Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Dennis Pietruck

Konzeption von fachlichen REST APIs für Prozessanwendungen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Dennis Pietruck

Konzeption von fachlichen REST APIs für Prozessanwendungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 24. September 2018

Dennis Pietruck

Thema der Arbeit
Konzeption von fachlichen REST APIs für Prozessanwendungen

Stichworte
Business Process Model and Notation, Representational State Transfer, HATEOAS, Process Engine, Camunda BPM

Kurzzusammenfassung
Process Engines ermöglichen die Steuerung von Prozessen durch REST APIs. Diese sind allerdings generisch und geben keinerlei Informationen über die in der Engine laufenden Prozesse. In dieser Arbeit wird die Entwicklung einer fachlichen REST Schnittstelle für einen Beispielprozess einer Process Engine beschrieben mit dem Ziel, den Nutzen sowie die Nachteile einer fachlichen gegenüber einer generischen Schnittstelle herauszuarbeiten. Um die Schnittstellen vergleichen zu können, wird eine Beispielanwendung einmal mit Hilfe der fachlichen Schnittstelle und ein weiteres Mal mit Hilfe der generischen Schnittstelle entwickelt.

Dennis Pietruck

Title of the paper
Conception of professional REST APIs for process applications

Keywords
Business Process Model and Notation, Representational State Transfer, HATEOAS, Process Engine, Camunda BPM

Abstract
Process Engines allow the control of processes through REST API's. Those are generic and don't give any information about running processes in the engine. This work shows the development of a professional REST interface for an example process which is running in an process engine. The aim of it is to find the advantages and also disatvantages of an professional interface in comparison to a generic interface. A sample application will be developed to compare those two interfaces. This sample application is developed by using a professional interface and again by using the generic interface.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

API application programming interface, Seite 1

BPMN Business Process Model and Notation, Seite 2

HAL Hypertext Application Language, Seite 4

HATEOAS Hypermedia As The Engine Of Application State, Seite 14

JSON JavaScript Object Notation, Seite 14

OMG Object Management Group, Seite 5

REST representational state transfer, Seite 1

XML Extensible Markup Language, Seite 11

1 Einleitung

Diese Arbeit wurde in Zusammenarbeit mit der NovaTec Consulting GmbH verfasst. Die 1996 gegründete Firma ist mit acht Standorten in Deutschland und mit einem Standort in Spanien vertreten. Die NovaTec besteht aus sieben Competence Areas (CA). Die Betreuung der Arbeit seitens der NovaTec fand durch die CA "Business Process Management" statt. Die CA BPM bietet BPM-Technologieberatung sowie Anwendungsentwicklung mit BPM-Systemen. Nicht selten müssen hierbei Process Engines in bestehende Systeme integriert werden. Dafür bieten die Process Engines verschiedene Schnittstellen an. Allerdings setzen diese Schnittstellen Wissen über die verwendete Process Engine voraus. Dieser Umstand wird in diesem Kapitel in 1.2 noch genauer erläutert.

1.1 Motivation

Für die Kommunikation von Komponenten in verteilten Systemen werden häufig REST APIs verwendet. "Eine API spezifiziert eine Softwarekomponente hinsichtlich ihrer Ein- und Ausgaben, mit dem Zweck, eine Menge von Funktionalitäten zu definieren, die unabhängig von ihrer Implementierung sind, sodass eine Änderung der Implementierung die Benutzer der API nicht beeinträchtigt." (vgl. Bloch (2014)) Eine gute API zeichnet sich dadurch aus, dass sie einfach zu benutzen, schwer falsch zu benutzen, leicht erweiterbar und für die Benutzer angemessen ist. (vgl. Bloch (2006)) Schwer verwendbare Schnittstellen haben zur Folge, dass Programmierer mehr Zeit benötigen, um diese zu bedienen. Zusätzlich muss mehr Code geschrieben werden, der die Softwarekomplexität und den Testaufwand erhöht. (vgl. Henning (2007))

1.2 Problemstellung und Zielsetzung

Process Engines ermöglichen die Ausführung von Geschäftsprozessen und die Verwaltung der für die Prozesse benötigten Daten ohne komplexe Programmierung durch die Verwendung von

standardisierten BPMN-Modellen. Die Camunda BPMN Workflow Engine bietet zur Steuerung der Prozesse und Konfiguration der Engine, neben einer grafischen Benutzeroberfläche und einer Java API, eine REST API. Diese Schnittstelle verwendet generische Bezeichner für den Zugriff auf die von der Engine verwalteten Entitäten. Damit ist es nicht möglich, auf die Fachlichkeit der ausgeführten Prozesse zu schließen. Entwickler, die diese Schnittstelle verwenden, müssen sich die Terminologie und Bedienung der Engine erarbeiten. Eine Möglichkeit, diesen Aufwand zu umgehen, ist eine neue fachliche API zu entwickeln, welche für Entwickler, die mit der Fachlichkeit der Prozesse vertraut sind, leicht zu bedienen ist. Allerdings erzeugt diese Lösung ebenfalls weiteren Entwicklungsaufwand, wie im Verlauf der Arbeit aufgezeigt wird. Ziel dieser Arbeit ist es, Nutzen und Nachteile einer selbst entwickelten fachlichen Schnittstelle für einen Prozess in einer Process Engine zu erarbeiten. Dazu wird eine fachliche API für einen Beispielprozess entwickelt. Anschließend soll eine Frontend Anwendung in zwei Versionen implementiert werden. Einmal unter Verwendung der generischen und ein weiteres Mal mit der fachlichen API, um diese im Anschluss zu vergleichen.

1.3 Aufbau der Arbeit

Kapitel 2, Einleitung

Nach der Einleitung werden in Kapitel 2 die Grundlagen für diese Arbeit vermittelt. Zuerst wird das Thema Business Process Management erklärt. Anschließend wird eine kurze Übersicht über den REST Architekturstil gegeben.

Kapitel 3, REST und BPMN - Ein Überblick

Im dritten Kapitel werden einige Arbeiten vorgestellt, in denen Vorschläge präsentiert werden, REST und BPMN zu vereinen. Außerdem wird ein kurzer Überblick über zwei andere Process Engines und deren REST Schnittstellen gegeben, um zu zeigen, wie die REST Schnittstellen anderer BPM Systeme aussehen.

Kapitel 4, Fachliches Szenario

Im vierten Kapitel wird der Beispielprozess, für den eine fachliche Schnittstelle entwickelt werden soll, vorgestellt und daraus Anforderungen an die Referenzanwendung abgeleitet.

Kapitel 5, Spezifikation

Das fünfte Kapitel dient der Spezifikation der zu entwickelnden Anwendung. Dafür werden Use Cases, Wireframes sowie das fachliche Datenmodell der Anwendung vorgestellt.

Kapitel 6, Architektur

Kapitel sechs beschäftigt sich mit der Architektur des entwickelten Systems. Als Erstes wird die Architektur des Backends und die angebotene REST Schnittstelle beschrieben. Anschließend wird die Architektur des Frontends vorgestellt.

Kapitel 7, Realisierung

Kapitel sieben zeigt einige Realisierungsdetails der Anwendung und gibt eine kurze Übersicht über die verwendeten Werkzeuge und Bibliotheken.

Kapitel 8, Evaluation

Kapitel acht widmet sich der Evaluation der API Varianten anhand der zuvor implementierten Frontendanwendungen. Dazu werden einige Metriken vorgestellt und angewendet, um die Schnittstellenvarianten vergleichen zu können.

Kapitel 9, Zusammenfassung und Ausblick

Abschließend wird die Arbeit zusammengefasst und Vorschläge für weiterführende Arbeiten gegeben.

2 Grundlagen

In diesem Kapitel wird zunächst das Thema Business Process Management vorgestellt. In den Unterkapiteln wird dann die Modellsprache BPMN und ihre Basiselemente mit der dazugehörigen Semantik erklärt. Außerdem werden die BPMN Erweiterungen CMMN und DMN erläutert. Anschließend wird die Camunda Process Engine vorgestellt und erklärt, wie BPMN Modelle mit Hilfe der Engine ausgeführt und gesteuert werden können. Im Kapitel 2.3 werden dann die Grundlagen des REST Architekturstils erläutert und die *Hypermedia Application Language* (HAL) vorgestellt.

2.1 Business Process Management

Für das Thema Business Process Management gibt es verschiedene Definitionen. Nach der European Association of BPM lautet diese folgendermaßen: Business Process Management ist ein systematischer Ansatz, um sowohl automatisierte als auch nicht automatisierte Prozesse zu erfassen, zu gestalten, auszuführen, zu dokumentieren, zu messen, zu überwachen und zu steuern und damit nachhaltig die mit der Unternehmensstrategie abgestimmten Ziele zu erreichen. (vgl. European Association of Business Process Management (2014)) Als Prozess wird eine Abfolge von Aktivitäten definiert, die ausgeführt werden, um ein bestimmtes Ziel zu erreichen (vgl. Freund (2016))

Zur Modellierung und Ausführung von Prozessen wurden einige Notationen und Darstellungsarten wie Ereignisgesteuerte Prozessketten und Kommunikationsstrukturdiagramme entwickelt. Die standardisierte BPMN ist ebenfalls eine Notation für Prozessmodelle die sich durchgesetzt hat. Diese wurde durch die CMMN und DMN erweitert. In den folgenden Unterkapiteln wird die BPMN sowie die Erweiterungen erläutert.

2.1.1 BPMN

Die Business Process Model and Notation ist ein von der Object Management Group (OMG) entwickelter Standard, um Geschäftsprozesse grafisch darzustellen. Die Basiselemente sind dabei in die fünf Kategorien, Flussobjekte, Verbindende Objekte, Teilnehmer, Artefakte und Daten aufgeteilt. Für die vollständige BPMN Spezifikation siehe OMG (2011). Im Folgenden werden die für den Beispielprozess relevanten Basiselemente und Symbole kurz erläutert.

Flussobjekte bestimmen das grundlegende Verhalten eines Modells.

- *Activites* (Aktivitäten) sind Aufgaben, die im Verlauf des Prozesses erledigt werden müssen. Aktivitäten lassen sich in atomar und nicht atomar unterteilen. Bei atomaren Aktivitäten handelt es sich um *Tasks*. Diese lassen sich nicht in weitere Unteraufgaben aufteilen. Tasks können von Menschen (User Task) oder von Maschinen (Service Task) ausgeführt werden. Weiterhin kann ein Task auch das Auswerten einer DMN Entscheidungstabelle sowie das Senden oder Empfangen einer Nachricht sein. Nicht atomare Aktivitäten lassen sich durch weitere Prozessmodelle oder Casemodelle in kleinere Unteraufgaben aufteilen und können als *Call Activity* dargestellt werden.

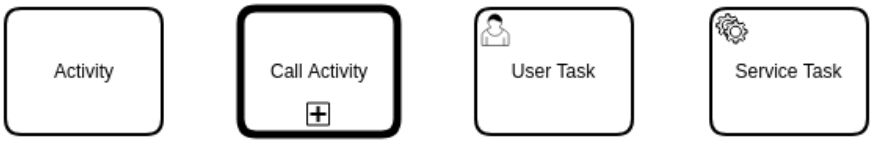

Abbildung 2.1: BPMN Aktivitäten (vgl. OMG (2011))

- *Events* sind Ereignisse, die in einem Prozess auftreten, wie das Empfangen einer Nachricht (Message Event). Ein Ereignis kann auch wiederholt in Intervallen auftreten (Timer Event). Diese Ereignisse können für sich alleine stehend als Start-, End oder Intermediate Event, oder an Tasks als unterbrechend und nicht unterbrechend angeheftet sein, um auf Ereignisse in einem laufenden Task reagieren zu können.

Abbildung 2.2: BPMN Events (vgl. OMG (2011))

- *Gateways* steuern den Fluss eines Prozesses durch Aufteilen und Zusammenführen von parallelen Flüssen (Parallel Gateway) oder durch Auswahl eines bestimmten Prozesspfades auf Grundlage von gegebenen Daten (XOR-Gateway). Die gängigsten sind XOR- und Parellel Gateways. Die BPMN spezifiziert allerdings noch weitere unterschiedliche Arten von Gateways.

Abbildung 2.3: BPMN Gateways (vgl. OMG (2011))

Verbindende Objekte definieren Verbindungen und Beziehungen zwischen Flussobjekten eines Modells.

- *Sequence Flows* verbinden Flusselemente und zeigen an, in welcher Reihenfolge diese ausgeführt werden.

Abbildung 2.4: BPMN Sequence Flow (vgl. OMG (2011))

Teilnehmer strukturieren Prozesse auf bis zu zwei Ebenen:

- *Pools* repräsentieren einen Prozessteilnehmer. Ein Prozessteilnehmer kann eine Person oder eine Organisation sein. Alle Tasks in einem Pool werden von einem Teilnehmer ausgeführt.

Abbildung 2.5: BPMN Pool (vgl. OMG (2011))

- *Lanes* teilen einen Pool in feinere Einheiten. Innerhalb einer Firma könnte es sich hier zum Beispiel um Abteilungen handeln.

Abbildung 2.6: BPMN Lanes (vgl. OMG (2011))

2.1.2 CMMN

Es gibt Prozesse, in denen Tasks in keiner festen Reihenfolge ausgeführt werden müssen. Dies lässt sich mit der BPMN zwar durch Nebenläufigkeit modellieren, führt allerdings zu stark verzweigten und unübersichtlichen Diagrammen. Für diese Fälle wurde die Case Management Model and Notation von der OMG veröffentlicht, welche im März 2016 in der Version 1.1 verabschiedet wurde. (vgl. Freund (2016))

Nach der OMG lässt sich ein Case folgendermaßen definieren: Ein *Case* ist ein Vorgang, der Ausführung von Aktivitäten hinsichtlich eines Subjekts in einer bestimmten Situation erfordert, um ein Ergebnis zu erzielen. Die Bearbeitung eines Case kann zur Laufzeit geplant werden. OMG (2016a) Hier zeigt sich der Unterschied zur BPMN. Während in einem BPMN-Modell die Abarbeitung von Aktivitäten durch Sequenzflüsse klar definiert ist, gibt die CMMN keine feste Reihenfolge vor. Diese wird von einem Bearbeiter mit Anwendungswissen während der Bearbeitung eines Case gewählt. Mit der CMMN lassen sich die möglichen Aufgaben in einem Case, und unter welchen Bedingungen diese ausgeführt werden können, modellieren. Abbildung 2.7 zeigt die Grundlegenden Elemente der CMMN.

Ein Case wird in der CMMN durch eine Aktenmappe visualisiert. In einem Case befinden sich Aktivitäten, die keine Reihenfolge haben. Aktivitäten können, wie in der BPMN, Tasks oder Subprozesse und Cases sein. Um verschiedene Phasen oder Abschnitte in der Bearbeitung eines Case zu modellieren, werden *Stages* (Abschnitte) verwendet. Auch wenn es in der CMMN keinen Sequenzfluss gibt, kann es sein, dass Aufgaben erst unter bestimmten Bedingungen erledigt werden dürfen. Dieser Sachverhalt wird durch *Sentries* (Wächter) dargestellt. Es kann sichergestellt werden, dass Tasks erst dann begonnen werden dürfen, wenn zuvor bestimmte

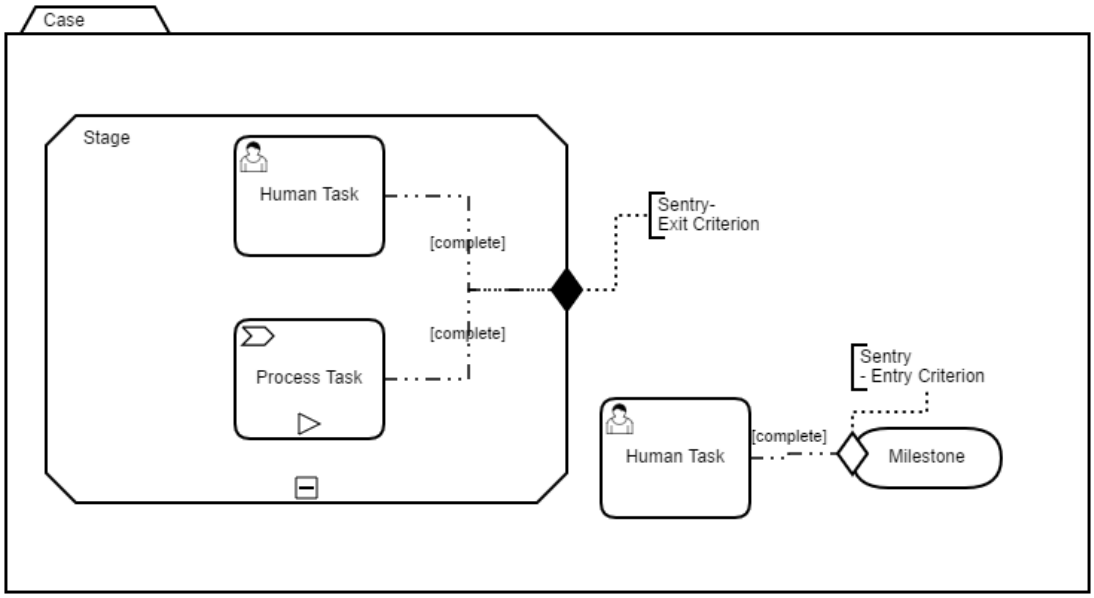

Abbildung 2.7: Grundlegende CMMN Elemente

Aufgaben beendet wurden oder wenn ausgewählte Daten einen bestimmten Wert annehmen. Stellt ein Sentry sicher, ob ein Task begonnen werden kann, wird ein *Entry Criterion* (Eintrittskriterium) geprüft. Wenn geprüft wird, ob ein Task beendet werden kann, handelt es sich um ein *Exit Criterion* (Austrittskriterium). Sentries können nicht nur an Tasks, sondern auch an Milestones als Entry Criterion angebracht werden. Da die CMMN keinen Kontrollfluss besitzt, werden Milestones verwendet um einen Fortschritt in der Bearbeitung eines Case darzustellen. Ein *Milestone* (Meilenstein) repräsentiert ein wichtiges Zwischenereignis wie das Beenden einer Stage. OMG (2016a)

In der BPMN lässt sich die Verfügbarkeit eines Tasks durch Verfolgen des Sequenzflusses nachvollziehen. Dies ist in der CMMN nicht möglich. Hierbei wird die Verfügbarkeit durch einen Lebenszyklus beschrieben, der in Abbildung 2.8 visualisiert ist.

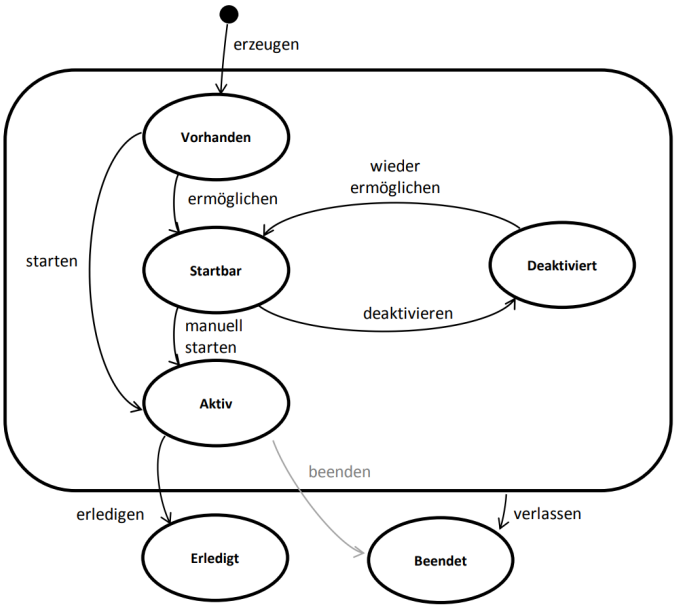

Abbildung 2.8: Lebenszyklus eines Tasks (vgl. Freund (2016))

Wird ein Case gestartet, werden alle darin enthaltenen Tasks erzeugt und sind im Zustand *vorhanden*. Für jeden Task, an den ein Sentry angeheftet ist, wird geprüft, ob die Bedingungen erfüllt sind und ein Wechsel in den Zustand *startbar* stattfinden kann. Sollten sich Daten verändern, sodass Bedingungen nicht erfüllt sind, wird ein Task *deaktiviert*. Tasks ohne Sentry wechseln sofort in den Zustand startbar. Mit dem manuellen Starten findet ein Wechsel in den Zustand *aktiv* statt. Von dort aus können Tasks *erledigt* oder *beendet* werden.

