die korrekte Prozessinstanz gewährleistet ist.

Listing 5.2: Nachricht vom externen IT-System

```
1 {
2   "TRANSFER-ID": "1-1-7",
3   "TYPE": "HotelBooking",
4   "Name": "Hotel Mulino di Firenze",
5   "Stars": "4"
6 }
```

5 Beispielprozesse

Für das Senden von HTTP-Anfragen kann beispielsweise das Tool Restlet Client¹, welches für Google Chrome verfügbar ist, verwendet werden. Allerdings kann jedes beliebige Tool, das HTTP-Anfragen senden kann, genutzt werden. In Abbildung 5.22 wird das Senden mittels Restlet Client dargestellt. Der *Content-Type* muss mit *application/json* definiert werden. Zudem muss die *HTTP-Methode POST* ausgewählt werden. Der *Body* muss das Listing 5.2 beinhalten.

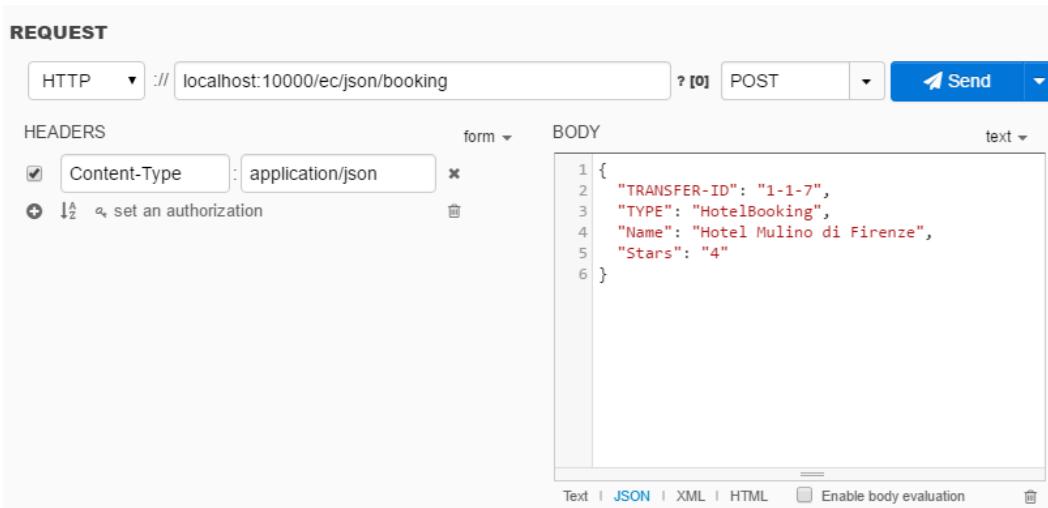


Abbildung 5.22: Senden von HTTP-Anfragen mittels Restlet Client

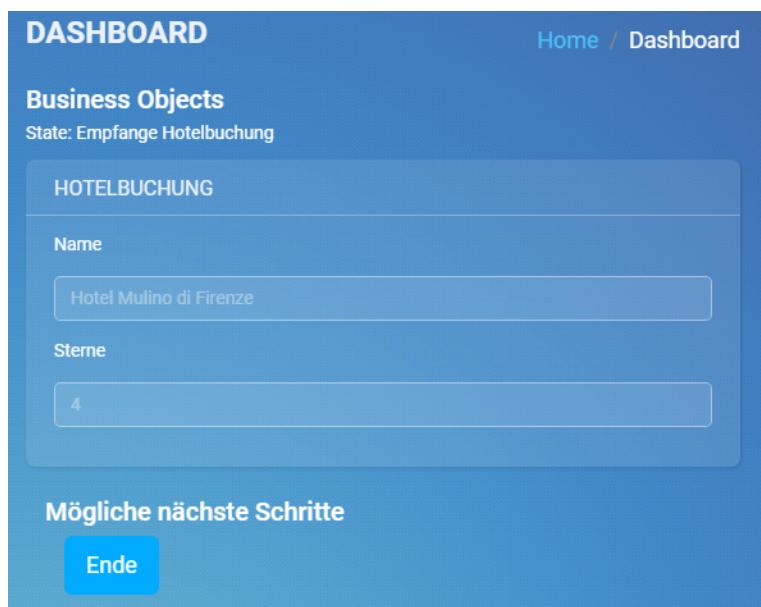


Abbildung 5.23: State: Empfange Hotelbuchung im Beispielprozess 2

Nach dem Senden der Anfrage des externen IT-Systems verarbeitet der ExternalCommunicator diese und gibt diese an die ProcessEngine weiter. In diesem Fall wird eine Hotelbuchung aus der externen Nachricht generiert und persistiert. Dementsprechend ist diese Hotelbuchung nun für das Subjekt MitarbeiterIn verfügbar. Um dies zu prüfen, muss man sich mit jener Benutzerin bzw. jenem Benutzer einloggen, die/der diesen Reiseantrag gestartet hat. In der Liste der offenen Tasks ist nun erkenntlich, dass eine

¹ <https://chrome.google.com/webstore/detail/restlet-client-rest-api-t/aejoelaoggembahagimdilamlcdmfm>

5 Beispielprozesse

Hotelbuchung empfangen wurde. Bei Auswahl von dieser, wird die Hotelbuchung angezeigt (siehe Abbildung 5.23). Nach Auswahl von *Ende* ist der Reiseantragsprozess beendet.

6 Erweiterungen und Verbesserungen für den Prototypen

In diesem Abschnitt werden mögliche Erweiterungen und Verbesserungen für den Prototypen vorgestellt. Hierbei wird die neue Architektur in Abbildung 6.1 dargestellt. Anhand dieser Abbildung ist erkennbar, dass drei neue Microservices vorgeschlagen werden. Infolgedessen werden die Microservices AuthenticationService, LogServer sowie LifeCycleService im Abschnitt 6.1 näher erläutert. Zudem werden im Abschnitt 6.2 die bereits vorgestellten Microservices des Prototypen behandelt.

Eine weitere Neuerung der überarbeiteten Architektur ist die Einführung eines Message Brokers, der von den Microservices AuthenticationService, ProcessModelStorage, ProcessEngine und ExternalCommunicator genutzt wird. Ein Message Broker ist ein Softwarearchitektur-Pattern, das in der Lage ist, Nachrichten von beliebig vielen Sendern zu empfangen. Des Weiteren bestimmen Message Broker den korrekten Empfänger und leiten die Nachrichten an diesen weiter. Demgemäß ermöglicht der Message Broker die Kommunikation der eben genannten Microservices. Beim Message Broker handelt es sich um eine zentralisierte Komponente, damit die Kontrolle und Verwaltung aller Nachrichten gegeben ist. Infolgedessen werden alle ein- und ausgehenden Nachrichten an den Message Broker gesendet, welcher diese an das richtige Microservice weiterleitet [8].

Für die angegebenen Microservices werden zwei Topics erstellt. Alle eingehenden Nachrichten werden an das Topic IN gesendet. Folglich nimmt das zugewiesene Microservice die Nachrichten entgegen und verarbeitet diese. Die Resultate der Verarbeitung oder andere veröffentlichte Events werden als Nachricht an das zugewiesene Topic OUT gesendet. Microservices können sich für die OUT-Topics anderer Microservices als Subscriber registrieren und dementsprechend auf neue Nachrichten reagieren. Mit diesem Konzept wird das Publish/Subscribe-Pattern umgesetzt. Publish baut eine Verbindung zu einem Topic des Message Brokers auf und veröffentlicht neue Nachrichten. Mittels Subscribe wird eine Subscription für ein Topic definiert, womit die Nachrichten des Topics für den Subscriber verfügbar sind [38].

Aufgrund der Einführung des Message Brokers sowie des Publish/Subscribe-Patterns ist die Kommunikation zwischen den Microservices asynchron. Die asynchrone Kommunikation von Microservices wurde bereits im Abschnitt 2.2.4.2 erläutert. Folglich ist eine lose Kopplung und einfache Skalierbarkeit der einzelnen Microservices gegeben. Zudem ist die Performance eines Message Brokers im Vergleich zu REST deutlich besser. Insofern ist der Aufwand für die Einbindung und Verwaltung einer zusätzlichen Infrastrukturkomponente vernachlässigbar. Das Projekt Spring Cloud Stream ermöglicht die schnelle Einbindung von Message Brokern (u.a. Apache Kafka¹, RabbitMQ²).