2.1.3 DMN

Komplexe Entscheidungsregeln lassen sich mit der BPMN durch Gateways darstellen, führen allerdings durch viele Verzweigungen zu schwer lesbaren Diagrammen.

Die Decision Model and Notation wird, wie die BPMN und die CMMN, von der OMG verwaltet und liegt seit Juni 2016 in der Version 1.1 vor. Sie dient der grafischen Modellierung und Dokumentation von Business Rules. Business Rules sind Geschäftsregeln, welche auf Daten angewendet werden können, um eine Entscheidung zu treffen. OMG (2016b)

Mit Hilfe der DMN sollen BPMN Diagramme vereinfacht werden. Abbildung 2.9 zeigt einen Beispielprozess zur Entscheidung über einen Kreditantrag.

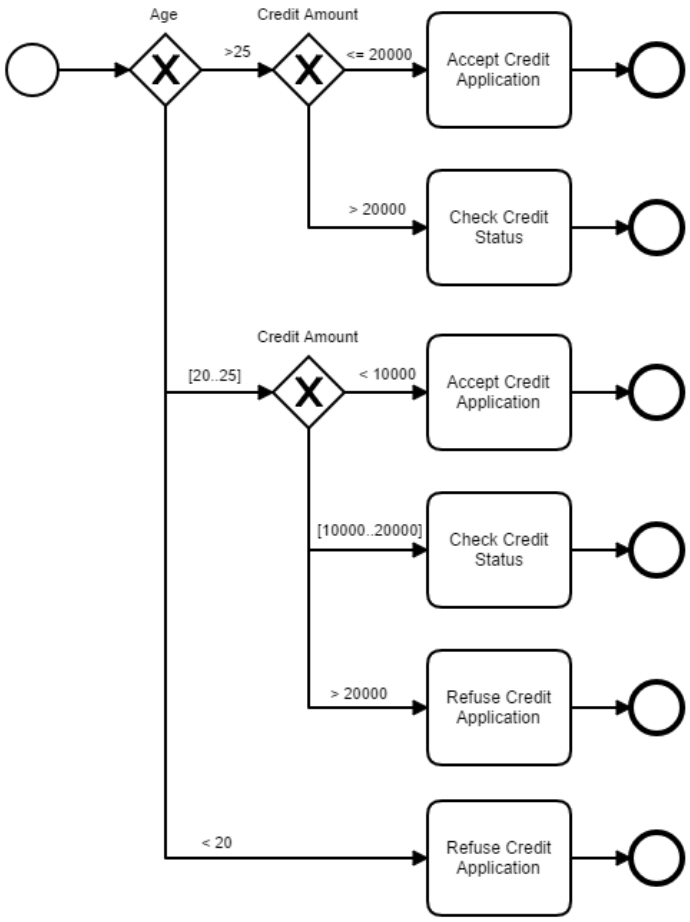

Abbildung 2.9: Regeln zur Annahme eines Kreditantrags (BPMN)

Die in diesem Prozess aufgestellten Regeln sind in Tabelle 2.1 als Entscheidungstabelle dargestellt. Die linken Spalten enthalten die Eingabewerte (Input) und Bedingungen, die gelten müssen, damit das Ergebnis (Output) in der rechten Spalte zurück gegeben wird.

Für jede Tabelle muss eine *Hit Policy* in der linken oberen Ecke der Tabelle angegeben werden. Diese bestimmt die Reihenfolge in der die gegebenen Regeln ausgewertet werden und wie viele Ergebnisse ausgegeben werden. Die Hit Policy Unique wird mit einem “U“ abgekürzt und bedeutet, dass für die übergebenen Eingabewerte genau eine Zeile zutreffen darf. Alternativ kann auch ein “F“ für First verwendet werden. Nach dieser Policy können mehrere Regeln gelten. Wenn dies der Fall ist, wird nur die von oben nach unten, erste geltende Regel verwendet. Mit der Hit Policy Collect, abgekürzt mit “C“, werden die Ergebnisse aller geltenden Regeln gesammelt als Liste zurückgegeben.

U	input		Output
	Age	Credit amount	Accept Loan
	integer	integer	string
1	< 20	-	“Refuse“
2	[20..25]	< 10000	“Accept“
3	[20..25]	[10000..20000]	“Background Check“
4	[20..25]	> 20000	“Refuse“
5	> 25	<= 20000	“Accept“
6	> 25	> 20000	“Background Check“

Tabelle 2.1: Regeln zur Annahme eines Kreditantrags (DMN)

Die Anwendung von Geschäftsregeln lassen sich in BPMN als Business Rule Task modellieren. Abbildung 2.10 zeigt die Entscheidungstabelle als einen Business Rule Task. Das Ergebnis kann dann weiter in BPMN Modellen genutzt werden und führt zu einer einfacheren Darstellung.

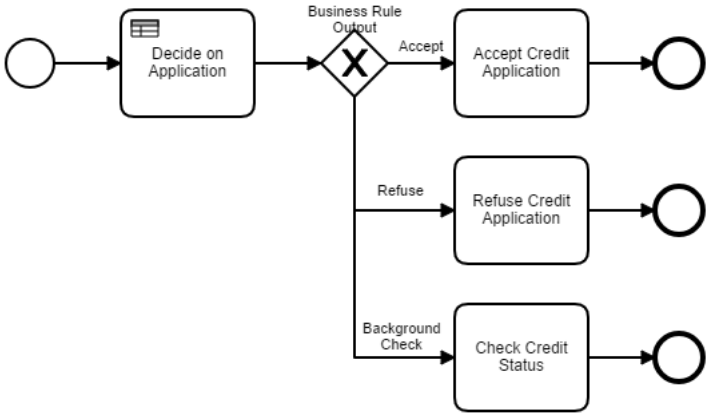

Abbildung 2.10: Verwendung von Geschäftsregeln in BPMN

2.2 Camunda BPM

Für die BPMN, die CMMN und die DMN sind nicht nur die grafischen Elemente, und wie diese angeordnet werden können, spezifiziert, sondern auch wie die in der Modellierungssprache erstellten Diagramme einheitlich in XML festgehalten werden sollen. Process Engines können diese Modelle einlesen und automatisch ausführen. Camunda ist eine BPM Plattform zur Prozessautomatisierung. Die Plattform stellt eine Reihe von Applikationen bereit um BPMN, CMMN und DMN Modelle auszuführen und zu verwalten. Im Folgenden wird ein Überblick

über die Camunda Anwendungen gegeben und die wichtigsten Konzepte der Process Engine kurz vorgestellt.

2.2.1 Camunda Stack

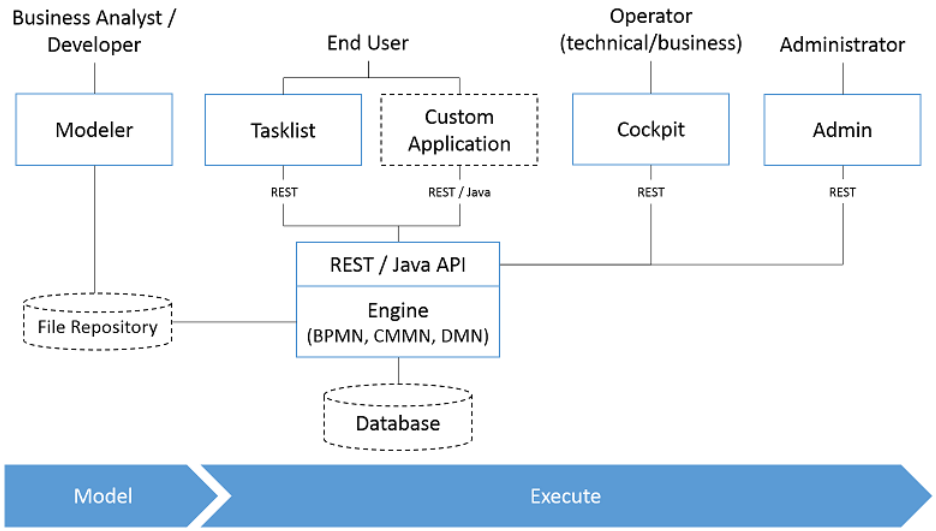

Abbildung 2.11: Camunda Systemarchitektur (vgl. Camunda (a))

Der Kern der Plattform ist die Core Engine, welche Prozessmodelle, Case Modelle und Decision Modelle ausführt. Diese wird über REST oder Java API von anderen Anwendungen angesteuert. Endanwender starten und steuern die Modelle über die Webanwendung Camunda Tasklist oder einem eigenen Frontend.

Der Camunda Modeler ist eine eigenständige Anwendung mit der BPMN, CMMN und DMN Modelle erstellt und mit den für die Ausführung nötigen technischen Eigenschaften versehen werden können.

Camunda Cockpit ist eine Anwendung für das Monitoring und Verwalten von Prozessinstanzen. So kann der Verlauf einer Prozessinstanz beobachtet oder unterbrochen und Variablenwerte geändert werden.

Mit der Anwendung Camunda Admin können Systemadministratoren Benutzer und Gruppen sowie deren Rechte verwalten.

2.2.2 Modellausführung

Zur Modellausführung benötigt die Engine eine *Prozessdefinition* bzw. eine *Case Definition*. Dies ist ein Modell in der entsprechenden Notation, das alle technischen Informationen enthält, die zur Ausführung verwendet werden. Dazu gehören Referenzen zu weiteren Prozessen oder Cases und Entscheidungstabellen, die als Aktivität aus dem Prozess bzw. Case aufgerufen werden können. Darüber hinaus müssen Klassennamen und Skripte für die Bearbeitung von Service bzw. Script Tasks angegeben werden, um das Modell mit dem Anwendungscode zu verknüpfen.

Für jeden gestarteten Prozess erzeugt die Engine eine neue *Prozessinstanz*. Jede Instanz repräsentiert genau einen individuellen Prozess. Das Verhältnis zwischen Prozessdefinition und Prozessinstanz ist mit dem zwischen Klasse und Objekt in der objektorientierten Programmierung zu vergleichen. Zu einer Prozessinstanz gehört eine oder mehrere *Prozessausführungen*, die von der Engine gesteuert werden. Eine Ausführung stellt eine Aktivität dar, an der sich ein Prozess gerade befindet. Wird die Aktivität beendet, wird auch die Prozessausführung beendet und eine neue für die folgende Aktivität erzeugt. (vgl. Camunda (d))

Analog zu Prozessen erzeugt die Engine für jeden gestarteten Case eine *Case Instanz*. Zu jeder Instanz gehören eine oder mehrere *Case Ausführungen*. Da es in der CMMN keinen vorgeschriebenen Verlauf gibt, wird mit dem Erzeugen einer Case Instanz eine Case Ausführung für jede Aktivität erzeugt. Eine Ausführung enthält den Zustand eines Tasks (siehe Abbildung 2.8). Dieser Zustand wird von der Engine gesteuert. (vgl. Camunda (b))

In beiden Fällen gehört zu einer Ausführung genau ein Task. Ein Benutzer kann sich in der Camunda Tasklist oder einem eigenen Frontend offene Tasks anzeigen lassen, und sich selber oder andere als Bearbeiter des Tasks eintragen. Der Bearbeiter eines Tasks wird in Camunda *assignee* genannt.

Wird ein Human Task ausgeführt, trägt der Bearbeiter des Tasks Daten in eine Eingabemaske der Camunda Tasklist oder des eigenen Frontends ein. Diese Daten werden mit Beenden des Tasks als *Prozessvariablen* in der Engine gespeichert und verwaltet. Variablen können in der Tasklist oder über die REST API angezeigt sowie zur Auswertung von Gateways oder Sentries verwendet werden. (vgl. Camunda (d))

2.3 Representational State Transfer

REST ist ein Architekturstil für verteilte hypermedia Systeme, der von Roy Thomas Fielding im Rahmen seiner Dissertation entwickelt wurde. Ein Architekturstil ist eine Menge von Einschränkungen über Elemente einer Architektur, die einen bestimmten Stil umsetzt. Die Einhaltung der Einschränkungen hat die gewünschten Eigenschaften einer Architektur zur Folge. (vgl. Fielding (2000))

In seiner Dissertation führt Fielding Einschränkungen ein, die für eine Anwendung gelten, die REST als Architekturstil verwendet. Hervorzuheben ist, dass Komponenten eines Systems nach dem Client-Server Model kommunizieren. Der Server muss dabei keine Sitzungsinformationen speichern. Die Kommunikation findet also zustandslos statt. Außerdem wird über ein einheitliches Interface kommuniziert, das unabhängig von einer Anwendungsdomäne definiert ist. Für das Interface soll gelten, dass Ressourcen durch einen Identifier eindeutig identifiziert werden können. Ressourcen können durch Repräsentationen manipuliert werden und Nachrichten sollen selbstbeschreibend sein. Das heißt, dass Anfragen an einen Server alle Daten enthalten, die für die Verarbeitung einer Anfrage benötigt werden. Dazu gehört die Repräsentation einer Resource, die verarbeitet wird, sowie Metadaten zur Beschreibung der Repräsentation. Zuletzt soll für das Interface gelten, dass HATEOAS umgesetzt wird, also dass eine Repräsentation Links zu Ressourcen enthält, die mit der dargestellten Ressource verknüpft sind. Für eine vollständige Beschreibung des Architekturstils siehe Fielding (2000).

Um HATEOAS umzusetzen existieren verschiedene Möglichkeiten. Darunter auch die Hypermedia Application Language. Die HAL spezifiziert für XML und JSON Objekte, wie Verknüpfungen zwischen Resourcen dargestellt werden sollen. Für JSON Objekte wurden dafür zwei Felder vorgesehen: _links und _embedded. _links enthält ein JSON Objekt, welches ein Link Objekt oder ein Array aus Linkobjekten als Werte enthält. Ein Linkobjekt muss das Feld *href* besitzen, dessen Wert eine URI oder ein URI Template ist. Die Feldnamen sind *link relation types*. Link relation types beschreiben die Semantik eines Links. (vgl. Nottingham (2010)) Das _embedded Objekt enthält als Wertnamen ebenfalls link relation types. Allerdings sind die Werte keine Linkobjekte, sondern Arrays aus Resourcenobjekten oder ein einzelnes Resourcenobjekt. (vgl. Kelly (2016))

3 REST und BPMN - Ein Überblick

In diesem Abschnitt werden zwei Arbeiten aus dem Bereich REST Schnittstellen und Process Engines vorgestellt. Diese Arbeiten beschreiben Vorschläge wie Process Engines mit dem REST Architekturstil vereint werden können. Hierdurch soll eine Übersicht über mögliche Ansätze Process Engines mit REST anzusprechen gegeben werden. Zuletzt werden weitere Process Engines mit Hinblick auf die angebotenen Schnittstellen vorgestellt.

3.1 ROsWeL Workflow Language

Die in Brzeziński u. a. (2012) vorgestellte ROsWel Workflow Language bietet die Möglichkeit, Geschäftsprozesse deklarativ zu erstellen. Dabei wird definiert, welches Ziel mit einem Geschäftsprozess erreicht werden soll und welche Teilziele erreicht werden müssen, um das Endziel eines Geschäftsprozesses zu erreichen. Die ROsWeL Workflow Language wurde mit dem Ziel entwickelt, die mit der Sprache entwickelten Prozesse als REST Service bereitzustellen. Dazu bietet die Sprache die Möglichkeit, URIs zu definieren und diese mit den erstellten Prozessen zu verknüpfen. Resourcen können mit Hilfe von HTML Templates dargestellt werden und unterstützen durch das Einsetzen von URIs in HTML-Forms sogar HATEOAS. Ein Nachteil der vorgestellten Sprache ist, dass die Ziele eines Prozesses als Horn-Formeln beschrieben werden und keinen Ansatz für die grafische Darstellung von Prozessen bietet, was die Kommunikation von Prozessen erschwert. Die vorgestellte Sprache ermöglicht es, fachliche REST Schnittstellen für Prozesse zu erzeugen. Allerdings ist diese nicht mit BPMN Prozessmodellen kompatibel, weshalb die vorgestellte Sprache nicht im Rahmen dieser Arbeit verwendet werden kann.

3.2 BPMN for REST

Ein Vorschlag REST und BPMN zu verknüpfen wird in Pautasso (2011) gemacht. Dabei soll die BPMN um ein Resourcensymbol (Abbildung 3.1) erweitert werden, um Prozesse und Tasks

als REST Resource anbieten zu können. Die URIs zur Identifikation von Tasks oder Prozessen werden durch die Namen der Prozesse sowie die der markierten Tasks gebildet, was dazu führt, dass URIs durch die Fachlichkeit der Anwendung gebildet werden. Diese Erweiterung umfasst allerdings keine Vorschläge dazu, wie die Repräsentationen der Resourcen dargestellt werden und welche Informationen für das Bearbeiten eines Tasks benötigt werden.

Abbildung 3.1: Vorgeschlagenes Resourcen Symol (vgl. Pautasso (2011))

3.3 Weitere Process Engines

In dieser Arbeit wird die Camunda Process Engine verwendet, welche standardmäßig die Möglichkeit bietet, die Prozesse über eine REST Schnittstelle zu steuern. Weitere BPMN Engines wie die Activity Engine[1], oder jBPM[2] sind wie die Camunda Engine in Java geschriebene Open Source Projekte. Beide Engines bieten eine REST API, um Prozesse und Tasks zu steuern. Dabei werden für die Bezeichner von Resourcen und Attributen, wie bei der Camunda Process Engine auch, technische Begriffe verwendet, die sich nicht durch Konfiguration an die Fachlichkeit der Prozesse anpassen lassen. Da beide Engines in bestehende Projekte eingebettet werden können, ist es möglich die REST APIs durch zusätzliche fachliche Resourcen zu erweitern. Bei der Suche nach weitere Engines konnten keine gefunden werden, die es ermöglichen, die angebotene Schnittstelle an die Fachlichkeit der Anwendung durch Konfiguration anzupassen.

[1] activiti.org
[2] jbpm.org

4 Fachliches Szenario

Es soll eine Anwendung auf Grundlage eines fiktiven Versicherungsantragsprozess entwickelt werden, der als Beispiel in Freund (2016) veröffentlicht ist. Dieser Prozess eignet sich als Fallbeispiel, da die wichtigsten Elemente der BPMN, DMN und CMMN verwendet werden. Gleichzeitig ist das Prozessmodell überschaubar. Im Folgenden wird der gewählte Versicherungsantragsprozess vorgestellt und erklärt. Dann wird der Case für die Neuantragsprüfung und der Prozess zur Dokumentenanforderung erklärt. Anschließend wird die im Prozess verwendete Entscheidungstabelle erklärt. Zuletzt werden aus den vorgestellten Modellen Anforderungen an die Referenzanwendung abgeleitet.

4.1 Der Hauptprozess für Versicherungsanträge

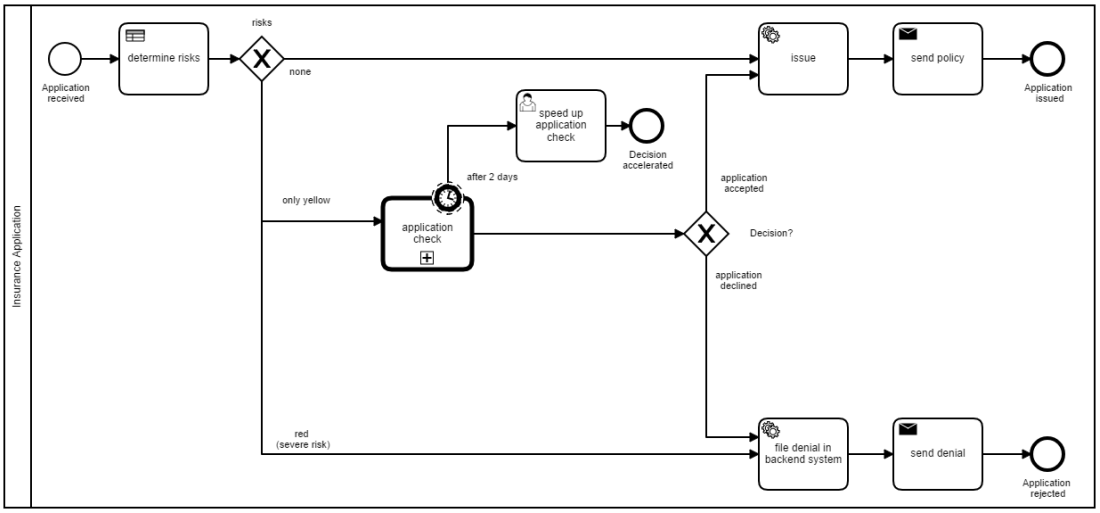

Abbildung 4.1: Hauptprozess der Versicherungsanwendung (vgl. Freund (2016))

Der Versicherungsantragsprozess beginnt, sobald die Versicherung einen Antrag erhält, mit dem Startereignis *Application received.* Jeder Prozess durchläuft als erstes den Business Rule Task *determine risks.* Dieser Task ist mit der DMN-Entscheidungstabelle *risk check* verknüpft, welche Fahrer anhand von Alter und Fahrzeug kategorisiert. Dies wird in Abschnitt 4.4 genauer erklärt. Im nächsten Schritt entscheidet ein XOR-Gateway auf Grundlage der vorherigen Risikoprüfung, ob der Antrag angenommen oder abgelehnt wird oder ob eine manuelle Neuantragsprüfung stattfinden soll. Falls keine Risiken ermittelt wurden, fährt der Prozess mit dem Service Task *issue policy* fort. Anschließend wird der Antragssteller durch den Send Task *send policy* darüber informiert, dass der Antrag angenommen wurde. Zuletzt wird das Endereignis *policy issued* ausgelöst und der Prozess ist beendet.

Alternativ kann nach dem Business Rule Task ein hohes Risiko ermittelt worden sein. Dann wird der Prozess nach dem XOR-Gateway mit dem Service Task *file denial in backend system* fortgesetzt. Im Anschluss wird der Antragssteller durch den Send Task *send denial* darüber informiert, dass der Antrag abgelehnt wurde. In diesem Fall endet der Prozess mit dem Endereignis *application denied.*

Sollte nach der Überprüfung mit der DMN-Entscheidungstabelle ein mittleres Risiko festgestellt worden sein, wird die Call Activity *application check*, bei der ein Antrag manuell geprüft wird, durchlaufen. Diese wird in Abschnitt 4.2 näher erläutert. Sobald die Aktivität abgeschlossen ist, trifft der Prozess auf ein weiteres XOR-Gateway. Wurde der Antrag in der manuellen Prüfung abgelehnt, wird die bereits erwähnte Aufgabe *file denial in backend system* durchgeführt. Danach verläuft der Prozess wie im Fall, in dem ein hohes Risiko festgestellt wurde. Der Antrag kann in der Call Activity auch angenommen werden. Dann verläuft der Prozess nach dem zweiten XOR-Gateway wie im Fall, in dem keine Risiken festgestellt wurden.

Dauert die Aufrufaktivität *application check* länger als zwei Tage an, wird das angeheftete Timer Event ausgelöst und die Aktivität *speed up application check* aktiv.

4.2 Neuantragsprüfung

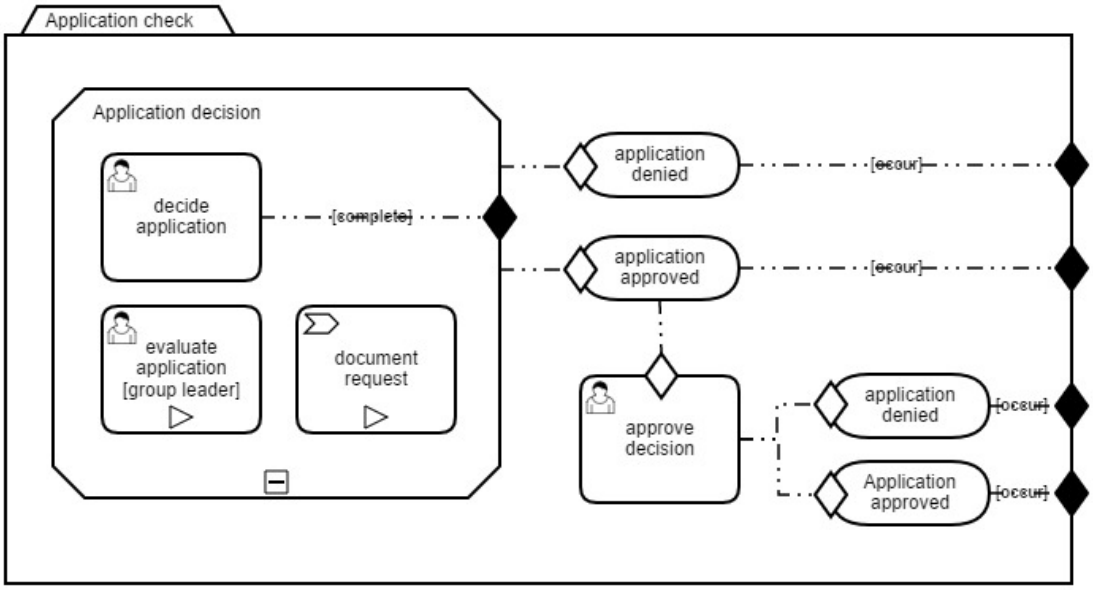

Abbildung 4.2: Neuantragsprüfung (vgl. Freund (2016))

Im Case *application check* ist zu Beginn die Stage *application decision* aktiv. Darin befindet sich der User Task *decide application*, welcher zu Beginn ebenfalls aktiv ist. Wenn der Antrag durch einen Gruppenleiter beurteilt werden soll, muss der User Task *evaluate application* manuell gestartet werden. Fehlen zur Beurteilung des Antrags noch Dokumente, können mit dem Starten der Aktivität *document request* weitere Dokumente angefordert werden. Damit wird der Prozess *Document Request*, der in Abschnitt 4.3 genauer erklärt wird, gestartet. Hat ein Sachbearbeiter die Aufgabe *application check* abgeschlossen und über einen Antrag sowie die Versicherungsprämie entschieden, ist das Austrittskriterium an der Stage *application decision* erfüllt. Damit ist, wenn der Antrag abgelehnt wurde, der Meilenstein *application denied* erreicht und das damit verbundene Austrittskriterium erfüllt. Wenn keine weiteren Aktivitäten aktiv sind, wird der Case verlassen. Wenn der Antrag angenommen wurde, ist der Meilenstein *application approved* erreicht und damit ein Austrittskriterium erfüllt. Wenn die Bedingung für den Wächter an der Benutzeraufgabe *approve decision* nicht erfüllt wird, ist keine Aktivität mehr aktiv und der Case wird verlassen. Übersteigt die festgelegte Prämie 300 Euro, ist die Bedingung für den Wächter an der Aufgabe *approve decision* erfüllt und es muss die Entscheidung aus der Aufgabe *decide application* freigegeben werden. Dafür wird der User Task *approve decision*

aktiv. Wenn dieser beendet wurde, ist je nach Entscheidung der Meilenstein *application denied* oder *application approved* erreicht. In beiden Fällen ist ein Austrittskriterium erfüllt und der Case beendet.

4.3 Dokumentenanforderung

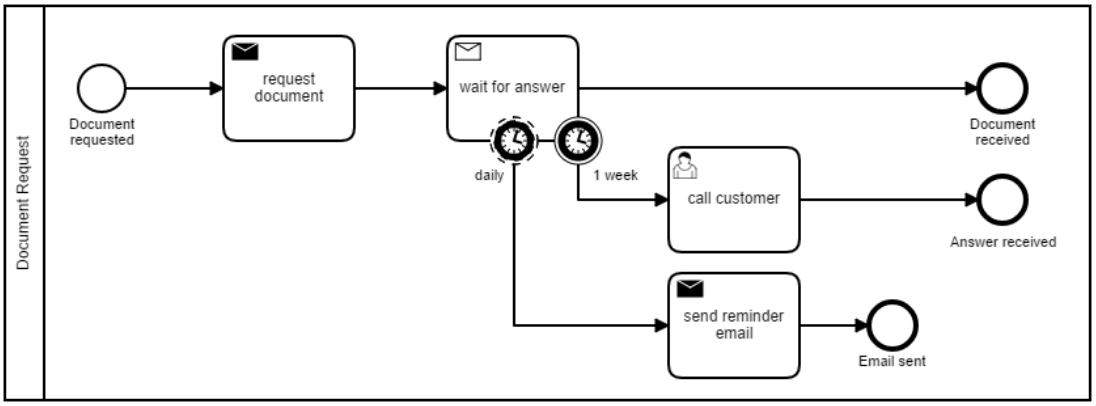

Abbildung 4.3: Dokumentenanforderung nach Freund (2016)

Falls Dokumente zur Beurteilung eines Antrags benötigt werden, wird der Prozess *document request* gestartet. Dieser beginnt mit dem Send Task *request document*. Dabei wird der Antragsteller benachrichtigt, dass noch weitere Dokumente benötigt werden. Anschließend beginnt der Receive Task *wait for answer*. Dieser ist beendet, wenn die benötigten Dokumente eingereicht wurden. Dieser Aufgabe sind zwei Timer Events angeheftet. Ein nicht unterbrechendes, das täglich den Send Task *send reminder email* auslöst, und ein unterbrechendes, welches nach einer Woche ausgelöst wird und den User Task *call customer* aktiviert sowie den Task *wait for answer* abbricht.

4.4 Risikoprüfung

C	Input			Output	
	Age	Vehicle Manufacturer	Vehicle Type	Risk	Risk Level
	integer	string	string	string	string
1	<= 21	-	-	"beginner"	"yellow"
2	<= 30	"BMW"	-	"young and fast"	"yellow"
3	-	"Porsche"	"911"	"careless speeding"	"yellow"
4	-	"BMW"	"X3"	"Premium car"	"yellow"
5	<= 25	"Porsche"	“911“	"young and too fast"	"red"

Tabelle 4.1: Risikoprüfung nach Freund (2016)

Zur Risikoprüfung werden Alter des Antragstellers, Fahrzeugtyp und Fahrzeugmarke bewertet. Das Ergebnis besteht aus einer Liste von Risiken, die für den Antragssteller festgestellt wurden, und einer Liste aus Risikobewertungen, die durch Farbcodes dargestellt werden. Für die Prüfung gilt die Hit Policy Collect, weshalb alle zutreffenden Ergebnisse ausgegeben werden. Für Antragssteller unter 21 gilt grundsätzlich die erste Zeile. Damit sind diese in der Sicherheitsstufe *yellow* mit dem Risiko *beginner* eingestuft. Für alle unter 30 ist das Ergebnis nicht nur vom Alter, sondern auch vom Fahrzeughersteller und dem Fahrzeugtyp abhängig. BMW Fahrer unter 30 werden grundsätzlich mit dem Risiko *young and fast* und der Risikostufe *yellow* bewertet. Sollte es sich beim Fahrzeugtyp um einen X3 handeln, kommt das Risiko *premium car*, unabhängig vom Alter, hinzu. Für Antragssteller mit einem Porsche und dem Modell 911 gilt ebenfalls die Risikostufe *yellow* mit dem Risiko *careless speeding*. Sollte eine Person mit dem selben Fahrzeug unter 25 sein, wird diese mit dem Risiko *young and too fast* in der Risikostufe *red* eingestuft.

4.5 Anforderungen

Aus den vorgestellten Geschäftsprozessen lassen sich Anforderungen an die zu entwickelnde Anwendung ableiten. Diese sind im Folgenden aufgelistet:

1. Ein KFZ-Besitzer kann über die Anwendung eine Versicherung beantragen. Dazu muss dieser seine Personendaten, also Geburtstag, Vor- und Nachname, Geschlecht und seine

E-Mail Adresse und Informationen über das zu versichernde Fahrzeug - Fahrzeugtyp und Marke - der Versicherung übermitteln.

2. Ein Sachbearbeiter kann sich Antragsdetails für eine manuelle Prüfung anzeigen lassen.
3. Ein Sachbearbeiter kann einen Antrag ablehnen oder annehmen. Wenn ein Antrag angenommen wird, muss eine Versicherungsprämie bestimmt werden.
4. Ein Sachbearbeiter kann eine Antragsbewertung von einem Teamleiter anfordern.
5. Ein Teamleiter kann einen Antrag bewerten, wenn dies angefordert wurde.
6. Ein Sachbearbeiter kann weitere Dokumente von einem Antragsteller anfordern. Dazu gibt ein Sachbearbeiter eine Beschreibung des benötigten Dokuments an. Der Antragsteller wird per E-Mail darüber benachrichtigt, dass dieser Dokumente hochladen soll.
7. Ein Antragsteller kann angeforderte Dokumente hochladen.
8. Ein Sachbearbeiter kann hochgeladene Dokumente herunterladen.
9. Ein Teamleiter kann Entscheidungen über einen Antrag bestätigen. Dazu gibt er an, ob der Antrag tatsächlich angenommen oder abgelehnt werden soll. Wird er angenommen, kann er die Versicherungsprämie verändern.

Die aus den Prozesse abgeleiteten Funktionalitäten können nun genutzt werden, um das zu entwickelnde System, wie im folgenden Kapitel, weiter zu spezifizieren.

5 Spezifikation

Um ein genaueres Verständnis davon zu schaffen, was das System leisten soll, werden die im vorherigen Kapitel aufgelisteten Anforderungen genutzt, um eine Spezifikation zu erstellen. Teil einer Spezifikation sind Use Cases, Wireframes, fachliche Datenmodelle und Geschäftsprozesse. Die Geschäftsprozesse wurden im letzten Kapitel bereits präsentiert. Deshalb werden im Folgenden einige Use Cases und dazugehörige Wirframes vorgestellt. Außerdem wird das fachliche Datenmodell, welches durch die fachliche REST Schnittstelle abgebildet werden soll, erklärt.

5.1 Use Cases

Ein Use Case beschreibt die Interaktion zwischen einem System und seinen Benutzern. Diese Interaktion wird aus der Sicht der Benutzer, also ohne Implementierungsdetails vorzugeben, beschrieben. Für die zu entwickelnde Anwendung werden im Folgenden einige Use Cases vorgestellt.

Use Case 1: Beantragen einer Kfz-Versicherung
Akteur: Antragsteller
Ziel: Ein Antragsteller beantragt eine Kfz Versicherung.
Auslöser: Ein Antragsteller entscheidet sich, sein Kraftfahrzeug versichern zu lassen.
Vorbedingungen: Antragsteller besitzt ein Kfz.
Nachbedingungen: Antrag wurde aufgenommen.
Erfolgsszenario:

1. Der Antragsteller ruft die Versicherungswebsite auf.
2. Der Antragsteller füllt das Formular für den Antrag mit folgenden persönlichen Daten aus: Name, Geburtstag, Geschlecht, E-Mail Adresse. Und folgenden Daten über das Fahrzeug: Hersteller und Fahrzeugtyp.

3. Der Antragsteller wählt ein Versicherungsprodukt über das angezeigte Dropdownmenü aus.
4. Das System zeigt einen Kostenvoranschlag für das ausgewählte Produkt.
5. Der Antragsteller klickt auf den “Submit“ Button.
6. Das System erstellt den Antrag und zeigt dem Kunden an, dass dieser erfolgreich erstellt wurde.

Fehlerfälle:

2.a Der Antragsteller hat nicht alle Felder im Formular ausgefüllt. Das System lässt eine Bestätigung des Antrags nicht zu, indem der “Submit“ Button ausgegraut und nicht anklickbar gemacht wird.

2.b Die angegebene E-Mail Adresse entspricht nicht dem Format einer E-Mail Adresse. Das System zeigt beim Klicken auf den “Submit“ Button an, dass die E-Mail Adresse nicht gültig ist.

Use Case 2: Einen Antrag annehmen
Akteur: Sachbearbeiter der Versicherung
Ziel: Ein Sachbearbeiter nimmt einen Versicherungsantrag an.
Auslöser: Es wurde ein Antrag gestellt, der manuell entschieden werden muss.
Vorbedingungen: Der Sachbearbeiter ist eingeloggt.
Nachbedingungen: Antrag wurde angenommen.
Erfolgsszenario:

1. Der Sachbearbeiter ruft die Antragsübersicht auf.
2. Der Sachbearbeiter wählt den Tab “Decision“ unter dem zu entscheidenden Antrag aus.
3. Das System zeigt einen Button, mit dem sich der Bearbeiter der Entscheidung zuweisen kann.
4. Der Sachbearbeiter ordnet sich der Aufgabe mit einem Klick auf den angezeigten Button zu.
5. Das System zeigt das Formular zum Entscheiden über einen Antrag.
6. Der Sachbearbeiter setzt einen Haken in der angezeigten Checkbox und bestimmt die Versicherungsprämie.
7. Der Sachbearbeiter klickt auf den “Submit“ Button.
8. Das System zeigt eine Übersicht über die getätigte Entscheidung an.

Fehlerfälle:

4.a Ein anderer Sachbearbeiter hat sich dem Antrag bereits zugeordnet. Das System zeigt an, dass sich der Bearbeiter nicht dem Antrag zuweisen kann.

7.a Der Sachbearbeiter wurde als Bearbeiter für den Antrag entfernt. Das System zeigt an, dass der Bearbeiter nicht mehr dem Antrag zugeordnet ist.

Use Case 3: Dokumente Anfordern

Akteur: Sachbearbeiter der Versicherung

Ziel: Ein Sachbearbeiter fordert Dokumente beim Antragssteller an.

Auslöser: Ein Sachbearbeiter benötigt Dokumente, um über einen Antrag entscheiden zu können.

Vorbedingungen: Der Sachbearbeiter ist eingeloggt.

Nachbedingungen: Der Antragssteller wurde informiert,dass Dokumente benötigt werden.

Erfolgsszenario:

1. Der Sachbearbeiter ruft die Antragsübersicht auf.
2. Der Sachbearbeiter wählt den Tab “Documents“ unter dem Antrag aus für den Dokumente benötigt werden.
3. Das System zeigt eine Übersicht über die bereits eingereichten Dokumente und ein Formular, um Dokumente anzufordern.
4. Der Sachbearbeiter gibt in das angezeigte Textfeld eine Beschreibung für das benötigte Dokument ein.
5. Der Sachbearbeiter bestätigt seine Eingabe mit dem “Submit“ Button.
6. Das System aktualisiert die Dokumentenansicht.
7. Das System benachrichtigt den Antragsteller per E-Mail, dass Dokumente benötigt werden.

Fehlerfälle:

5.a Ein anderer Sachbearbeiter hat bereits ein Dokument angefordert, das noch nicht eingereicht wurde. Das System zeigt an, dass keine Dokumente angefordert werden können.

Use Case 4: Dokumente Hochladen
Akteur: Antragsteller
Ziel: Ein Antragsteller lädt ein Dokument hoch.
Auslöser: Ein Sachbearbeiter hat ein Dokument angefordert.
Vorbedingungen: Der Antragsteller wurde per E-Mail informiert, dass Dokumente benötigt werden.
Nachbedingungen: Der Antragssteller hat fehlende Dokumente hochgeladen.
Erfolgsszenario:

1. Der Antragsteller klickt den angezeigten Link in der E-Mail an, die darüber informiert, dass Dokumente benötigt werden.
2. Das System zeigt eine Übersicht über den gestellten Antrag.
3. Der Antragsteller klickt auf den Tab “Documents“.
4. Das System zeigt eine Übersicht über die bereits hochgeladenen Dokumente, die Beschreibung des benötigten Dokuments und einen “Choose File“ Button.
5. Der Antragsteller klickt den “Choose File“ Button an.
6. Das System zeigt einen Dateiauswahldialog.
7. Der Antragsteller klickt die hochzuladende Datei an.
8. Das System zeigt den Namen der ausgewählten Datei an.
9. Der Antragsteller klickt auf den “Upload“ Button.
10. Das System speichert die ausgewählte Datei.
11. Das System aktualisiert die Übersicht über die hochgeladenen Dokumente.

Fehlerfälle:

1.a Der Antrag wurde bereits entschieden. Das System leitet den Antragsteller auf die Startseite.
9.a Das benötigte Dokument wurde bereits hochgeladen. Das System zeigt an, dass keine Dokumente mehr benötigt werden.

5.2 Wireframes

Für die Entwicklung des Frontends werden vorher Skizzen gemacht, die das Konzept des Frontends darstellen sollen. Das Ziel ist dabei, nicht das Frontend mit Details wie Farben und Bildern zu spezifizieren. Zusätzlich können die Wireframes als Ergänzung für die erarbeiteten Use Cases verwendet werden, um das Verständnis für diese zu fördern. Im Folgenden werden Wireframes vorgestellt, die als Ergänzung für die zuvor gezeigten Use Cases erstellt wurden. Die Wireframes wurden mit dem Tool balsamiq erstellt[1].