¹ <https://kafka.apache.org/>

² <https://www.rabbitmq.com/>

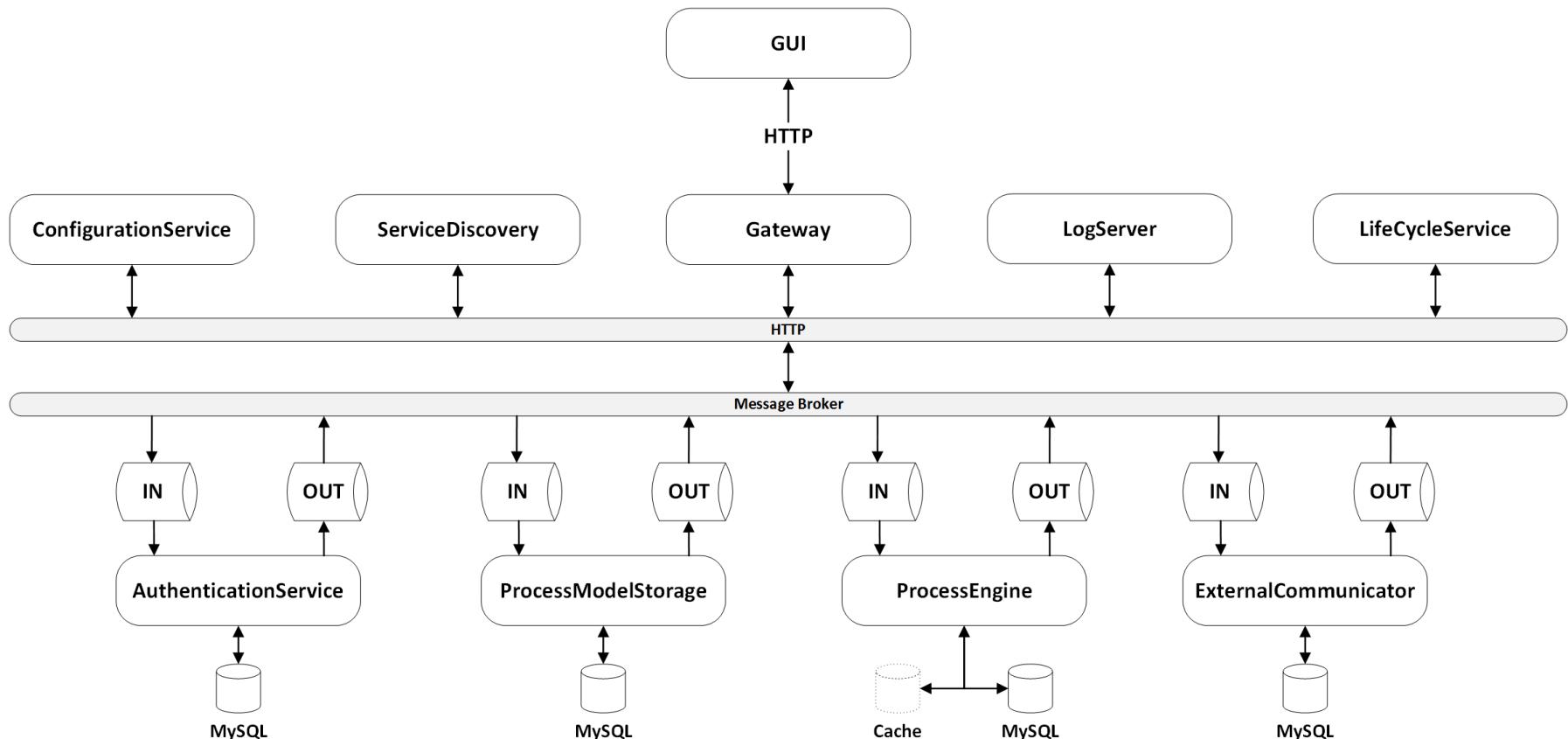


Abbildung 6.1: Verbesserungen für den Microservice-basierten Prototyp

Für die Microservices Gateway, ServiceDiscovery, ConfigurationService sowie LifeCycleService ist keine Inkludierung des Message Brokers notwendig, da es sich hier um vorhandene Spring Cloud-Komponenten handelt, die standardmäßig auf REST, HTTP und JSON basieren. Insofern müssten die bereits vorhandenen und vollständig getesteten Funktionen angepasst werden, damit diese mit einem Message Broker kompatibel sind.

6.1 Neue Microservices

In diesem Abschnitt werden jene Microservices vorgestellt, die im Abschnitt 4.2 noch nicht Bestandteil der Architektur waren.

6.1.1 AuthenticationService

Im derzeitigen Prototypen wird die Authentifizierung aller Anfragen im Microservice Gateway durchgeführt. Vor allem für prototypische Implementierungen ist dieser Ansatz definitiv valide. Dennoch sollten die Authentifizierungsmechanismen in das Microservice AuthenticationService ausgelagert werden. Auf die notwendigen Funktionen dieses Microservices wird hier nicht eingegangen, da diese bereits im Abschnitt 4.2.3.2 veranschaulicht wurden. Alle externen Anfragen an das Gateway werden demgemäß von diesem an das AuthenticationService gesendet, welches den mitgesendeten JWT prüft bzw. die Authentifizierung von Benutzerinnen bzw. Benutzern ermöglicht.

6.1.2 LogServer

In Microservice-basierten Architekturen ist eine zentrale Stelle, an welche sämtliche Log-Dateien zur Auswertung gesendet werden, essentiell. Zipkin ist ein Framework das für den LogServer eingesetzt werden kann. Hierbei werden sogenannte Spans an den LogServer gesendet. Ein Span ist eine zusammenhängende Arbeitseinheit und ist Bestandteil eines Traces. Spans enthalten in der Regel Informationen wie den Start- sowie den Stop-Zeitstempel und zusätzliche Log-Informationen. Weitere Metadaten können einfach hinzugefügt werden. Bei Empfang der ersten Anfrage wird ein Trace mit einem eindeutigen Identifikator gestartet. Ist für die Verarbeitung der Anfrage ein weiteres Microservice notwendig, dann wird eine neue Anfrage an dieses gesendet. Nun wird ein weiterer Span erstellt, der wiederum Bestandteil des erstellten Traces ist. Somit werden die einzelnen, jedoch zusammenhängenden Anfragen an die Microservices nachvollziehbar, da sie einem nachvollziehbaren Trace zugeordnet sind [29].

In Abbildung 6.2 wird die interne Architektur des Microservices LogServer veranschaulicht. Wie bereits dargelegt basiert dieses Microservice auf Zipkin. Die Logs – in Zipkin als Spans bezeichnet – der Microservices werden hierbei mittels HTTP an den Collector gesendet. Der Collector leitet die empfangenen Logs an die Komponente Storage weiter, welche für die Speicherung der Logs zuständig ist. Ursprünglich erfolgte die Speicherung der Daten in einer Casandra-Datenbank, da diese skalierbar ist und ein flexibles Datenbankschema bereitstellt. Mittlerweile werden konfigurierbare Plugins zur Speicherung des Logs angeboten, die die Speicherung u.a. in MySQL- und InMemory-Datenbanken ermöglichen. Bei der Komponente Zipkin Query Service handelt es sich um eine JSON-API. Mithilfe von dieser können Logs aus dem Datenspeicher gelesen werden. Des Weiteren können Suchanfragen an diese API gesendet werden. Die Komponente GUI dient zur grafischen Darstellung der Logs und nutzt

hierfür das Zipkin Query Service. Zudem bietet sie diverse Filter (z.B. Service, Zeitraum, etc.) an, damit die Auswertung des Logs erleichtert wird [42].

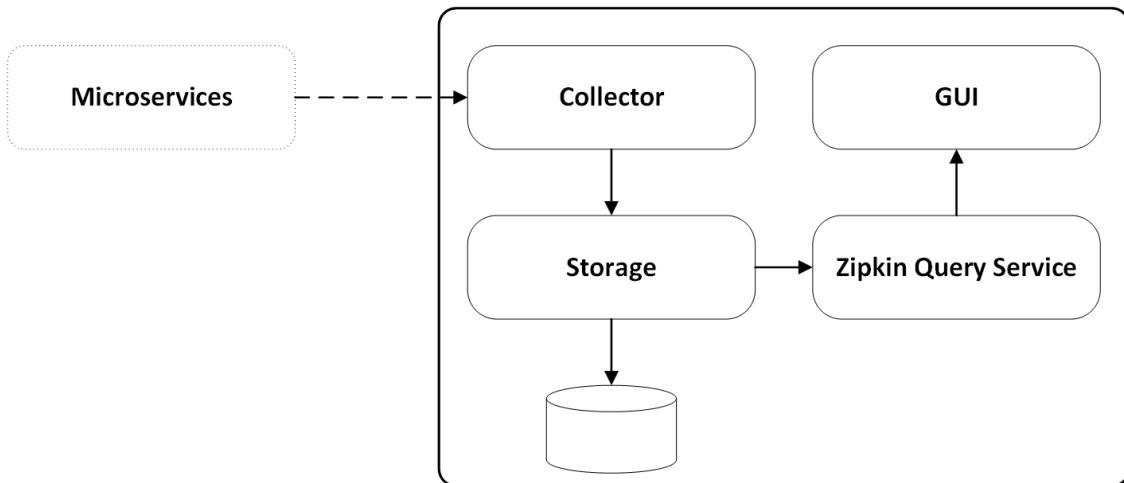


Abbildung 6.2: Architektur des Microservices LogServer (modifiziert übernommen von [42])

Mittels des Spring Cloud-Projekts Spring Cloud Sleuth wird der LogServer in den Microservices integriert. Dazu muss Spring Cloud Sleuth als externes Projekt in der Datei build.gradle der Microservices deklariert werden. Folglich werden die Logs der Microservices per HTTP automatisch an den Collector von Zipkin gesendet. Standardmäßig werden die Zipkin-HTTP-Anfragen an die URI 127.0.0.1:9411 gesendet. In der Konfigurationsdatei application.properties ermöglicht der Parameter spring.zipkin.baseUrl die Änderung der URI [4].

6.1.3 LifeCycleService

Das Microservice LifeCycleService ist für das dynamische Starten bzw. Stoppen von Serviceinstanzen auf Basis derer Auslastung verantwortlich. Mit diesem wird die automatische Skalierung der Microservice-basierten Architektur ermöglicht. Das LifeCycleService, das in Abbildung 6.3 dargestellt wird, besteht aus den folgenden Komponenten [36]:

- Metrics Collector: Diese Komponente ist zuständig für den Empfang sowie die Speicherung der Metriken der einzelnen Serviceinstanzen. Diese Metriken (z.B. Hauptprozessorauslastung, Transaktionen per Minute, etc.) werden vom LifeCycleService aggregiert, um die Auslastung der Serviceinstanzen zu bestimmen.
- Scaling Policies: Hierbei handelt es sich um ein Set von Regeln, welche bestimmen, ob Instanzen der Microservices gestartet bzw. gestoppt werden. Z.B. 90% Hauptprozessorauslastung in den letzten fünf Minuten.
- Decision Engine: Diese Komponente nutzt den Metrics Collector sowie die Scaling Policies und entscheidet demgemäß, ob neue Instanzen gestartet oder gestoppt werden.
- Deployment Rules: Diese legen zusätzliche Regeln fest, die beim Deployment der Microservices zu berücksichtigen sind. Beispielsweise kann festgelegt werden, dass die Instanzen der Services nur auf unterschiedlichen Servern gestartet werden.
- Deployment Engine: Die Komponente ist zuständig für den Start bzw. Stopp – basierend auf den Entscheidungen der Decision Engine – der Serviceinstanzen.