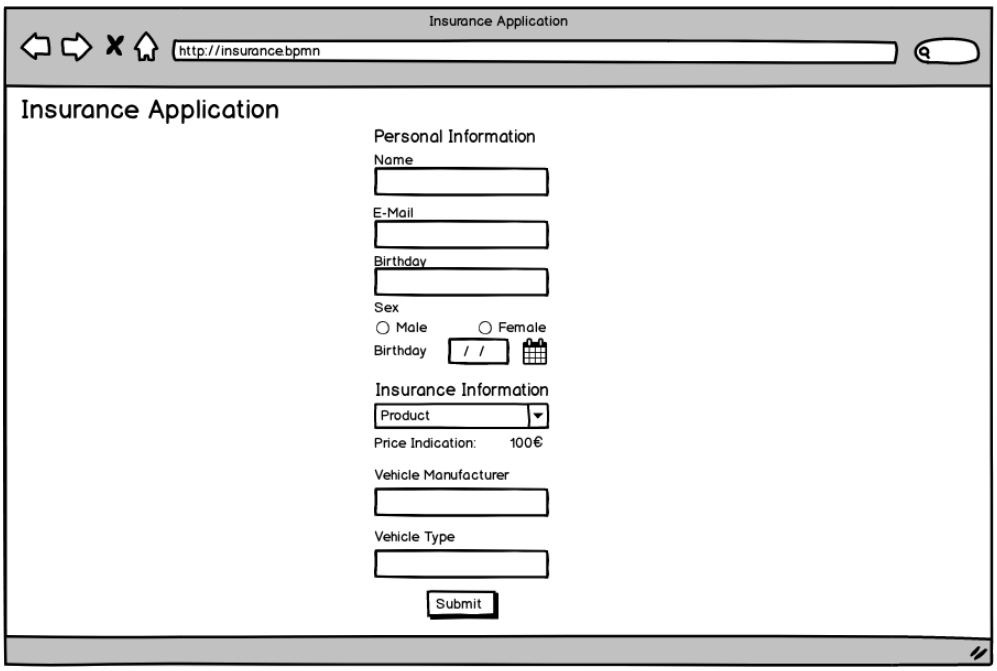

Abbildung 5.1: Wireframe für Beantragung einer Versicherung (Use Case 1)

[1] balsamiq.com

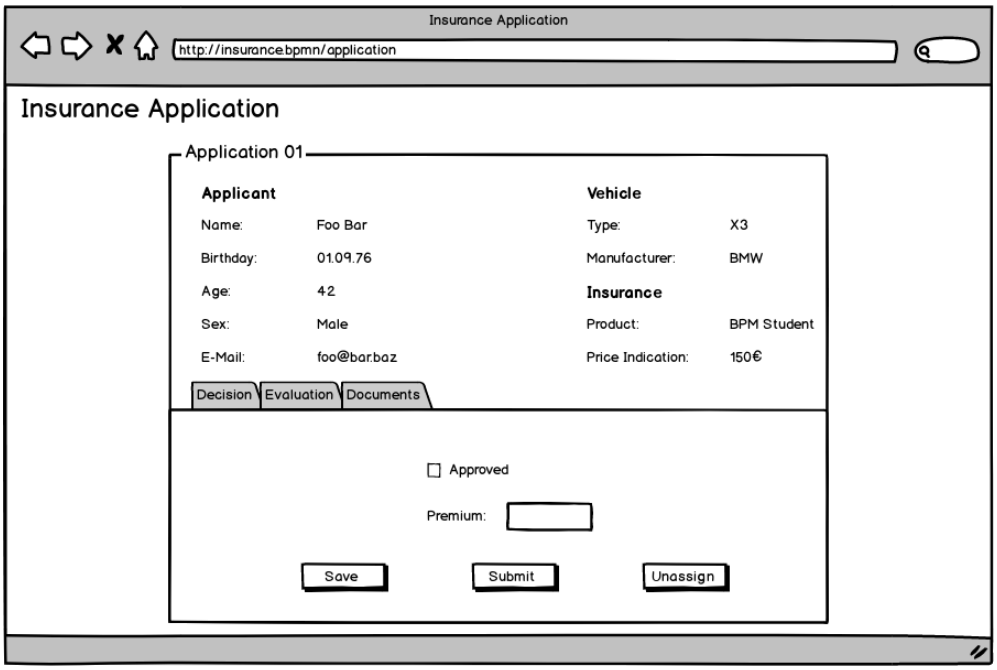

Abbildung 5.2: Wireframe für Entscheidung über einen Antrag (Use Case 2)

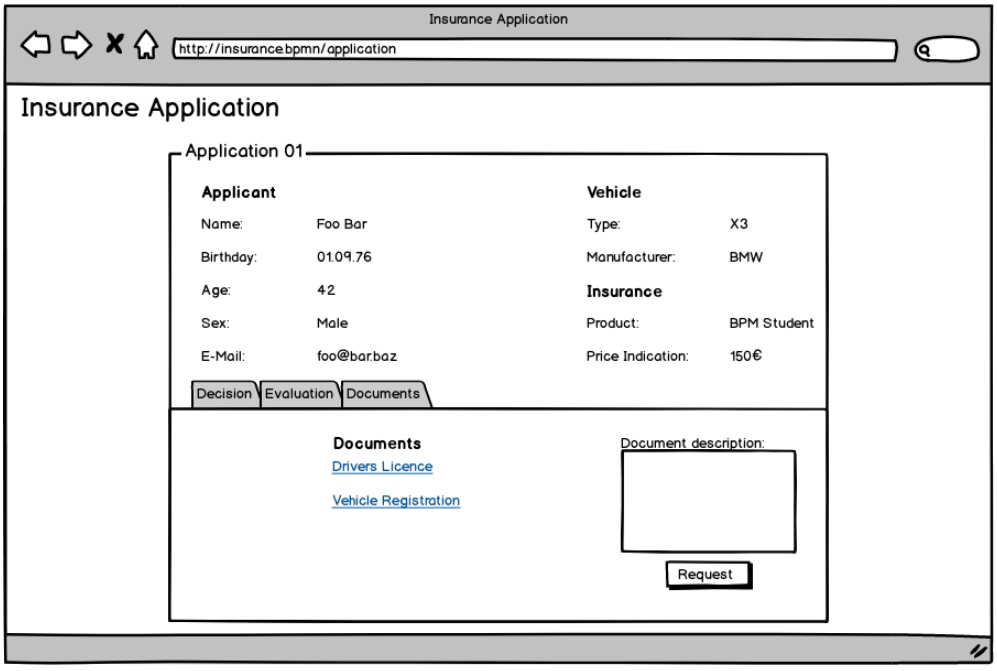

Abbildung 5.3: Wireframe für Dokumentenanforderung (Use Case 3)

5.3 Fachliches Datenmodell

Das fachliche Datenmodell dient der Darstellung wichtiger Konzepte aus dem Anwendungsbereich und kann als Grundlage für weitere Artefakte wie einem physischen Datenmodell oder einer Schnittstellenspezifikation verwendet werden. Abbildung 5.4 zeigt das Datenmodell der Anwendung als UML Klassendiagramm. Im Fokus der Abbildung steht der Antrag, welcher von einer Person gestellt wird. Zu einem Antrag gehört dann eine Entscheidung die diesen annimmt oder ablehnt. Diese wird von einem Sachbearbeiter gefällt. Wenn, wie im Prozessmodell spezifiziert, die Prämie der Entscheidung größer als 300 ist, muss die Entscheidung genehmigt werden. Weiterhin gehören zu einem Antrag Dokumente. Diese werden von Sachbearbeitern angefordert und von Personen eingereicht.

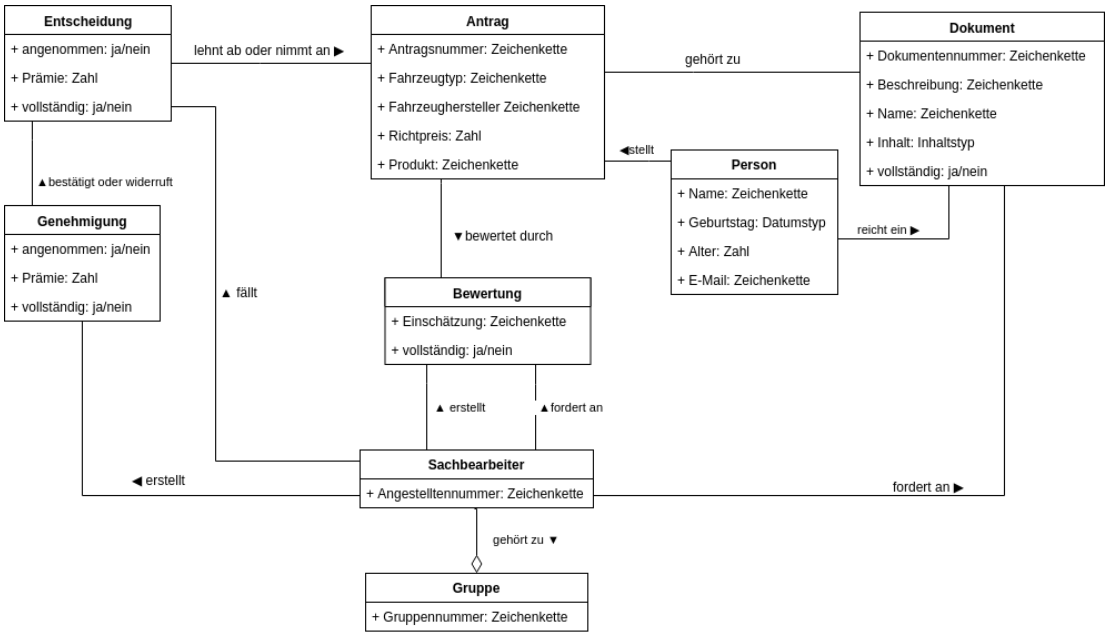

Abbildung 5.4: Fachliches Datenmodell der Anwendung

6 Architektur

In diesem Kapitel soll die Architektur der Beispielanwendung vorgestellt werden. Dazu wird als erstes die fachliche API vorgestellt sowie die Architektur des REST Service erklärt. In Kapitel 6.2 wird die Architektur der zu entwickelnden Frontendanwendung beschrieben. Dabei wird zunächst auf die Architektur des Frontends, welche die fachliche API verwendet, und dann auf die Architektur des Frontends, welche die generische API verwendet, eingegangen.

6.1 Backend

Als Referenzarchitektur wurde die von Professor Sarstedt im Modul “Architekur von Informationssystemen“ vorgestellte Q3/HAW-Referenzarchitektur verwendet (vgl. Sarstedt (2017)).

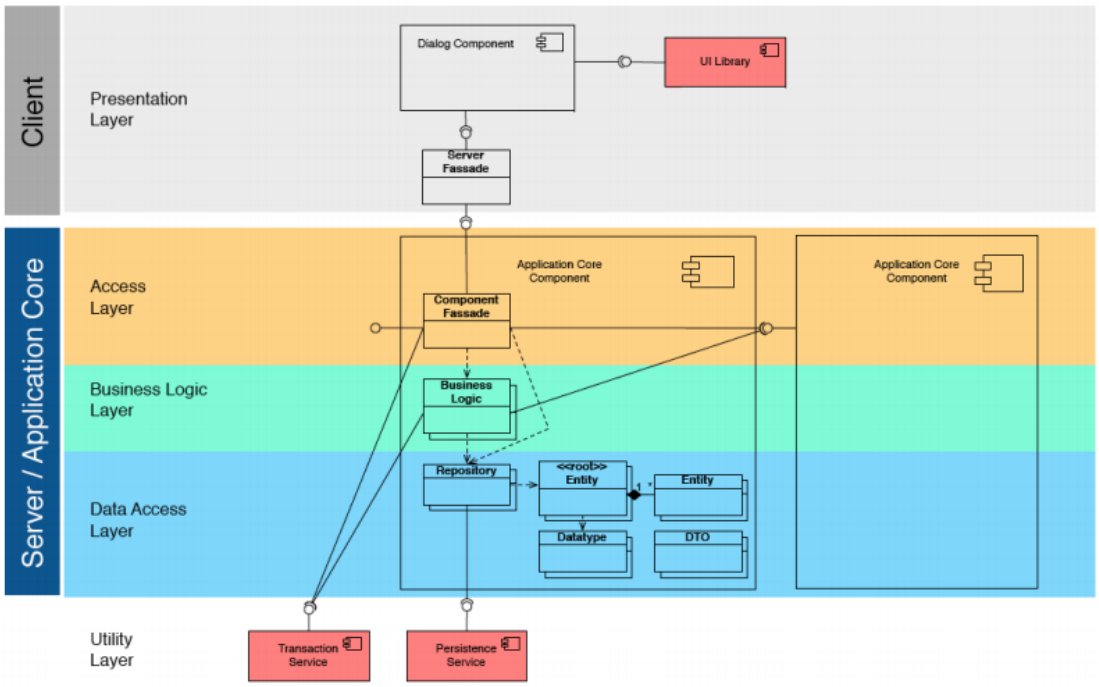

Abbildung 6.1: Q3/HAW-Referenzarchitektur (vgl. Sarstedt (2017))

Durch das Verwenden dieser Architektur soll eine klare Trennung von Technik und Fachlichkeit erreicht werden. Dafür ist eine Unterteilung in Präsentationsschicht und Anwendungskern vorgesehen. Der Anwendungskern teilt sich vertikal in Komponenten auf, die von der Fachlichkeit bestimmt werden, um eine fachliche Trennung von Zuständigkeiten zu erreichen. Horizontal teilt sich der Anwendungskern in Access Layer, Business Logic Layer und Data Access Layer auf. Die Data Access Layer bietet die Schnittstellen einer Komponente sowie eine Fassade, die diese Schnittstellen an andere Komponenten oder unter ihr liegende Schichten weiter delegiert. Die Business Logic Layer implementiert die Anwendungslogik und kann andere Komponenten sowie die von der Data Access Layer angebotenen Repositories verwenden. Die Data Access Layer ermöglicht Zugriff auf persistierte Daten durch Implementierung der Repositories.

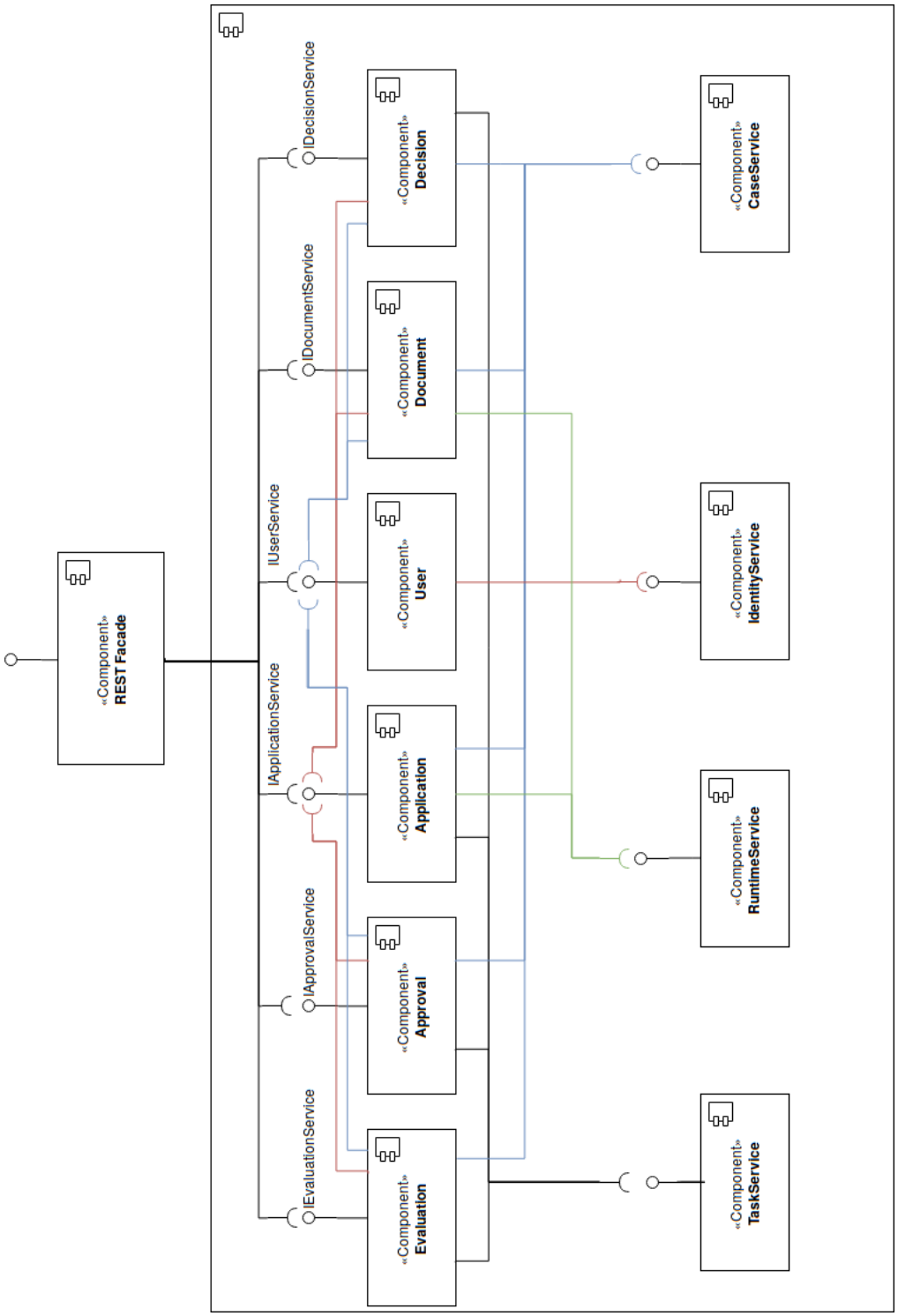

Abbildung 6.2: Architektur des fachlichen REST Service

Abbildung 6.2 zeigt die Komponenten der Anwendung. Da die Process Engine sämtliche Anwendungsdaten verwaltet, realisiert diese die Datenzugriffsschicht der Referenzarchitektur. Jede Komponente ist für Operationen auf einen Entitätstyp zuständig. Die Evaluation-Komponente verwaltet Bewertungsobjekte und die Decision-Komponente Entscheidungsobjekte. Hier ist auch das Mapping zwischen der Fachlichkeit der Anwendung und der Fachlichkeit der Engine zu erkennen. Jeder Entitätstyp, außer Angestellter und Antrag, ist mit einem Task verknüpft. Jede Komponente verwaltet einen Enitätstypen, indem diese den damit verknüpften Task steuert und die Variablen in der Process Engine modifiziert. Durch die Verwendung der Referenzarchitektur kann auf Änderungen des Prozessmodels, wie das Hinzufügen oder Entfernen von Tasks, reagiert werden, ohne dass komponentenübergreifend Code verändert werden muss. Da jede Resource die einen Human Task repräsentiert, das Zuweisen, das Entfernen und das Betrachten des Bearbeiters unterstützen soll, muss dies auch in jeder Komponentenklasse realisiert werden. Um Codeduplikate und Aufwand zu reduzieren, wird die abstrakte Klasse AbstractHumanTaskComponent eingefügt. Jede Komponente, die einen Human Task verwaltet, erbt die Implementierung dieser Klasse. Außerdem wird die Abstrakte Klasse AbstractDecisionTaskComponent eingeführt, um die Implementierung der Decision Komponente sowie der Approval Komponente zu vereinfachen. Denn beide Komponenten realisieren eine Entscheidung über einen Versicherungsantrag. Die Vererbungshierachie der Komponentenklassen für Human Tasks ist in Abbildung 6.3 dargestellt.

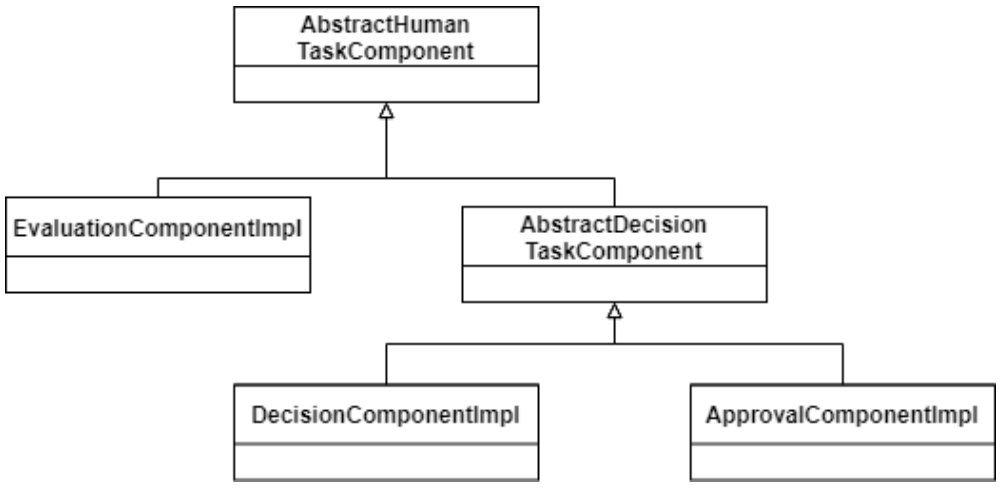

Abbildung 6.3: Vererbungshierachie der Komponentenklassen

6.1.1 Deployment

Um die Kopplung zwischen Process Engine und fachlicher REST Schnittstelle zu reduzieren, ist es denkbar, dass die fachliche Schnittstelle die REST API der Process Engine verwendet, um mit dieser zu kommunizieren. Dies würde in einem Microservice Szenario dem *API Gateway Pattern* entsprechen. (vgl. Microsoft Developer Division) Hierdurch würde sich allerdings der Aufwand für Test und Deployment erhöhen. Da das Ziel dieser Arbeit der Vergleich von Schnittstellen ist, und um die Komplexität für Test und Deployment nicht unnötig zu erhöhen ist die REST Schnittstelle in Java implementiert und verwendet die Java Programmiersprachen API der Process Engine. Abbildung 6.4 zeigt das Deployment für die Umsetzung mit der Programmiersprachen API. Abbildung 6.5 zeigt das alternative Deployment in dem die fachliche API in Python realisiert ist.