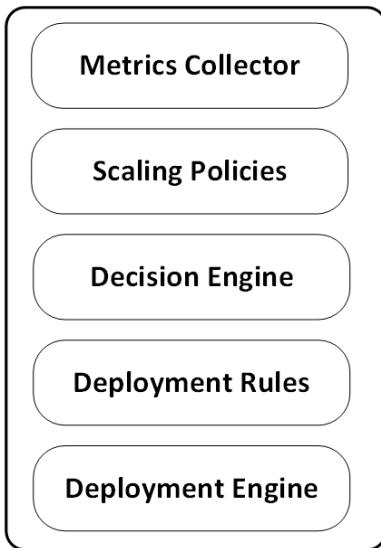


Abbildung 6.3: Architektur des Microservices LifeCycleService (modifiziert übernommen von [36])

Das LifeCycleService führt den Metrics Collector als Hintergrundjob aus, der Informationen zu den Serviceinstanzen vom Microservice ServiceDiscovery bezieht. Der Metrics Collector nutzt das Modul Spring Boot Actuator für die Ermittlung der Metriken der einzelnen Instanzen. Basierend auf den hinterlegten Scaling Policies entscheidet die Decision Engine, ob eine Instanz gestartet oder gestoppt wird. Ist ein Stopp einer Instanz notwendig, dann wird diese benachrichtigt, womit diese heruntergefahren wird. Bei Start einer neuen Serviceinstanz, nutzt die Deployment Engine die deklarierten Deployment Rules, um zu entscheiden, wo die Instanz gestartet wird [36].

6.2 Vorhandene Microservices

In diesem Abschnitt werden für die Microservices des implementierten Prototypen (siehe Abschnitt 4.2) Verbesserungen sowie Erweiterungen vorgestellt. Die Microservices ProcessEngine, ProcessModelStorage und Gateway sind Bestandteil dieses Abschnitts. Folglich werden für die Microservices ExternalCommunicator, ServiceDiscovery sowie ConfigurationService keine Verbesserungen vorgeschlagen.

6.2.1 ProcessEngine

In diesem Abschnitt werden Verbesserungen und Erweiterungen für das Microservice ProcessEngine vorgestellt.

6.2.1.1 Verbesserungen für die Verwendung von Akka

Im derzeitigen Prototypen werden u.a. die Aktoren ProcessSupervisorActor, ProcessActor, UserSupervisorActor sowie UserActor für die Ausführung von Prozessinstanzen eingesetzt (siehe Abschnitt 4.2.5.2). Demgemäß wird für jede Prozessinstanz ein ProcessActor und für jede Benutzerin bzw. für jeden Benutzer ein UserActor erstellt, welche die Nachrichten von den Aktoren ProcessSupervisorActor und UserSupervisorActor empfangen. Dieser Ansatz hat einen großen Nachteil, weil

die Zustandslosigkeit des Microservices ProcessEngine nicht mehr gegeben ist. D.h., dass eine Prozessinstanz nur verarbeitet werden kann, wenn ein aktiver ProcessActor dafür existiert. Dasselbe gilt für alle Anfragen an eine Benutzerin/einen Benutzer, weil ein UserActor vorhanden sein muss. Dies ist vor allem für die Skalierung der ProcessEngine problematisch, da jede Serviceinstanz Aktoren für alle Prozessinstanzen sowie für die Benutzerinnen/Benutzer ausführen muss, damit jede Serviceinstanz in der Lage ist, jeden Prozess zu verarbeiten. Dieser Ansatz ist bei der Ausführung von mehreren parallelen Instanzen des Microservices ProcessEngine nicht sonderlich effizient.

Insofern ist ein Ansatz zu empfehlen, der die Zustandslosigkeit des Microservices garantiert. Dafür können die Tasks, die bereits im Abschnitt 4.2.5.3 vorgestellt wurden, verwendet werden. Dementsprechend muss jede einzelne Operation zur Ausführung von Prozessinstanzen (z.B. Starten von Prozessen, Stoppen von Prozessen, etc.) in einen Task ausgelagert werden, womit die Aktoren ProcessSupervisorActor, ProcessActor, UserSupervisorActor sowie UserActor entfernt werden können, weil diese zur Prozessausführung nicht mehr benötigt werden. Somit ist die Ausführung eines Aktors für jede Prozessinstanz bzw. jeder Benutzerin/jeden Benutzer nicht mehr notwendig, da Tasks zustandslos sind. Folglich können neue Serviceinstanzen der ProcessEngine problemlos gestartet werden.

In Abbildung 6.4 wird dieses neue Konzept dargestellt. In diesem Konzept gibt es nur mehr eine einzige Mailbox für alle Aktoren. In dieser Mailbox werden alle Tasks, welche abzuarbeiten sind, gespeichert. Die Komponente Balancing Dispatcher ist für die Verteilung der Tasks an die TaskWorker zuständig. Der Balancing Dispatcher gibt die Tasks an die untätigen TaskWorker weiter, welche anschließend mit der Verarbeitung der Tasks beginnen. Ist kein untätigter Aktor vorhanden, dann bleibt der Task so lange in der Mailbox, bis wieder ein Aktor für die Verarbeitung zur Verfügung steht. Dieser Ansatz ist sehr skalierbar, da je nach Auslastung neue TaskWorker gestartet bzw. gestoppt werden können [22].

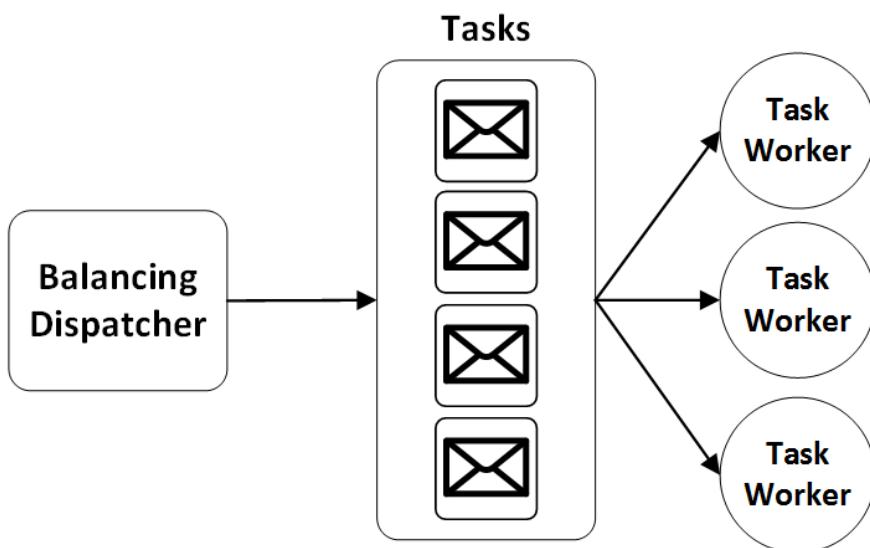


Abbildung 6.4: Balancing Dispatcher für die ProcessEngine (modifiziert übernommen von [22])

6.2.1.2 Strategien für den Empfang von S-BPM-Nachrichten

Im vorgestellten Prototypen gibt es derzeit keine Beschränkungen für den Nachrichtenempfang eines Subjekts. Folglich sind die Konfigurationen und Strategien des Input Pools, welche

im Abschnitt 2.1.2.1 dargelegt wurden, nicht umgesetzt, womit jedes Subjekt beliebig viele Nachrichten empfangen kann. Um Beschränkungen für den Nachrichtenempfang einzuführen, wird das Interface `ReceiveStrategy` vorgeschlagen. Die Methode `canReceive` liefert hierbei `true` zurück, wenn die Nachricht vom Subjekt empfangen werden kann. Ist dies nicht der Fall, dann wird `false` zurückgeliefert und demgemäß konnte die Nachricht nicht ordnungsgemäß an ein anderes Subjekt gesendet werden. Die konkreten Implementierungen, die überprüfen, ob eine Nachricht empfangen werden kann, müssen somit vom Interface `ReceiveStrategy` abgeleitet werden. Beispielsweise kann die Anzahl der Nachrichten im Input Pool eines Subjekts überprüft werden. Zur Kombination von mehreren verschiedenen Strategien, können diese in einem Set eingefügt werden. Beim Senden einer Nachricht wird jede hinterlegte `ReceiveStrategy` überprüft, indem die Methode `canReceive` ausgeführt wird. Nur wenn alle Strategien `true` zurückliefern, war der Sendevorgang erfolgreich.

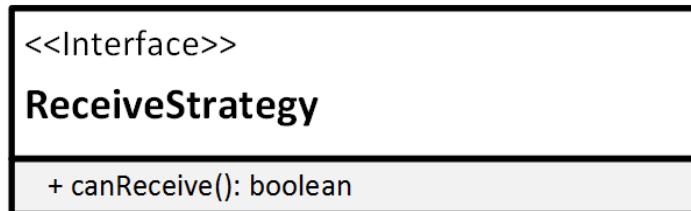


Abbildung 6.5: Klassendiagramm des Interfaces `ReceiveStrategy`

6.2.1.3 Autonome Datenbankmodelle

Wie in Abbildung 6.1 erkennbar ist, ist das Java-Package `Persistence` nicht mehr Bestandteil der Architektur. Dieses Package wird im Prototypen in den Microservices `ProcessModelStorage` und `ProcessEngine` inkludiert, womit diese ein gemeinsames Datenbankschema für die Speicherung und Ausführung von Prozessen nutzen. Grundsätzlich sollte jedes Microservice über einen autonomen Datenspeicher verfügen, weshalb das Java-Package `Persistence` nicht mehr Bestandteil der neuen Architektur ist. Dementsprechend verfügt die `ProcessEngine` über ein eigenes Datenbankschema, in welchem alle notwendigen Tabellen zur Ausführung von Prozessinstanzen enthalten sind. Insofern würde die `ProcessEngine` erst beim Start einer Prozessinstanz das notwendige Prozessmodell direkt vom `ProcessModelStorage` beziehen, das den Ablauf des auszuführenden Geschäftsprozesses vorgibt. Damit nicht bei jedem Start einer Prozessinstanz eine Anfrage an das `ProcessModelStorage` notwendig ist, werden bereits empfangene Prozessmodelle in einem Cache temporär gespeichert. Dafür kann das Framework `cache2k`³ eingesetzt werden, welches die Speicherung von Java-Objekten im Cache ermöglicht.