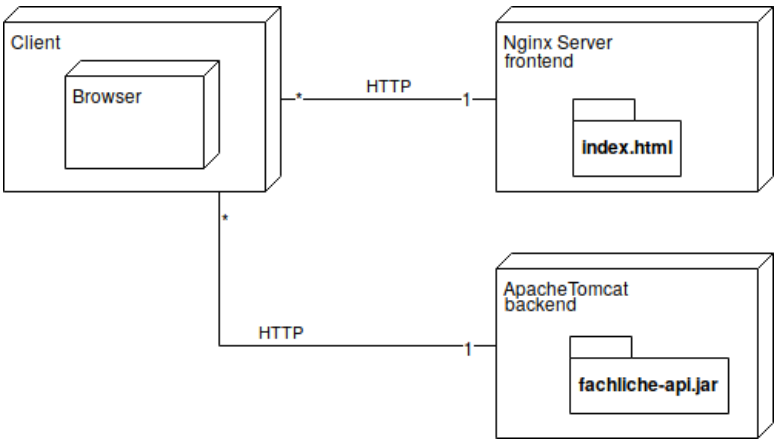

Abbildung 6.4: Umgesetztes Deployment

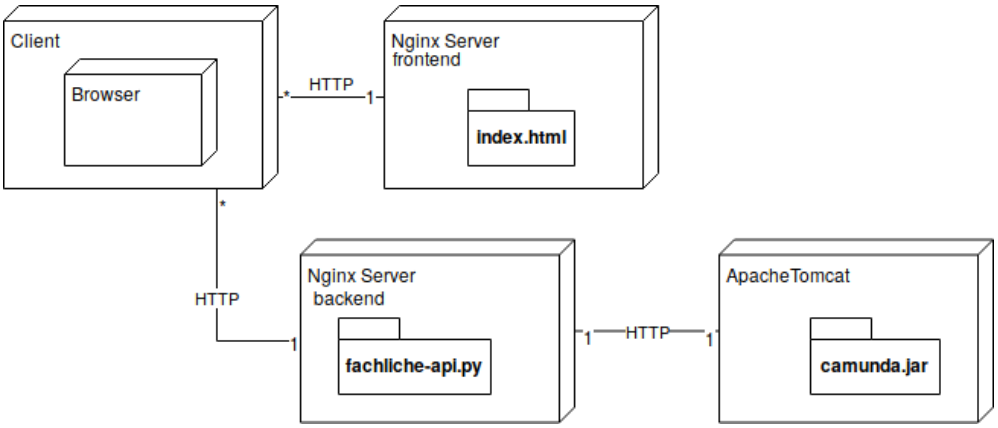

Abbildung 6.5: Alternatives Deployment

6.1.2 Beschreibung der REST API

Die fachliche Schnittstelle soll von Entwicklern ohne Wissen über die verwendete Process Engine, also ohne Wissen über Prozessinstanzen oder Prozessausführungen, verwendet werden können. Um das zu erreichen, bildet die REST API das Domänenmodell (Abbildung 5.4) der Anwendung durch die angebotenen Resourcen und ihre Links ab.

Jede erzeugte Prozessinstanz wird in der fachlichen API durch eine neue Antragsresource dargestellt. Einzelne Tasks werden dann durch Verwendung der Operationen, welche die REST API zur Verfügung stellt, gestartet, bearbeitet und beendet.

Endpunkte und Methoden

Beispielhaft werden im Folgenden die möglichen Operationen auf eine Bewertung für einen Antrag mit der Antragsnummer 42 dargestellt:

Betrachten einer Bewertung
URI: /application/42/evaluation
Methode: GET
Parameter: Basic-Auth Header
Rückgabe: Bewertungsobjekt

Modifizieren einer Bewertung
URI: /application/42/evaluation
Methode: PUT
Parameter: modifiziertes Bewertungsobjekt, Basic-Auth Header
Rückgabe: modifiziertes Bewertungsobjekt

Betrachten des aktuellen Bearbeiters
URI: /application/42/evaluation/assignee
Methode: GET
Parameter: Basic-Auth Header
Rückgabe: Bearbeiterobjekt

Bearbeiter zuordnen
URI: /application/42/evaluation/assignee
Methode: PUT
Parameter: Basic-Auth Header
Rückgabe: modifiziertes Bewertungsobjekt (mit neuem Assignee)

Entfernen des aktuellen Bearbeiters
URI: /application/42/evaluation/assignee
Methode: DELETE
Parameter: Basic-Auth Header
Rückgabe: modifiziertes Bewertungsobjekt (ohne Assignee)

Beantragen einer Bewertung
URI: /application/42/evaluation
Methode: POST
Parameter: Basic-Auth Header
Rückgabe: leeres Bewertungsobjekt

Da neu erzeugte Bewertungen alle mit Standardwerten instantiiert werden, ist bei dem POST Request für das Beantragen einer Bewertung kein Bewertungsobjekt im Requestbody notwendig. Bei der Zuordnung eines Bearbeiters wird die ID des Bearbeiters durch die Authentifikation des Clients ermittelt. Deshalb müssen im Requestbody keine Daten angegeben werden. Es sind nicht alle Operationen zu jeder Zeit ausführbar. Die Ausführbarkeit einer Operation auf eine Resource wird durch den Zustand der Resource bestimmt. Die erreichbaren Zustände sowie die

möglichen Operationen einer Bewertungsresource sind in Abbildung 6.6 als Zustandsdiagramm dargestellt.

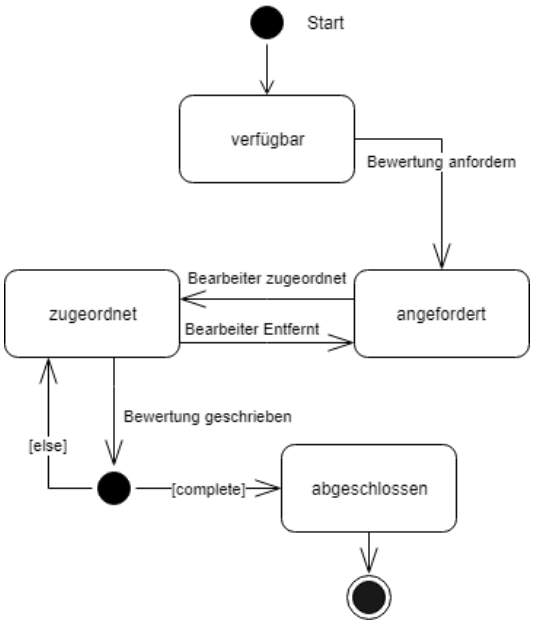

Abbildung 6.6: Zustandsdiagramm für eine Bewertungsresource

Hier ist auch zu erkennen, dass die Zustände einer Resource (Abbildung 6.6) aus dem CMMN Lebenszyklus eines Tasks (Abbildung 2.8) abgeleitet sind. Der Zustand *verfügbar* aus dem Zustandsdiagramm entspricht dem Zustand *vorhanden* im Lebenszyklus des Tasks zur Bewertung eines Antrags. Die Zustände *zugeordnet* und *angefordert* lassen sich auf den Zustand aktiv des Lebenszyklus abbilden. Wenn die Bewertung *abgeschlossen* ist, befindet sich der Task im Zustand *erledigt*.

Repräsentation der Resourcen

Die Resourcen werden im JSON Format dargestellt und setzen HATEOAS durch die Realisierung der HAL Spezifikation um. Dadurch lässt sich über das _links Feld der aktuelle Zustand einer Resource ermitteln, indem das Vorhandensein bestimmter Felder überprüft wird. Deshalb werden alle Requests mit der aktualisierten Resource beantwortet. Clients erhalten nach der Änderung einer Resource ihren aktualisierten Zustand. Als Beispiel ist eine Entscheidungsresource, die den Antrag mit der Nummer 42 mit einer Prämie in Höhe von 150 Euro angenommen

hat, in Listing 6.1 abgebildet. Hierbei befindet sich die Resource im Zustand *abgeschlossen*. Der Inhalt des _links Feldes zeigt immer die Operationen, die aktuell auf eine Resource anwendbar sind. Das _links Feld einer Entscheidungsresource, welche sich im Zustand *zugeordnet* befindet, ist in Listing 6.2 abgebildet.

```
{
  "approved": true,
  "premium": 15000,
  "complete": true,
  "_links":     {
    "self": {
          "href": "/application/42/decision"
        },
        "assignee": {
          "href": "/application/42/decision/assignee"
        },
        "application": {
          "href": "/application/42"
        }
  }
}
```

Listing 6.1: Schema für Entscheidungsobjekte im Zustand “abgeschlossen“

```
{
  "self": {
        "href": "/application/42/decision"
  },
  "decide": {
        "href": "/application/42/decision"
  },
  "unassign": {
        "href": "/application/42/decision/assignee"
  },
  "assignee": {
        "href": "/application/42/decision/assignee"
  },
  "application": {
        "href": "/application/42"
  }
}
```

Listing 6.2: _links Feld einer Entscheidungsresource im Zustand “zugeordnet“

Das Feld *decide* zeigt den Link an mit dem eine Entscheidung abgegeben werden kann. Das Feld unassign signalisiert, dass der Bearbeiter mit gegebenem Link entfernt werden kann.

Fehlerfälle

Im Falle eines Fehlers wird mit einem String, der den Fehler beschreibt, geantwortet sowie mit den in Fielding und Reschke (2014) spezifizierten HTTP Statuscodes. Mögliche Fehlerfälle sind:

- Die Angeforderte Resource existiert nicht.
  Responsecode: 404 Not Found
- Ein Bearbeiter ist nicht autorisiert eine Resource zu modifizieren.
  Responsecode: 401 Unauthorized
- Das Content-Type Headerfeld ist nicht auf application/json gesetzt.
  Responsecode: 415 Unsupoorted Media Type
- Die Daten im Requestbody sind nicht im JSON Format.
  Responsecode: 400 Bad Request

- Eine Operation kann aufgrund des Zustands einer Resource nicht ausgeführt werden. Aus fachlicher Sicht können dies folgende Fälle sein:
  Responsecode: 409 Conflict

  - Ein Angestellter hat sich für eine Aufgabe zugeteilt, die bereits einen Bearbeiter hat.
  - Ein Angestellter bearbeitet eine Aufgabe, für die er nicht zugeteilt ist.
  - Ein Angestellter bearbeitet eine Aufgabe, die bereits beendet ist.
  - Ein Angestellter fordert für einen Antrag eine Bewertung an, für den dies bereits getätigt wurde.
  - Ein Angestellter fordert für einen Antrag Dokumente an, für den dies bereits getätigt wurde.
  - Ein Antragsteller versucht Dokumente hochzuladen, obwohl keine angefordert wurden.

Die vollständige Dokumentation der API ist über swagger als OpenAPI Spezifikation einsehbar[1].

6.2 Frontend

Das Frontend orientiert sich, wie das Backend, an dem fachlichen Datenmodell der Anwendung. Hierbei findet ebenfalls eine Aufteilung in fachliche Komponenten statt. Es wird das Frontendframework Angular verwendet, da dies eine Aufteilung der Anwendung in Komponenten vorsieht. Jede Komponente ist für die Darstellung der ihr zugeordneten Daten zuständig. Um die Komponenten in ihrer Implementierung übersichtlich zu halten, werden diese in weitere Unterkomponenten aufgeteilt. Abbildung 6.7 zeigt die vollständige Komponentenhierarchie der Frontendanwendung. Die ApplicationListComponent realisiert die Ansicht für Sachbearbeiter und stellt alle offenen Anträge dar. Dies macht auch die ApplicationCustomerViewComponent, allerdings stellt diese nur einen einzigen Antrag für den Antragsteller dar. Die Kommunikation mit dem Backend wird durch Services realisiert, die von Komponenten verwendet werden können.

[1] https://app.swaggerhub.com/apis/dennis_p/application-process-api/1.0.0

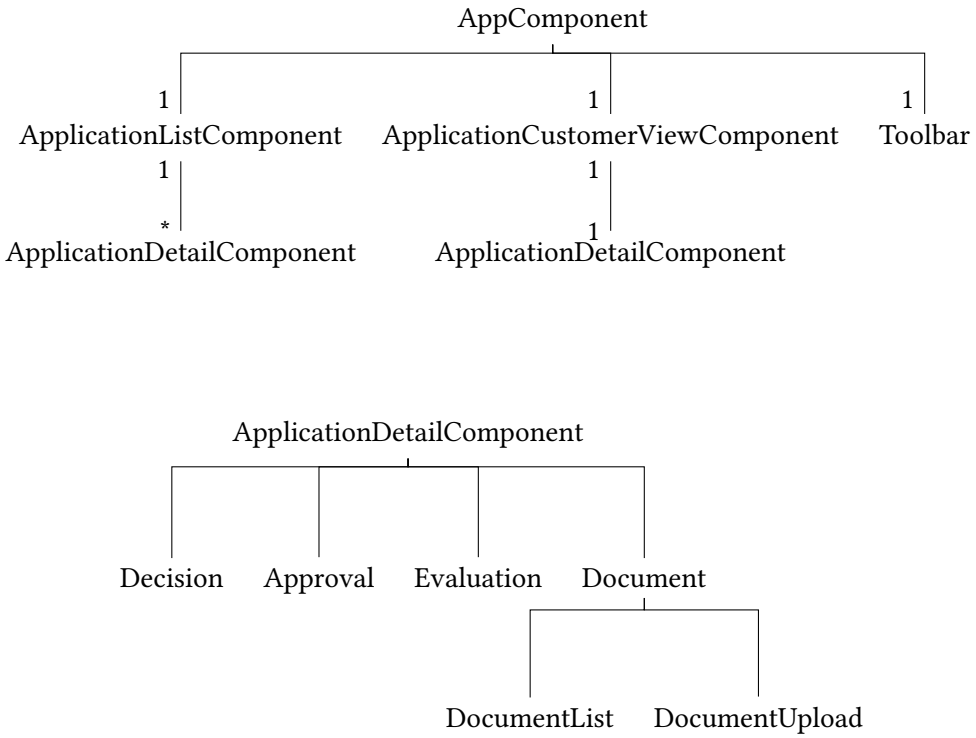

Abbildung 6.7: Komponentenhierarchie der Frontendanwendung

6.2.1 Unter Verwendung der fachlichen API

Im Frontend, in dem die fachliche API verwendet wird, sind die Services nach der Fachlichkeit der Anwendung aufgeteilt (siehe Abbildung 6.8). Jeder Service stellt einer Komponente die benötigten Zugriffe auf die Entitäten der Anwendung über die REST API zur Verfügung. Der einzige Service, der von allen Komponenten verwendet wird, ist der User Service, da dieser die Daten verwaltet, die für die Authentifizierung von Requests notwendig sind. Um eine Liste mit allen zu bearbeitenden Anträgen darzustellen, macht die ApplicationList Komponente ein Request, um eine Liste mit allen Anträgen zu erhalten. Abbildung 6.9 zeigt die Kommunikation zwischen Frontend und Backend sowie zwischen den einzelnen Komponenten der Frontendanwendung. In diesem Beispiel existiert ein zu bearbeitender Antrag bei dem keine Tasks bearbeitet wurden. Nachdem die ApplicationList Komponente eine Liste mit einem offenen Antrag erhalten hat, verteilt diese den Antrag auf eine Application Komponente. Da die Darstellung des Antrags im _embedded Feld alle verknüpften Resourcen enthält, können diese an die jeweils zuständigen Komponenten verteilt werden. Die Approval Komponente erhält dabei null, weil die Bestätigung einer Entscheidung erst dann notwendig ist, wenn die Entscheidung gefällt wurde. Deshalb existiert noch kein Approval Objekt. Die DocumentComponent erhält eine Liste mit Linkobjekten zu Dokumenten, die bereits hochgeladen wurden. Da dies in dem Beispiel noch nicht geschehen ist, erhält die DocumentComponent eine leere Liste.