6.2.2 ProcessModelStorage

Im Abschnitt 6.2.1.3 wurde bereits dargelegt, dass das Java-Package `Persistence` auch aus dem `ProcessModelStorage` entfernt wird. Infolgedessen steht dem `ProcessModelStorage` ein eigenes Datenbankschema zur Speicherung der Prozessmodelle zur Verfügung. Des Weiteren muss eine API geschaffen werden, damit die `ProcessEngine` in der Lage ist, Prozessmodelle vom `ProcessModelStorage` abzurufen.

³ <https://cache2k.org/>

6.2.3 Gateway

Wie im Abschnitt 6.1.1 bereits angeführt wurde, werden sämtliche Authentifizierungsmechanismen aus dem Gateway entfernt und in das neue Microservice AuthenticationService ausgelagert. Das Gateway leitet die Authentifizierungsanfragen an dieses neue Microservice weiter.

7 Vergleich mit einem klassischen Workflow-Management-System

In diesem Kapitel wird ein Softwarearchitekturvergleich zwischen dem vorgestellten Prototypen, welcher auf Microservices basiert und einem klassischen Workflow-Management-System durchgeführt. Die Softwarearchitektur beschreibt die Organisation eines Systems, wie im Abschnitt 2.2 bereits dargelegt wurde. Das klassische Workflow-Management-System, das eine monolithische Softwarearchitektur einsetzt, wird im Abschnitt 7.1 definiert. Für diesen Vergleich wird eine Architekturbewertung für beide genannten Ansätze herangezogen. Mit Hilfe einer Architekturbewertung kann geprüft werden, ob eine Softwarearchitektur den Anforderungen und geforderten Qualitätsmerkmalen entspricht. Laut [33] gibt es zwei verschiedene Ansätze zur Bewertung von Softwarearchitekturen:

- Qualitative Methoden: Fragebögen, Checklisten, Szenarien und Bewertungen basierend auf Erfahrungen sind Bestandteil der qualitativen Methoden. Für die qualitativen Methoden können beliebige Bewertungsfaktoren verwendet werden.
- Quantitative Methoden: Zur quantitativen Architekturbewertung können Simulationen, Metriken sowie mathematische Modelle eingesetzt werden. Damit diese Bewertung durchführbar ist, muss der Sourcecode zur Verfügung stehen.

Aufgrund der Tatsache, dass der Sourcecode eines klassischen Workflow-Management-Systems nicht zugänglich ist, ist eine quantitative Softwarearchitekturbewertung nicht durchführbar. Folglich werden Szenarien für den Vergleich herangezogen, da diese die Bewertung beliebiger Anforderungen unterstützen. Zudem erlauben Szenarien die Bewertung von komplexen Qualitätsmerkmalen [33].

7.1 Klassisches Workflow-Management-System

In Abbildung 7.1 wird die Architektur eines klassischen Workflow-Management-Systems veranschaulicht. In dieser Architektur werden die Prozessmodelle im Process Repository abgelegt. Die Modellierung der Prozessmodelle erfolgt mittels des Moduls Process Definition Tool. Des Weiteren ist eine Zuordnung der Prozessaktivitäten zu den Elementen (z.B. Benutzerinnen bzw. Benutzer, Rollen, etc.) einer Organisationseinheit notwendig, um festzulegen, wer welche Tätigkeiten in einem Prozess durchführen kann. Prozessmodelle können in die Workflow Engine importiert werden, welche die Prozessmodelle in konkrete Prozessinstanzen überführt und folglich die Ausführung von Prozessen ermöglicht. Das Modul Human Interaction Module & Interface stellt den Benutzerinnen bzw. Benutzern eine GUI zur Verfügung, damit diese mit der Workflow Engine interagieren können. Für die Kommunikation mit externen IT-Systemen werden sogenannte Service Calls eingesetzt [25].

Nahezu alle Workflow-Management-Systeme basieren hierbei auf monolithischen Softwarearchitekturen. Dementsprechend sind alle Module Bestandteil einer einzigen großen Anwendung, welche häufig als EAR- oder WAR-Datei bereitgestellt wird. Diese Dateien werden in Anwendungsservern importiert und mithilfe von diesen gestartet.

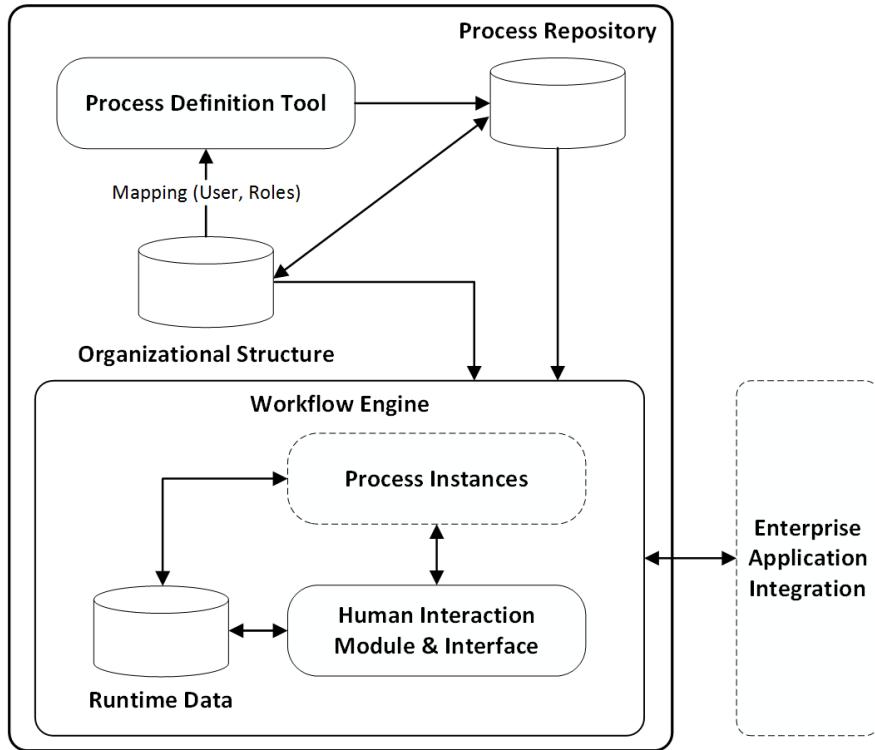


Abbildung 7.1: Architektur des klassischen Workflow-Management-Systems (modifiziert übernommen von [25])

Modul des monolithischen Ansatzes	Microservice	Bemerkung
Process Definition Tool	-	Microservice zur Modellierung von Prozessmodellen grundsätzlich vorhanden, jedoch nicht Bestandteil dieser Arbeit
Process Repository	ProcessModelStorage	
Workflow Engine	ProcessEngine	
Human Interaction Module & Interface	GUI	
Enterprise Application Integration	ExternalCommunicator	
-	Gateway	Kein zentrales Gateway im monolithischen Ansatz vorhanden

Tabelle 7.1: Zuordnung der Module des monolithischen Ansatzes zu Microservices des Prototypen

In Tabelle 7.1 werden die Module des klassischen Workflow-Management-Systems – im Folgenden als monolithischer Ansatz bezeichnet – den Microservices des vorgestellten Prototypen zugeordnet. D.h. zugeordnete Module und Microservices decken gleiche bzw. ähnliche Anforderungen und Aufgabenbereiche ab. Wie bereits dargelegt wurde, sind die Module des Workflow-Management-Systems Bestandteil einer einzigen Anwendung, wohingegen es sich bei den Microservices um unabhängige Services handelt, die miteinander interagieren.

7.2 Szenarien

In diesem Abschnitt werden verschiedene Szenarien für den Softwarearchitekturvergleich zwischen dem monolithischen Ansatz und dem Microservice-basierten Ansatz vorgestellt. Grundsätzlich wird geprüft, wie viel Aufwand notwendig ist, um die Szenarios zu erfüllen. Die folgenden Szenarien wurden gewählt, damit zum einen Standardanforderungen von Softwarearchitekturen, wie z.B. Skalierbarkeit, geprüft werden. Zum anderen wurden auch Szenarien definiert, welche die Softwarearchitekturen hinsichtlich der Anforderungen an ein Workflow-Management-System testen. In jedem Szenario werden die zwei Softwarearchitekturen nach folgendem Schema bewertet:

- **-1:** Das Szenario ist mit der Architektur nicht umsetzbar bzw. die Erfüllung des Szenarios ist nur mit erheblichem Aufwand umsetzbar.
- **0:** Das Szenario kann mit vertretbarem Aufwand erfüllt werden.
- **1:** Für die Erfüllung des Szenarios ist kein bzw. kaum Aufwand notwendig.

7.2.1 Skalierbarkeit

Ein essentieller Geschäftsprozess des Unternehmens wird in das Workflow-Management-System eingebunden. Infolgedessen muss das Workflow-Management-System nun in der Lage sein 100.000 zusätzliche Anfragen von Benutzerinnen bzw. Benutzern pro Tag zu verarbeiten.

Grundsätzlich gibt es zwei verschiedene Varianten, um die zusätzlichen Anfragen zu bewältigen. Zum einen ist das Hinzufügen von zusätzlichen Ressourcen (vertikale Skalierung) möglich. Jedoch muss sichergestellt werden, dass die Software die zusätzlichen Ressourcen auch tatsächlich nutzt. Beim monolithischen Ansatz gibt es nur eine einzelne Anwendung, in welcher alle Module inkludiert sind. Demgemäß kann jedes Modul die zusätzlichen Ressourcen nutzen, obwohl diese eigentlich nur im Modul Workflow Engine – zur Ausführung von Prozessinstanzen – benötigt werden. Aufgrund der engen Kopplung der Module muss jedes Modul sicherstellen, dass es in der Lage ist, die zusätzlichen Ressourcen (z.B. mittels Multithreading) zu nutzen. Beim Microservice-basierten Ansatz ist dies nicht der Fall, da nur dem Microservice ProcessEngine zusätzliche Ressourcen zugewiesen werden und demgemäß nur dieses angepasst werden muss.

Die zweite Möglichkeit ist das Starten von zusätzlichen Serviceinstanzen (horizontale Skalierung). Auch dies ist mit dem monolithischen Ansatz nicht so einfach möglich, da ein Load Balancer, der die Anfragen an die Instanzen weiterleitet, in die Architektur mitaufgenommen werden muss. Aufgrund der Verwendung des zentralen Microservices Gateway, das bereits einen Load Balancer inkludiert und dementsprechend die Anfragen an die verfügbaren

Serviceinstanzen aufteilt, sind in der Microservice-basierten Architektur keine Änderungen notwendig. Demgemäß ist es mit der Verwendung von Microservices deutlich einfacher die Skalierbarkeit von Architekturen sicherzustellen, wohingegen im monolithischen Ansatz einige Anpassungen notwendig sind.

Bewertung:

Monolithischer Ansatz: -1
Microservice-basierter Ansatz: 1

7.2.2 Erweiterbarkeit

Der Aufbau und die Struktur der Business Objects kann hochflexibel sein, wofür eine spezielle Datenbank notwendig ist. Infolgedessen entschieden sich die Entwicklerinnen bzw. Entwickler für die Verwendung einer No-SQL-Datenbank, die die Speicherung von flexiblen Objekten unterstützt.

Beim Einsatz eines Monolithen muss ein neues Modul implementiert werden, welches die Persistierung der Business Objects in einer No-SQL-Datenbank vornimmt. Weiters muss ein neues ORM-Framework für No-SQL-Datenbanken integriert werden. Das neue Modul muss in die Workflow Engine eingebunden werden, da diese die Speicherung von Business Objects vornimmt. Folglich entstehen neue Abhängigkeiten, womit die Wartbarkeit der Software erschwert wird. Bei der Verwendung der Microservice-basierten Architektur ist die Entwicklung eines neuen Microservices, das die Speicherung von Business Objects in einer No-SQL-Datenbank ermöglicht, empfehlenswert. Mittels klar definierter API werden die Funktionen zum Abrufen oder Speichern von Business Objects für andere Microservices zur Verfügung gestellt. Diese API wird von der ProcessEngine aufgerufen. Grundsätzlich ist die Umsetzung des Szenarios mit beiden Ansätzen möglich, aber aufgrund der neuen Abhängigkeiten beim monolithischen Ansatz schneidet dieser etwas schlechter ab.

Bewertung:

Monolithischer Ansatz: 0
Microservice-basierter Ansatz: 1

7.2.3 Verfügbarkeit

Das Workflow-Management-System ist ein signifikantes IT-System des Unternehmens und wird in nahezu allen Abteilungen genutzt. Demgemäß wird eine Verfügbarkeit von 99,9 % (maximal erlaubte Ausfallzeit in Stunden: ca. 9) verlangt.

Das große Problem der monolithischen Anwendung ist, dass bei Fehlschlag eines einzigen Moduls, dies Auswirkungen auf das Gesamtsystem hat. Eventuell kann eine einzige Komponente sogar das Gesamtsystem zum Absturz bringen. Wie in Abschnitt 7.2.1 bereits erläutert wurde, ist das Bereitstellen von mehreren Anwendungsinstanzen nicht ohne weiteres möglich. Ein Vorteil von Microservices ist deren Unabhängigkeit. D.h. bei Absturz eines Microservices

sind die anderen Microservices trotzdem noch im Stande weiterzuarbeiten. Wird dieses abgestürzte Microservice jedoch von anderen benötigt, muss sichergestellt sein, dass für die anderen Microservices Funktionen deklariert sind, die im Falle des Nichterreichens des abgestürzten Microservices ausgeführt werden. Des Weiteren ist es bei Microservices einfacher möglich mehrere aktive Serviceinstanzen bereitzustellen, womit die Operationen einer abgestürzten Instanz von einer anderen aktiven Instanz übernommen werden können.

Bewertung:

Monolithischer Ansatz: -1

Microservice-basierter Ansatz: 1

7.2.4 Antwortzeiten

Für die Benutzerinnen bzw. Benutzer ist es wichtig, dass ihre Anfragen an das Workflow-Management-System schnell abgearbeitet werden. Infolgedessen werden Antwortzeiten von maximal 500 Millisekunden akzeptiert.

Mit einem leistungsfähigen Anwendungsserver können monolithische Anwendungen definitiv sehr niedrige Antwortzeiten erreichen, da alle Module Bestandteil einer einzigen Anwendung sind. Somit ist die Kommunikation zwischen den Modulen sehr schnell zu bewältigen, da z.B. keine REST-Anfragen an externe Services gesendet werden müssen. Bei Microservice-basierten Architekturen ist es sehr wahrscheinlich, dass bei einer Anfrage von Benutzerinnen bzw. Benutzern mehrere Microservices involviert sind, um die Anfrage korrekt zu verarbeiten. Basiert die Kommunikation zwischen den Microservices beispielsweise auf REST sowie JSON und sind mehrere Microservices involviert, dann ist es sehr unwahrscheinlich, dass die niedrigen Antwortzeiten des Monolithen erreicht werden. Um dies zu verbessern, können Message Broker eingesetzt werden. Dies ist wiederum sehr aufwändig, womit der monolithische Ansatz im Vergleich zu Microservices definitiv Vorteile bei Antwortzeiten bringt.

Bewertung:

Monolithischer Ansatz: 1

Microservice-basierter Ansatz: -1

7.2.5 Deployment

Da das Workflow-Management-System ein essentieller Bestandteil des Unternehmens ist, ist es von großer Bedeutung, dass das Deployment schnell abgewickelt wird und somit den laufenden Betrieb nicht stört.

Der Vorteil des monolithischen Ansatzes ist, dass nur eine einzelne EAR- oder WAR-Datei deployt werden muss, um die gesamte Anwendung zu starten. Dies ist einfach und sogar manuell umsetzbar. Ein Nachteil ist jedoch, dass auch bei kleinen Bugs die gesamte Anwendung mit allen Modulen neu gestartet werden muss. Zudem kommt es zu Ausfallzeiten während des Deployments, wenn der parallele Betrieb der Anwendung nicht unterstützt wird. Bei einer

großen Anzahl von Microservices ist ein manuelles Deployment kaum zu bewerkstelligen, weshalb das Deployment automatisiert und koordiniert werden muss. Allerdings erlauben Microservices den parallelen Betrieb von Serviceinstanzen, womit die Ausfallzeiten während eines Deployments gering gehalten werden können, indem die alte Serviceinstanz erst nach dem Deployment und Start der neuen Serviceinstanz heruntergefahren wird. Ein weiterer Vorteil von Microservices ist, dass die gesamte Anwendung inklusive deren Ausführungsumgebung in einem einzelnen JAR verpackt ist. Demgemäß ist kein Anwendungsserver notwendig und der Start der JAR-Datei reicht aus. Somit gibt es bei beiden Ansätzen Vorteile, aber auch Nachteile.

Bewertung:

Monolithischer Ansatz:	0
Microservice-basierter Ansatz:	0

7.2.6 Verantwortlichkeiten

Der Geschäftsführung des Unternehmens ist es wichtig, dass es klare Verantwortlichkeiten bei der Entwicklung des Workflow-Management-Systems gibt. Dementsprechend soll jedes Team für einen klar abgegrenzten Themenbereich zuständig sein, um Expertinnen bzw. Experten für diese Bereiche aufzubauen.

Dieses Anforderung ist mit dem monolithischen Ansatz häufig schwierig zu erfüllen. Es gibt zwar einzelne Module im Monolithen, jedoch sind diese eng gekoppelt und dies führt zu Überschneidungen bei den Themenbereichen, womit Bereiche in den Verantwortungsbereich von mehreren Teams fallen. Folglich ist eine klare Festlegung von Verantwortlichkeiten nicht immer möglich. Bei Microservices ist dies nicht der Fall, da jedes Team genau ein Microservice übernimmt und für dieses zuständig ist. Aufgrund der Autonomie der Microservices gibt es keine Überschneidung von Bereichen und jedes Team kann selbstständig arbeiten. Einzig bei der Definition der APIs sind mehrere Teams involviert. Dennoch gibt es hier klare Vorteile bei der Verwendung von Microservice-basierten Architekturen.

Bewertung:

Monolithischer Ansatz:	0
Microservice-basierter Ansatz:	1

7.2.7 Flexibilität bei neuen Anforderungen

Der Geschäftsführung müssen ständig aktuelle statistische Auswertungen bereitgestellt werden, wodurch sich die Entwicklerinnen bzw. Entwickler für die Verwendung der Programmiersprache Go entschließen, da diese hochperformant ist.

Beim monolithischen Ansatz gibt es zwei Möglichkeiten für die Erreichung dieser Anforderung. Zum einen besteht die Möglichkeit ein neues Modul, welches die Auswertungen bereitstellt, zu entwickeln. Der Nachteil dieser Variante ist allerdings, dass die verwendete

Programmiersprache des Monolithen eingesetzt werden muss. Ist der Monolith beispielsweise in Java entwickelt, gibt es keine Möglichkeit, dass Go eingesetzt werden kann. Die zweite Möglichkeit sieht die Entwicklung einer externen Anwendung vor, die die geforderten Auswertungen vornimmt. Folglich sind beide Möglichkeiten nicht ideal. Bei der Verwendung der Microservice-basierten Architektur ist die Bereitstellung von statistischen Auswertungen, basierend auf Go, leicht umzusetzen. Hierfür wird ein neues Microservice – implementiert mit der Programmiersprache Go – erstellt. Aufgrund der vollständigen Unabhängigkeit von Microservices ist der Einsatz von verschiedenen Programmiersprachen und Technologien leicht möglich.

Bewertung:

Monolithischer Ansatz: -1

Microservice-basierter Ansatz: 1

7.2.8 Datenbanktransaktionsmanagement

Damit die Konsistenz der Daten immer gewährleistet ist, ist die Notwendigkeit eines Datenbanktransaktionsmanagements im Workflow-Management-System gegeben.

Beim monolithischen Ansatz ist das Datenbanktransaktionsmanagement deutlich leichter umsetzbar, da Frameworks, wie z.B. Spring dieses automatisch inkludiert. Dementsprechend wird bei einer Anfrage eine Datenbanktransaktion gestartet, die über alle Module hinweggeht. Schlägt eine Operation in den Modulen fehl, dann wird die gesamte Datenbanktransaktion zurückgestellt. Erst bei vollständiger Abarbeitung aller Operationen wird die Datenbanktransaktion in der Datenbank festgeschrieben. Folglich wird die Konsistenz der Daten sichergestellt.