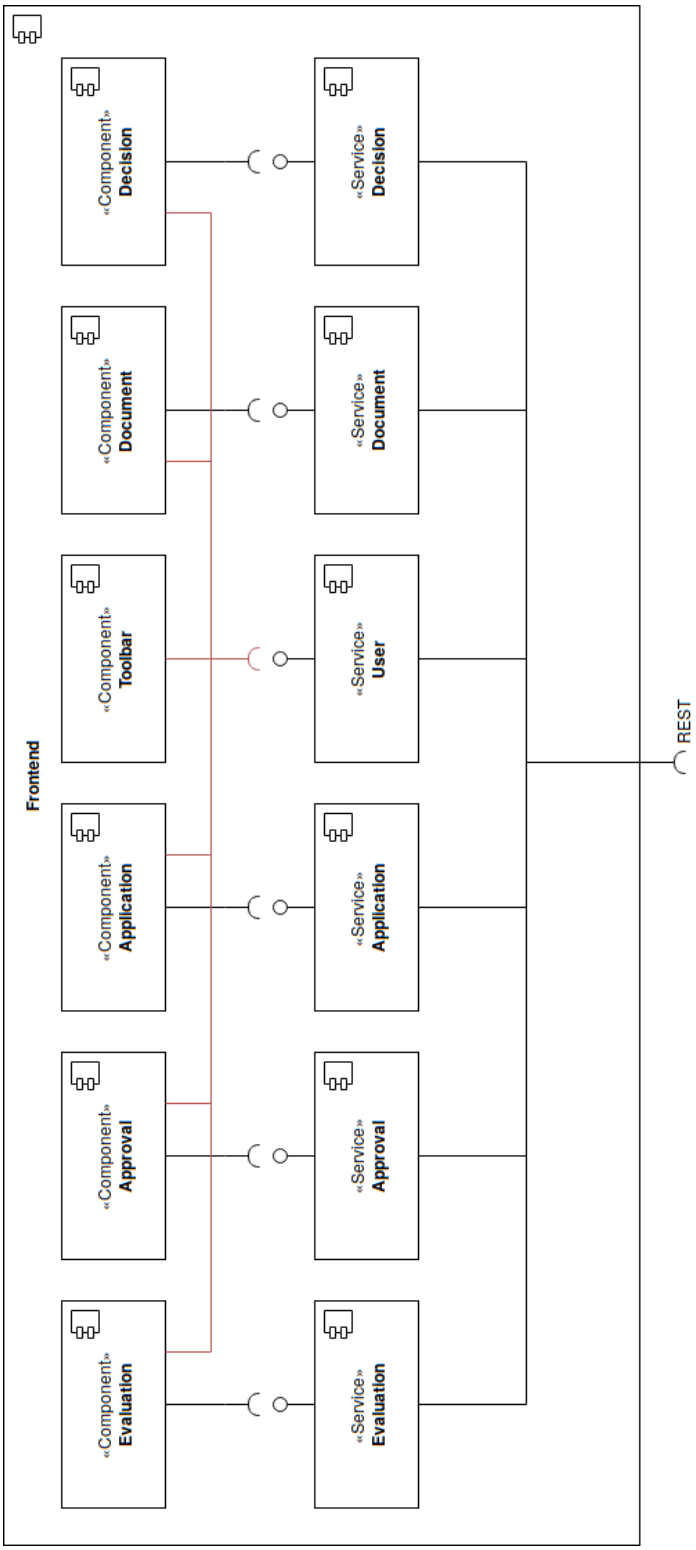

Abbildung 6.8: Komponenten und Services für ein Frontend mit fachlicher API

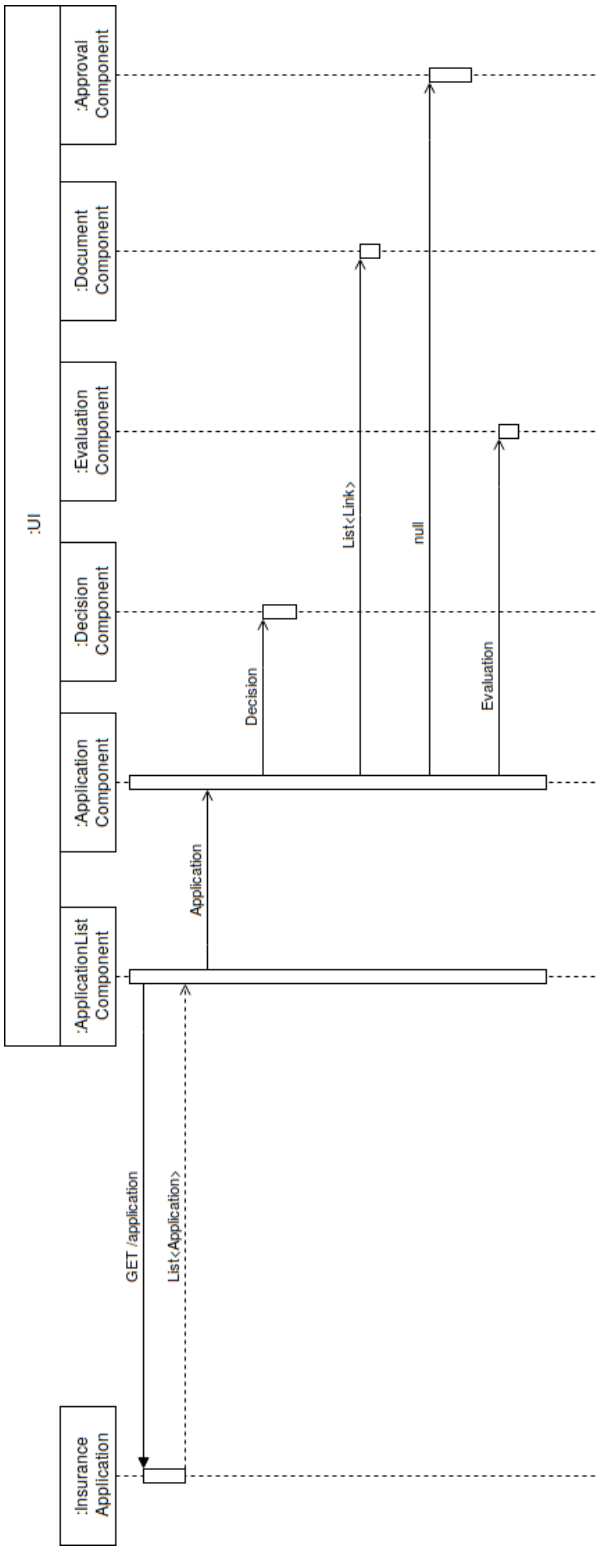

Abbildung 6.9: Requests zur Darstellung des Frontends mit fachlicher API

6.2.2 Unter Verwendung der generischen API

Im Frontend, in dem die generische API verwendet wird, sind die Services nach den Typen der Process Engine aufgeteilt. Jeder Service ist für Operationen auf einen Datentyp der Process Engine (Tasks, Variablen, Ausführungen, Instanzen) zuständig. Eine Ausnahme ist dabei der Identity Service, der sowohl für die Verwaltung von Benutzern als auch Benutzergruppen zuständig ist. Abbildung 6.11 zeigt die Kommunikation zwischen Frontend und Backend sowie zwischen den einzelnen Komponenten der Frontendanwendung. Allerdings wird dies hier für die Anwendung, welche die generische Schnittstelle verwendet, abgebildet. In diesem Beispiel exisitert nur ein Antrag für den noch keine Tasks bearbeitet wurden. Die ApplicationList Komponente macht zu Beginn ein Request, um eine Liste mit allen aktiven Case Instanzen zu erhalten. Nachdem die ApplicationList Komponente eine Liste mit einer aktiven Case Instanz erhalten hat, wird diese an eine ApplicationComponente weitergegeben, die nun für die Darstellung dieser Instanz zuständig ist. Um die Antragsdaten zu laden, wird zunächst die Variable *application* geladen. Dann werden alle Case Ausführungen, die zu der zuvor übergebenen Instanz gehören, geladen und an die Komponenten weitergegeben, die für den Task zuständig sind, welcher mit der gegebenen Case Ausführung verknüpft ist. Wenn die Case Ausführung einer Komponente aktiv ist, wird der dazugehörige Task geladen. In diesem Beispiel trifft dies nur auf die Decision Komponente zu. Diese lädt dann die Variablen *approved* und *premium*, um diese anzuzeigen. Die Document Komponente lädt immer eine Liste mit allen Dokumenten, da es möglich ist, dass der Dokumentenanforderungsprozess zuvor bereits ausgeführt wurde.

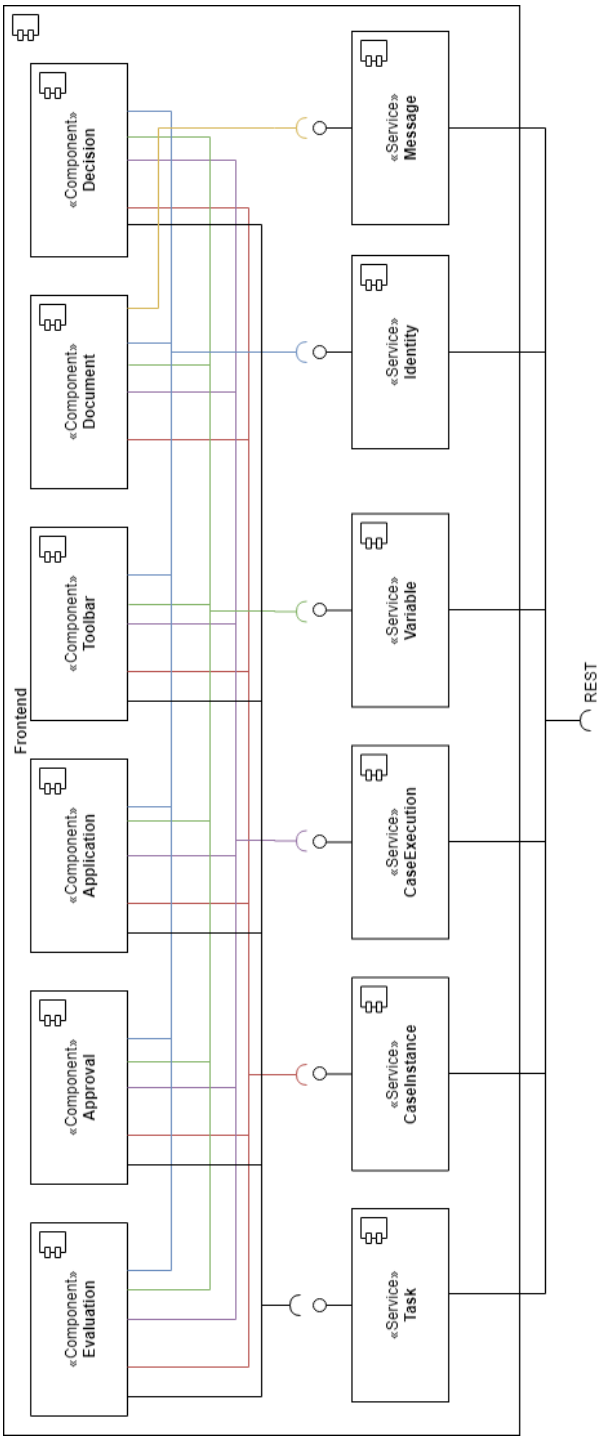

Abbildung 6.10: Komponenten und Services für ein Frontend mit generischer API

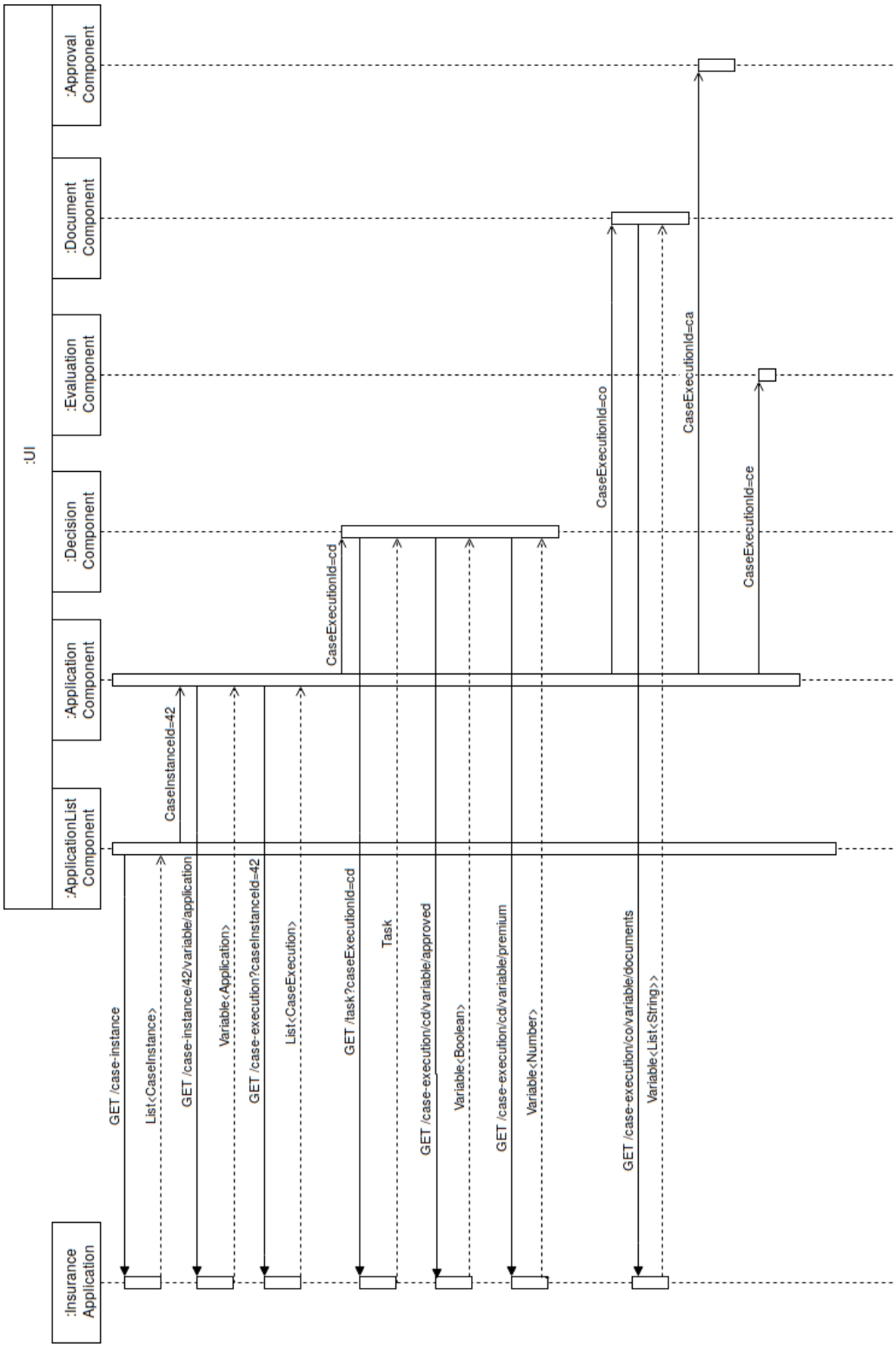

Abbildung 6.11: Requests zur Darstellung des Frontends mit generischer API

7 Realisierung

Im Folgenden wird die Umsetzung der Anwendung beschrieben. Diese basiert auf den zuvor getroffenen Entwurfsentscheidungen. Beginnend mit der Umsetzung des Backends werden die Bibliotheken vorgestellt, welche für die Realisierung von Tests, HATEOAS sowie der Authentifizierung verwendet wurden. Einige Besonderheiten werden durch Codebeispiele vorgestellt.

7.1 Backend

7.2 Der Beispielprozess

Als Grundlage für die Realisierung des Beispielprozesses dient eine von Camunda veröffentlichte Demo[1]. Die Dateien aus dem Package com.camunda.demo sind aus diesem Beispiel übernommen und wurden an das Szenario angepasst. In der Demo ist die manuelle Prüfung nicht als Case, sondern als Prozess modelliert und wurde deshalb mit dem Camunda Modeler umgesetzt. Der in 2.1.2 beschriebene Case konnte nicht wie spezifiziert umgesetzt werden, da in der aktuellen Version von Camunda, entgegen der CMMN 1.1 Spezifikation, keine Sentries als Austrittskriterium an Stages verwendet werden können. (vgl. Camunda (c)) Deshalb wird in den Sentries nicht auf das Austrittskriterium geprüft, sondern auf den Zustand der Stage. Die Sentries an den Meilensteinen wurden als logischer Ausdruck realisiert. Listing 7.1 zeigt den Ausdruck am Meilenstein "application denied". Listing 7.2 zeigt den Ausdruck am Meilenstein "application approved". Damit sieht der umgesetzte Case wie in Abbildung 7.1 aus.

```
1 #{!approved}
```

Listing 7.1: Bedingung für Meilenstein "application denied"

[1] https://github.com/camunda-consulting/camunda-showcase-insurance-application

```
1 #{approved}
```

Listing 7.2: Bedingung für Meilenstein “application approved“

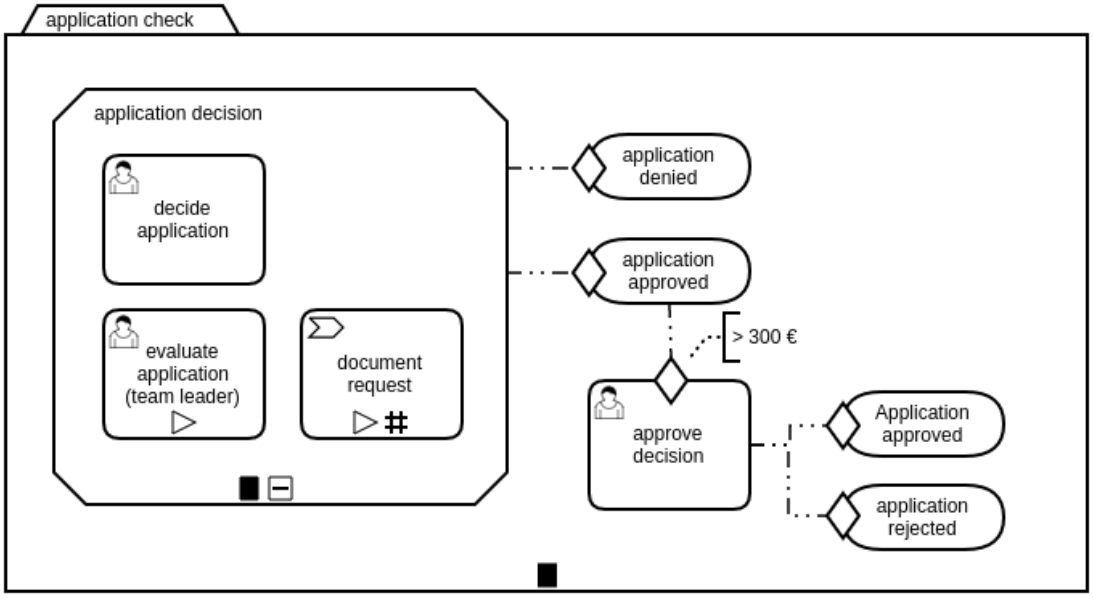

Abbildung 7.1: Umgesetzter Case ohne Exit Criterion

Die Send Tasks implementieren keine vollständige Anbindung an einen Mail Server, sondern geben den zu versendenden Mailtext lediglich auf der Konsole aus.

7.2.1 Test und Deployment

Spring Boot folgt dem Grundatz “convention over configuration“ und erspart somit aufwendige Konfigurationen. Darüber hinaus bietet das Framework einen eingebeteten HTTP Server an, der das Deployment der Anwendung stark vereinfacht. Camunda bietet die Bibliothek camunda-bpm-spring-boot-starter, welche eine problemlose Integration der Cmaunda BPM Engine in eine Spring Boot Anwendung ermöglicht. Um die einzelnen Komponenten zu testen, wurde die Bibliothek Camunda BPM Assert[2] verwendet, welche das Testen von Prozessen und Cases durch ein fluent interface vereinfachen soll. Listing 7.3 zeigt eine Beispielmethode, in der das fluent interface bei Tests für die Entscheidungskomponente verwendet wurde.

[2] https://github.com/camunda/camunda-bpm-assert

```
@Before
public void createDecisionTask(){
        ProcessInstance processInstance = createProcessInstance();
        assertThat(processInstance).job();
        execute(job());
        assertThat(processInstance)
                        .isWaitingAt("UserTask_DecideOnApplication");
}
```

Listing 7.3: Beispielmethode BPM Assert

7.2.2 HATEOAS

Für die Umsetzung von HATEOAS wurde das Projekt Spring HATEOAS verwendet, welches ohne großen Aufwand ermöglicht, REST Resourcen entsprechend der HAL Spezifikation darzustellen. Dazu müssen Klassen, dessen Objekte HAL konform dargestellt werden sollen, von der Klasse *ResourceSupport* erben. Dadurch können Linkobjekte für die Darstellung des _links Feldes mit der Methode ResourceSupport::add(Link link) hinzugefügt werden. Um die Links nicht per Hand zu erstellen, kann die Methode ControllerLinkBuilder:linkTo(Class<?> controller, Method method, Object... parameters) verwendet werden. Diese erzeugt aus den übergebenen Parametern ein Linkobjekt, dass auf die Methode der übergebenen REST Controller Klasse verweist. Dies hat den Vorteil, dass das URL Schema der Anwendung jederzeit geändert werden kann, ohne dass Strings für das Erzeugen der Links angepasst werden müssen.

7.2.3 Authentifizierung und Authorisierung

Um die API abzusichern, wurde die Erweiterung Spring Security verwendet, wodurch Zugriffsberechtigungen für Endpunkte und darauf aufgerufene Methoden konfiguriert werden können (Listing 7.4). Darüber hinaus kann dadurch die Identität des Zugreifenden ermittelt werden. Dadurch ist es möglich, die Darstellung der Resourcen für jeden Zugreifer angemessen darzustellen. So sieht ein Kunde die Bewertung und Entscheidung eines Antrags nicht als eingebettete Resource.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
        .antMatchers(HttpMethod.OPTIONS, "/**").permitAll()
        .antMatchers( "/application").permitAll()
        .antMatchers(HttpMethod.GET, "/application/*").permitAll()
        .antMatchers(HttpMethod.GET,"/application/*").permitAll()
        .antMatchers("/application/*/document/**").permitAll()
        .antMatchers(HttpMethod.GET, "/application/**").authenticated()
        .antMatchers(HttpMethod.GET, "/login").authenticated()
        .and()
        .httpBasic()
        .and()
        .cors().and()
        .csrf().ignoringAntMatchers("/**");
}
```

Listing 7.4: Spring Security Konfiguration

7.2.4 Dependency Management

Für das Dependency Management wurde das Tool gradle verwendet. Durch das Tool lassen sich die Abhängigkeiten der Anwendung in einer auf Groovy basierenden Domain Specific Language beschreiben. Neben den bereits genannten Bibliotheken werden noch weitere benötigt. Die Bibliotheken camunda-spin-dataformat-all und camunda-engine-plugin-spin werden benötigt, um Java Objekte als Prozessvariablen im JSON Format zu speichern. Durch die Bibliothek camunda-template-engine-freemarker kann die Engine JavaScript ausführen, um Templates für E-Mails auszufüllen. Listing 7.2.4 zeigt alle Abhängigkeiten, die im gradle build Script angegeben sind.