Bei der Microservice-basierten Architektur kann ebenso Spring für das Datenbanktransaktionsmanagement eingesetzt werden. Jedoch stellt das Datenbanktransaktionsmanagement nur die Datenkonsistenz eines einzelnen Microservices sicher. D.h. wenn die Operationen des ersten Microservices erfolgreich abgeschlossen und festgeschrieben wurden, jedoch die Operationen im zweiten Microservice fehlschlagen, müssen auch die bereits festgeschriebenen Operationen des ersten Microservices zurückgesetzt werden, um die Konsistenz der Daten sicherzustellen. Infolgedessen ist das Datenbanktransaktionsmanagement in der Microservice-basierten Architektur deutlich komplexer.

Bewertung:

Monolithischer Ansatz: 1

Microservice-basierter Ansatz: -1

7.2.9 Monitoring

Um die Analyse von Bugs zu ermöglichen, müssen entsprechende Log-Dateien für die Entwicklerinnen bzw. Entwickler zur Verfügung stehen.

Im monolithischen Ansatz ist der Einsatz eines Log-Frameworks leichter, da Operationen zwar in verschiedenen Modulen, jedoch in derselben Anwendung ausgeführt werden. Folglich gibt es eine zentrale Log-Datei, womit die Analyse erleichtert wird. Bei der Verwendung von Microservices erstellt jedes Service eine einzelne Log-Datei. Infolgedessen ist die Analyse von Bugs deutlich schwieriger, wenn Operationen unterschiedliche Microservices nutzen. Wie in Abschnitt 6.1.2 bereits erläutert wurde, gibt es Frameworks, die die Nachvollziehbarkeit von verteilten Log-Dateien ermöglichen. Demgemäß bedarf es der Inkludierung eines zusätzlichen Frameworks und somit ist das Monitoring aufwändiger zu bewerkstelligen, als bei Verwendung des monolithischen Ansatzes.

Bewertung:

Monolithischer Ansatz:	1
Microservice-basierter Ansatz:	0

7.2.10 Import und Ausführung bestehender Prozessmodelle

Der Geschäftsführung ist es wichtig, dass bereits bestehende Business Process Model and Notation (BPMN)¹-Prozessmodelle in das Workflow-Management-System importiert und ausgeführt werden können, damit bereits getätigte Investitionen nicht verloren gehen.

Damit bestehende BPMN-Prozessmodelle importiert und ausgeführt werden können, muss das jeweilige Workflow-Management-System in der Lage sein, diese zu importierenden Prozessmodelle in der Datenbank in einer Art und Weise zu persistieren, dass das Workflow-Management-System diese Prozessmodelle ausführen kann. Im Fall der Microservice-basierten Architektur müssen BPMN-Prozessmodelle in subjekt-orientierte Prozessmodelle transformiert werden, da die internen Architekturen – im Besonderen die Microservices ProcessEngine und ProcessModelStorage – auf Basis von S-BPM konzipiert und umgesetzt wurden. Dementsprechend sind die internen Architekturen und Konzepte der Microservices nicht unabhängig von der Modellierungssprache. Jedoch unabhängig von der Modellierungssprache ist die vorgestellte Gesamtarchitektur (siehe Abbildung 4.1) des Microservice-basierten Ansatzes. Infolgedessen müssen die autonomen Microservices, welche die Speicherung (ProcessModelStorage) sowie die Ausführung (ProcessEngine) von Prozessen vornehmen, vorhanden sein. Für die Gesamtarchitektur ist es hierbei nicht relevant, welche Modellierungssprache die eben genannten Microservices unterstützen, womit die Gesamtarchitektur unabhängig von der eingesetzten Modellierungssprache betrachtet werden kann.

Wie bereits dargelegt ist die Transformation von BPMN-Prozessmodellen in subjekt-orientierte Prozessmodelle eine mögliche Vorgangsweise, um die vorhandenen Prozessmodelle zu importieren und ausführbar zu machen. Ein Nachteil dieser Transformation ist allerdings, dass nicht alle Konzepte von BPMN in S-BPM vorhanden sind, womit der Import von BPMN-Prozessmodellen mittels Transformation nur eingeschränkt möglich ist und definitiv nicht für alle Prozessmodelle funktionieren wird. Wird die Transformation trotzdem als adäquates Mittel gesehen, dann muss ein neues Microservice konzipiert werden, das die Transformation der BPMN-Prozessmodelle vornimmt. Für den monolithischen Ansatz trifft

¹ Etablierter Standard der Geschäftsprozessmodellierung [6]

dies auch zu. Demnach muss ein neues Modul erzeugt werden, welches die Transformation von Prozessmodellen übernimmt. Auch die Module des Monolithen basieren auf der eingesetzten Modellierungssprache, weshalb eine Transformation notwendig ist. Des Weiteren ist auch die Gesamtarchitektur von der Modellierungssprache abhängig, da die Module gegenseitige Abhängigkeiten erzeugen und eng gekoppelt sind, womit dies Einfluss auf die Gesamtarchitektur des Monolithen hat.

Aufgrund der Nachteile der Transformation wäre eine parallele Ausführung von S-BPM- sowie BPMN-Prozessen die bessere Lösung. In der Microservice-basierten Architektur müssen dafür zwei neue Microservices – ProcessModelStorage und ProcessEngine zur Speicherung und Ausführung von BPMN-Prozessen – entwickelt und eingebunden werden. Zur Inkludierung in die Gesamtarchitektur muss das Gateway angepasst werden, damit die Anfragen an die richtigen Microservices weiterdelegiert werden. Die anderen bereits vorhandenen Microservices müssen aufgrund der Autonomie der einzelnen Services nicht adaptiert werden. Beim monolithischen Ansatz müssen neue Module zur Persistierung und Ausführung von BPMN-Prozessen umgesetzt und eingebunden werden. Folglich würden neue Abhängigkeiten zwischen den einzelnen Modulen entstehen und die Gesamtarchitektur verkomplizieren sowie die Wartbarkeit und Erweiterbarkeit reduzieren. Aufgrund dessen wird der monolithische Ansatz schlechter als der Microservice-basierte Ansatz bewertet. Zudem muss erwähnt werden, dass die Umsetzung der Transformierung von BPMN-Prozessen und die parallele Ausführung von S-BPM- sowie BPMN-Prozessen auch mit der Microservice-basierten Architektur durchaus komplex ist.

Bewertung:

Monolithischer Ansatz: -1

Microservice-basierter Ansatz: 0

7.2.11 Standortübergreifende Prozessausführung

Das Unternehmen ist an mehreren Standorten vertreten. Infolgedessen muss die standortübergreifende Ausführung von Prozessen mit dem Workflow-Management-System möglich sein.

Grundsätzlich gibt es zwei Möglichkeiten zur Erfüllung dieses Szenarios. Zum einen kann das Workflow-Management-System an einer zentralen Stelle (z.B. Server, der für alle Standorte verfügbar ist oder Cloud) für alle Standorte bereitgestellt werden. Die einzelnen Standorte greifen über die jeweilige Netzwerkverbindung auf das Workflow-Management-System zu. Hierfür sind sowohl für den monolithischen als auch für den Microservice-basierten Ansatz keinerlei Adaptionen für die Erfüllung des Szenarios notwendig.

Bei der zweiten Möglichkeit verfügt jeder Standort über eine eigene IT-Infrastruktur, wo das Workflow-Management-System autonom ausgeführt wird. Dementsprechend benutzt jeder Standort eine eigenständige Datenbank. D.h. zur Erfüllung des Szenarios muss ein Datenbankabgleich der Datenbanken der einzelnen Standorte durchgeführt werden, damit die autonomen Workflow-Management-Systeme über den selben Datenstand verfügen und folglich die standortübergreifende Ausführung von Prozessen ermöglichen. Mit einer zentralen Datenbank, wo alle Standorte darauf zugreifen, wäre der zweite Ansatz leichter zu bewerkstelligen.

Folglich verfügen alle Standorte über ein eigenständiges Workflow-Management-System, die jedoch alle die selbe zentrale Datenbank und somit den selben Datenstand nutzen. Insofern besteht bei der Erfüllung dieses Szenarios kein Unterschied zwischen dem Monolithen und dem Microservice-basierten Ansatz.

Bewertung:

Monolithischer Ansatz:	0
Microservice-basierter Ansatz:	0

7.2.12 Unternehmensübergreifende Prozesse

Eine essentielle Anforderung an das Workflow-Management-System ist die Einbindung von Workflow-Management-Systemen anderer Unternehmen, um unternehmensübergreifende Prozesse zu ermöglichen.

Zur Erfüllung dieser Anforderung muss ein Kommunikationsmechanismus geschaffen werden, der das interne Workflow-Management-System mit dem eines anderen Unternehmens – dem externen Workflow-Management-System – verbindet. Im Microservice-basierten Ansatz wurde hierfür das Microservice ExternalCommunicator (siehe Abschnitt 4.2.6) geschaffen, das die Einbindung von externen Subjekten ermöglicht. Dies stellt u.a. die Verbindung zu externen IT-Systemen und folglich auch externen Workflow-Management-Systemen bereit. Des Weiteren unterstützt dieses Microservice verschiedene Datenformate (im derzeitigen Prototypen JSON sowie XML) und Dateiübertragungsprotokolle zur Kommunikation mit externen IT-Systemen. Dieser Ansatz ist deshalb flexibel anpassbar und leicht erweiterbar.

Auch der monolithische Ansatz ermöglicht die Einbindung von externen IT-Systemen, jedoch basierend auf Service Calls (siehe Abschnitt 7.1). Im Vergleich zum Microservice-basierten Ansatz ist der monolithische Ansatz jedoch nicht so flexibel und schwierig erweiterbar, womit der monolithische Ansatz zur Anbindung an andere Workflow-Management-Systeme Einschränkungen unterworfen ist, da beispielsweise die Einbindung von neuen Datenformaten schwierig umzusetzen ist. Demnach ist die Unterstützung von unternehmensübergreifenden Prozessen mit dem Microservice-basierten Ansatz besser umgesetzt.

Bewertung:

Monolithischer Ansatz:	0
Microservice-basierter Ansatz:	1

7.3 Gesamtbewertung

In der Tabelle 7.2 ist die Gesamtbewertung des Softwarearchitekturvergleichs zwischen dem monolithischen Ansatz und dem Microservice-basierten Ansatz veranschaulicht. Auf eine Gewichtung der verschiedenen Szenarien wurde bewusst verzichtet, weil diese subjektiv wäre. Der Architekturvergleich zeigt, dass die Microservice-basierte Architektur für Workflow-Management-Systeme besser geeignet ist, als der monolithische und häufiger

eingesetzte Ansatz, weil die beschriebenen Szenarien mit Microservices mit weniger Aufwand umsetzbar bzw. bereits umgesetzt sind. Zum einen werden Standardanforderungen von Softwarearchitekturen mittels dem Microservice-basierten Ansatz besser umgesetzt. Skalierbarkeit, Erweiterbarkeit sowie Verfügbarkeit sind essentielle Anforderungen an Softwarearchitekturen und sind mit Microservices deutlich einfacher erreichbar. Zudem ist die Microservice-basierte Architektur flexibler und folglich problemlos änderbar. Auch die Vergabe von klar definierten Verantwortlichkeiten für diverse Aufgabenbereiche ist mit Microservices leichter umzusetzen, da Microservices autonom sind und jedes Team unabhängig an einem Service arbeiten kann. Dennoch gibt es Punkte, wie z.B. Datenbanktransaktionsmanagement, Antwortzeiten und Monitoring, die mit dem monolithischen Ansatz simpler umsetzbar sind.

Des Weiteren sind typische Anforderungen an ein Workflow-Management-System leichter mit dem Microservice-basierten Ansatz zu erfüllen. Insbesondere der Import und die Ausführung von bestehenden Prozessmodellen ist mit Microservices einfacher umsetzbar, da die Gesamtarchitektur unabhängig von der Modellierungssprache ist. Aufgrund der Unabhängigkeit der einzelnen Microservices können die Services zur Speicherung und Ausführung von Prozessen ohne Schwierigkeiten ersetzt werden. Beim monolithischen Ansatz ist eine solche Unabhängigkeit und folglich der Austausch von einzelnen Modulen nur mit großem Aufwand erreichbar. Auch die Integration von unternehmensübergreifenden Prozessen ist mit Microservices einfacher möglich. Insgesamt hat der Softwarearchitekturvergleich jedenfalls ergeben, dass der Microservice-basierte Ansatz für Workflow-Management-Systeme besser geeignet ist als der monolithische Ansatz.

Szenario	Monolithischer Ansatz	Microservice-basierter A.
Skalierbarkeit	-1	1
Erweiterbarkeit	0	1
Verfügbarkeit	-1	1
Antwortzeiten	1	-1
Deployment	0	0
Verantwortlichkeiten	0	1
Flexibilität bei neuen Anforderungen	-1	1
Datenbanktransaktionsmanagement	1	-1
Monitoring	1	0
Import und Ausführung bestehender Prozessmodelle	-1	0
Standortübergreifende Prozessausführung	0	0
Unternehmensübergreifende Prozesse	0	1
Gesamt	-1	4

Tabelle 7.2: Gesamtbewertung des Softwarearchitekturvergleichs

8 Fazit

Diese Masterarbeit behandelte die Fragestellung, wie eine Microservice-basierte Architektur zur Speicherung und Ausführung von subjekt-orientierten Prozessen aufgebaut sein kann und welche Vor- bzw. Nachteile diese Architektur gegenüber von monolithischen Architekturen hat. Zu diesem Zweck wurde aufgezeigt, dass es sich bei S-BPM um eine Notation zur Beschreibung und Ausführung von Geschäftsprozessen handelt, welche die Subjekte (Prozessbeteiligten) eines Prozesses in den Mittelpunkt stellt. Zudem wurden die Anforderungen an ein Workflow-Management-System, das S-BPM unterstützt, herausgearbeitet. Insbesondere die Skalierbarkeit eines solchen Systems ist eine der wichtigsten Anforderungen, damit eine beliebig große Anzahl von Prozessinstanzen gleichzeitig verarbeitet werden kann. Softwarearchitekturen, die auf Microservices basieren, erleichtern den Aufbau von skalierbaren Systemen, weshalb sie für den Prototypen eingesetzt wurden. Bei Microservices handelt es sich um kleine autonome Services, die für genau einen Aufgabenbereich zuständig sind. Zudem kann jedes Microservice einen eigenen Technologiestack (z.B. Programmiersprache, Datenbank) nutzen. Microservices enthalten all jene Komponenten, die zur Ausführung des Services benötigt werden. Folglich ist kein zusätzlicher Anwendungsserver zur Ausführung notwendig.

Ein weiterer Bestandteil dieser Arbeit war die Vorstellung eines Datenbankmodells, das die Speicherung von subjekt-orientierten Prozessen ermöglicht und somit eine essentielle Komponente des vorgestellten Prototypen ist. Das Datenbankmodell besteht aus zwei wesentlichen Komponenten. Zum einen wird das Prozessmodell gespeichert, das die beteiligten Subjekte, deren Verhalten, den Aufbau der Business Objects sowie die Interaktion zwischen den Subjekten definiert. Zum anderen wird mit diesem Datenbankmodell die Möglichkeit geschaffen, Prozessinstanzen, die konkrete Ausführungen von Prozessmodellen sind, zu speichern. Mit dem ORM-Framework Hibernate wurde das Datenbankmodell in den Prototypen integriert.

Als Basistechnologie wurde für den Microservice-basierten Prototypen Spring Boot verwendet, da dieses Framework zahlreiche Funktionen, umfangreiche Konfigurationsmöglichkeiten sowie die nahtlose Integration von Hibernate ermöglicht. Des Weiteren wurde Spring Cloud, welches auf Spring Boot aufbaut, eingesetzt, das weitere, für Microservicearchitekturen nützliche Komponenten (z.B. zentrales Konfigurationsmanagement, Service Registry, Routing, etc.), anbietet. Diese vorhandenen Komponenten wurden genutzt, um ein typisches Microservice-Ökosystem für den Prototypen aufzubauen. Dementsprechend konnte gezeigt werden, dass der Aufbau eines Ökosystems für Microservices mit den beiden Frameworks einfach möglich ist.

Die vorgestellte Microservice-basierte Architektur zur Speicherung und Ausführung von subjekt-orientierten Prozessen besteht aus insgesamt sieben verschiedenen Microservices. Beim Start neuer Microserviceinstanzen registrieren sich diese bei der ServiceDiscovery, womit diese Instanzen nun auch für andere Microservices auffindbar sind. Zudem bezieht jedes Microservice beim Start die zugehörigen Konfigurationsdateien vom ConfigurationService, in welchem die Konfigurationsdateien zentral hinterlegt sind.

Diese beiden genannten Services sind typische Services eines Microservice-Ökosystems und Standardkomponenten von Spring Cloud. Das Microservice ProcessModelStorage ist für den Import von Prozessmodellen verantwortlich. Im derzeitigen Prototypen werden nur Prozessmodelle basierend auf standard-passont unterstützt. Der Aufgabenbereich des Microservices ProcessEngine liegt in der Ausführung von subjekt-orientierten Prozessen. Dementsprechend werden die Prozessmodelle, welche mittels ProcessModelStorage importiert werden, in Prozessinstanzen überführt. Für die Ausführung der Prozessinstanzen wurde das Akka-Framework eingesetzt. Akka, welches für Java und Scala verfügbar ist, ermöglicht die Integration des Aktorenmodells, wo alle Entitäten als unabhängige Komponenten modelliert werden, die nur bei empfangenen Nachrichten reagieren. Folglich ermöglicht Akka den Aufbau von skalierbaren und verteilten Systemen, womit dieses Framework hervorragend in Microservice-basierte Architekturen integriert werden kann. Da nur fünf Symbole zur Modellierung von subjekt-orientierten Prozessen notwendig sind, sind die Microservices ProcessModelStorage und ProcessEngine sehr leichtgewichtig. Mit anderen Notationen, die deutlich mehr Symbole zur Modellierung benötigen, wäre dies kaum möglich. Im Zuge dieser Masterarbeit wurde der ExternalCommunicator für die Einbindung von externen Subjekten geschaffen. Hierbei wird der Aufbau von Nachrichten mittels Konfigurationen deklariert. Weiters werden im derzeitigen Prototypen JSON- sowie XML-basierende Nachrichten unterstützt. Das Microservice GUI stellt den Benutzerinnen bzw. Benutzern eine grafische Oberfläche bereit, damit diese die Komponenten des Prototypen bedienen können. Die GUI sendet alle Anfragen an das Gateway, welches diese Anfragen an die anderen Microservices des Prototypen weiterleitet. Darüber hinaus stellt das Gateway alle Authentifizierungsmechanismen bereit.

Mithilfe dieses Prototypen wurde eine erste Microservice-basierte Architektur zur Speicherung und Ausführung von subjekt-orientierten Prozessen bereitgestellt, welche bereits ein breites Spektrum an möglichen S-BPM Funktionen abdeckt. In einem Szenario-basierten Softwarearchitekturvergleich mit einem klassischen monolithischen Workflow-Management-System konnte aufgezeigt werden, dass die Microservice-basierte Architektur präferiert werden sollte. Insbesondere in den Punkten Skalierbarkeit, Erweiterbarkeit sowie Verfügbarkeit konnte die Microservice-basierte Architektur deutlich besser abschneiden. Des Weiteren ist die Ausführung von unternehmensübergreifenden Prozessen sowie der Import und die Ausführung von bereits bestehenden Prozessmodellen mit Microservices leichter umsetzbar. Dennoch bringt die Einführung von Microservices auch neue Herausforderungen mit sich, da u.a. das Deployment von vielen Services schwieriger ist und folglich automatisiert werden muss. Auch das Datenbanktransaktionsmanagement ist mit Microservices deutlich schwieriger umzusetzen und noch nicht vollständig im Prototypen inkludiert.