```
dependencies {
compile("org.camunda.bpm.springboot:
                    camunda-bpm-spring-boot-starter-webapp:
                    ${camundaSpringBootVersion}")

compile("org.camunda.bpm.springboot:camunda-bpm-spring-boot-starter:
                    ${camundaSpringBootVersion}")

compile("org.camunda.template-engines:
                    camunda-template-engines-freemarker")

testCompile("org.camunda.bpm.springboot:
                    camunda-bpm-spring-boot-starter-test:
                    ${camundaSpringBootVersion}")

compile("org.camunda.spin:camunda-spin-dataformat-all")
compile("org.camunda.bpm:camunda-engine-plugin-spin")
compile('org.springframework.boot:spring-boot-starter-security')
testCompile 'io.rest-assured:rest-assured:3.1.0'
runtime("com.h2database:h2")
compile("org.springframework.boot:spring-boot-starter-hateoas")
}
```

Listing 7.5: Dependency management mit Gradle

7.2.5 Fehlerbehandlung

Wie in 6.1.2 beschrieben, muss die REST API auf Fehlerfälle reagieren. Spring erkennt und reagiert auf Fehler wie nicht gesetzte Header Felder oder Daten, die nicht dem JSON Format entsprechen. Allerdings sind damit nicht anwendungsspezifische Fehler abgedeckt. Die Prüfung auf mögliche Fehler findet in den einzelnen Komponenten statt. Ist ein Fehler gefunden, wird eine fachliche Exception geworfen. Das Spring Framework ermöglicht es, die Fehlerbehandlung außerhalb der Anwendungslogik durchzuführen. Dazu müssen Methoden einer Klasse, die von ResponseEntityExceptionHandler erbt, mit @ExceptionHandler annotiert werden. Der Annotation werden die Exceptions übergeben, die von der annotierten Methode behandelt werden sollen. Dem Client wird dann der Rückgabewert der ausgeführten Methode übergeben. Listing 7.6 zeigt exemplarisch die Fehlerbehandlung für einen unautorisierten Zugriff.

```
@ExceptionHandler(value = {UserNotFoundException.class})
private ResponseEntity<Object> handleUnauthorizedException
                (ApplicationException e, WebRequest webRequest){
        LOGGER.info(e.getMessage());
        return handleExceptionInternal(e, e.getMessage(),
                new HttpHeaders(),
                HttpStatus.UNAUTHORIZED,
                webRequest);
}
```

Listing 7.6: Codebeispiel für Exceptionhandling

7.3 Frontend

Das Frontend ist mit Angular 6 entwickelt. Das Framework bringt alle benötigten Abhängigkeiten für das Testen des geschriebenen Codes und das Ausführen von HTTP Requests mit. Durch die Verwendung der statisch typisierten Sprache Typescript soll die Wartbarkeit und Lesbarkeit des Codes gefördert werden. Für die Gestaltung wurde das CSS-Framework bootstrap verwendet.

8 Evaluation

Während der Realisierung der Frontendanwendung haben sich einige Unterschiede in den API Varianten herausgestellt. Diese werden in diesem Kapitel einzeln vorgestellt. Als erstes wird die RESTfulness der Schnittstellen verglichen. Dann wird mit verschiedenen Metriken gezeigt wie sich die Schnittstellenvarianten auf die Komplexität der Anwendungen auswirken. Anschließend wird die Effizienz und die Bedienbarkeit der Schnittstellen verglichen um im letzten Abschnitt eine abschließende Bewertung vorzunehmen.

8.1 RESTfulness

Das Richardson Maturity Model ermöglicht es den Reifegrad einer API, die den REST Architekturstil umsetzt, zu bewerten, indem diese in eine von vier Stufen eingeordnet ist. Je höher die Stufe, umso stärker wird der Architekturstil umgesetzt. (vgl. Fowler (2010))Tabelle 8.1 zeigt eine Übersicht über die Reifegrade und die Eigenschaften die von der API umgesetzt werden müssen. Eine API mit einem hohen Reifegrad bietet die Vorteile die durch die Verwendung des REST Architekturstils entstehen sollen, wie zum Beispiel Erweiterbarkeit ohne verschiedene API Versionen einführen zu müssen. Die generische API der Process Engine antwortet zwar auf einige Requests in denen der Accept Header auf application/hal+json gesetzt ist mit JSON Objekten die der HAL Spezifikation entsprechen, aber nicht für alle angebotenen Resourcen. So wird beispielsweise auf Anfragen auf die Resource */case-instance* mit entsprechend gesetztem Header, mit HTTP Status 406 (Not Acceptable), anstatt mit einer Repräsentation der Case Instanz im HAL Format, geantwortet. Tasks hingegen können nach HAL Spezifikation dargestellt werden. Das _links Feld enthält Linkobjekte zu verknüpften Resourcen (Anhang A.1). Allerdings lassen sich hierbei keine Aussagen über den Zustand der Resource machen und damit auch keine Aussagen über die möglichen Operationen. Damit ist HATEOAS nicht umgesetzt und die generische API auf Level 2 einzustufen. Dies führt dazu, dass eine Clientseitige Prüfung über die möglichen Operationen implementiert werden muss, was die Komplexität im

Client erhöht. Wie in Listing 6.2 gezeigt, ermöglichen die _links Felder der fachlichen API eine Prüfung auf den Zustand der Resource. Damit ist diese in Level 3 einzuordnen. Listing 8.1 und 8.2 zeigen wie sich dieser Unterschied auf den Clientcode auswirkt. Die Codebeispiele zeigen die Überprüfungen die nötig sind, um zu entschieden, ob ein Button angezeigt werden soll oder nicht.

```
evaluateable(): boolean {
        return this.caseExecution != null
                && this.caseExecution.active
                && this.assignee != null;
}
```

Listing 8.1: Prüfung für Bewertbarkeit - generisch

```
evaluateable(): boolean {
        return this.evaluation._links[REL_EVALUATE] != null;
}
```

Listing 8.2: Prüfung für Bewertbarkeit - fachlich

Unter Verwendung der generischen API müssen drei Werte geprüft werden, um eine Entscheidung zu treffen. In der Anwendung welche die generische API verwendet wird muss lediglich das Vorhandensein eines Wertes im Objekt unter _links geprüft werden.

Level	
0	• Verwendet HTTP als Transportprotokoll • Ein Endpunkt für alle Anfragen
1	• Identifizierung von Objekten durch Resourcen
2	• Vewendet HTTP Verben entsprechend der HTTP Spezifikation für Operationen auf Resourcen • Verwendet Response Codes entsprechen der HTTP Spezifikation um über das Ergebnis des Aufrufs zu informieren
3	• Setzt HATEOAS um

Tabelle 8.1: Übersicht über REST Reifegrade

8.2 Wartbarkeit

Nach DIN 9126 ist Wartbarkeit ein Qualitätsmerkmal für Software. Die Wartbarkeit eines Systems ergibt sich aus der Modifizierbarkeit, Analysierbarkeit sowie der Testbarkeit. (vgl. ISO9126 (2000)) Die fachliche API erhöht die Wartbarkeit der Frontendanwendung sowohl auf Entwurfsebene als auch im Code selber. Die Wartbarkeit eines Systems lässt sich durch verschiedene Metriken ermitteln. Zwei davon, die in diesem Abschnitt zum Vergleich herangezogen werden, sind Kopplung und Komplexität. (vgl. Riaz u. a. (2009)) Da das Frontend, welches die fachliche Schnittstelle verwendet, auf dem Code und dem Entwurf des Backend aufbaut, werden die Metriken auch darauf angewendet.

8.2.1 Modulkopplung

Die Modulkopplung misst, inwiefern Module miteinander verbunden sind. Geringe Kopplung fördert die Wartbarkeit von Software. Deshalb sollte die Anzahl an Verbindungen so weit

wie möglich reduziert werden (vgl. Sneed (2010)). Die Modulkopplung wird folgendermaßen ermittelt:

$$\frac{\text{Anzahl der Modulverbindungen}}{\text{Anzahl der Module} + \text{Anzahl der Modulverbindungen}} \tag{8.1}$$

Wendet man diese Metrik auf den Entwurf der Frontendanwendungen (Abbildung 6.10 und 6.8) an, ergibt sich für das Frontend mit der generischen API ein Wert von 0.714 und für das Frontend mit der fachlichen API ein Wert von 0.478. Für den Entwurf des Backends ergibt sich damit ein Wert von 0.703. Aufgrund der klaren Aufteilung nach Fachlichkeit muss im Frontend mit der fachlichen API jede Komponente lediglich mit dem Service kommunizieren, der die Operationen auf den von der Komponente verwaltete Entitätstyp anbietet. Damit sinkt die Anzahl der Modulverbindungen im Verhältnis zur der Anzahl der Module und damit insgesamt die Modulkopplung, was angestrebt werden sollte. Hier wird deutlich, dass durch die Entwicklung der fachlichen API die Kopplung, welche die Process Engine mit sich bringt in das Backend verschoben wird. Trotzdem ist ein gewisser Grad an Kopplung unumgänglich, wenn die Komponenten einer Anwendung miteinander kommunizieren.

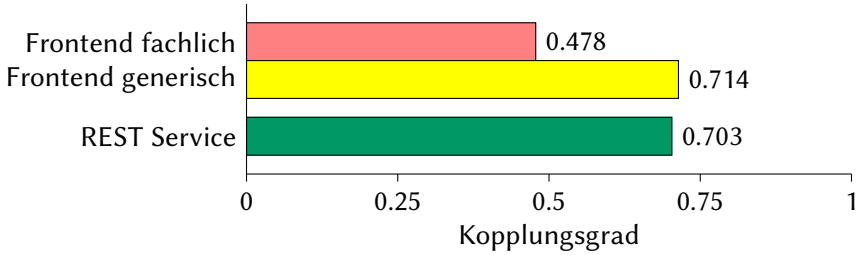

Abbildung 8.1: Übersicht über die Modulkopplung der einzelnen Anwendungen

8.2.2 Entfwurfs-Komplexität

Die Komplexität eines Systems kann sowohl auf Entwurfsebene als auch direkt im Code gemessen werden. Auf Entwurfsebene ergibt sich die Komplexität, ähnlich wie die Kopplung, aus den Verbindungen zwischen den Modulen eines Systems. Eine Möglichkeit, konkrete Werte zu erfassen, bietet die strukturelle Systemkomplexität. Diese wird folgendermaßen ermittelt:

$$\frac{\sum f(i)^2}{n} \tag{8.2}$$

Dabei ist $f(i)$ der fan-out für jedes Modul und n die Anzahl der Module. Fan-out ist die Anzahl ausgehender Aufrufe. (vgl. Sneed (2010)) Wendet man dies auf den auf den Entwurf der Anwendungen an, ergibt sich für die Anwendung, welche die generische API verwendet, ein Wert von 12.5 und für die Anwendung, welche die fachliche API verwendet, ein Wert von 1.75. Im Backend beträgt der Wert für die Entwurfs-Komplexität 10. An diesen Werten zeigt sich, dass die Komplexität durch die Verwendung der fachlichen API in das Backend verlagert.

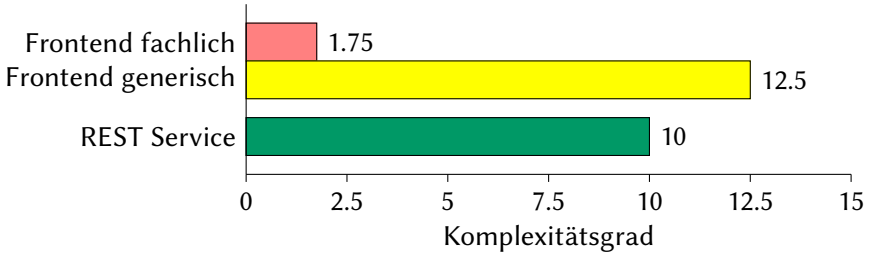

Abbildung 8.2: Übersicht über die Entwurfs-Komplexität der einzelnen Anwendungen

8.2.3 McCabe Metrik

Neben der Komplexität auf Entwurfsebene wird auch die Komplexität des Codes durch die fachliche API reduziert. Um die Codekomplexität zu messen, wird die McCabe-Metrik (zyklomatische Komplexität) verwendet. Die zyklomatische Komplexität einer Methode wird gemessen, indem diese als Graph dargestellt wird. Jede Anweisung und jede Kontrollstruktur wird durch einen Knoten abgebildet. Der mögliche Programmfluss wird durch Kanten dargestellt. Außerdem ist jede Außenverbindung ein Teilgraph. Auf diesen Graphen wird folgende Berechnung durchgeführt:

$$V(s) = e - n + 2p \tag{8.3}$$

Dabei ist e die Anzahl der Kanten n die Anzahl der Knoten und p die Anzahl der Teilgraphen. Zu beachten ist, dass Bedingungen in Kontrollstrukturen, welche durch verknüpfende logische Operatoren gebildet werden, in geschachtelte Verzweigungen aufgeteilt werden müssen. Die Komplexität eines Moduls kann gemessen werden, indem die Werte für die einzelnen Methoden eines Moduls aufaddiert werden. (vgl. McCabe (1976))

Listing 8.3 zeigt die Initialisierungsmethode der Application Komponente, aus der Anwendung mit der generischen Schnittstelle. Der Wert der zyklomatischen Komplexität beträgt hierbei 3. Für die Initialisierungsmethode der Komponente mit der fachlichen API, abgebildet in Listing 8.4, ergibt sich ein Wert von 1.

```
ngOnInit() {
        this.premium = this.application.premiumInCent / 100;
        if (this.caseExecution != null &&
                this.caseExecution.active) {
                this.getTask();
        } else {
                this.complete = true;
        }
        this.getDecision();
}
```

Listing 8.3: Initialisierung der Application-Komponente mit generischer API

```
ngOnInit() {
        this.premium = this.decision.premiumInCent / 100;
}
```

Listing 8.4: Initialisierung der Application-Komponente mit fachlicher API

Für einzelne Methoden zeigt sich, dass die Verwendung der generischen Schnittstelle keinen starken Unterschied ausmacht. Dennoch zeigt sich darin eine Tendenz, die sich bestätigt, sobald diese Metrik auf den gesamten Code angewendet wird. Um die zyklomatische Komplexität der Anwendungen zu messen, wurde das Tool sonarqube[1] verwendet. Für die Anwendung, welche die generische Schnittstelle verwendet, ergibt sich ein Gesamtwert von 382. In der Variante, welche die fachliche Schnittstelle verwendet, liegt der Wert bei 245. Um das Backend in den Vergleich einzubeziehen, wird auch hierfür das Tool sonarqube verwendet. Dabei ergibt sich ein Wert von 308.

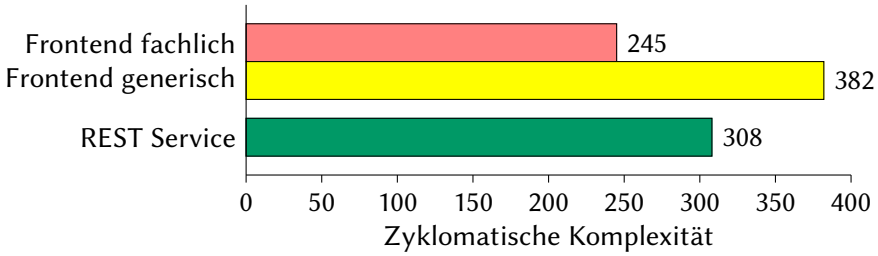

Abbildung 8.3: Übersicht über die zyklomatische Komplexität der einzelnen Anwendungen

[1] sonarqube.org

Hierbei fällt auf, dass die fachliche API die Komplexität im Frontend tatsächlich reduziert. Dennoch liegt die zyklomatsiche Komplexität des Backends dicht an der des Frontends mit der generischen Schnittstelle. Dies lässt sich dadurch erklären, dass in der fachlichen Schnittstelle sämtliche übergebenen Werte auf null sowie das korrekte Format überprüft werden müssen. Im Frontend lassen sich diese Prüfungen durch HTML Attribute in input Tags realisieren. Dadurch wird die Codekomplexität reduziert. Das in Listing 8.5 abgebildete HTML-Tag verdeutlicht dies.