Grundsätzlich wurde der Prototyp so konzipiert, dass neue Microservices und somit neue Funktionen ohne großen Aufwand in die Architektur inkludiert werden können. Zudem können bereits vorhandene Microservices leicht ersetzt werden. Einzig die definierten APIs müssen eingehalten werden, um die korrekte Kommunikation der Services sicherzustellen. Dennoch gibt es noch großes Potential für Verbesserungen des Prototypen. Beispielsweise nutzen die Microservices ProcessModelStorage und ProcessEngine dasselbe Datenbankschema. Grundsätzlich sollte jedes Microservice über ein eigenes Schema verfügen, damit Abhängigkeiten der Microservices vermieden werden, womit dies demgemäß verbessert werden muss. Des Weiteren kann die Verwendung von Akka in der ProcessEngine verbessert werden, indem die Zustandslosigkeit sichergestellt wird.

Anhand des Softwarearchitekturvergleichs konnte aufgezeigt werden, dass sowohl der Microservice-basierte Ansatz als auch der monolithische Ansatz grundsätzlich die gleichen Aufgabenbereiche abdecken. Dementsprechend sind Funktionen notwendig, die den Import, die Speicherung sowie die Ausführung von Prozessen bereitstellen. Zudem sind Schnittstellen zu externen IT-Systemen zur Integration von diesen notwendig. In der Microservice-basierten Architektur werden diese Funktionen mittels autonomes Services bereitgestellt. Die Gesamtheit der Services bildet die Gesamtarchitektur. Beim monolithischen Ansatz werden die geforderten Funktionen in eng gekoppelten Modulen implementiert und als einzelne Anwendung deployt. Infolgedessen haben die einzelnen Module und folglich die eingesetzte Modellierungssprache Einfluss auf die Gesamtarchitektur des Monolithen. Bei der Microservice-basierten Architektur ist dies nicht der Fall, da sich die Gesamtarchitektur aus unabhängigen Microservices zusammensetzt. D.h. die Gesamtarchitektur kann unabhängig von der Modellierungssprache betrachtet werden. Allerdings hat die Modellierungssprache sehr wohl Einfluss auf die interne Architektur und die Konzepte der Microservices zur Speicherung und Ausführung von Prozessen, die im vorgestellten Prototypen auf S-BPM basieren. Auch andere Modellierungssprachen können in dieser Gesamtarchitektur inkludiert werden. Beispielsweise kann die Speicherung und Ausführung von BPMN-Prozessen mit derselben Architektur umgesetzt werden. Einzig die Services zur Speicherung und Ausführung müssen angepasst oder ersetzt werden, damit diese BPMN unterstützen. Allerdings ist erwähnenswert, dass die Microservices zur Speicherung und Ausführung von BPMN-Prozessen komplexer wären, als jene, welche die Unterstützung von S-BPM bereitstellen. Der Grund dafür ist die große Anzahl an verschiedenen Modellierungssymbolen und Konzepten in BPMN, welche in den Microservices implementiert werden müssten. Demgemäß werden Speicherungs- und Ausführungsumgebungen für S-BPM im Vergleich zu BPMN immer leichtgewichtiger sein.

Insgesamt lässt sich hieraus der Schluss ziehen, dass Microservice-basierte Architekturen für Workflow-Management-System besser geeignet sind, als monolithische Ansätze. Dies trifft jedoch nicht nur für Workflow-Management-Systeme, sondern auch für andere Geschäftsbereiche zu. Langfristig werden sich Microservice-basierte Architekturen durchsetzen, da sie einfacher zu erweitern und zu warten sind. Zudem ist die Skalierbarkeit höher und neue Technologien oder sogar Programmiersprachen können problemlos eingeführt werden. Microservices ermöglichen eine in Monolithen nie dagewesene Flexibilität und vereinfachen die Entwicklung von komplexer Software. Weiters sind Microservices kleiner und einfacher, womit sie kostengünstiger ersetzt werden können.

Literaturverzeichnis

- [1] Introduction to JSON Web Tokens, 2017. <https://jwt.io/introduction/>. Abgerufen am 18.05.2017.
- [2] Spring Cloud, 2017. <http://projects.spring.io/spring-cloud/spring-cloud.html>. Abgerufen am 11.05.2017.
- [3] Spring Cloud Config, 2017. <https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html>. Abgerufen am 12.05.2017.
- [4] Spring Cloud Sleuth, 2017. <https://cloud.spring.io/spring-cloud-sleuth/>. Abgerufen am 08.06.2017.
- [5] ALLEN, J. Effective Akka. O'Reilly Media, Inc., Sebastopol, 2013.
- [6] ALLWEYER, T. BPMN 2.0 - Business Process Model and Notation. Books on Demand, Norderstedt, 2015.
- [7] AMUNDSEN, M., AND MCLARTY, M. Microservice Architecture. O'Reilly Media, Inc., Sebastopol, 2016.
- [8] AYANOGLU, E. Mastering Rabbitmq. Pack Publishing, Birmingham, 2015.
- [9] BAUER, C., KING, G., AND GREGORY, G. Java Persistence with Hibernate. Manning, New York, 2015.
- [10] BERG, A. M. Jenkins Continuous Integration Cookbook - Second Edition. Packt Publishing, Birmingham, 2015.
- [11] BLOCH, J. Effective Java. Addison Wesley, Santa Clara, 2008.
- [12] BÖRGER, E., FLEISCHMANN, A., OBERMEIER, S., SCHMIDT, W., AND STARY, C. Subject-Oriented Business Process Management. Springer, Berlin, 2012.
- [13] DEMICHEIL, L. Jsr 338: Java Persistence API, Version 2.1. Tech. rep., Oracle America, Inc., 2013.
- [14] EISELE, M. Developing Reactive Microservices - Enterprise Implementation in Java. O'Reilly Media, Inc., Sebastopol, 2016.
- [15] ELSTERMANN, M. Proposal for Using Semantic Technologies As a Means to Store and Exchange Subject-Oriented Process Models. In Proceedings of the 9th Conference on Subject-oriented Business Process Management (2017), S-BPM ONE '17, ACM.
- [16] FERRAIOLI, D. F., AND KUHN, D. R. Role-Based Access Controls. 15th National Computer Security Conference (1992), 554 – 563.
- [17] FISCHERMANN, G. Praxishandbuch Prozessmanagement. Schmidt Dr. Goetz, Wettberg, 2013.
- [18] FISHER, P., AND MURPHY, B. D. Spring Persistence with Hibernate. Apress, New York, 2016.
- [19] FLEISCHMANN, A. What Is S-BPM? In S-BPM ONE - Setting the Stage for Subject-Oriented Business Process Management: First International Workshop, Karlsruhe, Germany, October 22, 2009. Revised Selected Papers, H. Buchwald, A. Fleischmann, D. Seese, and C. Stary, Eds. Springer, Berlin, 2010, pp. 85–106.
- [20] FLEISCHMANN, A., RASS, S., AND SINGER, R. S-BPM Illustrated: A Storybook about Business Process Modeling. Springer, Berlin, 2013.

Literaturverzeichnis

- [21] GRÄSSLE, P., BAUMANN, H., AND BAUMANN, P. UML 2.0 projektorientiert: Geschäftsprozessmodellierung, IT-System-Spezifikation und Systemintegration. Galileo Press, 2004.
- [22] GUPTA, M. K. Akka Essentials. Packt Publishing, Birmingham, 2012.
- [23] GUTIERREZ, F. Pro Spring Boot. Apress, Albuquerque, 2016.
- [24] HAGEN, C. R.-V., AND STUCKY, W. Business-Process- und Workflow-Management. B. G. Teubner Verlag, Wiesbaden, 2004.
- [25] HOLLINGSWORTH, D. The Workflow Reference Model. 1995.
- [26] IKKINK, H. K. Gradle Effective Implementations Guide - Second Edition. Packt Publishing, Birmingham, 2016.
- [27] INDEN, M. Der Weg zum Java-Profi. dpunk.verlag GmbH, Heidelberg, 2012.
- [28] LOELIGER, J. Versionskontrolle mit Git. O'Reilly Media, Inc., Köln, 2010.
- [29] LONG, J. Distributed Tracing with Spring Cloud Sleuth and Spring Cloud Zipkin, Feb. 2016. <https://spring.io/blog/2016/02/15/distributed-tracing-with-spring-cloud-sleuth-and-spring-cloud-zipkin>. Abgerufen am 07.06.2017.
- [30] MCCLUSKEY, G. Using Java Reflection, 1998. <http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html>. Abgerufen am 29.05.2017.
- [31] MUSCHKO, B. Gradle in Action. Manning, New York, 2014.
- [32] NEWMANN, S. Building Microservices. O'Reilly Media, Inc., Sebastopol, 2015.
- [33] POSCH, T., BIRKEN, K., AND GERDOM, M. Basiswissen Softwarearchitektur. dpunk.verlag GmbH, Heidelberg, 2011.
- [34] REDDY, K. S. P. SpringBoot: Learn By Example. Leanpub, Hyderabad, 2016.
- [35] ROESTENBURG, R., BAKKER, R., AND WILLIAMS, R. Akka in Action. Manning, New York, 2017.
- [36] RV, R. Spring Microservices. Packt Publishing, Birmingham, 2016.
- [37] STARKE, G. Effektive Softwarearchitekturen. Hanser Fachbuchverlag, München, 2015.
- [38] WAHER, P. Learning Internet of Things. Packt Publishing, Birmingham, 2015.
- [39] WALLS, C. Spring Boot in Action. Manning, New York, 2016.
- [40] WEBB, P., SYER, D., LONG, J., NICOLL, S., WINCH, R., WILKINSON, A., OVERDIJK, M., DUPUIS, C., DELEUZE, S., AND SIMONS, M. Spring Boot Reference Guide, 2017. <http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>. Abgerufen am 06.05.2017.
- [41] WOLFF, E. Microservices - Grundlagen flexibler Softwarearchitekturen. dpunk.verlag GmbH, Heidelberg, 2016.
- [42] ZIPKIN. Zipkin Architecture, 2017. <http://zipkin.io/pages/architecture.html>. Abgerufen am 08.06.2017.