```
1 <input type="date" required/>
```

Listing 8.5: HTML Tag für die Eingabe von Daten

Durch das type Attribut wird verhindert, dass Eingaben, welche nicht dem Format eines Datums entsprechen, bestätigt werden können. Durch das required Attribut kann das Formular mit dem gezeigten Eingabefeld nicht abgeschickt werden, solange dies nicht ausgefüllt ist. Dies verhindert die Übergabe von null Werten.

8.3 Effizienz

Ebenso wie Wartbarkeit ist auch Effizienz nach DIN 9126 ein Merkmal für Softwarequalität. Die Effizienz von Software lässt sich durch das Messen von physikalischen Vorgängen und Größen feststellen. (vgl. Sneed (2010)) Ein wichtiger Faktor für eine verzögerungsfreie Darstellung einer Web Anwendung ist die Anzahl an Requests, die benötigt werden, um eine Seite aufzubauen sowie die Menge der übertragenen Daten. Deshalb werden im Folgenden die Requests gezählt, die benötigt werden, um die Liste aller offenen Anträge darzustellen. Die im vorherigen Kapitel vorgestellten Abbildungen 6.11 und 6.9 visualisieren die benötigten Requests zur Darstellung einer Liste mit einem Antrag, in dem noch kein Task bearbeitet wurde. Für die Darstellung mit der fachlichen Schnittstelle wird genau ein Request benötigt. Es wird eine Liste aller Bewerbungen angefordert. Jede Bewerbung enthält alle Daten die benötigt werden, um diese darzustellen. Unter Verwendung der generischen API werden 7 Requests benötigt, um die Liste mit einem Antrag darzustellen. Der erste Request wird benötigt, um alle laufenden Prozessinstanzen zu erhalten. Anschließend laden die Komponenten weitere Daten, die sie zur Darstellung der Seite benötigen. Dadurch werden sechs weitere Requests benötigt.

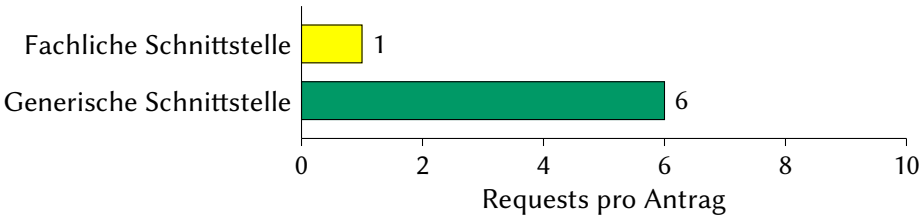

Abbildung 8.4: Übersicht über die Effizienz der einzelnen Anwendungen

8.4 Bedienbarkeit

Die Bedienbarkeit von Software beschreibt, wie gut diese verstanden, erlernt und verwendet werden kann, und wie angenehm es für einen Benutzer ist, diese zu verwenden. Um die Bedienbarkeit der Schnittstellen zu vergleichen wird wie in Macvean u. a. (2016) vorgeschlagen, Code vorgestellt, der die zu bewertende API verwendet. Die Bewertung der Schnittstelle erfolgt allerdings nicht wie in Macvean u. a. (2016) durch Kommentare, sondern durch einen Fragebogen, der das subjektive Empfinden über die vorgestellte API feststellen soll. Um die Bedienbarkeit von Benutzeroberflächen zu bewerten, wurden in der Vergangenheit einige Fragebögen entwickelt. Ein bewährte Möglichkeit ist die System Usability Scale (Brooke). Der Fragebogen besteht aus zehn Aussagen über die Bedienbarkeit, die abwechselnd positiv und negativ sind. Personen, die den Bogen ausfüllen, geben auf einer Skala von eins bis fünf an, wie sehr sie den Aussagen zustimmen. Eins steht dabei für keine Zustimmung. Fünf steht für absolute Zustimmung. Die Auswertung für einen ausgefüllten Bogen findet folgendermaßen statt:

1. Für jede positive Aussage: subtrahiere 1 von dem angekreuzten Wert der Zustimmungsskala.

2. Für jede negative Aussage: subtrahiere den angekreuzten Wert der Zustimmungsskala von 5.

3. Addiere die neuen Werte und multipliziere das Ergebnis mit 2.5.

Wurden die Werte für alle Bögen errechnet, muss für das Endergebnis der Durchschnitt aller ausgefüllten Bögen ermittelt werden. Mit dem Ergebnis dieser Rechnung lassen sich keine genauen Defizite in der Bedienbarkeit ermitteln. Vielmehr wird durch das Ergebnis die empfundene Bedienbarkeit auf einer Skala von 0 bis 100 ausgedrückt. Dabei steht 0 für nicht bedienbar und 100 für perfekt bedienbar. Für die Bewertung der Schnittstellen wurden die

Fragen fünf und sechs ersetzt, da diese mit dem Umfang des Codebeispiels nicht beantwortet werden können. Außerdem wurden die Fragen in der Formulierung so angepasst, dass diese sich auf die vorgestellte API beziehen. Der vollständige Fragebogen ist dem Anhang A.2 beigefügt. Für den Vergleich wurden zwei Gruppen aus jeweils zehn Entwicklern befragt. Zu Beginn wurde jeder Gruppe die Versicherungsanwendung und ihre Funktionalitäten vorgestellt. Anschließend wurden die dem Fragebogen beigefügten Codebeispiele (Anhang A.3) erläutert sowie Fragen dazu beantwortet. Der ersten Gruppe wurde Code gezeigt, der die fachliche Schnittstelle verwendet. Der anderen wurde Code präsentiert, der die generische Schnittstelle verwendet. Die Codebeispiele der beiden Gruppen implementieren dieselbe Funktionalität, verwendeten jedoch die zwei unterschiedlichen APIs. In den gezeigten Beispielen wurde nicht Code aus den entwickelten Frontend Anwendungen, sondern Python Code gezeigt, da dieser ähnlich lesbar wie Pseudocode ist und ohne die für JavaScript bzw. Typescript typischen Callbacks auskommt. Die Codebeispiele zeigen die wichtigsten Unterschiede der Schnittstellen, indem gezeigt wird, wie Daten dargestellt werden und welche Prüfungen ein Client tätigen muss. Die Ergebnisse sind aufgrund des Umfangs der Befragung nicht repräsentativ, zeigen allerdings eine Tendenz, die Aufschlüsse über die Bedienbarkeit gibt. Die Auswertung zeigt, dass die fachliche Schnittstelle mit einem SUS-Score von 72.75 als bedienbarer empfunden wird als die generische Schnittstelle, bei der ein SUS-Score von 47.25 erreicht wurde.

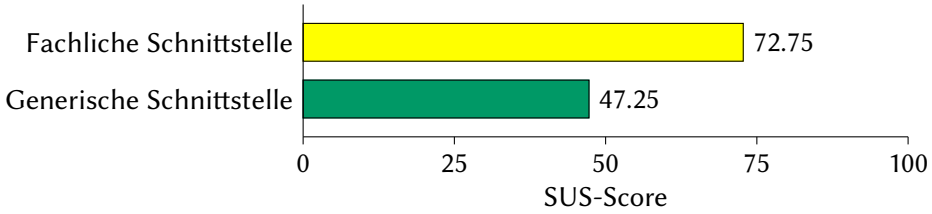

Abbildung 8.5: Übersicht über den SUS-Score der einzelnen Anwendungen

8.5 Bewertung

Metrik	fachliche Schnittstelle	generische Schnittstelle
REST Reifegrad	3	2
SUS-Score	72.75	47.25
Requests pro dargestelltem Antrag	1	6

Tabelle 8.2: Übersicht über die ermittelten Werte für die Schnittstellenarten

Metrik	Frontend - fachlich	Frontend - generisch	REST Service
Modulkopplung	0.478	0.714	0.703
Entwurfskomplexität	1.75	12.5	10
McCabe Metrik	245	382	308

Tabelle 8.3: Übersicht über die ermittelten Werte für die entwickelten Anwendungen

Unter Betrachtung der zuvor erläuterten Metriken zeigt sich, dass eine fachliche REST API die Komplexität im Frontend senkt. Dies wird unter anderem durch eine hochwertige Umsetzung von REST gefördert. Dabei ist allerdings zu beachten, dass der Aufwand, eine fachliche Schnittstelle zu entwickeln, nicht eins zu eins den Aufwand ersetzt, der entsteht, wenn eine generische Schnittstelle im Frontend verwendet wird. Dem API Entwickler obliegt die Aufgabe, sämtliche Eingaben zu prüfen. Dies sollte zwar ebenfalls von einem Frontendentwickler getan werden, lässt sich allerdings, wie in 8.2.3 gezeigt, durch die Verwendung von HTML Attributen stark vereinfachen. Zusätzlich können die in 6.2.2 beschriebenen generischen Services wiederverwendet werden, was den Entwicklungsaufwand mit einer generischen Schnittstelle weiter reduziert. Außerdem muss die fachliche Schnittstelle getestet werden. Dies ist mit Bibliotheken wie REST Assured möglich, erzeugt aber zusätzliche Kosten. Die Schnittstelle der Camunda Process Engine ist bereits getestet, weshalb der Testaufwand hierfür entfällt. Ein nicht gemessener, aber dennoch wichtiger Unterschied in den Schnittstellenvarianten ist die Austauschbarkeit der Engine. Mit der generischen Schnittstelle ist es möglich, die verwendete Engine durch ein anderes Produkt oder eine eigene Implementierung auszutauschen. Dabei muss kein Client an die Änderung angepasst werden. Die generische Schnittstelle koppelt die Clients stark an die verwendete Engine. Ob für einen Prozess eine fachliche Schnittstelle entwickelt wird, sollte von dem Einsatzszenario sowie den Benutzern der Schnittstelle abhängig gemacht werden.

Zunächst sollte ein nicht öffentliches BPMN Projekt betrachtet werden, in dem Entwickler, die mit einer Process Engine vertraut sind, eine Anwendung realisieren, die auf einer Process Engine basiert. In diesem Fall erzeugt eine fachliche Schnittstelle weiteren Aufwand und damit zusätzliche Kosten, die sich durch Verwenden der generischen Schnittstelle vermeiden lassen. Jedoch kann es sein, dass in einem größeren Projekt nicht alle Beteiligten mit der BPMN vertraut sind. Wenn beispielsweise Frontendentwickler beteiligt sind, die keine Erfahrung mit der BPMN sowie der Process Engine haben, ersetzt die fachliche Schnittstelle das Einarbeiten in die Process Engine. Allerdings sind hierbei Kosten und Nutzen einer Weiterbildung im Vergleich mit der Entwicklung einer fachlichen Schnittstelle abzuwägen.

In einer Microservice Umgebung hingegen erscheint eine fachliche Schnittstelle durchaus sinnvoll. Bei der Aufteilung von Microservices sollte jeder Service eine fachliche Einheit bilden. Jeder Service bildet einen Bounded Context. (vgl. Wolff (2018)) Nach dem Entwurfsmuster aus dem Domain Driven Design, grenzt ein Bounded Context einen Domänen- oder Subdomänenbereich ein. (vgl. Vernon u. a. (2017)) Für die Kommunikation von Services über Kontextgrenzen hinweg gibt es eine Reihe von Pattern (Shared Kernel,Customer/Supplier, Conformist, Anticorruption Layer, Open Host Service, Published Language). (vgl. Wolff (2018)) Alle Pattern vereint, dass ein Service eine Schnittstelle bereitstellt, die das Domänenmodell des Service verwendet. Die fachliche Schnittstelle fördert hierbei den Gedanken des Domain Driven Desgin. Außerdem werden durch die Schnittstelle Entwickler anderer Services unterstützt, die keine Erfahrung mit Process Engines haben, aber die Schnittstelle verwenden müssen. Deshalb sollte die Anwendung in diesem beschriebenen Szenario eine fachliche Schnittstelle anbieten.

Zuletzt sollte der Fall betrachtet werden, in dem die API nicht nur intern verwendet wird, sondern der Öffentlichkeit zur Verfügung gestellt ist. In dem Fall sollte ein Unternehmen die angebotene API als Produkt betrachten. Twitter steigerte die Nutzung der angebotenen Plattform durch die Twitter-API, welche eine Vielzahl an Twitter-Apps hervorbrachte. Andere Unternehmen wie FedEx oder Walmart haben das Ziel, mit einer öffentlichen API einen zusätzlichen Verkaufskanal zu schaffen. (vgl. Spichale (2017)) Wenn es sich bei der API um ein Produkt handelt, sollte diese dem Kunden dienen und komfortabel sein. Ein fachliche Schnittstelle erhöht die Wartbarkeit des Clientcodes und ist bedienbarer als eine generische. Daher ist in diesem Anwendungsbeispiel die fachliche API der generischen vorzuziehen.

9 Zusammenfassung und Ausblick

Im Folgenden werden kurz die Ergebnisse der Arbeit zusammengefasst. Zudem sollen mögliche Ansatzpunkte für aufbauende Arbeiten kurz vorgestellt werden.

Das Ziel dieser Arbeit war, den Nutzen und die Nachteile einer fachlichen Schnittstelle für einen Prozess in einer Process Engine herauszuarbeiten. Hierfür wurde ein Fallbeispiel ausgewählt und vorgestellt, um dafür eine fachliche Schnittstelle sowie eine Anwendung, die darauf aufbaut, zu entwickeln. Dafür musste die fachliche Schnittstelle spezifiziert werden. Darüber hinaus wurde ein Vorschlag für die Umsetzung einer fachlichen Schnittstelle für eine Process Engine erarbeitet und anschließend implementiert. Im Vergleich zeigte sich, dass die fachliche Schnittstelle einfacher zu bedienen ist und die Wartbarkeit sowie die Effizienz von Clientcode fördert. Trotzdem ist der Aufwand für die Entwicklung einer fachlichen Schnittstelle mit einem Client insgesamt höher als der Aufwand, einen Client zu entwickeln, der die generische Schnittstelle benutzt. Trotzdem sollte der Rahmen beachtet werden, in dem die Schnittstelle eingesetzt wird. Hierfür wurden im letzten Kapitel einige mögliche Einsatzszenarien aufgezeigt sowie die Vor- und Nachteile einer fachlichen Schnittstelle, die im jeweiligen Szenario zu beachten sind, erläutert.

Um die in dieser Arbeit vorgestellten Vorteile einer fachlichen Schnittstelle zu erhalten, ohne dass zusätzlicher Entwicklungsaufwand entsteht, ist es denkbar, eine fachliche Schnittstelle sowie das Mapping zwischen Fachlichkeit und Process Engine automatisch generieren zu lassen. Dazu könnte das XML-Dokument, welches die Prozesse und Cases definiert, um eigene Tags erweitert werden, welche die Schnittstelle mit ihren Endpunkten und Datentypen spezifizieren. Die BPMN und CMMN Spezifikation lässt Erweiterungen der Dokumente durch eigene Tags zu. Für weiterführende Arbeiten wäre ebenfalls denkbar, die fachliche Schnittstelle mit Hilfe einer Domain Specific Language zu generieren, welche die Endpunkte der Schnittstelle sowie die benötigte Übersetzung beschreibt. Diese Lösung beschränkt sich nicht nur auf Process Engines, sondern kann beispielsweise auch in der Übersetzung von API Versionen oder unterschiedlichen Bounded Contexts eingesetzt werden.

Literaturverzeichnis

[ISO9126 2000] Informationtechnology - Softwareproductquality / International Organization for Standardization. Geneva, CH, 2000. – Standard

[Bloch 2006] Bloch, Joshua: How to Design a Good API and Why It Matters. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA : ACM, 2006 (OOPSLA '06), S. 506–507. – URL http://doi.acm.org/10.1145/1176617.1176622. – ISBN 1-59593-491-X

[Bloch 2014] Bloch, Joshua: A Brief, Opinionated History of the API, 2014

[Brooke] Brooke, John / Redhatch Consulting Ltd. – Forschungsbericht

[Brzeziński u. a. 2012] Brzeziński, Jerzy ; Danilecki, Arkadiusz ; Flotyński, Jakub ; Kobusińska, Anna ; Stroiński, Andrzej: ROsWeL Workflow Language: A Declarative, Resource-oriented Approach. In: *New Generation Computing* 30 (2012), Jun, Nr. 2, S. 141–164. – URL https://doi.org/10.1007/s00354-012-0203-y. – ISSN 1882-7055

[Camunda a] Camunda: *Camunda Docs*. – URL https://docs.camunda.org/manual/7.8/. – Letzter Zugriff: 28.08.2018

[Camunda b] Camunda: Case Management and CMMN for Developers / Camunda. – Forschungsbericht

[Camunda c] Camunda: *Feature Request - CAM-6087*. – URL https://app.camunda.com/jira/browse/CAM-6087. – Letzter Zugriff: 15.09.2018

[Camunda d] Camunda: *User Guide*. – URL https://docs.camunda.org/manual/7.8/user-guide/. – Letzter Zugriff: 28.08.2018

[European Association of Business Process Management 2014] European Association of Business Process Management: *Business process management BPM common body of*

knowledge - BPM CBOK - Leitfaden für das Prozessmanagement ; Version 3.0. Mainz : Schmidt, 2014. – ISBN 978-3-921-31380-0

[Fielding und Reschke 2014] Fielding, R. ; Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content / RFC Editor. RFC Editor, June 2014 (7231). – RFC. – URL http://www.rfc-editor.org/rfc/rfc7231.txt. http://www.rfc-editor.org/rfc/rfc7231.txt. – ISSN 2070-1721

[Fielding 2000] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Dissertation, 2000

[Fowler 2010] Fowler, Martin: *Richardson Maturity Model.* 2010. – URL https://martinfowler.com/articles/richardsonMaturityModel.html. – Letzter Zugriff: 28.08.2018

[Freund 2016] Freund, R: *Praxishandbuch BPMN 2.0 5.A.* Carl Hanser Verlag GmbH & Co, 2016. – ISBN 3446450548

[Henning 2007] Henning, Michi: API Design Matters. In: *Queue* 5 (2007), Mai, Nr. 4, S. 24–36. – URL http://doi.acm.org/10.1145/1255421.1255422. – ISSN 1542-7730

[Kelly 2016] Kelly, Mike: JSON Hypertext Application Language / IETF Secretariat. URL http://www.ietf.org/internet-drafts/draft-kelly-json-hal-08.txt, May 2016 (draft-kelly-json-hal-08). – Internet-Draft

[Macvean u. a. 2016] Macvean, Andrew ; Maly, Martin ; Daughtry, John: API Design Reviews at Scale. In: *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems.* New York, NY, USA : ACM, 2016 (CHI EA '16), S. 849–858. – URL http://doi.acm.org/10.1145/2851581.2851602. – ISBN 978-1-4503-4082-3

[McCabe 1976] McCabe, T. J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), Dec, Nr. 4, S. 308–320. – ISSN 0098-5589

[Microsoft Developer Division] Microsoft Developer Division: *.NET-Microservices-Architektur für .NET-Containeranwendungen.* URL https://docs.microsoft.com/de-de/dotnet/standard/microservices-architecture/index

[Nottingham 2010] Nottingham, M.: Web Linking / RFC Editor. RFC Editor, October 2010 (5988). – RFC. – URL http://www.rfc-editor.org/rfc/rfc5988.txt. http://www.rfc-editor.org/rfc/rfc5988.txt. – ISSN 2070-1721

[OMG 2011] OMG: *Business Process Model and Notation (BPMN), Version 2.0.* January 2011. – URL https://www.omg.org/spec/BPMN/2.0

[OMG 2016a] OMG: *Case Model Management and Notation (CMMN), Version 1.1.* December 2016. – URL https://www.omg.org/spec/CMMN/About-CMMN/

[OMG 2016b] OMG: *Decision Model and Notation (DMN), Version 1.1.* June 2016. – URL https://www.omg.org/spec/DMN/About-DMN/

[Pautasso 2011] Pautasso, Cesare: BPMN for REST. In: Dijkman, Remco (Hrsg.) ; Hofstetter, Jörg (Hrsg.) ; Koehler, Jana (Hrsg.): *Business Process Model and Notation.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, S. 74–87. – ISBN 978-3-642-25160-3

[Riaz u. a. 2009] Riaz, M. ; Mendes, E. ; Tempero, E.: A systematic review of software maintainability prediction and metrics. In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, Oct 2009, S. 367–377. – ISSN 1949-3770

[Sarstedt 2017] Sarstedt, Stefan: *Vorlesung: Architektur von Informationssystemen, Wintersemester 2017.* 2017

[Sneed 2010] Sneed, B.: *Software in Zahlen.* Carl Hanser Verlag GmbH & Co, 2010. – ISBN 3446421750

[Spichale 2017] Spichale, K.: *API-Design: Praxishandbuch für Java- und Webservice-Entwickler.* dpunkt.verlag, 2017. – ISBN 9783960880783

[Vernon u. a. 2017] Vernon, V. ; Lilienthal, C. ; Schwentner, H.: *Domain-Driven Design kompakt.* Dpunkt.Verlag GmbH, 2017. – ISBN 9783864904394

[Wolff 2018] Wolff, Eberhard: *Microservices - Grundlagen flexibler Softwarearchitekturen.* dpunkt, 2018. – ISBN 3864905559

Anhang

A.1 _links Feld für eine Task Resource

```
"_links": {
        "caseDefinition": {
        "href": "/case-definition/check_application:1:
        318d064e-a473-11e8-9a5-0242791beae7"
        },
        "caseExecution": {
        "href": "/case-execution
        /fef58e40-a473-11e8-9a5-0242791beae7"
        },
        "caseInstance": {
        "href": "/case-instance
        /feeb55ec-a473-11e-9a52-0242791beae7"
        },
        "identityLink": {
        "href": "/task/ff025f85-a473-11e8-9a52-0242791beae7
        /identity-links"
        },
        "self": {
        "href": "/task/ff025f85-a473-11e8-9a52-0242791beae7"
        }
}
```

A.2 Fragebogen für APIs

1. I think that I would like to use this API frequently

Strongly disagree				Strongly agree
1	2	3	4	5

2. I think the API is unnecessarily complex

Strongly disagree				Strongly agree
1	2	3	4	5

3. I think the API is easy to use

Strongly disagree				Strongly agree
1	2	3	4	5

4. I think that I would need the support of a technical person to be able to use this API

Strongly disagree				Strongly agree
1	2	3	4	5

5. The naming supports the understandability

Strongly disagree				Strongly agree
1	2	3	4	5

6. The use of the API seems unintuitive.

Strongly disagree				Strongly agree
1	2	3	4	5

7. I would imagine that most people would learn to use this API very quickly

Strongly disagree				Strongly agree
1	2	3	4	5

8. I find the API very cumbersome to use

Strongly disagree				Strongly agree
1	2	3	4	5

9. I feel very confident using the API

Strongly disagree				Strongly agree
1	2	3	4	5

10. I need to learn a lot of things before I could get going with this API

Strongly disagree				Strongly agree
1	2	3	4	5

A.3 Codebeispiele für SUS Bewertung

```
import requests
def request_evaluation(case_execution: CaseExecution):
   # check if application can be evaluated
   if case_execution.enabled or case_execution.available:
       raise Exception('An evaluation can\'t be requested for the given
application')

   response = requests.post('http://foo.bar/case-execution/'+case_execution.id)
   response_data = response.json()
   # check for errors
   if response.status_code >= 400:
       error_message = response_data['message']
       raise Exception(error_message)
   return response_data

def decide_on_application(case_execution: CaseExecution, approve, premium):
   # check if application can be decided
   if not case_execution.active:
       raise Exception('Can\'t decide on  given application.')

   response =
requests.get('http://foo.bar/task?caseExecutionId='+case_execution.id)
   response_data = response.json()
   # a decision is a task. Get the task id of the decision for the given
application
   if response.status_code >= 400:
       raise Exception(response.json()['message'])

   task_id = response_data['id']
   # check if task has an assignee
   response = requests.get('http://foo.bar/task/' + task_id + '/assignee')
   response_data = response.json()
   # check if task is assigned
   if response.status_code >= 400 or 'assignee' not in response_data:
       raise Exception(response_data['message'])
   # build decision object for the REST API
   decision = {'approve':
                   {'value': approve,
                    'type':'boolean',
                    'valueInfo':{}},
               'premium':
                   {'value': premium,
                    'type': 'number',
                    'valueInfo': {}}}
   # complete the decision
   response = requests.put('http://foo.bar/task/' + task_id + '/complete',
json=decision)
   response_data = response.json()
   if response.status_code >= 400:
       error_message = response_data['message']
       raise Exception(error_message)
   return response_data
```

Abbildung 1: Codebeispiel für die generische Schnittstelle

```
import requests

def request_evaluation(application: Application):
   # check if application can be evaluated
   if 'evaluate' not in application._links:
       raise Exception('An evaluation can\'t be requested for the given
application')

   evaluation_uri = application._links['evaluate']
   response = requests.post(evaluation_uri)
   response_data = response.json()
   # check for errors
   if response.status_code >= 400:
       error_message = response_data['message']
       raise Exception(error_message)

   return response_data

def decide_on_application(application: Application, approve, premium):
   # check if application can be decided
   if 'decide' not in application._links:
       raise Exception('Can\'t decide on  given application.')

   decision_uri = application._links['decide']
   # build decision object for the REST API
   decision = {'approve': approve,
               'premium': premium,
               'complete': True}

   response = requests.put(decision_uri, json=decision)
   response_data = response.json()
   # check for errors
   if response.status_code >= 400:
       error_message = response_data['message']
       raise Exception(error_message)

   return response_data
```

Abbildung 2: Codebeispiel für die fachliche Schnittstelle

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 24. September 2018 Dennis Pietruck